

O'REILLY®

Wydanie II

Kafka w praktyce

Przetwarzanie strumieniowe
i potoki danych o dużej skali



Helion

Gwen Shapira, Todd Palino
Rajini Sivaram, Krit Petty

Tytuł oryginału: Kafka: The Definitive: Guide Real-Time Data and Stream Processing at Scale, 2nd Edition

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-9321-9

© 2022 Helion S.A.

Authorized Polish translation of the English *Kafka: The Definitive Guide* ISBN 9781492043089

© 2022 Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

https://helion.pl/user/opinie/kafpra_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/kafpra.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Opinie o książce

Kafka w praktyce zawiera wszystko, co musisz wiedzieć, aby wydobyć jak najwięcej z Kafki, czy to w chmurze, czy w sieci lokalnej. Niezbędnik zarówno dla programistów, jak i operatorów. Gwen, Todd, Rajini i Krit zamieniają lata doświadczeń w jedną zwięzłą książkę. Potrzebujesz tego przewodnika, jeśli uruchamiasz lub wykorzystujesz Kafkę.

— *Chris Riccomini, inżynier oprogramowania, doradca ds. start-upów i współautor książki Brakujący plik README*

Kompleksowy przewodnik po podstawach Kafki i korzystaniu z niej.
— *Sumant Tambe, starszy inżynier oprogramowania w firmie LinkedIn*

Ta książka jest niezbędną lekturą dla każdego programisty lub administratora Kafki. Przeczytaj ją od deski do deski, aby zanurzyć się w szczegółach Kafki, lub trzymaj ją pod ręką, by zawsze móc po nią sięgnąć. Tak czy inaczej znakomicie napisana i dokładna pod względem technicznym pozycja.

— *Robin Moffatt, specjalista ds. rozwoju kadr w firmie Confluent*

Jest to fundamentalna lektura dla wszystkich inżynierów zainteresowanych Kafką. Bardzo pomogła firmie Robinhood w skalowaniu, modernizacji i dostrajaniu Kafki, aby obsłużyć nasz szybki przyrost liczby użytkowników.

— *Jaren M. Glover, jeden z pierwszych inżynierów w firmie Robinhood, anioł biznesu*

Niezbędnik dla wszystkich, którzy pracują z platformą Apache Kafka, bez względu na to, czy są programistami czy administratorami, początkującymi czy ekspertami, użytkownikami czy kontrybutorami.

— *Matthias J. Sax, inżynier oprogramowania w firmie Confluent i członek Apache Kafka PMC*

Świetne wskazówki dla każdego zespołu poważnie wykorzystującego platformę Apache Kafka w środowisku produkcyjnym i generalnie dla inżynierów pracujących nad systemami rozproszonymi. Ta książka wykracza daleko poza zwykłe wprowadzenie, a autorzy omawiają, w jaki sposób faktycznie działa Kafka, jak powinna być używana i gdzie kryją się pułapki. Dla każdej wspaniałej funkcjonalności Kafki opisują wszelkie zastrzeżenia, o których usłyszysz tylko od posiwiących weteranów Kafki.

Informacje te nie są łatwo dostępne w żadnym innym miejscu. Opisy są na tyle jasne i pogłębione, że poleciłbym ją nawet inżynierom, którzy nie korzystają z Kafki — poznanie zasad, wyborów projektowych i operacyjnych pułapek pomoże im w podejmowaniu lepszych decyzji przy tworzeniu innych systemów.

— *Dmitriy Ryaboy, wiceprezes ds. inżynierii oprogramowania w firmie Zymergen*

Spis treści

Przedmowa do wydania II	15
Przedmowa do wydania I	17
Wstęp	21
1. Poznaj Kafkę	25
Publikuj-Subskrybuj	25
Jak to się zaczyna?	26
Indywidualne systemy kolejkowe	27
Pojawienie się Kafki	28
Komunikaty i partie	28
Schematy	28
Tematy i partycje	29
Producenci i konsumenci	30
Brokery i klastry	31
Wiele klastrów	32
Dlaczego Kafka?	33
Wiele producentów	33
Wiele konsumentów	34
Retencja na dysku	34
Skalowalność	34
Wysoka wydajność	34
Funkcjonalności platformy	35
Ekosystem danych	35
Przypadki użycia	36
Pochodzenie Kafki	37
Problem LinkedIna	37
Narodziny Kafki	38
Open source	39

Zaangażowanie komercyjne	39
Nazwa	40
Pierwsze kroki z Kafką	40
2. Instalowanie Kafki	41
Konfiguracja środowiska	41
Wybór systemu operacyjnego	41
Instalowanie Javy	41
Instalowanie ZooKeepera	42
Instalowanie brokera Kafki	45
Konfiguracja brokera	46
Ogólne parametry brokera	46
Domyślne ustawienia tematu	49
Wybór sprzętu	54
Przepustowość dysku	54
Pojemność dysku	55
Pamięć	55
Konfiguracja sieciowa	55
Procesor	56
Kafka w chmurze	56
Microsoft Azure	56
Amazon Web Services	57
Konfiguracja klastrów Kafki	57
Ile brokerów?	57
Konfiguracja brokera	59
Dostosowywanie systemu operacyjnego	59
Zagadnienia produkcyjne	62
Opcje mechanizmu odzyskiwania pamięci	62
Układ centrum danych	64
Współlokowanie aplikacji na ZooKeeperze	64
Podsumowanie	66
3. Producenty Kafki — zapisywanie komunikatów w Kafce	67
Omówienie producenta	68
Konstruowanie producenta Kafki	69
Wysyłanie komunikatów do Kafki	71
Synchroniczne wysyłanie komunikatów	72
Asynchroniczne wysyłanie komunikatów	73
Konfiguracja producentów	74
client.id	74
acks	74

Czas dostarczania komunikatów	75
linger.ms	77
buffer.memory	78
compression.type	78
batch.size	78
max.in.flight.requests.per.connection	78
max.request.size	79
receive.buffer.bytes i send.buffer.bytes	79
enable.idempotence	80
Serializatory	80
Serializatory niestandardowe	80
Serializacja przy użyciu Apache Avro	82
Używanie rekordów Avry z Kafką	84
Partycje	86
Wdrażanie niestandardowej strategii partycjonowania	88
Nagłówki	89
Przechwytywacze	89
Kwoty i dławienie przepływności	91
Podsumowanie	92
4. Konsumenty Kafki — odczytywanie danych z Kafki	94
Koncepcje konsumentenckie Kafki	94
Konsumenty i grupy konsumentów	94
Grupy konsumentów i równoważenie partycji	97
Statyczne członkostwo w grupie	100
Tworzenie konsumenta Kafki	101
Subskrybowanie tematów	101
Pętla odpytywania	102
Bezpieczeństwo wątków	103
Konfiguracja konsumentów	104
fetch.min.bytes	104
fetch.max.wait.ms	105
fetch.max.bytes	105
max.poll.records	105
max.partition.fetch.bytes	105
session.timeout.ms i heartbeat.interval.ms	106
max.poll.interval.ms	106
default.api.timeout.ms	107
request.timeout.ms	107
auto.offset.reset	107
enable.auto.commit	107

partition.assignment.strategy	107
client.id	109
client.rack	109
group.instance.id	109
receive.buffer.bytes i send.buffer.bytes	109
offsets.retention.minutes	109
Zatwierdzenia i przesunięcia	110
Zatwierdzanie automatyczne	111
Zatwierdzanie bieżących przesunięć	112
Zatwierdzanie asynchroniczne	113
Łączenie zatwierdzeń synchronicznych i asynchronicznych	114
Zatwierdzanie określonego przesunięcia	115
Nasłuchiwanie równoważenia obciążenia	116
Konsumowanie rekordów z określonymi przesunięciami	118
Wychodzenie z pętli	119
Deserializatory	121
Niestandardowe deserializatory	122
Stosowanie deserializacji Avry z konsumentem Kafki	124
Samodzielny konsument — dlaczego i jak korzystać z konsumenta bez grupy?	124
Podsumowanie	125
5. Programowe zarządzanie Kafką	127
Przegląd interfejsu AdminClient	128
Asynchroniczny i ostatecznie spójny interfejs API	128
Opcje	128
Hierarchia płaska	128
Dodatkowe uwagi	129
Cykl życia AdminClient API — tworzenie, konfiguracja i zamykanie	129
client.dns.lookup	130
request.timeout.ms	131
Podstawowe zarządzanie tematami	131
Zarządzanie konfiguracją	135
Zarządzanie grupą konsumentów	136
Eksploracja grup konsumentów	136
Modyfikowanie grup konsumentów	138
Metadane klastra	139
Zaawansowane operacje administracyjne	140
Dodawanie partycji do tematu	140
Usuwanie rekordów z tematu	141
Wybór lidera	141
Ponowne przypisywanie replik	142

Testowanie	143
Podsumowanie	145
6. Wewnętrzne mechanizmy działania Kafki	146
Przynależność do klastra	146
Kontroler	147
KRaft — nowy kontroler Kafki oparty na algorytmie Raft	148
Replikacja	150
Przetwarzanie żądań	152
Żądania produkcji	154
Żądania pobierania	155
Inne żądania	157
Fizyczna pamięć masowa	159
Warstwowy system pamięci masowej	159
Przydzielanie partycji	161
Zarządzanie plikami	162
Format plików	163
Indeksy	165
Kompaktowanie	165
Jak działa kompaktowanie?	166
Usunięte zdarzenia	167
Kiedy tematy są kompaktowane?	168
Podsumowanie	169
7. Niezawodne dostarczanie danych	170
Gwarancje niezawodności	170
Replikacja	171
Konfiguracja brokera	173
Współczynnik replikacji	173
Wybór nieczystego lidera	175
Minimalna liczba replik zsynchronizowanych	176
Utrzymywanie synchronizacji replik	176
Utrwalanie na dysku	177
Korzystanie z producentów w niezawodnym systemie	177
Wysyłanie potwierdzeń	178
Konfigurowanie prób ponawiania przez producenta	179
Dodatkowa obsługa błędów	180
Korzystanie z konsumentów w niezawodnym systemie	180
Ważne właściwości konfiguracji konsumenta	
w celu niezawodnego przetwarzania	181
Bezpośrednie zatwierdzanie przesunięć w konsumentach	182

Walidacja niezawodności systemu	184
Walidacja konfiguracji	184
Walidacja aplikacji	185
Monitorowanie niezawodności w środowisku produkcyjnym	186
Podsumowanie	187
8. Semantyka „dokładnie raz”	189
Producent idempotentny	190
Jak działa idempotentny producent?	190
Ograniczenia producenta idempotentnego	192
Jak korzystać z producenta idempotentnego Kafki?	193
Transakcje	193
Przypadki użycia transakcji	194
Jakie problemy rozwiązują transakcje?	194
W jaki sposób transakcje gwarantują semantykę „dokładnie raz”?	195
Jakich problemów nie rozwiązują transakcje?	198
Jak korzystać z transakcji?	200
Identyfikatory transakcyjne i odgradzanie	203
Jak działają transakcje?	204
Wydajność transakcji	206
Podsumowanie	206
9. Budowanie potoków danych	208
Zagadnienia związane z budowaniem potoków danych	209
Terminowość	209
Niezawodność	209
Wysoka i zmienna przepustowość	210
Format danych	211
Transformacje	211
Bezpieczeństwo	212
Postępowanie w razie awarii	213
Powiązania i zwinność	213
Przypadki użycia dla frameworku Kafka Connect	
oraz dla producentów i konsumentów	214
Kafka Connect	215
Uruchamianie frameworku Kafka Connect	216
Przykład konektora — źródło plików i ujście plików	218
Przykład konektora — z MySQL-a do Elasticsearcha	220
Pojedyncze transformacje komunikatów	226
Jak działa framework Kafka Connect?	228

Alternatywy dla frameworku Kafka Connect	231
Frameworki pobierania dla innych magazynów danych	231
Narzędzia ETL oparte na GUI	231
Frameworki przetwarzania strumieniowego	232
Podsumowanie	232
10. Mirroring danych między klastrami	234
Przypadki użycia dla mirroringu między klastrami	234
Architektury wieloklastrowe	236
Realia komunikacji między centrami danych	236
Architektura piasty i szprych	237
Architektura aktywny-aktywny	239
Architektura aktywny-pasywny	241
Klastry rozciągnięte	246
MirrorMaker platformy Apache Kafki	247
Konfigurowanie MirrorMakera	249
Topologia replikacji wieloklastrowej	251
Zabezpieczanie MirrorMakera	252
Wdrażanie MirrorMakera w środowisku produkcyjnym	253
Regulowanie MirrorMakera	256
Inne rozwiązania do mirroringu między klastrami	258
uReplicator Ubera	259
Brooklin LinkedIna	260
Rozwiązania do mirroringu między centrami danych firmy Confluent	260
Podsumowanie	262
11. Zabezpieczanie Kafki	263
Zamykanie dostępu do Kafki	263
Protokoły bezpieczeństwa	265
Uwierzytelnianie	267
SSL	267
SASL	272
Ponowne uwierzytelnianie	283
Aktualizowanie zabezpieczeń bez przestojów	284
Szyfrowanie	285
Szyfrowanie kompleksowe	286
Autoryzacja	287
AclAuthorizer	288
Dostosowywanie autoryzacji	291
Kwestie bezpieczeństwa	293
Audyt	294

Zabezpieczanie ZooKeepera	294
SASL	295
SSL	295
Autoryzacja	296
Zabezpieczanie platformy	296
Zabezpieczanie hasłem	297
Podsumowanie	298
12. Administrowanie Kafką	300
Operacje na tematach	300
Tworzenie nowego tematu	301
Wyświetlanie listy wszystkich tematów w klastrze	302
Generowanie opisu tematu	302
Dodawanie partycji	304
Redukcja liczby partycji	305
Usuwanie tematu	305
Grupy konsumentów	306
Wyświetlanie listy i generowanie opisu grup	306
Usuwanie grup	307
Zarządzanie przesunięciami	307
Dynamiczne zmiany konfiguracji	309
Nadpisywanie wartości domyślnych konfiguracji tematu	309
Nadpisywanie wartości domyślnych konfiguracji klienta i użytkownika	311
Nadpisywanie wartości domyślnych konfiguracji brokera	312
Generowanie opisów nadpisanych konfiguracji	312
Usuwanie nadpisanych konfiguracji	313
Produkowanie i konsumowanie	313
Producent konsolowy	313
Konsument konsolowy	315
Zarządzanie partycjami	318
Wybór repliki preferowanej	318
Zmienianie replik partycji	320
Zrzucanie segmentów dziennika	324
Weryfikacja replik	326
Pozostałe narzędzia	326
Niebezpieczne operacje	327
Przenoszenie kontrolera klastra	327
Rozwiązywanie problemów z usuwaniem tematów	328
Ręczne usuwanie tematów	328
Podsumowanie	329

13. Monitorowanie Kafki	330
Podstawy monitorowania	330
Gdzie są wskaźniki?	330
Jakich wskaźników potrzebujesz?	332
Kontrolowanie kondycji aplikacji	333
SLO	333
Definicje dotyczące poziomu usług	334
Jakie wskaźniki nadają się na SLI?	335
Stosowanie SLO w ostrzeganiu	335
Wskaźniki brokerów Kafki	336
Diagnozowanie problemów z klastrami	337
Sztuka posługiwania się wskaźnikiem URP	338
Wskaźniki brokera	343
Wskaźniki tematów i partycji	351
Monitorowanie JVM	353
Monitorowanie systemu operacyjnego	354
Rejestrowanie	356
Monitorowanie klienta	357
Wskaźniki producenta	357
Wskaźniki konsumenta	359
Kwoty	362
Monitorowanie opóźnień	363
Monitorowanie kompleksowe	364
Podsumowanie	364
14. Przetwarzanie strumieniowe	366
Czym jest przetwarzanie strumieniowe?	367
Koncepcje przetwarzania strumieniowego	370
Topologia	370
Czas	370
Stan	372
Dualność strumieniowo-tablicowa	373
Okna czasowe	374
Gwarancje przetwarzania	376
Wzorce projektowe przetwarzania strumieniowego	376
Przetwarzanie pojedynczego zdarzenia	376
Przetwarzanie ze stanem lokalnym	377
Przetwarzanie wielofazowe (repartycjonowanie)	379
Przetwarzanie z wyszukiwaniem zewnętrznym	
— łączenie strumieniowo-tablicowe	380

Łączenie tablicowo-tablicowe	381
Łączenie strumieniowe	382
Zdarzenia poza kolejnością	383
Ponowne przetwarzanie	384
Zapytania interaktywne	384
Kafka Streams na przykładach	385
Licznik słów	385
Statystyki giełdowe	388
Wzbogacanie strumienia kliknięć	390
Strumienie Kafki — przegląd architektury	392
Budowanie topologii	392
Optymalizacja topologii	393
Testowanie topologii	393
Skalowanie topologii	394
Obsługa awarii	397
Przypadki użycia dla przetwarzania strumieniowego	398
Jak wybrać framework przetwarzania strumieniowego	399
Podsumowanie	401
A Instalowanie Kafki w innych systemach operacyjnych	403
Instalowanie w systemie Windows	403
Korzystanie z windowsowego podsystemu dla Linuksa	403
Korzystanie z natywnej Javy	404
Instalowanie w systemie macOS	406
Korzystanie z Homebrew	406
Instalacja ręczna	407
B Dodatkowe narzędzia Kafki	409
Kompleksowe platformy	409
Wdrażanie klastra i zarządzanie nim	410
Monitorowanie i eksploracja danych	411
Biblioteki klienckie	412
Przetwarzanie strumieniowe	412

Przedmowa do wydania II

Pierwsze wydanie książki *Kafka w praktyce* ukazało się pięć lat temu. Szacowaliśmy wówczas, że platforma Apache Kafka była używana przez ok. 30% firm z listy Fortune 500. Dziś z Kafki korzysta ponad 70% firm z tej listy. To wciąż jeden z najpopularniejszych projektów *open source* na świecie i znajduje się w centrum ogromnego ekosystemu.

Skąd tak duży zachwyt? Myślę, że jest to spowodowane istnieniem ogromnej luki w naszej infrastrukturze danych. Tradycyjnie zarządzanie danymi opierało się na przechowywaniu — w magazynach plików i bazach danych, które zapewniały bezpieczeństwo naszych danych i pozwalały znaleźć odpowiedni fragment we właściwym czasie. W systemy te zainwestowano ogromne ilości energii intelektualnej i komercyjnych nakładów finansowych. Jednak nowoczesna firma to nie tylko jeden program z jedną bazą danych. Nowoczesna firma to niezwykle złożony system, zbudowany z setek, a nawet tysięcy niestandardowych aplikacji, mikrousług, baz danych, warstw SaaS i platform analitycznych. Coraz częściej stajemy więc przed problemem, jak połączyć to wszystko i sprawić, aby poszczególne składowe współpracowały ze sobą w czasie rzeczywistym.

Ten problem nie dotyczy zarządzania danymi w spoczynku — chodzi o zarządzanie danymi w ruchu. A w samym sercu znajduje się Apache Kafka, która stała się *de facto* fundamentem każdej platformy danych w ruchu.

Od początku swojej podróży Kafka nie pozostawała statyczna. Ewoluoowało również to, co zaczęło się jako podstawowy dziennik zatwierdzeń: po drodze dodane zostały konektory i funkcjonalności przetwarzania strumieniowego oraz przeprojektowana została cała architektura. Społeczność nie tylko rozwinęła istniejące interfejsy API, opcje konfiguracyjne, wskaźniki i narzędzia, by poprawić użyteczność i niezawodność Kafki, ale wprowadziła także nowe programowe API administracyjne, nową generację globalnej replikacji i odzyskiwania sprawności po katastrofie przy użyciu Mirror-Makera 2.0, nowy protokół konsensusu oparty na algorytmie Raft, umożliwiający uruchamianie Kafki w pojedynczym pliku wykonywalnym, oraz prawdziwą elastyczność dzięki obsłudze warstwowej pamięci masowej. A co najważniejsze, dodając obsługę zaawansowanych opcji zabezpieczeń, takich jak uwierzytelnianie, autoryzacja i szyfrowanie, najprawdopodobniej uczyniliśmy Kafkę łatwym w zastosowaniu oprogramowaniem dla kluczowych przypadków użycia w przedsiębiorstwie.

Wraz z ewolucją Kafki obserwujemy również ewolucję przypadków użycia. Kiedy opublikowane zostało pierwsze wydanie tej książki, znacząca część instalacji Kafki nadal znajdowała się w tradycyjnych lokalnych centrach danych i korzystała z tradycyjnych skryptów wdrożeniowych.

Najpopularniejszymi przypadkami użycia były ETL i system wymiany komunikatów; przetwarzanie strumieniowe dopiero raczkowało. Pięć lat później większość instalacji Kafki znajduje się w chmurze, a wiele instalacji jest uruchamianych w Kubernetesie. ETL i systemy wymiany komunikatów są nadal popularne, ale dołączają do nich mikrousługi oparte na zdarzeniach, przetwarzanie strumieniowe w czasie rzeczywistym, internet rzeczy, potoki uczenia maszynowego oraz setki branżowych przypadków użycia i wzorców, od przetwarzania roszczeń w firmach ubezpieczeniowych, przez systemy tradingowe w bankach po obsługę rozgrywek w czasie rzeczywistym i personalizację w grach wideo i usługach przesyłania strumieniowego.

Chociaż Apache Kafka cały czas rozszerza swój zasięg na nowe środowiska i przypadki użycia, pisanie aplikacji, które dobrze wykorzystują tę platformę i niezawodnie wdrażają ją w środowisku produkcyjnym, wymaga dostosowania się do unikatowego sposobu myślenia w kategoriach Kafki. W tej książce opisane zostało wszystko, czego programiści i inżynierowie SRE potrzebują, aby w pełni wykorzystać potencjał Kafki, od najbardziej podstawowych interfejsów API i konfiguracji po najnowsze i najnowocześniejsze funkcjonalności. Omówiono w niej nie tylko to, co można zrobić z Kafką i jak to zrobić, ale także, czego nie robić i jakich antywzorców należy unikać. Ta książka może być godnym zaufania przewodnikiem po świecie Kafki zarówno dla nowych użytkowników, jak i doświadczonych praktyków.

— Jay Kreps,
współzałożyciel i prezes firmy Confluent

Przedmowa do wydania I

To ekscytujący czas dla platformy Apache Kafka. Jest ona wykorzystywana przez dziesiątki tysięcy organizacji, w tym ponad jedną trzecią firm z listy Fortune 500. To jeden z najszybciej rozwijających się projektów udostępnianych na licencji *open source*, który wytworzył wokół siebie ogromny ekosystem. Stanowi samo serce ruchu ukierunkowanego na zarządzanie strumieniami danych i ich przetwarzanie.

Skąd więc wzięła się Kafka, dlaczego ją zbudowaliśmy i czym dokładnie jest?

Początkowo Kafka była wewnętrznym systemem infrastruktury, który zbudowaliśmy w LinkedInie. Dokonałiśmy naprawdę prostej obserwacji: do *przechowywania* danych zbudowano wiele baz danych i innych systemów, ale w naszej architekturze brakowało czegoś, co pomogłoby nam obsługiwać ciągły *przepływ* danych. Zanim zbudowaliśmy Kafkę, eksperymentowaliśmy z różnego rodzaju gotowymi opcjami, od systemów wymiany komunikatów po narzędzia do agregacji dzienników i narzędzia ETL, ale żadna z nich nie oferowała tego, czego potrzebowaliśmy.

Ostatecznie postanowiliśmy zbudować coś od zera. Nasz pomysł polegał na tym, byśmy zamiast skupiać się na przechowywaniu stosów danych, takich jak nasze relacyjne bazy danych, magazyny klucz-wartość, indeksy wyszukiwania czy pamięci podręczne, skoncentrowali się na potraktowaniu danych jako stale ewoluującego i rosnącego strumienia i wokół tej koncepcji zbudowali system (a tak właściwie nawet architekturę) danych.

Pomysł ten okazał się mieć znacznie szersze zastosowanie, niż się spodziewaliśmy. Chociaż na początku Kafka wspierała działanie aplikacji czasu rzeczywistego i przepływ danych za kulisami sieci społecznościowej, teraz stanowi serce architektur nowej generacji w każdej możliwej branży. Wielcy detaliści przerabiają swoje podstawowe procesy biznesowe z wykorzystaniem ciągłych strumieni danych, firmy samochodowe zbierają i przetwarzają strumienie danych w czasie rzeczywistym z samochodów podłączonych do internetu, a banki przeprojektowują swoje podstawowe procesy i systemy wokół Kafki.

O co więc chodzi z tą Kafką? Jak wypada ona w porównaniu z systemami, które już znasz i z których korzystasz?

Zaczeliliśmy traktować Kafkę jako *platformę streamingową*: system umożliwiający publikowanie i subskrybowanie strumieni danych, przechowywanie ich i przetwarzanie — czyli dokładnie to, do czego została stworzona platforma Apache Kafka. Taki sposób traktowania danych może różnić

się nieco od tego, do czego przywykliśmy, ale okazuje się niesamowicie potężną abstrakcją dla budowania aplikacji i architektur. Kafka jest często porównywana z kilkoma istniejącymi kategoriami technologii: systemami wymiany komunikatów klasy enterprise, systemami wielkich zbiorów danych (*big data*), takimi jak Hadoop, oraz narzędziami integracji danych lub ETL. Każde z tych porównań jest po części zasadne, ale żadne nie uwzględnia wszystkich aspektów.

Kafka przypomina system wymiany komunikatów w tym sensie, że pozwala publikować i subskrybować strumienie komunikatów. W tym zakresie jest podobna do takich produktów, jak ActiveMQ, RabbitMQ, MQSeries firmy IBM itd. Jednak mimo tych podobieństw Kafka znacznie różni się od tradycyjnych systemów wymiany komunikatów, przez co jest zwierzęciem zupełnie innego gatunku. Możemy wymienić trzy istotne różnice. Po pierwsze, Kafka działa jak nowoczesny system rozproszony, który jest uruchamiany jako klaster i może być skalowany do obsługi wszystkich aplikacji w nawet największych korporacjach. Zamiast uruchamiać dziesiątki indywidualnych brokerów komunikatów ręcznie podłączonych do różnych aplikacji, możemy mieć centralną platformę, która może skalować się elastycznie, aby obsłużyć wszystkie strumienie danych w firmie. Po drugie, Kafka to prawdziwy system pamięci masowej zaprojektowany do przechowywania danych tak długo, jak jest to wymagane. To duża zaleta, gdy używa się Kafki jako warstwy łączącej, ponieważ zapewnia rzeczywiste gwarancje dostarczania danych — jej dane są replikowane, trwałe i mogą być przechowywane dowolnie długo. I po trzecie, świat przetwarzania strumieniowego dość znacząco podnosi poziom abstrakcji. Systemy wymiany komunikatów głównie przekazują komunikaty. Funkcjonalności przetwarzania strumieniowego Kafki umożliwiają dynamiczne obliczanie strumieni pochodnych i zbiorów danych ze strumieni przy użyciu znacznie mniejszej ilości kodu. Różnice te sprawiają, że Kafka jest na tyle wyjątkowa, że nie ma sensu traktowanie jej jak „jeszcze jednej kolejki”.

Innym ze sposobów postrzegania Kafki — i jednym z naszych czynników motywujących w jej projektowaniu i budowaniu — było potraktowanie jej jak swoistej wersji Hadoopa w czasie rzeczywistym. Hadoop umożliwia przechowywanie i okresowe przetwarzanie plików danych na bardzo dużą skalę. Z kolei Kafka umożliwia przechowywanie i ciągłe przetwarzanie strumieni danych, również na dużą skalę. Na poziomie technicznym zdecydowanie istnieją pewne podobieństwa, a wiele osób postrzega wyłaniającą się dziedzinę przetwarzania strumieniowego jako nadzbiór takiego rodzaju przetwarzania wsadowego, jakie wykonywało się za pomocą Hadoopa i jego różnych warstw przetwarzania. To porównanie nie uwzględnia faktu, że ciągłe przetwarzanie o niewielkich opóźnieniach wprowadza nowe przypadki użycia, różniące się od tych, które naturalnie mieszczą się w kategoriach systemów przetwarzania wsadowego. Podczas gdy Hadoop i wielkie zbiory danych są ukierunkowane na aplikacje analityczne, często w przestrzeni hurtowni danych, niskie opóźnienia Kafki sprawiają, że ma ona zastosowanie do tego rodzaju bazowych aplikacji, które bezpośrednio wspierają prowadzenie działalności. Jest to sensowne, gdyż w firmie cały czas mają miejsce jakieś zdarzenia, a możliwość reagowania na nie w miarę ich występowania znacznie ułatwia budowanie usług, które bezpośrednio napędzają funkcjonowanie firmy, umożliwiają dostosowywanie doświadczenia klienta itd.

Ostatnia technologia, do której porównywana jest Kafka, to narzędzia ETL lub narzędzia do integracji danych. W końcu narzędzia te służą do przenoszenia danych, a Kafka przenosi dane. Pod pewnymi względami jest to zasadne, ale myślę, że główna różnica polega na tym, że Kafka odwróciła ten problem. Zamiast być narzędziem do scrapingu danych z jednego systemu i wstawiania ich

do drugiego, Kafka jest platformą zorientowaną na strumienie zdarzeń czasu rzeczywistego. Oznacza to, że nie tylko może łączyć gotowe aplikacje i systemy danych, ale również stanowić podstawę niestandardowych aplikacji zbudowanych w celu wyzwiania samych tych strumieni danych. Uważamy, że ta architektura skupiona wokół strumieni zdarzeń jest czymś naprawdę istotnym. W pewnym sensie te przepływy danych są najważniejszym aspektem nowoczesnego przedsiębiorstwa cyfrowego, równie ważnym jak przepływy pieniężne, które można znaleźć w sprawozdaniu finansowym.

To właśnie możliwość połączenia tych trzech obszarów — zebrania wszystkich strumieni danych dla wszystkich przypadków użycia — sprawia, że idea platformy streamingowej jest tak atrakcyjna.

Tak czy inaczej wszystko to jest nieco odmienne, a nauka myślenia w kategoriach ciągłych strumieni danych i budowania opartych na nich aplikacji jest dużą zmianą, jeśli pochodzisz ze świata aplikacji budowanych w stylu żądanie-odpowiedź i relacyjnych baz danych. Ta książka, napisana przez osoby, które najlepiej znają Kafkę, to bezwzględnie najlepsze źródło wiedzy o tej platformie, od wewnętrznych mechanizmów jej działania po interfejsy API. Mam nadzieję, że ta lektura przyniesie Ci tyle samo satysfakcji co mnie!

— Jay Kreps,
współzałożyciel i prezes firmy Confluent

Największy komplement, jaki można powiedzieć autorowi książki technicznej, brzmi: „Szkoda, że nie miałem tej książki, gdy zaczynałem pracę nad tym tematem”. Właśnie taki cel postawiliśmy sobie, kiedy zaczynaliśmy pisać tę książkę. Przeanalizowaliśmy nasze doświadczenia związane z pisaniem Kafki, uruchamianiem jej w środowisku produkcyjnym i pomaganiem wielu firmom w wykorzystywaniu Kafki do budowania architektur oprogramowania i zarządzania ich potokami danych i zadaliśmy sobie pytanie: „Jakie są najbardziej przydatne elementy, którymi możemy podzielić się z początkującymi użytkownikami, aby mogli szybko przejść do poziomu ekspertów?”. Ta książka jest odzwierciedleniem pracy, którą wykonujemy na co dzień: uruchamiamy Apache Kafkę i pomagamy innym korzystać z niej w najlepszy możliwy sposób.

Zawarliśmy w tej książce to, co naszym zdaniem powinieneś wiedzieć, by z powodzeniem uruchamiać Apache Kafkę w środowisku produkcyjnym i na jej podstawie budować solidne i wydajne aplikacje. Wskazaliśmy popularne przypadki użycia: magistrale komunikatów dla mikrousług opartych na zdarzeniach, aplikacje przetwarzania strumieniowego i potoki danych o dużej skali. Skupiliśmy się również na uczynieniu tej książki na tyle ogólnej i kompleksowej, aby była przydatna dla każdego, kto korzysta z Kafki, bez względu na przypadek użycia czy architekturę. Omówiliśmy kwestie praktyczne, takie jak instalacja i konfiguracja Kafki oraz sposoby korzystania z jej interfejsów API, ale poświęciliśmy miejsce także zasadom projektowym i gwarancjom niezawodności Kafki i zbadaliśmy kilka wspaniałych szczegółów jej architektury: protokół replikacji, kontroler i warstwę pamięci masowej. Jesteśmy przekonani, że poznanie projektu i wewnętrznych mechanizmów działania Kafki będzie nie tylko przyjemnością dla osób zainteresowanych systemami rozproszonymi, ale okaże się też niezwykle przydatne dla tych, którzy starają się podejmować świadome decyzje podczas wdrażania Kafki w środowisku produkcyjnym i projektowania aplikacji wykorzystujących Kafkę. Im lepiej zrozumiesz działanie Kafki, tym bardziej świadome będą Twoje decyzje dotyczące wielu kompromisów, które będziesz musiał podejmować w zakresie inżynierii.

Jednym z problemów inżynierii oprogramowania jest to, że zawsze można zrobić coś na kilka różnych sposobów. Platformy takie jak Apache Kafka zapewniają dużą elastyczność, co jest cenione przez ekspertów, ale tworzy stromą krzywą uczenia się dla początkujących. Bardzo często Apache Kafka instruuje Cię, jak korzystać z określonej funkcjonalności, ale nie zapewnia wskazówek, dlaczego powinieneś z niej korzystać lub nie. Gdzie było to możliwe, staraliśmy się wyjaśniać, na czym polegają możliwe wybory, związane z nimi kompromisy oraz kiedy powinieneś (lub nie powinieneś) wykorzystywać różne opcje oferowane przez tę platformę.

Dla kogo przeznaczona jest ta książka?

Książkę *Kafka w praktyce* napisaliśmy z myślą o inżynierach oprogramowania, którzy opracowują aplikacje wykorzystujące interfejsy API Kafki, oraz o inżynierach produkcji (zwanymi również SRE, DevOps lub sysadminami), którzy instalują, konfiguruje, dostrajają i monitorują Kafkę w środowisku produkcyjnym. Nie zapomnieliśmy także o architektach i inżynierach danych — tych, którzy są odpowiedzialni za projektowanie i budowanie całej infrastruktury danych organizacji. Niektóre z rozdziałów, zwłaszcza rozdziały 3., 4. i 14., są skierowane do programistów Javy. W tych rozdziałach przyjęliśmy założenie, że znasz podstawy języka programowania Java, w tym takie zagadnienia, jak obsługa wyjątków i współbieżność. W innych rozdziałach, zwłaszcza 2., 10., 12. i 13., przyjęliśmy założenie, że masz pewne doświadczenie w uruchamianiu Linuksa oraz konfigurowaniu pamięci masowej i sieci w tym systemie operacyjnym. W pozostałych fragmentach książki omówiliśmy Kafkę i architektury oprogramowania w sposób bardziej ogólny, co nie wymaga posiadania wiedzy z konkretnej dziedziny.

Tę książkę mogą uznać za interesującą także menedżerowie i architekci niepracujący bezpośrednio z Kafką, ale pracujący z ludźmi, którzy to robią. Oni też powinni rozumieć gwarancje zapewniane przez Kafkę oraz kompromisy, których będą musieli dokonywać ich pracownicy i współpracownicy podczas budowania systemów opartych na Kafce. Lektura ta może dostarczyć amunicji menedżerom, którzy chcieliby przeszkolić swój personel w używaniu platformy Apache Kafka lub upewnić się, że ich zespoły wiedzą to, co muszą wiedzieć.

Konwencje stosowane w tej książce

W tej książce wykorzystywane są następujące konwencje typograficzne.

Czcionka pogrubiona

Stosowana do podkreślania ważnych pojęć i wprowadzania nowych terminów.

Czcionka pochylona

Stosowana do zapisywania nazw folderów i plików, rozszerzeń plików, nazw ścieżek, adresów URL, nazw opcji, nazw okien programów oraz nazw przycisków.

Czcionka o stałej szerokości znaków

Stosowana do zapisywania listingów oraz poleceń i fragmentów kodu występujących w tekście.

Pogrubiona czcionka o stałej szerokości znaków

Stosowana do zaznaczania fragmentów kodu, poleceń itp. wpisywanych bezpośrednio przez użytkownika.

Pochylona czcionka o stałej szerokości znaków

Stosowana w poleceniach i listingach do zaznaczania argumentów, pod które należy samodzielnie podstawić określone wartości.



Ta ikona oznacza wskazówki lub sugestie.



Ta ikona oznacza ogólne uwagi.



Ta ikona oznacza ostrzeżenie.

Korzystanie z przykładów kodu

Umieszczone w książce przykłady kodu możesz pobrać pod adresem <https://ftp.helion.pl/przyklady/kafpra.zip>. Po ściągnięciu archiwum rozpakuj je na swoim dysku.

Książka ta ma pomóc Ci w pracy. Ogólnie rzecz biorąc, kodu znajdującego się w tej książce można używać we własnych programach i dokumentacjach bez proszenia kogokolwiek o zgodę, chyba że wykorzystasz duże fragmenty. Jeśli np. w pisany program użyjesz kilku fragmentów kodu z tej książki, nie musisz pytać o pozwolenie. Aby sprzedawać i rozprowadzać płyty CD-ROM z przykładami, trzeba mieć zezwolenie. Aby odpowiedzieć komuś na pytanie, cytując fragment tej książki wraz z kodem źródłowym, nie trzeba mieć zezwolenia. Aby wykorzystać dużą ilość kodu źródłowego z tej książki w dokumentacji własnego produktu, trzeba mieć pozwolenie.

Informacje o źródle użytych fragmentów są mile widziane, ale niewymagane. Notka powinna zawierać nazwisko autora, tytuł publikacji, nazwę wydawcy oraz datę i miejsce publikacji, np. Gwen Shapira, Todd Palino, Rajini Sivaram, Krit Petty, *Kafka w praktyce. Wydanie II*, Helion, Gliwice 2022.

Podziękowania

Chcielibyśmy podziękować wielu kontrybutorom projektu Apache Kafka i jego ekosystemu. Bez ich pracy ta książka by nie powstała. Szczególne podziękowania dla Jaya Krepsa, Nehy Narkhede i Juna Rao, a także ich kolegów i kierownictwa z firmy LinkedIn za współtworzenie Kafki i udostępnienie jej organizacji Apache Software Foundation.

Wiele osób dostarczyło cennych informacji zwrotnych na temat wczesnych wersji książki i doceniamy ich czas i wiedzę: Apurva Mehta, Arseniy Tashoyan, Dylan Scott, Ewen Cheslack-Postava, Grant Henke, Ismael Juma, James Cheng, Jason Gustafson, Jeff Holoman, Joel Koshy, Jonathan Seidman, Jun Rao, Matthias Sax, Michael Noll, Paolo Castagna i Jesse Anderson. Chcemy również podziękować wielu czytelnikom, którzy pisali komentarze i opinie za pośrednictwem przedpremierowej strony internetowej.

Pomogło nam wielu recenzentów, którzy znacznie poprawili jakość tej książki, więc wszelkie pozostałe błędy są nasze.

Chcielibyśmy podziękować naszej redaktorce I wydania z wydawnictwa O'Reilly, Shannon Cutt, za jej zachętę i cierpliwość, i za to, że znacznie bardziej kontrolowała sytuację niż my. Redaktorzy naszego II wydania, Jess Haberman i Gary O'Brien, pomogli nam sprostać globalnym wyzwaniom. Współpraca z wydawnictwem O'Reilly jest wspaniałym doświadczeniem dla autora — zapewnia ono niezrównane wsparcie, od narzędzi po podpisywanie książek. Jesteśmy wdzięczni wszystkim zaangażowanym w realizację tego zadania i doceniamy ich wybór współpracy z nami.

Chcielibyśmy podziękować naszym menedżerom i współpracownikom za umożliwienie nam napisania tej książki i za ich zachętę.

Gwen chce podziękować swojemu mężowi Omerowi Shapirze za jego wsparcie i cierpliwość podczas wielu miesięcy spędzonych na pisaniu kolejnej książki, swoim kotom Luke'owi i Lei za przytulanie oraz swojemu tacie Liorowi Shapirze za nauczanie jej, by zawsze mówiła „tak” możliwościom, nawet jeśli wydaje się to niechęć.

Todd nie osiągnąłby niczego bez wsparcia ze strony swojej żony Marcy oraz córek Belli i Kaylee. Ich wyrozumiałość dla długich godzin przeznaczanych na pisanie oraz bieganie, aby oczyścić umysł, napędza go do działania.

Rajini chciałaby podziękować swojemu mężowi Manjunathowi i synowi Tarunowi za ich niezachwiane wsparcie i zachętę, za spędzanie weekendów na przeglądaniu wczesnych projektów i za to, że zawsze byli przy niej.

Krit dzieli się swoją miłością i wdzięcznością z żoną Cecilią i dwójką dzieci, Lucasem i Lizabeth. Ich miłość i wsparcie sprawiają, że każdy dzień jest radością, a bez nich nie byłby w stanie realizować swoich pasji. Chce również podziękować swojej mamie Cindy Petty za zaszczepienie w nim pragnienia, aby zawsze być najlepszą wersją siebie.

Poznaj Kafkę

Funkcjonowanie każdego przedsiębiorstwa opiera się na danych. Przyjmujemy informacje, analizujemy je, manipulujemy nimi i jako dane wyjściowe tworzymy jeszcze więcej informacji. Każda aplikacja tworzy dane: komunikaty dziennika, wskaźniki, aktywność użytkowników, komunikaty wychodzące itd. Każdy bajt danych ma do opowiedzenia jakąś historię, coś ważnego, co wskaże następną rzecz do zrobienia. Aby dowiedzieć się, co to ma być, musimy przesłać dane z miejsca ich powstania do miejsca, w którym można je analizować. Spotykamy się z tym codziennie na stronach takich jak Amazon, gdzie kliknięcia interesujących nas elementów zamieniają się w rekomendacje, które są nam wyświetlane nieco później.

Im szybciej jesteśmy w stanie to robić, tym sprawniejsze i bardziej responsywne mogą być nasze organizacje. Im mniej wysiłku wkładamy w przenoszenie danych, tym mocniej możemy skoncentrować się na podstawowej działalności biznesowej. Właśnie dlatego w przedsiębiorstwie opartym na danych potok jest kluczowym komponentem. Sposób przenoszenia danych staje się prawie tak samo ważny jak dane.

Jeżeli naukowcy się ze sobą nie zgadzają, jest to spowodowane brakiem wystarczającej ilości danych. Można wtedy uzgodnić, jakiego rodzaju dane są potrzebne, pozyskać te dane, a dane rozwiążą problem. Ja mam rację, ty masz rację albo oboje się mylimy. I możemy pójść dalej.

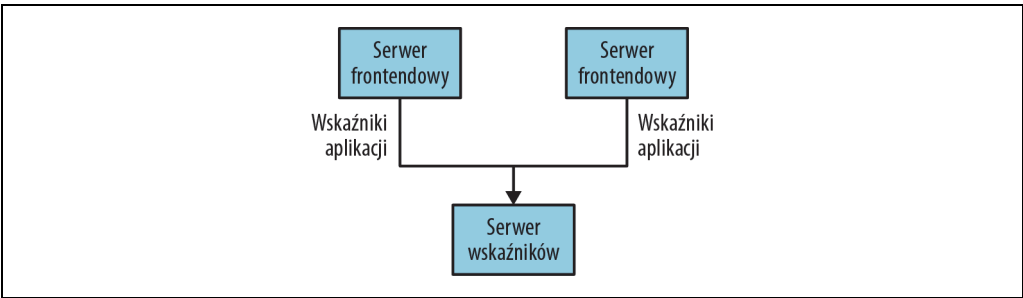
— Neil deGrasse Tyson

Publikuj-Subskrybuj

Zanim przejdziemy do szczegółów platformy Apache Kafka, musimy omówić koncepcję wzorca Publikuj-Subskrybuj i wyjaśnić, dlaczego jest to kluczowy komponent aplikacji opartych na danych. **System wymiany komunikatów publikuj-subskrybuj** (*pub-sub*) to wzorzec charakteryzujący się tym, że nadawca (wydawca) elementu danych (komunikatu) nie kieruje go do konkretnego odbiorcy. Zamiast tego wydawca klasyfikuje komunikaty w określony sposób, a odbiorca (subskrybent) dokonuje subskrypcji w celu otrzymywania określonych klas komunikatów. Aby ułatwić zastosowanie tego wzorca, systemy pub-sub często mają centralny punkt (broker), w którym publikowane są komunikaty.

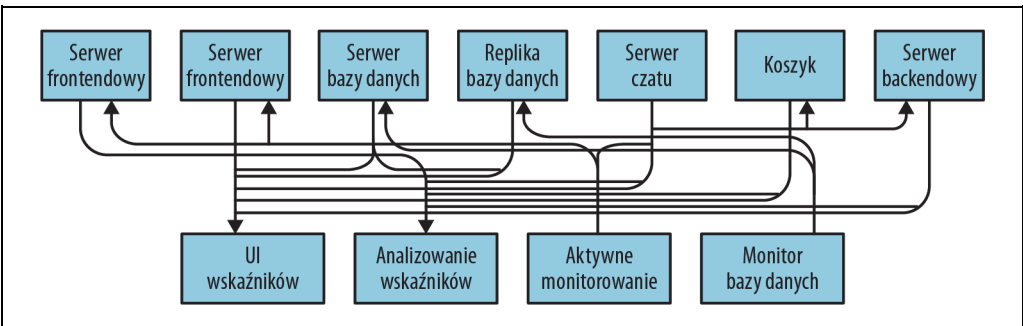
Jak to się zaczyna?

Początkiem dla wielu przypadków użycia mechanizmu pub-sub jest prosta kolejka komunikatów lub kanał komunikacji międzyprocesowej. Tworzysz np. aplikację, która musi gdzieś wysyłać informacje z monitorowania, a następnie otwierasz bezpośrednie połączenie z tej aplikacji do programu wyświetlającego Twoje wskaźniki na dashboardzie i przez to połączenie przesyłasz te wskaźniki, jak pokazaliśmy na rysunku 1.1.



Rysunek 1.1. Pojedynczy bezpośredni wydawca wskaźników

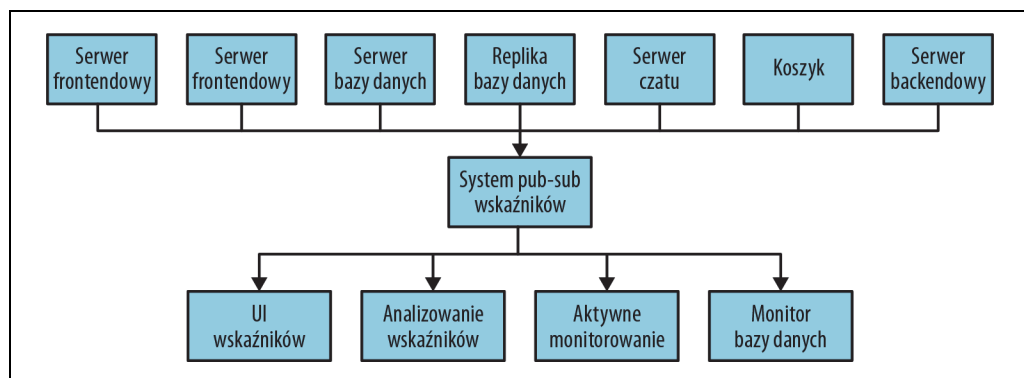
Jest to proste rozwiązanie prostego problemu, które sprawdza się, gdy rozpoczynasz zabawę w monitorowanie. Wkrótce jednak postanawiasz, że chcesz analizować swoje wskaźniki w dłuższej perspektywie, a dashboard niezbyt się do tego nadaje. Uruchamiasz nową usługę, która może odbierać wskaźniki, przechowywać je i analizować. W tym celu modyfikujesz aplikację w taki sposób, aby zapisywała wskaźniki w obu systemach. Na tym etapie masz już trzy kolejne aplikacje, które generują wskaźniki i nawiązują takie same połączenia z tymi dwiema usługami. Twój współpracownik uważa, że dobrym pomysłem byłoby również aktywne odpytywanie usług w celu uruchamiania alarmów, dodajesz więc do każdej aplikacji serwer obsługujący dostarczanie wskaźników na żądanie. Po pewnym czasie masz jeszcze więcej aplikacji, które korzystają z tych serwerów, by pobierać indywidualne wskaźniki i używać ich do różnych celów. Ta architektura może wyglądać mniej więcej tak, jak pokazaliśmy na rysunku 1.2, z jeszcze trudniejszymi do prześledzenia połączeniami.



Rysunek 1.2. Wiele wydawców wskaźników korzystających z połączeń bezpośrednich

Nagromadzony tutaj dług techniczny jest oczywisty, dlatego decydujesz się częściowo go spłacić. Konfigurujesz pojedynczą aplikację odbierającą wskaźniki ze wszystkich aplikacji i każdemu systemowi,

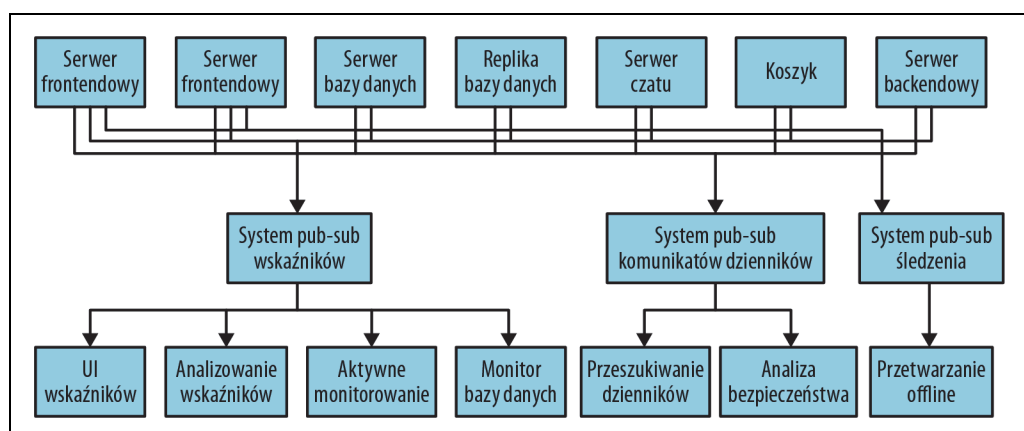
który potrzebuje tych wskaźników, udostępniasz serwer do ich kwerendowania. Zmniejsza to złożoność architektury do mniej więcej takiej postaci, jaką pokazaliśmy na rysunku 1.3. Gratulacje, zbudowałeś system wymiany komunikatów publikuj-subskrybuj!



Rysunek 1.3. System publikuj-subskrybuj dla wskaźników

Indywidualne systemy kolejkowe

W tym samym czasie, gdy prowadziłeś wojnę ze wskaźnikami, jeden z Twoich współpracowników prowadził podobną batalię z komunikatami dzienników, a drugi zmagał się ze śledzeniem zachowań użytkowników na stronie frontendowej i z dostarczaniem tych informacji programistom pracującym nad uczeniem maszynowym oraz tworzeniem raportów dla kierownictwa. Wszyscy podążaliście podobną ścieżką budowania systemów, które oddzielają wydawców informacji od subskrybentów tych informacji. Taką infrastrukturę z trzema osobnymi systemami pub-sub przedstawiliśmy na rysunku 1.4.



Rysunek 1.4. Wiele systemów publikuj-subskrybuj

Jest to z pewnością o wiele lepsze niż korzystanie z połączeń punkt-punkt (jak pokazaliśmy na rysunku 1.2), ale istnieje wiele duplikacji. Twoja firma utrzymuje wiele systemów do kolejkowania

danych, z których każdy ma swoje własne, indywidualne błędy i ograniczenia. Wiesz ponadto, że wkrótce pojawi się więcej przypadków użycia dla mechanizmu wymiany komunikatów. Chciałbyś mieć jeden scentralizowany system, umożliwiający publikowanie ogólnych typów danych, który będzie rozwijał się wraz z rozwojem firmy.

Pojawienie się Kafki

Platforma Apache Kafka została opracowana jako system wymiany komunikatów publikuj-subskrybuj, który ma rozwiązać ten problem. Często jest opisywana jako „rozproszony dziennik zatwierdzania”, a ostatnio również jako „rozproszona platforma przesyłania strumieniowego”. Dziennik zatwierdzania systemu plików lub bazy danych został zaprojektowany w celu zapewnienia trwałego zapisu wszystkich transakcji, aby można je było odtwarzać w celu spójnego budowania stanu systemu. Dane w Kafce również są przechowywane w sposób trwały, uporządkowany i mogą być odczytywane deterministycznie. Ponadto dane mogą być dystrybuowane w obrębie systemu w celu zagwarantowania dodatkowych zabezpieczeń przed awariami oraz możliwości skalowania wydajności.

Komunikaty i partie

Jednostka danych w Kafce nazywa się **komunikatem** (ang. *message*). Jeżeli masz doświadczenie w dziedzinie baz danych, jednostkę danych możesz potraktować jak **wiersz** lub **rekord**. W przypadku Kafki komunikat to po prostu tablica bajtów, dlatego zawarte w niej dane nie mają określonego formatu ani konkretnego znaczenia dla tej platformy. Komunikat może zawierać opcjonalny fragment metadanych nazywany **kluczem**. Klucz również jest tablicą bajtów i, podobnie jak komunikat, nie ma dla Kafki konkretnego znaczenia. Kluczy używa się, gdy komunikaty mają być zapisywane na partycjach w bardziej kontrolowany sposób. Najprostszym takim schematem jest wygenerowanie spójnego skrótu klucza, a następnie wybranie numeru partycji dla danego komunikatu na podstawie reszty z dzielenia skrótu przez całkowitą liczbę partycji w temacie. Zapewnia to zapisywanie komunikatów z tym samym kluczem zawsze na tej samej partycji (pod warunkiem, że liczba partycji się nie zmienia).

Aby zwiększyć wydajność, komunikaty są zapisywane w Kafce partiami. **Partia** (ang. *batch*) to po prostu zbiór komunikatów tworzonych dla tego samego tematu i tej samej partycji. Indywidualny obieg w obie strony przez sieć dla każdego komunikatu generowałby nadmierne obciążenie, które można zredukować, gromadząc komunikaty w partie. Oczywiście jest to kompromis między opóźnieniem a przepustowością: im większe partie, tym więcej komunikatów można obsłużyć w jednostce czasu, ale tym dłużej trwa propagacja pojedynczego komunikatu. Partie są ponadto zwykle kompresowane, co zapewnia wydajniejszy transfer i bardziej efektywne przechowywanie danych kosztem pewnej mocy obliczeniowej. Zarówno klucze, jak i partie omówiliśmy szerzej w rozdziale 3.

Schematy

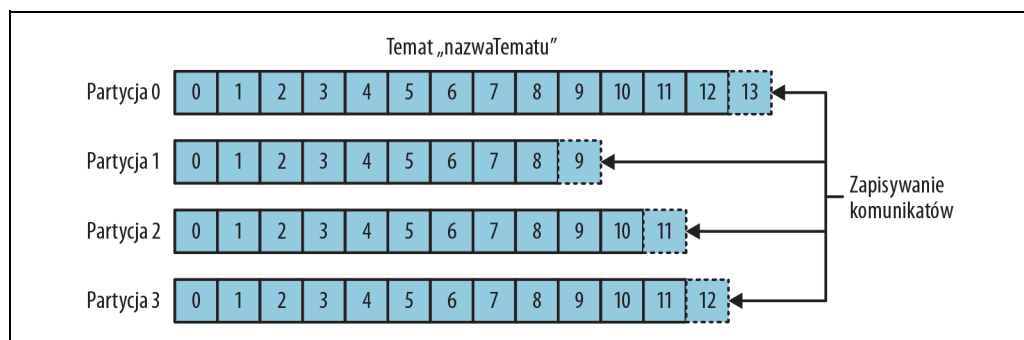
Chociaż dla samej Kafki komunikaty są nieprzezroczystymi tablicami bajtów, zaleca się nałożenie na zawartość komunikatu dodatkowej struktury lub schematu, aby można było tę zawartość łatwo zrozumieć. Dostępnych jest wiele opcji **schematu** (ang. *schema*) komunikatów, w zależności od

indywidualnych potrzeb aplikacji. Uproszczone systemy, np. JSON (ang. *JavaScript Object Notation*) i XML (ang. *Extensible Markup Language*), są łatwe w użyciu i czytelne dla człowieka, brakuje im jednak takich funkcjonalności jak niezawodna obsługa typów i zgodność między wersjami schematu. Wielu programistów Kafki preferuje korzystanie z Apache Avro, który jest frameworkiem serializacji opracowanym pierwotnie dla Hadoopa. Avro zapewnia kompaktowy format serializacji, schematy oddzielone od ładunków komunikatów i niewymagające generowania kodu po ich zmianie oraz silne typowanie danych i ewolucję schematu z kompatybilnością wstecz i w przód.

W Kafce spójny format danych jest ważny, ponieważ umożliwia oddzielenie zapisywania i odczytywania komunikatów. Gdy te zadania są ściśle powiązane, aplikacje subskrybujące komunikaty muszą być aktualizowane, aby obsługiwały nowy format danych równolegle ze starym. Dopiero wtedy aplikacje publikujące komunikaty mogą zostać zaktualizowane do korzystania z nowego formatu. Dzięki używaniu dobrze zdefiniowanych schematów i przechowywaniu ich we wspólnym repozytorium komunikaty w Kafce można zrozumieć bez koordynacji. Schematy i serializację omówiliśmy szerzej w rozdziale 3.

Tematy i partycje

Komunikaty w Kafce są dzielone na **tematy** (ang. *topic*). Najbliższe analogie do tematu to tablica bazy danych lub folder w systemie plików. Tematy są dodatkowo podzielone na kilka **partycji** (ang. *partition*). Jeśli przywołamy opis Kafki jako „dziennika zatwierdzania”, partycja będzie pojedynczym dziennikiem. Komunikaty są w nim zapisywane tylko przez ich dodawanie, a odczytywane w kolejności od początku do końca. Zwróć uwagę, że skoro temat ma zazwyczaj wiele partycji, nie ma gwarancji zachowywania kolejności komunikatów w całym temacie, a jedynie w ramach poszczególnych partycji. Na rysunku 1.5 pokazaliśmy temat z czterema partycjami, z zapisami dołączanymi na końcu każdej z nich. Partycje to także sposób zapewniania przez Kafkę redundancji i skalowalności. Każda partycja może być hostowana na innym serwerze, co oznacza, że pojedynczy temat można skalować poziomo na wielu serwerach, aby zapewnić wydajność znacznie przekraczającą możliwości pojedynczego serwera. Partycje mogą być ponadto replikowane, dzięki czemu różne serwery będą przechowywać kopię tej samej partycji na wypadek awarii jednego z serwerów.



Rysunek 1.5. Reprezentacja tematu z wieloma partycjami

Podczas omawiania danych w systemach takich jak Kafka często używane jest pojęcie **strumienia** (ang. *stream*). Strumień jest przeważnie traktowany jak pojedynczy temat danych, niezależnie od liczby partycji. Reprezentuje to pojedynczy strumień danych przechodzący od producentów do konsumentów. Ten sposób odwoływania się do komunikatów jest najczęstszy podczas omawiania przetwarzania strumieniowego, czyli wtedy, gdy frameworki (takie jak Kafka Streams, Apache Samza i Storm) operują na komunikatach w czasie rzeczywistym. Taką metodę działania można porównać do sposobu, w jaki frameworki offline, np. Hadoop, są projektowane do późniejszej pracy na danych zbiorczych. Przegląd przetwarzania strumieniowego znajdziesz w rozdziale 14.

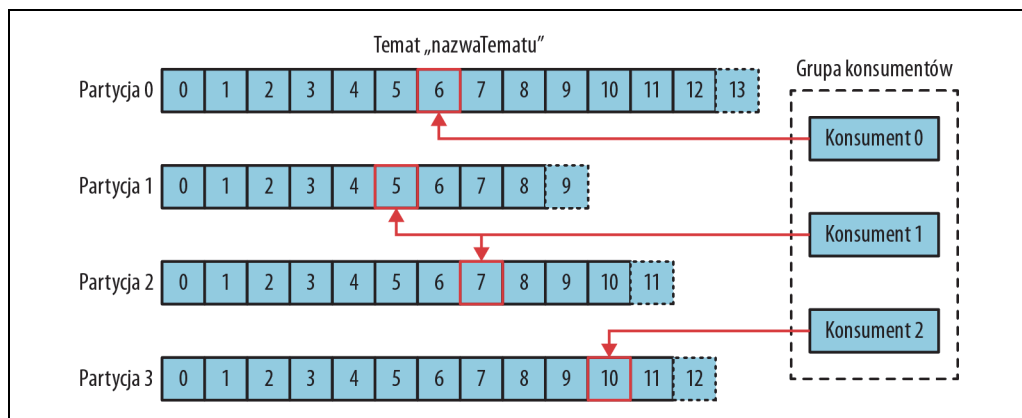
Producenci i konsumenci

Klientami Kafki są użytkownicy systemu i rozróżniamy dwa podstawowe typy: producenci i konsumenci. Dostępne są również zaawansowane klienckie interfejsy API — Kafka Connect API do integracji danych oraz Kafka Streams do przetwarzania strumieniowego. Zaawansowane klienty używają producentów i konsumentów jako elementów konstrukcyjnych i zapewniają funkcjonalność wyższego poziomu.

Producenci (ang. *producer*) tworzą nowe komunikaty. W innych systemach publikuj-subskrybuj producenci mogą być nazywane **wydawcami** (ang. *publisher*) lub **pisarzami** (ang. *writer*). Komunikaty są produkowane dla określonego tematu. Domyślnie producent równomiernie rozkłada komunikaty na wszystkie partycje tematu. W niektórych przypadkach producent kieruje komunikaty do konkretnych partycji. Zwykle odbywa się to za pomocą klucza komunikatu i partycjonera, który generuje skrót klucza i mapuje go na określoną partycję. Dzięki temu wszystkie komunikaty tworzone z danym kluczem są zapisywane na tej samej partycji. Do mapowania komunikatów na partycje producent może używać także niestandardowego partycjonera, który jest zgodny z innymi regułami biznesowymi. Producenci omówiliśmy szczegółowo w rozdziale 3.

Konsumenty odczytują komunikaty. W innych systemach publikuj-subskrybuj te klienty mogą być nazywane **subskrybentami** lub **czytelnikami**. Konsument subskrybuje co najmniej jeden temat i odczytuje komunikaty w kolejności, w jakiej zostały wyprodukowane do poszczególnych partycji. Konsument na podstawie przesunięcia komunikatów śledzi, które komunikaty już przetworzył. **Przesunięcie** (ang. *offset*) — liczba całkowita, która stale rośnie — to kolejny element metadanych, który Kafka dodaje do każdego produkowanego komunikatu. Każdy komunikat w danej partycji ma unikatowe przesunięcie, a następujący po nim komunikat ma przesunięcie większe (choć niekoniecznie monotonicznie większe). Przechowując następne możliwe przesunięcie dla każdej partycji, zwykle w samej Kafce, konsument może zatrzymać przetwarzanie i zrestartować się, nie tracąc swojego miejsca.

Konsumenty pracują w ramach **grupy konsumentów**, składającej się z co najmniej jednego konsumenta. Konsumenty w grupie współpracują ze sobą w celu konsumowania tematu. Grupa zapewnia, że każda partycja jest przetwarzana tylko przez jednego jej członka. Na rysunku 1.6 przedstawiliśmy trzy konsumenty w jednej grupie przetwarzające określony temat. Dwa konsumenty pracują z pojedynczymi partycjami, podczas gdy trzeci konsument konsumuje z dwóch partycji. Mapowanie konsumenta na partycję jest często nazywane posiadaniem przez konsumenta **własności** partycji.



Rysunek 1.6. Grupa konsumentów odczytująca temat

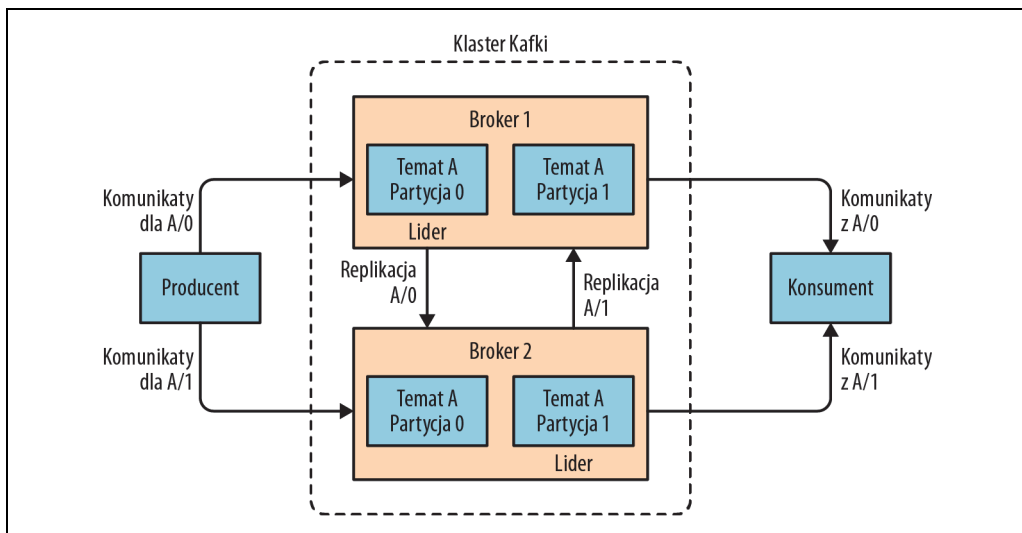
W ten sposób konsumenci mogą skalować się w poziomie, aby konsumować tematy z dużą liczbą komunikatów. Ponadto jeśli pojedynczy konsument ulegnie awarii, pozostali członkowie grupy otrzymają ponowne przypisanie konsumowanych partycji w celu przejęcia pracy brakującego członka grupy. Konsumenty i grupy konsumentów omówiliśmy szerzej w rozdziale 4.

Brokery i klastry

Pojedynczy serwer Kafki jest nazywany **brokerem**. Broker odbiera komunikaty od producentów, przypisuje im przesunięcia i zapisuje komunikaty do magazynu na dysku. Obsługuje również konsumenci, odpowiadając na żądania pobrania dla partycji i wysyłając komunikaty, które zostały opublikowane. W zależności od konkretnego sprzętu i jego charakterystyki wydajności pojedynczy broker może z łatwością obsługiwać tysiące partycji i miliony komunikatów na sekundę.

Brokery Kafki zostały zaprojektowane do działania w ramach **klastra**. W klastrze brokerów jeden z nich będzie działał również jako **kontroler** klastra (wybierany automatycznie spośród aktywnych członków klastra). Kontroler jest odpowiedzialny za operacje administracyjne, w tym przypisywanie partycji do brokerów i monitorowanie awarii brokerów. Partycja jest własnością pojedynczego brokera w klastrze, nazywanego **liderem** (brokerem głównym) danej partycji. Zreplikowana partycja (jak pokazaliśmy na rysunku 1.7) jest przypisywana do dodatkowych brokerów, zwanych **obserwatorami** (ang. *followers*) partycji lub brokerami podrzędnymi. Replikacja zapewnia redundancję komunikatów w partycji, dzięki czemu jeden z obserwatorów może przejąć przywództwo w przypadku awarii brokera. Aby publikować komunikaty, wszystkie producenci muszą połączyć się z liderem, ale konsumenci mogą pobierać komunikaty z lidera lub z jednego z brokerów podrzędnych. Operacje na klastrach, w tym replikację partycji, omówiliśmy szczegółowo w rozdziale 7.

Kluczową funkcjonalnością Apache Kafki jest **retencja**, czyli trwałe przechowywanie komunikatów przez określony przedział czasu. Brokery Kafki są skonfigurowane z domyślnym ustawieniem retencji tematów — przechowują komunikaty przez pewien czas (np. 7 dni) lub do momentu, gdy partycja osiągnie określony rozmiar w bajtach (np. 1 GB). Po osiągnięciu tych limitów ważność komunikatów wygasa i są one usuwane. W ten sposób konfiguracja retencji określa minimalną ilość danych dostępnych w dowolnym momencie. Poszczególne tematy można również skonfigurować z własnymi



Rysunek 1.7. Replikacja partycji w klastrze

ustawieniami retencji, dzięki czemu komunikaty są przechowywane tylko tak długo, jak są przydatne. Temat śledzenia może być np. przechowywany przez kilka dni, podczas gdy wskaźniki aplikacji mogą być przechowywane jedynie przez kilka godzin. Dla tematów można skonfigurować także **kompaktowanie dziennika** (ang. *log compaction*), co oznacza, że Kafka będzie zachowywać tylko ostatni komunikat utworzony z określonym kluczem. Może to być przydatne w wypadku danych dzienników zmian (ang. *changelog data*), w których interesująca jest tylko ostatnia aktualizacja.

Wiele klastrów

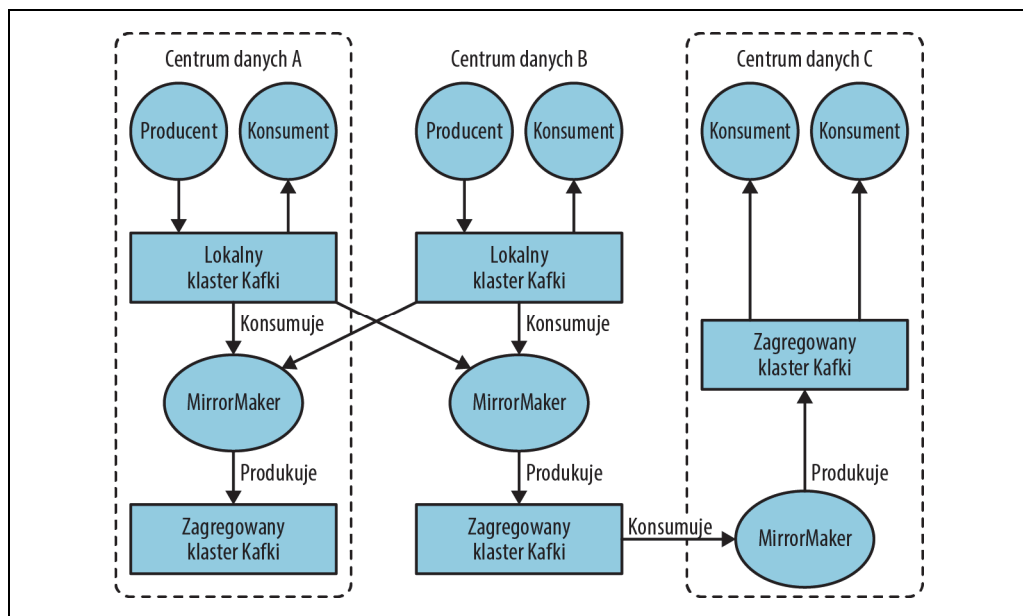
W miarę rozwoju wdrożeń Kafki często korzystne jest posiadanie wielu klastrów. Może to być przydatne z kilku powodów, takich jak:

- segregacja typów danych;
- izolacja ze względu na wymagania bezpieczeństwa;
- wiele centrów danych (odzyskiwanie sprawności po katastrofie).

Podczas pracy z wieloma centrami danych często wymagane jest kopiowanie komunikatów między nimi. W ten sposób aplikacje internetowe mogą mieć dostęp do aktywności użytkowników w obu lokalizacjach. Jeśli użytkownik zmieni np. informacje publiczne w swoim profilu, ta zmiana będzie musiała być widoczna niezależnie od centrum danych, w którym wyświetlane będą wyniki wyszukiwania. Dane monitorowania mogą być gromadzone z wielu lokalizacji w jednej centralnej lokalizacji, w której są hostowane systemy analizy i alarmowania. Mechanizmy replikacji w klastrach Kafki zostały zaprojektowane do pracy tylko w ramach jednego klastra, a nie między wieloma klastrami.

Projekt Kafka zawiera narzędzie o nazwie **MirrorMaker**, służące do replikacji danych do innych klastrów. W gruncie rzeczy MirrorMaker jest po prostu konsumentem i producentem Kafki połączonym

za pomocą kolejki. Komunikaty są konsumowane z jednego klastra Kafka i produkowane do drugiego. Na rysunku 1.8 pokazaliśmy przykład architektury, która korzysta z narzędzia MirrorMaker, agregując komunikaty z dwóch lokalnych klastrów w klastrowy zagregowany, a następnie kopiując ten klastrowy do innych centrów danych. Prosty charakter aplikacji skrywa swoją potęgę w możliwości tworzenia zaawansowanych potoków danych, które omówimy szerzej w rozdziale 9.



Rysunek 1.8. Architektura wielu centrów danych

Dlaczego Kafka?

Dla systemów wymiany komunikatów typu publikuj-subskrybuj dostępnych jest wiele opcji, dlatego więc Apache Kafka jest dobrym wyborem?

Wiele producentów

Kafka jest w stanie bezproblemowo obsługiwać wiele producentów, niezależnie od tego, czy te klienty korzystają z wielu tematów, czy z tego samego tematu. Dzięki temu ten system jest idealny do agregowania danych z wielu systemów frontendowych i do zapewniania ich spójności. Strona udostępniająca użytkownikom zawartość za pośrednictwem wielu mikrousług może mieć np. dla wyświetleń stron pojedynczy temat, w którym mogą zapisywać wszystkie usługi przy użyciu wspólnego formatu. Aplikacje konsumenckie mogą wówczas odbierać pojedynczy strumień wyświetleń stron dla wszystkich aplikacji w witrynie, bez konieczności koordynowania korzystania z wielu tematów, po jednym dla każdej aplikacji.

Wiele konsumentów

Oprócz wielu producentów, Kafka jest przeznaczona dla wielu konsumentów, które mogą odczytywać dowolny pojedynczy strumień komunikatów bez ingerencji w inne klienty. Różni się to od innych systemów kolejkowania, w których komunikat skonsumowany przez jeden klient przestaje być dostępny dla innych klientów. Wiele konsumentów Kafki może działać także w ramach grupy i współdzielić strumień, co gwarantuje, że cała grupa będzie przetwarzać dany komunikat tylko raz.

Retencja na dysku

Kafka nie tylko może obsługiwać wiele konsumentów, ale także trwale przechowywać komunikaty, co oznacza, że konsumenci nie zawsze muszą pracować w czasie rzeczywistym. Komunikaty są zapisywane na dysku i przechowywane zgodnie z konfigurowalnymi regułami retencji. Opcje te można wybierać dla poszczególnych tematów, dzięki czemu różne strumienie komunikatów mogą mieć różny okres retencji zależnie od potrzeb konsumentów. Trwale przechowywanie oznacza, że jeśli konsument pozostanie w tyle z powodu wolnego przetwarzania lub zwiększonego ruchu, nie ma niebezpieczeństwa utraty danych. Oznacza to również, że w konsumentach można przeprowadzać prace konserwacyjne, wyłączając aplikacje na krótki czas i nie przejmując się tworzeniem kopii zapasowych komunikatów w producencie lub utratą danych. Konsumenty mogą być zatrzymywane, a komunikaty zostaną zachowane w Kafce. Umożliwia to restartowanie konsumentów i wznowianie przetwarzania komunikatów bez utraty danych w miejscu, w którym zostało ono przerwane.

Skalowalność

Elastyczna skalowalność Kafki ułatwia obsługę dowolnej ilości danych. Użytkownicy mogą zacząć na próbę od jednego brokera, potem rozszerzyć konfigurację do niewielkiego klastra deweloperskiego składającego się z trzech brokerów, aż w końcu przejść do środowiska produkcyjnego z większym klastrem, złożonym z dziesiątek, a nawet setek brokerów, który będzie rozwijał się wraz ze wzrostem ilości danych. Rozszerzenia można przeprowadzać, gdy klaster jest w trybie online, bez wpływu na dostępność systemu jako całości. Oznacza to również, że klaster wielu brokerów może poradzić sobie z awarią pojedynczego brokera i kontynuować obsługę klientów. Klaster, który musi tolerować więcej jednoczesnych awarii, można skonfigurować z wyższymi współczynnikami replikacji. Replikację omówiliśmy szerzej w rozdziale 7.

Wysoka wydajność

Wszystkie te cechy sprawiają, że Apache Kafka jest systemem wymiany komunikatów typu publikuj-subskrybuj o doskonałej wydajności przy dużym obciążeniu. Producenci, konsumenci i brokerzy mogą być skalowane poziomo, by z łatwością obsługiwać bardzo duże strumienie komunikatów. Można to osiągnąć przy jednoczesnym zapewnieniu mniej niż sekundowego opóźnienia od wygenerowania komunikatu do udostępnienia go konsumentom.

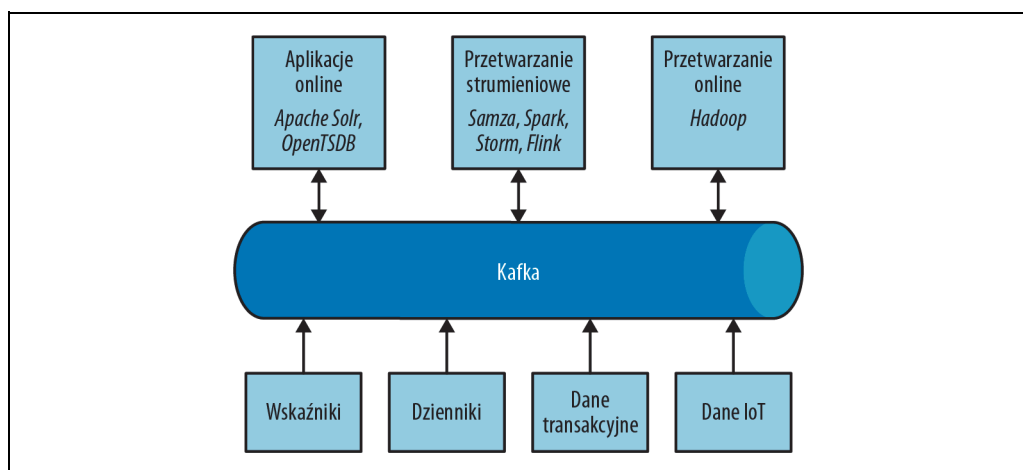
Funkcjonalności platformy

W głównym projekcie Apache Kafka dodano również kilka funkcjonalności platformy do przetwarzania strumieniowego, mogących znacznie ułatwić programistom wykonywanie typowych prac. Chociaż nie są to pełne platformy, które zwykle obejmują ustrukturyzowane środowisko wykonawcze takie jak YARN, funkcjonalności te mają postać interfejsów API i bibliotek, które zapewniają solidną podstawę do rozbudowywania oraz elastyczność w zakresie lokalizacji ich uruchamiania. Kafka Connect ułatwia zadanie pobierania danych z systemu danych źródłowych i przesyłania ich do Kafki lub pobierania danych z Kafki i przesyłania ich do systemu ujścia danych. Kafka Streams zapewnia bibliotekę do prostego tworzenia aplikacji przetwarzania strumieniowego, które są skalowalne i odporne na błędy. Connect omówiliśmy w rozdziale 9., a Streams w rozdziale 14.

Ekosystem danych

W budowanych przez nas środowiskach przetwarzania danych uczestniczy wiele aplikacji. Zdefiniowaliśmy dane wejściowe w postaci aplikacji, które tworzą dane lub w inny sposób wprowadzają je do systemu. Zdefiniowaliśmy dane wyjściowe w postaci wskaźników, raportów i innych produktów opartych na danych. Tworzymy pętle, w których określone komponenty odczytują dane z systemu, przekształcają je na podstawie danych z innych źródeł, a następnie wprowadzają z powrotem do infrastruktury danych w celu wykorzystania w innym miejscu. Wykonuje się to dla wielu typów danych, z których każdy ma unikatowe cechy dotyczące zawartości, rozmiaru i wykorzystania.

Apache Kafka stanowi układ krążenia dla ekosystemu danych, jak pokazaliśmy na rysunku 1.9. Przesyła komunikaty między różnymi składowymi infrastruktury, zapewniając spójny interfejs dla wszystkich klientów. W połączeniu z systemem gwarantującym schematy komunikatów producenci i konsumenci nie wymagają już żadnych ścisłych powiązań ani bezpośrednich połączeń. Komponenty można dodawać i usuwać w miarę tworzenia i usuwania przypadków biznesowych, a producenci nie muszą się przejmować tym, kto korzysta z danych, ani liczbą aplikacji konsumujących.



Rysunek 1.9. Ekosystem big data

Przypadki użycia

Śledzenie aktywności

Oryginalnym przypadkiem użycia Kafki, zaprojektowanym zgodnie z intencją serwisu LinkedIn, jest śledzenie aktywności użytkowników. Użytkownicy serwisu wchodzą w interakcję z aplikacjami frontendowymi, które generują komunikaty dotyczące działań podejmowanych przez użytkowników. Mogą to być informacje pasywne, takie jak wyświetlenia stron i śledzenie kliknięć, lub bardziej złożone działania, takie jak informacje dodawane przez użytkownika do jego profilu. Komunikaty są publikowane w co najmniej jednym temacie, który jest następnie konsumowany przez aplikacje backendowe. Aplikacje te mogą generować raporty, zasilać systemy uczenia maszynowego, aktualizować wyniki wyszukiwania lub wykonywać inne operacje, które są niezbędne do zapewnienia bogatego doświadczenia użytkownika.

Wymiana komunikatów

Kafka jest używana także do wymiany komunikatów, gdzie aplikacje muszą wysyłać użytkownikom powiadomienia (takie jak e-maile). Aplikacje te mogą produkować komunikaty bez konieczności przejmowania się formatowaniem lub sposobem, w jaki komunikaty będą faktycznie wysyłane. Pojedyncza aplikacja może wtedy odczytać wszystkie komunikaty do wysłania i konsekwentnie je obsługiwać. Obejmuje to:

- formatowanie komunikatów (zwane również **dekorowaniem**) przy użyciu wspólnego wyglądu i stylu;
- zbieranie wielu komunikatów w jedno powiadomienie do wysłania;
- stosowanie preferencji użytkownika dotyczących sposobu otrzymywania komunikatów.

Użycie do tego pojedynczej aplikacji pozwala uniknąć powielania tej funkcjonalności w wielu aplikacjach, a ponadto umożliwia operacje takie jak agregacja, które w innym przypadku nie byłyby możliwe.

Wskaźniki i rejestrowanie

Kafka idealnie nadaje się także do gromadzenia wskaźników i dzienników aplikacji i systemu. Jest to przypadek użycia, w którym szczególnie przydaje się możliwość posiadania wielu aplikacji produkujących ten sam typ komunikatów. Aplikacje regularnie publikują wskaźniki w temacie Kafki, a te wskaźniki mogą być wykorzystywane przez systemy do monitorowania i ostrzegania. Można ich również używać w systemie offline, takim jak Hadoop, do przeprowadzania analiz długoterminowych, np. prognoz wzrostu. Komunikaty dziennika mogą być publikowane w ten sam sposób i kierowane do dedykowanych systemów przeszukiwania dzienników, takich jak Elasticsearch lub aplikacje do analizy bezpieczeństwa. Kolejną zaletą Kafki jest to, że gdy system docelowy musi zostać zmieniony (np. nadszedł czas na aktualizację systemu przechowywania dzienników), nie ma potrzeby zmiany aplikacji frontendowych ani środków agregacji.

Dzienniki zatwierdzania

Ponieważ platforma Kafka opiera się na koncepcji dzienników zatwierdzania, zmiany w bazie danych mogą być publikowane do Kafki, a aplikacje mogą z łatwością monitorować ten strumień, aby otrzymywać aktualizacje na żywo w miarę ich pojawiania się. Ten strumień dziennika zmian może być również używany do replikacji aktualizacji bazy danych do systemu zdalnego lub do konsolidacji zmian z wielu aplikacji w pojedynczym widoku bazy danych. Trwałe przechowywanie jest przydatne do zapewnienia bufora dla dziennika zmian, co oznacza, że można go odtwarzać na wypadek awarii aplikacji konsumujących. Tematy skompaktowane w dzienniku mogą być ewentualnie używane w celu zapewnienia dłuższej retencji przez zachowywanie po jednej zmianie na klucz.

Przetwarzanie strumieniowe

Innym obszarem, który zapewnia wiele rodzajów aplikacji, jest przetwarzanie strumieniowe. Choć prawie każde użycie Kafki można traktować jako przetwarzanie strumieniowe, termin ten jest zwykle używany w odniesieniu do aplikacji, które oferują podobną funkcjonalność do przetwarzania MapReduce w Hadoopie. Hadoop zwykle opiera się na agregacji danych w długim przedziale czasu, liczonym w godzinach lub dniach. Przetwarzanie strumieniowe operuje na danych w czasie rzeczywistym, tak szybko, jak produkowane są komunikaty. Frameworki przetwarzania strumieniowego umożliwiają użytkownikom pisanie niewielkich aplikacji do operowania na komunikatach Kafki, wykonywania zadań takich jak zliczanie wskaźników, partycjonowania komunikatów w celu wydajnego przetwarzania przez inne aplikacje lub przekształcania komunikatów przy użyciu danych z wielu źródeł. Przetwarzanie strumieniowe omówiliśmy w rozdziale 14.

Pochodzenie Kafki

Kafka powstała w celu rozwiązania problemu potoku danych w LinkedInie. Została zaprojektowana, aby zapewnić wysokowydajny system wymiany komunikatów, który może obsługiwać wiele typów danych oraz dostarczać czyste, ustrukturyzowane dane na temat aktywności użytkowników i wskaźniki systemowe w czasie rzeczywistym.

Dane naprawdę napędzają wszystko, co robimy.

— Jeff Weiner, były prezes firmy LinkedIn

Problem LinkedIna

Podobnie jak w przykładzie opisanym na początku tego rozdziału, LinkedIn miał system do gromadzenia wskaźników systemu i aplikacji, który wykorzystywał niestandardowe kolektory i narzędzia *open source* do wewnętrznego przechowywania danych oraz ich prezentowania. Oprócz tradycyjnych wskaźników, takich jak wykorzystanie procesora i wydajność aplikacji, istniała zaawansowana funkcjonalność śledzenia żądań, która używała systemu monitorowania i umożliwiała introspekcję w sposób, w jaki pojedyncze żądanie użytkownika jest propagowane przez wewnętrzne aplikacje. System monitoringu miał jednak wiele wad. Obejmowało to gromadzenie wskaźników na podstawie odpytywania, długie interwały między wskaźnikami oraz brak możliwości zarządzania własnymi metrykami przez właścicieli aplikacji. System był skomplikowany, wymagający interwencji

człowieka w przypadku większości prostych zadań i niespójny: z różnymi nazwami wskaźników dla tego samego pomiaru w różnych systemach.

W tym samym czasie powstał system śledzenia informacji o aktywności użytkowników. Była to usługa HTTP, z którą serwery frontendowe łączyły się okresowo i publikowały w niej partie komunikatów (w formacie XML-a). Partie były następnie przenoszone na platformy przetwarzania offline, gdzie pliki były parsowane i porównywane. Ten system miał wiele wad. Formatowanie XML-a było niespójne, a jego parsowanie — kosztowne obliczeniowo. Zmiana typu śledzonej aktywności użytkownika wymagała znacznej ilości skoordynowanej pracy między frontendami a przetwarzaniem offline. Nawet wtedy system ciągle się psuł z powodu zmieniających się schematów. Śledzenie opierało się na codziennym przysyłaniu partii komunikatów, dlatego nie mógł być używany w czasie rzeczywistym.

Monitorowanie i śledzenie aktywności użytkowników nie mogły korzystać z tej samej usługi backendowej. Usługa monitorowania była zbyt nieporęczna, format danych nie był zorientowany na śledzenie aktywności, a model odpytywania dla monitorowania nie był zgodny z modelem wysyłania dla śledzenia. Jednocześnie usługa śledzenia była zbyt delikatna, aby używać jej do wskaźników, a przetwarzanie wsadowe (oparte na partiach komunikatów) nie było odpowiednim modelem dla monitorowania i ostrzegania w czasie rzeczywistym. Lecz dane z monitorowania i śledzenia miały wiele wspólnych cech, a korelacja informacji (np. wpływ określonych rodzajów aktywności użytkowników na wydajność aplikacji) była bardzo pożądana. Spadek określonych rodzajów aktywności użytkowników może wskazywać na problemy z aplikacją, która te aktywności obsługiwała, ale godziny opóźnienia w przetwarzaniu partii aktywności oznaczały spowolnioną reakcję na tego typu problemy.

Na początku dokładnie przeanalizowano istniejące, gotowe rozwiązania typu *open source*, by znaleźć nowy system, który zapewniłby dostęp do danych w czasie rzeczywistym i skalował się tak, aby obsłużyć wymaganą ilość przepływu komunikatów. Prototypowe systemy zostały skonfigurowane przy użyciu brokera ActiveMQ, ale w tamtym czasie nie radził on sobie z tą skalą. Było to również kruche rozwiązanie, jeśli chodzi o sposób, w jaki LinkedIn potrzebował z niego korzystać, i ujawniło wiele luk w ActiveMQ, które wstrzymywałyby pracę brokerów. Te postoje powodowały zapychanie się połączeń z klientami i zakłócały zdolność aplikacji do obsługi żądań użytkowników. Podjęto decyzję, aby pójść do przodu i wprowadzić niestandardową infrastrukturę dla potoku danych.

Narodziny Kafki

Zespołem programistów w LinkedInie kierował Jay Kreps, główny inżynier oprogramowania, który wcześniej był odpowiedzialny za rozwój i wydanie na licencji *open source* Voldemorta, czyli rozproszonego systemu pamięci masowej typu klucz-wartość. W początkowym zespole znaleźli się również Neha Narkhede, a później Jun Rao. Wspólnie postanowili stworzyć system wymiany komunikatów, który mógłby zaspokoić potrzeby zarówno systemów monitorowania, jak i śledzenia, a także skalować się na przyszłość. Oto główne cele, jakie zostały wytyczone:

- oddzielenie producentów i konsumentów dzięki użyciu modelu *push-pull* (wyślij-pobierz);
- zapewnienie trwałości danych komunikatów w systemie wymiany komunikatów, aby umożliwić korzystanie z wielu konsumentów;

- optymalizacja pod kątem wysokiej przepływności komunikatów;
- umożliwienie poziomego skalowania systemu w celu zapewnienia rozwoju w miarę powiększania się strumieni danych.

W rezultacie powstał system wymiany komunikatów typu publikuj-subskrybuj, który miał interfejs typowy dla tego rodzaju systemów, ale warstwa pamięci bardziej przypominała system agregacji dzienników. W połączeniu z platformą Apache Avro do serializacji komunikatów Kafka skutecznie radziła sobie zarówno ze wskaźnikami, jak i śledzeniem aktywności użytkowników w skali miliardów komunikatów dziennie. Skalowalność Kafki pomogła zwiększyć wykorzystanie LinkedIna do ponad 7 bilionów komunikatów (stan na luty 2020 r.) i ponad 5 petabajtów danych konsumowanych dziennie.

Open source

Kafka została wydana na GitHubie pod koniec 2010 r. jako projekt dostępny na licencji *open source*. Gdy zaczęła przyciągać uwagę społeczności *open source*, została zaproponowana i zaakceptowana jako projekt inkubatora Apache Software Foundation w lipcu 2011 r. Apache Kafka wyszła z inkubatora w październiku 2012 r. Od tego czasu jest stale rozwijana i znalazła solidną społeczność współtwórców i kontrybutorów spoza LinkedIna. Kafka jest obecnie używana w kilku największych potokach danych na świecie, w tym w Netfliksie, Uberze i wielu innych firmach.

Powszechne przyjęcie Kafki utworzyło zdrowy ekosystem również wokół głównego projektu. Powstały aktywne grupy spotkań w dziesiątkach krajów na całym świecie, zapewniające lokalne dyskusje i wsparcie dla przetwarzania strumieniowego. Istnieje też wiele projektów otwartoźródłowych związanych z platformą Apache Kafka. LinkedIn nadal utrzymuje kilka z nich, m.in.: Cruise Control, Kafka Monitor i Burrow. Oprócz oferty komercyjnej firma Confluent wydała takie projekty, jak ksqlDB, rejestr schematów i proxy REST na licencji wspólnotowej (która nie jest ściśle otwartoźródłowa, ponieważ obejmuje ograniczenia użytkowania). Kilka najpopularniejszych projektów wymieniliśmy w dodatku B.

Zaangażowanie komercyjne

Jesienią 2014 r. Jay Kreps, Neha Narkhede i Jun Rao opuścili LinkedIn, aby założyć Confluent, firmę skupiającą się na zapewnianiu rozwoju, wsparcia dla przedsiębiorstw i szkoleń z platformy Apache Kafka. Połączyli siły z innymi firmami (takich jak Heroku), aby dostarczać dla Kafki usługi w chmurze. Poprzez partnerstwo z Google'em firma Confluent dostarcza zarządzane klastry Kafki na Google Cloud Platform, a także podobne usługi na Amazon Web Services i Azure. Jedną z innych ważnych inicjatyw Confluenta jest organizacja cyklu konferencji Kafka Summit. Rozpoczęta w 2016 r. konferencjami odbywającymi się corocznie w Stanach Zjednoczonych i Londynie Kafka Summit stała się miejscem spotkań społeczności na skalę globalną i dzielenia się wiedzą na temat platformy Apache Kafka i powiązanych projektów.

Nazwa

Ludzie często pytają, skąd Kafka wzięła swoją nazwę i czy oznacza coś konkretnego dotyczącego aplikacji. Jay Kreps przedstawił następujący wywód:

Pomyślałem, że skoro Kafka jest systemem zoptymalizowanym pod kątem zapisywania, sensowne będzie użycie nazwiska jakiegoś pisarza. Na studiach brałem udział w wielu zajęciach z literatury i lubiłem Franza Kafkę. Dodatkowo nazwa brzmiała fajnie jak na projekt *open source*.

W zasadzie nazwa nie jest więc zbyt związana z samą aplikacją.

Pierwsze kroki z Kafką

Skoro wiesz już wszystko o Kafce i jej historii, możemy ją skonfigurować i zbudować własny potok danych. W następnym rozdziale zajmiemy się instalacją i konfiguracją Kafki. Omówimy również wybór odpowiedniego sprzętu do uruchomienia Kafki oraz kilka kwestii, o których należy pamiętać podczas przechodzenia do działania w środowisku produkcyjnym.

Instalowanie Kafki

W tym rozdziale opiszemy, jak rozpocząć pracę z brokerem platformy Apache Kafka oraz jak skonfigurować serwer Apache ZooKeeper, który jest używany przez Kafkę do przechowywania metadanych dla brokerów. Omówimy również podstawowe opcje konfiguracji dla wdrożeń Kafki i przedstawimy kilka sugestii dotyczących wyboru odpowiedniego sprzętu do uruchomienia brokerów. Na koniec zajmiemy się instalacją wielu brokerów Kafki w ramach jednego klastra i wskażemy najistotniejsze kwestie, które należy uwzględnić podczas korzystania z Kafki w środowisku produkcyjnym.

Konfiguracja środowiska

Zanim zaczniesz używać platformy Apache Kafka, musisz skonfigurować środowisko z kilkoma wymaganiami wstępnymi, by zapewnić prawidłowe działanie Kafki. Przez ten proces poprowadzimy Cię w kolejnych punktach.

Wybór systemu operacyjnego

Apache Kafka to aplikacja Javy, która może działać w wielu systemach operacyjnych, m.in.: Windows, macOS i Linux. Zalecany systemem operacyjnym do ogólnego przypadku użycia jest jednak Linux. W tym rozdziale skupimy się na konfiguracji i używaniu Kafki w środowisku linuxowym. Informacje na temat instalowania tej platformy w systemach Windows i macOS znajdziesz w dodatku A.

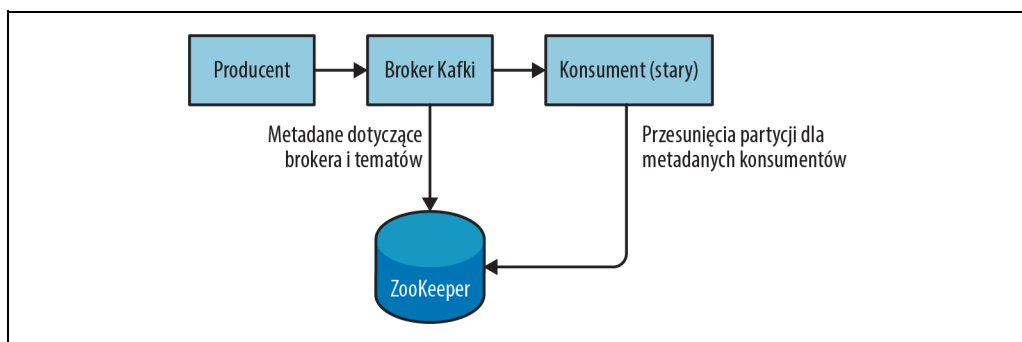
Instalowanie Javy

Do zainstalowania ZooKeepera lub Kafki będziesz potrzebować skonfigurowanego i działającego środowiska Javy. Kafka i ZooKeeper dobrze współpracują ze wszystkimi implementacjami Javy opartymi na OpenJDK, w tym Oracle JDK. Najnowsze wersje Kafki obsługują zarówno Javę 8, jak i Javę 11. Możesz korzystać z wersji dostarczonej przez system operacyjny lub pobrać bezpośrednio z internetu, np. z witryny Oracle dla wersji Oracle (<https://www.oracle.com/java>). Chociaż ZooKeeper i Kafka będą działać także z środowiskiem uruchomieniowym Javy (ang. *Java Runtime Environment* — JRE), podczas tworzenia narzędzi i aplikacji zaleca się posiadanie pełnego pakietu JDK (ang. *Java Development Kit*). Zaleca się ponadto zainstalowanie najnowszej wydanej wersji poprawki do środowiska Javy, ponieważ starsze wersje mogą mieć luki w zabezpieczeniach. W celu omówienia

kroków instalacyjnych przyjęliśmy założenie, że masz zainstalowany pakiet JDK w wersji 11 z aktualizacją 10 (*Version 11 Update 10*), wdrożony w lokalizacji `/usr/java/jdk-11.0.10`.

Instalowanie ZooKeepera

Do przechowywania metadanych dotyczących klastra Kafka oraz szczegółowych informacji na temat klientów konsumenckich Apache Kafka używa serwera Apache ZooKeeper, jak pokazaliśmy na rysunku 2.1. ZooKeeper to scentralizowana usługa do przechowywania informacji konfiguracyjnych, świadczenia usługi nazw, zapewniania rozproszonej synchronizacji i świadczenia usługi grup. W tej książce nie będziemy omawiać ZooKeepera szczegółowo, a ograniczymy się jedynie do wyjaśnienia tego, co jest potrzebne do obsługi Kafka. Chociaż możliwe jest uruchomienie serwera ZooKeeper przy użyciu skryptów zawartych w dystrybucji Kafka, zainstalowanie pełnej wersji ZooKeepera z dystrybucji jest w sumie trywialne.



Rysunek 2.1. Kafka i ZooKeeper

Kafka została gruntownie przetestowana ze stabilną wersją 3.5 ZooKeepera i jest regularnie aktualizowana o najnowsze wydania. W tej książce będziemy używać ZooKeepera 3.5.9, który możesz pobrać ze strony <https://oreil.ly/iMZjR>.

Autonomiczny serwer

ZooKeeper zawiera w lokalizacji `/usr/local/zookeeper/config/zoo_sample.cfg` podstawowy przykładowy plik konfiguracyjny, który będzie się sprawdzał w większości przypadków użycia. Jednak w tej książce do celów demonstracyjnych będziemy ręcznie tworzyć własne pliki konfiguracyjne z kilkoma bazowymi ustawieniami. Poniższy przykład powoduje zainstalowanie ZooKeepera z podstawową konfiguracją w lokalizacji `/usr/local/zookeeper` z przechowywaniem jego danych w lokalizacji `/var/lib/zookeeper`:

```
# tar -zxf apache-zookeeper-3.5.9-bin.tar.gz
# mv apache-zookeeper-3.5.9-bin /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cp > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
```

```
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

Aby sprawdzić, czy ZooKeeper działa poprawnie w trybie autonomicznym (ang. *standalone*), możesz połączyć się z portem klienta i wysłać czteroliterowe polecenie `srvr`. Spowoduje to zwrócenie z uruchomionego serwera podstawowych informacji o ZooKeeperze:

```
# telnet localhost 2181
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
srvr
Zookeeper version: 3.5.9-83df9301aa5c2a5d284a9940177808c01bc35cef, built on
01/06/2021 19:49 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 5
Connection closed by foreign host.
#
```

Ansambl ZooKeepera

Aby zapewnić wysoką dostępność, ZooKeeper został zaprojektowany do pracy jako klastery zwane **ansambelami** (ang. *ensemble*). Ze względu na zastosowany algorytm równoważenia zaleca się, aby ansamble zawierały nieparzystą liczbę serwerów (3, 5 itd.), ponieważ ZooKeeper będzie odpowiadał na żądania tylko wtedy, gdy działać będzie większość węzłów składowych ansambli (**kworum**). Oznacza to, że ansamble składające się np. z 3 węzłów mogą działać z 1 brakującym węzłem, a 5-węzłowe ansamble można uruchomić z brakującymi 2 węzłami.



Ustalanie rozmiaru ansambli ZooKeepera

Rozważ uruchomienie ZooKeepera w 5-węzłowym ansambli. Aby wprowadzić zmiany w konfiguracji ansambli, w tym podmienić węzeł, trzeba przeładować węzły pojedynczo. Jeśli Twój ansambl nie będzie tolerował awarii więcej niż jednego węzła, wykonywanie prac konserwacyjnych będzie się wiązało z dodatkowym ryzykiem. Nie zaleca się także uruchamiania więcej niż 7 węzłów, ponieważ wydajność może zacząć spadać ze względu na charakter protokołu konsensusu.

Ponadto, jeśli uważasz, że 5 lub 7 węzłów nie będzie w stanie obsłużyć obciążenia z powodu zbyt wielu połączeń klientów, rozważ uruchomienie dodatkowych węzłów obserwatorów, aby pomóc w równoważeniu ruchu związanego z odczytem.

Serwery ZooKeeper można skonfigurować w ansambli pod warunkiem, że mają wspólną konfigurację, która zawiera listę wszystkich serwerów, a każdy serwer ma w katalogu danych plik *myid*

określający numer identyfikacyjny serwera. Jeśli nazwy hostów serwerów w ansambli to *zoo1.example.com*, *zoo2.example.com* i *zoo3.example.com*, plik konfiguracyjny może wyglądać tak:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

W tej konfiguracji *initLimit* to ilość czasu, jaką węzły podrzędne, zwane obserwatorami (ang. *followers*), mają na połączenie się z węzłem głównym, określanym mianem lidera (ang. *leader*). Wartość *syncLimit* określa, jak długo węzły podrzędne mogą pozostawać w stanie braku synchronizacji z węzłem głównym. Obie wartości są liczbą jednostek *tickTime*, co w tym przypadku oznacza, że *initLimit* wynosi 20·2000 milisekund, czyli 40 sekund. Ta konfiguracja zawiera również listę wszystkich serwerów w ansambli. Serwery określa się w formacie *server.X=nazwa_hosta:port_równorzędny:port_lidera*. Oto opis poszczególnych parametrów:

X Numer identyfikacyjny serwera. Musi to być liczba całkowita, ale nie musi być liczbą od zera ani sekwencyjną.

nazwa_hosta

Nazwa hosta lub adres IP serwera.

port_równorzędny

Port TCP, przez który komunikują się ze sobą serwery w ansambli.

port_lidera

Port TCP, przez który przeprowadzany jest wybór węzła głównego.

Klienty muszą mieć tylko możliwość połączenia się z ansambliem przez *clientPort*, ale węzły składowe ansambli potrzebują komunikować się ze sobą przez wszystkie trzy porty.

Oprócz współdzielonego pliku konfiguracyjnego każdy serwer musi mieć w katalogu *dataDir* plik o nazwie *myid*. Ten plik musi zawierać numer identyfikacyjny serwera, który powinien być zgodny z numerem z pliku konfiguracyjnego. Po wykonaniu tych kroków konfiguracyjnych serwery zostaną uruchomione i będą się ze sobą komunikować w ansambli.



Testowanie ansambli ZooKeepera na pojedynczej maszynie

W celu uruchomienia i przetestowania ansambli ZooKeepera na pojedynczej maszynie należy w konfiguracji ustawić wszystkie nazwy hostów jako *localhost* i określić unikatowe wartości dla parametrów *port_równorzędny* i *port_lidera* dla każdej instancji. Dodatkowo dla każdej instancji należałoby utworzyć osobny plik *zoo.cfg* z unikatowymi *dataDir* i *clientPort*. Może to być przydatne tylko do celów testowych i *nie jest* zalecane w przypadku systemów produkcyjnych.

Instalowanie brokera Kafki

Po skonfigurowaniu Javy i ZooKeepera jesteś gotowy do zainstalowania platformy Apache Kafka. Aktualną wersję możesz pobrać ze strony Kafki (<https://oreil.ly/xLopS>). Gdy piszemy ten rozdział, jest to wersja 2.8.0 działająca pod Scalą w wersji 2.13.0. W przykładach przedstawionych w tym rozdziale używamy wersji 2.7.0.

Poniższy przykład powoduje zainstalowanie Kafki w lokalizacji `/usr/local/kafka` i skonfigurowanie jej do korzystania z uruchomionego wcześniej serwera ZooKeeper i przechowywania segmentów dziennika komunikatów w lokalizacji `/tmp/kafka-logs`:

```
# tar -zxf kafka_2.13-2.7.0.tgz
# mv kafka_2.13-2.7.0 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/
server.properties
#
```

Po uruchomieniu brokera Kafki możesz sprawdzić, czy działa. W tym celu musisz wykonać na klastrze kilka prostych operacji, takich jak utworzenie tematu testowego oraz wyprodukowanie i skonsumowanie kilku komunikatów.

Utwórz i zweryfikuj temat:

```
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --create
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe
--topic test
Topic:test      PartitionCount:1  ReplicationFactor:1  Configs:
      Topic: test      Partition: 0      Leader: 0      Replicas: 0      Isr: 0
#
```

Wyprodukuj komunikaty do tematu testowego (użyj `Ctrl+C`, aby zatrzymać producenta w dowolnym momencie):

```
# /usr/local/kafka/bin/kafka-console-producer.sh --bootstrap-server localhost:9092
--topic test
Test Message 1
Test Message 2
^C
#
```

Skonsumuj komunikaty z tematu testowego:

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic test --from-beginning
Test Message 1
Test Message 2
^C
Processed a total of 2 messages
#
```



Wycofanie połączeń ZooKeepera w narzędziach Kafka CLI

Jeśli znasz starsze wersje narzędzi Kafki, możesz być przyzwyczajony do używania dla połączenia łańcucha znaków `--zookeeper`. Zostało to zarzucone w prawie wszystkich przypadkach użycia. Obecnie najlepszą praktyką jest użycie nowszej opcji `--bootstrap-server` i łączenie się bezpośrednio z brokerem Kafki. Jeżeli pracujesz w klastrze, możesz podać wartości `host:port` dowolnego brokera w tym klastrze.

Konfiguracja brokera

Przykładowa konfiguracja dostarczona z dystrybucją Kafki jest wystarczająca do uruchomienia autonomicznego serwera jako weryfikacji koncepcji, ale najprawdopodobniej nie będzie wystarczająca dla dużych instalacji. Istnieje wiele opcji konfiguracyjnych Kafki, które kontrolują wszystkie aspekty konfiguracji i dostosowywania. Dla większości opcji można jednak pozostawić ustawienia domyślne, gdyż dotyczą one aspektów dostosowywania brokera Kafki, które nie będą miały zastosowania, dopóki nie pojawi się konkretny przypadek użycia wymagający ich zmiany.

Ogólne parametry brokera

Podczas wdrażania Kafki dla dowolnego środowiska innego niż samodzielny broker na pojedynczym serwerze należy przejrzeć kilka parametrów konfiguracyjnych brokera. Parametry dotyczą podstawowej konfiguracji brokera i większość z nich należy zmienić, aby działały poprawnie w klastrze z innymi brokerami.

broker.id

Każdy broker Kafki musi mieć identyfikator w postaci liczby całkowitej, ustawiany za pomocą konfiguracji parametru `broker.id`. Domyślnie ta liczba całkowita jest ustawiona na 0, ale może to być dowolna wartość. Istotne jest, aby identyfikator był unikatowy dla każdego brokera w ramach pojedynczego klastra Kafki. Z technicznego punktu widzenia można wybrać dowolną liczbę całkowitą i można ją przenosić między brokerami, jeśli jest to konieczne do wykonywania zadań konserwacyjnych. Zdecydowanie zaleca się jednak ustawienie wartości charakterystycznej dla danego hosta, aby podczas wykonywania konserwacji mapowanie numerów identyfikatorów brokerów na hosty nie było uciążliwe. Jeżeli nazwy hostów zawierają np. unikatową liczbę (tak jak `host1.example.com`, `host2.example.com` itd.), dobrym wyborem dla wartości `broker.id` będą odpowiednio 1 i 2.

listener

W starszych wersjach Kafki wykorzystywana była prosta konfiguracja parametru `port`. Można jej nadal używać jako konfiguracji zapasowej dla prostych przypadków użycia, ale jest ona przestarzała. Przykładowy plik konfiguracyjny powoduje uruchomienie Kafki z nasłuchiwcem (ang. *listener*) na porcie TCP 9092. Nowa konfiguracja `listener` to lista rozdzielonych przecinkami identyfikatorów URI oraz nazw nasłuchiwczy. Jeśli nazwą nasłuchiwachy nie jest powszechny protokół zabezpieczeń, należy zdefiniować także konfigurację `listener.security.protocol.map`. Nasłuchiwachy definiuje się w formacie `<protokół>://<nazwa_hosta>:<port>`. Przykładem poprawnej konfiguracji

nasłuchiacza może być `PLAINTEXT://localhost:9092,SSL://:9091`. Określenie nazwy hosta jako `0.0.0.0` spowoduje powiązanie go ze wszystkimi interfejsami. Pozostawienie pustej nazwy hosta spowoduje powiązanie go z domyślnym interfejsem. Pamiętaj, że jeśli wybrany zostanie port o wartości mniejszej niż 1024, Kafka musi zostać uruchomiona jako *root*. Uruchamianie Kafki jako *roota* nie jest zalecaną konfiguracją.

zookeeper.connect

Lokalizację ZooKeepera używaną do przechowywania metadanych brokera ustawia się za pomocą parametru konfiguracyjnego `zookeeper.connect`. W przykładowej konfiguracji używany jest ZooKeeper uruchomiony na porcie 2181 na hoście lokalnym, co określa się jako `localhost:2181`. Format tego parametru to rozdzielona średnikami lista łańcuchów znaków *nazwa_hosta:port/ścieżka*:

nazwa_hosta

Nazwa hosta lub adres IP serwera ZooKeeper.

port

Numer portu klienckiego dla serwera.

/ścieżka

Opcjonalna ścieżka ZooKeepera do użycia jako środowisko *chroot* dla klastra Kafki. W przypadku pominięcia tego parametru używana jest ścieżka główna.

Jeśli określona zostanie ścieżka *chroot* (ścieżka wyznaczona jako katalog główny dla danej aplikacji), która nie istnieje, zostanie ona utworzona przez broker podczas uruchamiania.



Dlaczego warto korzystać ze ścieżki *chroot*?

Powszechnie uważa się, że dobrą praktyką jest używanie dla klastra Kafki ścieżki *chroot*. Pozwala to na bezkonfliktowe współdzielenie ansamblu ZooKeepera z innymi aplikacjami, w tym innymi klastrami Kafki. W tej konfiguracji najlepiej jest ponadto określić wiele serwerów ZooKeeper (które są częścią tego samego ansamblu). Dzięki temu w przypadku awarii serwera broker Kafki może połączyć się z innym węzłem składowym ansamblu ZooKeepera.

log.dirs

Kafka utrzuwa wszystkie komunikaty na dysku, a te segmenty dziennika są przechowywane w katalogu określonym w konfiguracji `log.dir`. W przypadku wielu katalogów preferowana jest konfiguracja `log.dirs`. Jeśli ta wartość nie zostanie ustawiona, domyślnie przywrócony zostanie `log.dir`. Konfiguracja `log.dirs` to rozdzielona przecinkami lista ścieżek w systemie lokalnym. Jeżeli określona zostanie więcej niż jedna ścieżka, broker będzie zapisywał w nich partycje, wybierając „najmniej używaną” ścieżkę, przy czym segmenty dziennika jednej partycji będą przechowywane w tej samej ścieżce. Zwróć uwagę, że broker umieści nową partycję w ścieżce, w której w danym momencie będzie przechowywana najmniejsza liczba partycji, a która będzie miała najwięcej dostępnego miejsca na dysku, stąd równomierne rozmieszczenie danych w wielu katalogach nie jest gwarantowane.

num.recovery.threads.per.data.dir

Do obsługi segmentów dziennika Kafka używa konfigurowalnej puli wątków. Obecnie ta pula wątków jest używana w następujących przypadkach:

- podczas normalnego uruchamiania, aby otworzyć segmenty dziennika wszystkich partycji;
- podczas uruchamiania po awarii, aby sprawdzić i przyciąć segmenty dziennika wszystkich partycji;
- podczas wyłączania, aby czysto zamknąć segmenty dziennika.

Domyślnie używany jest tylko jeden wątek na każdy katalog dziennika. Ponieważ wątki te są wykorzystywane tylko podczas uruchamiania i zamykania, rozsądne jest ustawienie większej liczby wątków w celu zrównoleglenia operacji. W przypadku odzyskiwania sprawności po nieczystym zamknięciu może to oznaczać różnicę kilku godzin podczas ponownego uruchamiania brokera z dużą liczbą partycji! Ustawiając ten parametr, należy pamiętać, że skonfigurowana wartość to liczba wątków przypadająca na katalog dziennika określony za pomocą `log.dirs`. Oznacza to, że jeśli dla `num.recovery.threads.per.data.dir` ustawiona została wartość 8, a w `log.dirs` są określone 3 ścieżki, daje to w sumie 24 wątki.

auto.create.topics.enable

Domyślna konfiguracja Kafki określa, że broker powinien automatycznie utworzyć temat w następujących okolicznościach:

- kiedy producent zaczyna zapisywać komunikaty w temacie;
- kiedy konsument zaczyna odczytywać komunikaty z tematu;
- kiedy jakiegokolwiek klient zażąda metadanych dla tematu.

W wielu sytuacjach może to być niepożądane zachowanie, zwłaszcza że nie ma możliwości zweryfikowania istnienia tematu za pomocą protokołu Kafki bez spowodowania jego utworzenia. Jeśli zarządzasz tworzeniem tematów bezpośrednio — ręcznie lub za pośrednictwem systemu alokacji — możesz ustawić dla konfiguracji `auto.create.topics.enable` wartość `false`.

auto.leader.rebalance.enable

Aby mieć pewność, że klaster Kafki nie stanie się niezrównoważony przez posiadanie przywództwa wszystkich tematów na jednym brokerze, można określić tę konfigurację w celu zagwarantowania jak największego zrównoważenia przywództwa. Powoduje ona uruchomienie działającego w tle wątku, który w regularnych odstępach czasu (interwał ten można skonfigurować za pomocą parametru `leader.imbalance.check.interval.seconds`) sprawdza rozkład partycji. Jeśli brak równowagi przywództwa przekracza wartość określoną w konfiguracji `leader.imbalance.per.broker.percentage`, rozpoczyna się ponowne równoważenie preferowanych liderów dla partycji.

delete.topic.enable

W zależności od środowiska i wytycznych dotyczących przechowywania danych możesz chcieć zablokować klaster, by zapobiec arbitralnemu usuwaniu tematów. Możliwość usuwania tematów można wyłączyć przez ustawienie dla tej flagi wartości `false`.

Domyślne ustawienia tematu

Konfiguracja serwera Kafki określa wiele ustawień domyślnych dla tworzonych tematów. Kilka z tych parametrów, w tym liczbę partycji i czas przechowywania komunikatów, można ustawić dla poszczególnych tematów za pomocą narzędzi administracyjnych (które omówiliśmy w rozdziale 12.). Dla ustawień domyślnych w konfiguracji serwera należy zastosować wartości bazowe odpowiednie dla większości tematów w klastrze.



Korzystanie z nadpisywania ustawień dla poszczególnych tematów

W starszych wersjach Kafki możliwe było określenie nadpisań dla poszczególnych tematów dla tych ustawień w konfiguracji brokera za pomocą parametrów `log.retention.hours.per.topic`, `log.retention.bytes.per.topic` oraz `log.segment.bytes.per.topic`. Parametry te nie są już obsługiwane, a nadpisanie należy określać za pomocą narzędzi administracyjnych.

num.partitions

Parametr `num.partitions` określa liczbę partycji, z którymi tworzony jest nowy temat, głównie gdy włączone jest automatyczne tworzenie tematów (co jest ustawieniem domyślnym). Domyślną wartością tego parametru jest jedna partycja. Pamiętaj, że liczbę partycji dla tematu można tylko zwiększać, nigdy nie zmniejszać. Oznacza to, że jeśli jakiś temat musi mieć mniej partycji niż `num.partitions`, należy utworzyć go ręcznie (omówiliśmy to w rozdziale 12.).

Jak opisaliśmy w rozdziale 1., partycje umożliwiają skalowanie tematu w klastrze Kafki, dlatego ważne jest używanie liczby partycji, aby równoważyć obciążenie komunikatami w całym klastrze w miarę dodawania brokerów. Dla wielu użytkowników liczba partycji dla tematu będzie równa liczbie brokerów w klastrze lub będzie jej wielokrotnością. Dzięki temu partycje mogą być równomiernie dystrybuowane do brokerów, co z kolei powoduje równoważenie obciążenia komunikatami. Optymalną przepływność będzie miał np. temat z 10 partycjami działającymi w klastrze Kafki z 10 hostami z przywództwem rozłożonym równomiernie na wszystkie 10 hostów. Taka konfiguracja nie jest jednak wymaganiem, ponieważ obciążenie komunikatami można równoważyć także na inne sposoby, np. za pomocą wielu tematów.

Gdy weźmiesz pod uwagę wszystkie te kwestie, stanie się oczywiste, że potrzebujesz wielu partycji, ale niezbyt wielu. Jeśli masz jakieś oszacowanie dotyczące docelowej przepływności tematu i oczekiwanej przepływności konsumentów, liczbę partycji możesz obliczyć, dzieląc przepływność docelową przez oczekiwaną przepływność konsumentów. Jeżeli chcesz mieć możliwość zapisu w temacie i odczytu z niego z prędkością 1 GB/s i wiesz, że każdy konsument może przetworzyć tylko 50 MB/s, potrzebujesz wtedy co najmniej 20 partycji. Przy takiej konfiguracji możesz mieć 20 konsumentów odczytujących z tematu i osiągać 1 GB/s.

Jak wybrać liczbę partycji?

Przy wyborze liczby partycji należy wziąć pod uwagę kilka czynników. Oto one:

- Jaką przepływność chcesz osiągnąć w danym temacie? Czy oczekujesz np. zapisów z prędkością 100 kB/s czy 1 GB/s?
- Jakiej maksymalnej przepływności oczekujesz podczas konsumowania z pojedynczej partycji? Partycja będzie zawsze w całości konsumowana przez jeden konsument (nawet jeżeli nie używasz grup konsumentów, konsument musi odczytywać wszystkie komunikaty z partycji). Jeśli wiesz, że Twój wolniejszy klient zapisuje dane w bazie danych, która nigdy nie obsługuje przepustowości większej niż 50 MB/s dla każdego zapisującego w niej wątku, wtedy podczas korzystania z partycji masz przepływność ograniczoną do 50 MB/s.
- W ten sam sposób możesz oszacować maksymalną przepływność na producenta dla pojedynczej partycji, ponieważ jednak producenci są z reguły znacznie szybsze niż konsumenci, zwykle można to bezpiecznie pominąć.
- Jeśli komunikaty wysyłasz do partycji na podstawie kluczy, późniejsze dodawanie partycji może być bardzo trudne, dlatego przepływność obliczaj na podstawie oczekiwanego przyszłego użycia, a nie bieżącego użycia.
- Weź pod uwagę liczbę partycji, które umieścisz na każdym brokerze, oraz dostępną przestrzeń dyskową i przepustowość sieci dla każdego brokera.
- Unikaj przeszacowywania, ponieważ każda partycja wykorzystuje pamięć i inne zasoby brokera, co wydłuża czas aktualizacji metadanych i przenoszenia przywództwa.
- Jeśli będziesz stosować mirroring danych, konieczne może być również uwzględnienie przepustowości konfiguracji mirroringu. Duże partycje mogą stać się wąskim gardłem w wielu konfiguracjach mirroringu.
- Jeśli korzystasz z usług w chmurze, to czy masz ograniczenia liczby operacji we-wy na sekundę (ang. *Input/Output Operations Per Second* — IOPS) na swoich maszynach wirtualnych lub dyskach? W zależności od usługi w chmurze i konfiguracji maszyn wirtualnych mogą obowiązywać sztywne ograniczenia liczby dozwolonych operacji IOPS, które spowodują osiągnięcie limitów. Posiadanie zbyt wielu partycji może mieć efekt uboczny, polegający na zwiększeniu liczby IOPS ze względu na występującą równoległość.

W przypadku gdy nie masz tych szczegółowych informacji, nasze doświadczenie podpowiada, że często zadowalające rezultaty daje ograniczenie rozmiaru partycji na dysku do mniej niż 6 GB dziennej retencji. Rozpoczęcie od niewielkich wartości i rozwijanie się w miarę zwiększania zapotrzebowania jest łatwiejsze niż zaczynanie od zbyt dużych wartości.

default.replication.factor

Jeśli włączone jest automatyczne tworzenie tematów, ta konfiguracja określa współczynnik replikacji dla nowych tematów. Strategia replikacji może się różnić w zależności od pożądanej trwałości lub dostępności klastra i omówimy ją szerzej w kolejnych rozdziałach. Poniżej przedstawiliśmy zwięzłą rekomendację dla przypadku używania Kafki w klastrze — zapobiegnie ona przestojom spowodowanym czynnikami niezależnymi od wewnętrznych możliwości platformy Kafka, takimi jak awarie sprzętu.

Zdecydowanie zaleca się ustawienie dla współczynnika replikacji wartości co najmniej o 1 większej od `min.insync.replicas`. Jeśli masz wystarczająco duże klastry i odpowiednią ilość sprzętu, preferowane może być ustawienie dla współczynnika replikacji wartości o 2 większej niż `min.insync.replicas` (w skrócie `RF++`), co zapewnia większą odporność na awarie. `RF++` ułatwi utrzymywanie i zapobiegnie przestojom. Zastosowanie się do tego zalecenia umożliwia tolerowanie jednoczesnego wystąpienia jednego planowanego i jednego nieplanowanego przestoju w zbiorze replik. W przypadku typowego klastra będzie to oznaczało posiadanie co najmniej trzech replik każdej partycji. Jeżeli podczas kroczącego wdrażania lub uaktualnienia Kafki albo bazowego systemu operacyjnego dojdzie np. do awarii przełącznika sieciowego, awarii dysku lub powstania innego nieplanowanego problemu, będziesz miał pewność, że nadal dostępna jest dodatkowa replika. Omówimy to szerzej w rozdziale 7.

log.retention.ms

Najczęściej dla retencji komunikatów w Kafce stosuje się konfigurację czasową. Wartość domyślna jest określona w pliku konfiguracyjnym za pomocą parametru `log.retention.hours` i wynosi 168 godzin, czyli tydzień. Dozwolone są jednak jeszcze dwa inne parametry: `log.retention.minutes` i `log.retention.ms`. Wszystkie trzy kontrolują to samo (ilość czasu, po upływie którego komunikaty mogą zostać usunięte), ale zalecane jest korzystanie z parametru `log.retention.ms`, ponieważ mniejszy rozmiar jednostki będzie miał pierwszeństwo, jeśli zostanie określony więcej niż jeden rodzaj jednostki. Zapewni to, że zawsze używana będzie wartość ustawiona dla `log.retention.ms`. Jeśli określono więcej niż jeden parametr, pierwszeństwo ma mniejszy rozmiar jednostki.



Przechowywanie według czasu oraz czas ostatniej modyfikacji

Przechowywanie według czasu opiera się na sprawdzaniu czasu ostatniej modyfikacji (`mtime`) w każdym pliku segmentu dziennika na dysku. Przy normalnym działaniu klastra jest to czas zamknięcia segmentu dziennika i reprezentuje znacznik czasu ostatniego komunikatu w pliku. Jednak w przypadku używania narzędzi administracyjnych do przenoszenia partycji między brokerami czas ten nie zapewnia dokładności i powoduje wydłużenie retencji dla tych partycji. Więcej informacji na ten temat znajdziesz w rozdziale 12., w którym omówiliśmy przenoszenie partycji.

log.retention.bytes

Inny sposób określania retencji komunikatów opiera się na całkowitej liczbie zachowanych bajtów komunikatów. Tę wartość ustawia się za pomocą parametru `log.retention.bytes` i stosuje się ją do poszczególnych partycji. Oznacza to, że jeśli masz temat z 8 partycjami, a `log.retention.bytes` jest ustawiony na 1 GB, ilość danych zachowanych dla tego tematu będzie wynosić maksymalnie 8 GB. Zwróć uwagę, że retencja dotyczy poszczególnych partycji, a nie tematu. Dlatego jeśli przy korzystaniu z `log.retention.bytes` wzrośnie liczba partycji dla tematu, zwiększy się również retencja. Ustawienie wartości `-1` zapewnia nieskończoną retencję.



Konfigurowanie przechowywania według rozmiaru i czasu

Jeśli określiłeś wartość zarówno dla `log.retention.bytes`, jak i `log.retention.ms` (lub innego parametru przechowywania według czasu), komunikaty mogą zostać usunięte po spełnieniu któregośkolwiek z kryteriów. Jeżeli parametr `log.retention.ms` ma ustawioną np. wartość 86400000 (1 dzień), a `log.retention.bytes` ma ustawioną wartość 1000000000 (1 GB), wówczas komunikaty, które mają mniej niż 1 dzień, mogą zostać usunięte, gdy łączny rozmiar komunikatów w ciągu dnia będzie większy niż 1 GB. I odwrotnie: komunikaty mogą zostać usunięte po 1 dniu, nawet jeśli całkowity rozmiar partycji będzie mniejszy niż 1 GB. Dla uproszczenia zaleca się wybór przechowywania na podstawie rozmiaru lub czasu — a nie obu tych parametrów — by zapobiec niespodziankom i niechcianej utracie danych. Oba parametry można jednak wykorzystać w bardziej zaawansowanych konfiguracjach.

`log.segment.bytes`

Wspomniane wcześniej ustawienia retencji dzienników operują na segmentach dziennika, a nie na poszczególnych komunikatach. Komunikaty produkowane do brokera Kafki są dołączane do bieżącego segmentu dziennika partycji. Gdy segment dziennika osiągnie rozmiar określony przez parametr `log.segment.bytes`, który domyślnie ma wartość 1 GB, dany segment dziennika jest zamykany i otwierany jest nowy. Po zamknięciu segmentu dziennika można rozważyć jego wygaszenie. Mniejszy rozmiar segmentu dziennika oznacza, że pliki muszą być częściej zamykane i alokowane, co zmniejsza ogólną wydajność operacji zapisu na dysku.

Dostosowanie rozmiaru segmentów dziennika może być ważne, jeśli tematy charakteryzują się niskim współczynnikiem produkcji. Jeżeli temat otrzymuje dziennie tylko 100 MB komunikatów, a `log.segment.bytes` ma ustawioną wartość domyślną, wypełnienie jednego segmentu zajmie 10 dni. Ponieważ nie mogą zostać wygaszone, dopóki segment dziennika nie zostanie zamknięty, jeśli parametr `log.retention.ms` ma ustawioną wartość 604800000 (1 tydzień), komunikaty będą w rzeczywistości przechowywane nawet przez 17 dni, zanim wygaśnie zamknięty segment dziennika. Jest to spowodowane tym, że po zamknięciu segmentu dziennika zawierającego komunikaty z 10 dni trzeba przed wygaszeniem przechowywać go przez 7 dni zgodnie z regułami czasowymi (ponieważ segmentu nie można usunąć, dopóki nie zostanie wygaszony jego ostatni komunikat).



Pobieranie przesunięć według znacznika czasu

Rozmiar segmentu dziennika wpływa również na zachowanie pobierania przesunięć na podstawie znacznika czasu. Gdy zażądane zostanie przesunięcie dla partycji dla określonego znacznika czasu, Kafka odszuka plik segmentu dziennika, który był w tym czasie zapisywany. Użyje do tego celu czasu utworzenia oraz czasu ostatniej modyfikacji pliku i wyszuka plik, który został utworzony przed wskazanym znacznikiem czasu i zmodyfikowany po raz ostatni po znaczniku czasu. W odpowiedzi zwróci przesunięcie dla początku tego segmentu dziennika (które będzie również nazwą pliku).

`log.roll.ms`

Innym sposobem kontrolowania momentu zamykania segmentów dziennika jest użycie parametru `log.roll.ms`, określającego czas, po upływie którego segment dziennika powinien zostać zamknięty. Podobnie jak w przypadku parametrów `log.retention.bytes` i `log.retention.ms` parametry

`log.segment.bytes` i `log.roll.ms` nie wykluczają się wzajemnie. Kafka zamknie segment dziennika po osiągnięciu limitu rozmiaru lub po osiągnięciu limitu czasu, w zależności od tego, co nastąpi wcześniej. Parametr `log.roll.ms` nie ma ustawionej wartości domyślnej, co skutkuje zamykaniem segmentów dziennika tylko według rozmiaru.



Wydajność dysku w przypadku korzystania z segmentów opartych na czasie

Jeżeli korzysta się z czasowych limitów dla segmentów dziennika, należy wziąć pod uwagę, jaki wpływ na wydajność dysku będzie miało jednoczesne zamykanie wielu segmentów dziennika. Może się to zdarzyć, gdy będzie istniało wiele partycji, które nigdy nie osiągną limitu rozmiaru dla segmentów dziennika, ponieważ zegar limitu czasu zacznie odliczanie w momencie uruchomienia brokera i dla tych partycji o niewielkim wolumenie limit zawsze będzie egzekwowany w tym samym momencie.

`min.insync.replicas`

Podczas konfigurowania klastra pod kątem trwałości danych ustawienie dla parametru `min.insync.replicas` wartości 2 zapewnia synchronizowanie z producentem co najmniej dwóch replik. Jest to używane w połączeniu z ustawieniem konfiguracji producenta w taki sposób, aby potwierdzane były „wszystkie” żądania. Dzięki temu co najmniej dwie repliki (główna i jedna z pozostałych) potwierdzą, że zapis zakończył się powodzeniem. Może to zapobiec utracie danych w scenariuszach, w których replika główna potwierdzi zapis, a następnie ulegnie awarii, a przywództwo zostanie przeniesione na replikę, w której zapis się nie powiódł. Bez tych ustawień trwałości producent mógłby potraktować komunikat (lub komunikaty) jako wyprodukowany pomyślnie, podczas gdy tak naprawdę zostałyby on porzucony i utracony. Konfigurowanie pod kątem większej trwałości ma jednak efekt uboczny, polegający na utracie wydajności ze względu na dodatkowe obciążenie. Dlatego nie zaleca się zmiany wartości domyślnej 1 tego ustawienia w klastrach o wysokiej przepustowości, które mogą tolerować okazjonalną utratę komunikatów. Więcej informacji na ten temat znajdziesz w rozdziale 7.

`message.max.bytes`

W brokerze Kafki maksymalny rozmiar produkowanego komunikatu można skonfigurować za pomocą parametru `message.max.bytes`, który domyślnie ma wartość 1000000, czyli 1 MB. Producent, który próbuje wysłać komunikat o większym rozmiarze, otrzyma od brokera błąd, a komunikat nie zostanie przyjęty. Podobnie jak w przypadku wszystkich rozmiarów bajtowych określanych w brokerze ta konfiguracja dotyczy rozmiaru skompresowanego komunikatu. Oznacza to, że producenci mogą wysyłać komunikaty, które w stanie nieskompresowanym są znacznie większe, pod warunkiem że skompresują je do rozmiaru skonfigurowanego w parametrze `message.max.bytes`.

Zwiększenie dopuszczalnego rozmiaru komunikatów ma zauważalny wpływ na wydajność. Większe komunikaty będą oznaczały, że wątki brokera zajmujące się przetwarzaniem połączeń sieciowych i żądań będą pracowały dłużej nad każdym żądaniem. Większe komunikaty zwiększają również rozmiar zapisów na dysku, co wpływa na przepustowość we-wy. Inną metodą rozwiązywania problemów z dużymi zapisami na dyskach mogą być rozwiązania pamięci masowej, takie jak magazyny obiektów blob i (lub) warstwowa pamięć masowa, ale nie zostaną omówione w tym rozdziale.



Koordinacja konfiguracji rozmiaru komunikatów

Rozmiar komunikatów skonfigurowany w brokerze Kafki musi być skoordynowany z konfiguracją `fetch.message.max.bytes` w klientach konsumentów. Jeżeli ta wartość będzie mniejsza niż `message.max.bytes`, konsumenci, które napotkają większe komunikaty, nie będą mogły ich pobrać. W takiej sytuacji konsument utknie i nie będzie mógł kontynuować pracy. Ta sama reguła dotyczy konfiguracji `replica.fetch.max.bytes` w brokerach w klastrze.

Wybór sprzętu

Wybór odpowiedniej konfiguracji sprzętowej dla brokera Kafki może być bardziej sztuką niż nauką. Sama Kafka nie ma ścisłych wymagań dotyczących konkretnej konfiguracji sprzętowej i będzie działała bez problemów na większości systemów. Jednak gdy wydajność staje się problemem, istnieje kilka czynników, które mogą przyczynić się do powstawania ogólnych wąskich gardeł wydajności: przepustowość i pojemność dysku, pamięć, sieć i procesor główny. W przypadku bardzo dużego skalowania Kafki mogą ponadto wystąpić ograniczenia dotyczące liczby partycji, które może obsługiwać pojedynczy broker, co będzie spowodowane ilością metadanych wymagających aktualizowania. Po ustaleniu, które rodzaje wydajności są najbardziej kluczowe dla Twojego środowiska, możesz wybrać zoptymalizowaną konfigurację sprzętową, mieszczącą się w wyznaczonym budżecie.

Przepustowość dysku

Na wydajność klientów producentów najbardziej bezpośrednio wpływa przepustowość dysku brokera używanego do przechowywania segmentów dziennika. Produkowane komunikaty Kafki muszą być zatwierdzane w magazynie lokalnym, a większość klientów zanim uzna, że wysłanie zakończyło się pomyślnie, będzie czekać, aż co najmniej jeden broker potwierdzi, że komunikaty zostały zatwierdzone. Oznacza to, że szybszy zapis na dysku będzie skutkował mniejszym opóźnieniem produkcji.

Jeśli chodzi o przepustowość dysku, do wyboru są tradycyjne dyski twarde (HDD) oraz dyski półprzewodnikowe (SSD). Dyski SSD mają znacznie krótsze czasy wyszukiwania i dostępu, a ponadto zapewniają najlepszą wydajność. Z drugiej strony, dyski HDD są bardziej ekonomiczne i oferują większą pojemność na jednostkę. Wydajność dysków HDD można również poprawić, używając w brokerze większej ich liczby, czy to przez posiadanie wielu katalogów danych, czy przez skonfigurowanie dysków w redundantną macierz niezależnych dysków (ang. *Redundant Array of Independent Disks* — RAID). Na przepustowość mają także wpływ inne czynniki, takie jak określona technologia napędu (np. SAS lub SATA) oraz jakość kontrolera dysku. Generalnie obserwacje pokazują, że dyski HDD są zazwyczaj bardziej przydatne w przypadku klastrów o bardzo dużych potrzebach w zakresie przechowywania, ale mniejszych wymaganiach związanych z dostępem, natomiast dyski SSD są lepszymi opcjami, gdy mamy do czynienia z bardzo dużą liczbą połączeń klienckich.

Pojemność dysku

Pojemność to druga strona medalu w dyskusji na temat pamięci masowej. Wymagana ilość miejsca na dysku zależy od liczby komunikatów, które muszą być przechowywane w dowolnym momencie. Jeżeli broker ma otrzymywać codziennie 1 TB ruchu przy 7-dniowej retencji, będzie potrzebował co najmniej 7 TB użytecznej pamięci masowej dla segmentów dziennika. Oprócz dowolnego bufora, który ma być utrzymywany na wypadek wahań ruchu lub rozwoju wraz z upływem czasu, należy uwzględnić też co najmniej 10% narzutu dla pozostałych plików.

Pojemność pamięci masowej jest jednym z czynników, które należy wziąć pod uwagę przy ustalaniu rozmiaru klastra Kafka i określaniu, kiedy będzie rozszerzany. Całkowity ruch przeznaczony dla klastra można zrównoważyć przez zastosowanie wielu partycji przypadających na temat, co pozwoli dodatkowym brokerom zwiększyć dostępną pojemność, jeśli gęstość na pojedynczym brokerze nie będzie wystarczająca. Decyzja o tym, ile miejsca na dysku jest potrzebne, będzie również podyktowana wybraną dla klastra strategią replikacji (co omówiliśmy szczegółowo w rozdziale 7.).

Pamięć

Dla konsumenta Kafka normalnym trybem działania jest odczytywanie od końca partycji, przy czym konsument dogania producenty i tylko nieznacznie pozostaje za nimi w tyle, jeśli w ogóle. W takiej sytuacji komunikaty odczytywane przez konsument najlepiej przechowywać w systemowej pamięci podręcznej stron, dzięki czemu odczyty są szybsze, niż gdyby broker musiał ponownie odczytywać komunikaty z dysku. Dlatego posiadanie większej ilości pamięci, która może być przeznaczona dla systemowej pamięci podręcznej stron, poprawia wydajność klientów konsumentów.

Sama Kafka nie wymaga konfigurowania dużej ilości pamięci sterty dla maszyny wirtualnej Javy (ang. *Java Virtual Machine* — JVM). Nawet broker, który obsługuje 150 000 komunikatów na sekundę i prędkość transmisji danych rzędu 200 MB/s, może działać ze stertą o wielkości 5 GB. Reszta pamięci systemowej zostanie wykorzystana przez systemową pamięć podręczną stron i przyniesie Kafce korzyści w postaci umożliwienia systemowi buforowania używanych segmentów dziennika. Jest to główny powód, dla którego nie zaleca się współlokowania Kafka w systemie z jakąkolwiek inną znaczącą aplikacją, ponieważ musiałaby wtedy współdzielić wykorzystanie pamięci podręcznej stron. Spowodowałoby to spadek wydajności konsumentów Kafka.

Konfiguracja sieciowa

Dostępna przepustowość sieci będzie determinowała maksymalną ilość ruchu, jaki będzie mogła obsłużyć Kafka. W połączeniu z dyskową pamięcią masową może to być decydujący czynnik przy określaniu rozmiaru klastra. Komplikuje to immanentna nierównowaga między użyciem sieci przez ruch przychodzący i wychodzący, którą tworzy zapewnianie przez Kafkę obsługi wielu konsumentów. Producent może zapisywać 1 MB na sekundę dla danego tematu, ale może być dowolna liczba konsumentów, co będzie zwielokrotniać wykorzystanie sieci dla ruchu wychodzącego. Wymagania będą zwiększać również inne operacje, takie jak replikacja klastra (omówiona w rozdziale 7.) i mirroring (omówiony w rozdziale 10.). Jeśli interfejs sieciowy zostanie nasycony, istnieje spore prawdopodobieństwo, że replikacja klastra będzie pozostawała w tyle, przez co klastr może się

znaleźć w stanie podatnym na awarie. Aby sieć nie była głównym czynnikiem decydującym, zaleca się korzystanie z kart sieciowych o przepustowości co najmniej 10 Gb/s. W przypadku starszych maszyn z kartami sieciowymi 1 Gb/s łatwo może dojść do nasycenia interfejsów sieciowych, dlatego nie są one zalecane.

Procesor

Dopóki nie zaczniesz intensywnie skalować Kafki, moc obliczeniowa nie jest tak ważna jak dysk i pamięć, chociaż w pewnym stopniu wpływa na ogólną wydajność brokera. Najlepiej, gdyby klienci kompresowali komunikaty, aby zoptymalizować wykorzystanie sieci i dysku. Broker Kafki musi jednak zdekompresować wszystkie partie komunikatów w celu zweryfikowania sumy kontrolnej poszczególnych komunikatów i przypisania przesunięć. Następnie musi ponownie skompresować partię komunikatów, by zapisać ją na dysku. Stąd pochodzi większość wymagań Kafki dotyczących mocy obliczeniowej. Nie powinno to być jednak głównym czynnikiem przy wyborze sprzętu, chyba że klastry stają się bardzo duże, z setkami węzłów i milionami partycji w pojedynczym klastrze. W takiej sytuacji wybranie wydajniejszego procesora może pomóc w zmniejszeniu rozmiarów klastrów.

Kafka w chmurze

W ostatnich latach powszechne staje się instalowanie Kafki w środowiskach chmury obliczeniowej, takich jak Microsoft Azure, AWS Amazona czy Google Cloud Platform. Dostępnych jest wiele opcji konfigurowania platformy Kafka w chmurze i zarządzania nią za pośrednictwem dostawców takich jak Confluent, a nawet za pośrednictwem Kafki Azure w usłudze HDInsight. W kolejnych punktach przedstawiliśmy jednak kilka prostych porad, jeśli planujesz ręcznie zarządzać własnymi klastrami Kafki. W większości środowisk chmurowych masz do wyboru wiele instancji obliczeniowych, z których każda ma inną kombinację procesora, pamięci, liczby IOPS i dysku. Różnym charakterystykom wydajności Kafki należy nadać priorytety, by móc wybierać odpowiednią konfigurację instancji do użycia.

Microsoft Azure

Na platformie Azure możesz zarządzać dyskami niezależnie od maszyny wirtualnej (ang. *virtual machine* — VM), dlatego podejmowanie decyzji o potrzebach dotyczących pamięci masowej nie musi być powiązane z wybranym typem maszyny wirtualnej. Dobrym punktem dla dokonania wyboru jest ilość danych, które mają być przechowywane, a w następnej kolejności wydajność wymagana od producentów. Jeżeli warunkiem jest zapewnienie bardzo małych opóźnień, wymagane mogą być instancje zoptymalizowane pod kątem operacji we-wy, korzystające z magazynu danych Premium SSD. W przeciwnym razie wystarczające mogą być opcje zarządzanej pamięci masowej, takie jak Azure Managed Disks lub Azure Blob Storage.

Z naszego doświadczenia z Azure wynika, że dla mniejszych klastrów dobrym wyborem są typy instancji Standard D16s v3, które są wystarczająco wydajne w większości przypadków użycia. Wyższe wymagania dotyczące wydajności sprzętu i procesora spełniają instancje D64s v4, które można

skalować do większych klastrów. W celu zapewnienia dostępności zaleca się utworzenie klastra w zestawie dostępności Azure i równomierne rozłożenie partycji po domenach błędów obliczeniowych Azure. Gdy wybierzesz już maszynę wirtualną, możesz przejść do wyboru typu pamięci masowej. Zdecydowanie zaleca się używanie dysków zarządzanych Azure zamiast dysków efemerycznych. Jeżeli w przypadku tych ostatnich maszyna wirtualna zostanie przeniesiona, ryzykuje się utratę wszystkich danych brokera Kafki. Dyski zarządzane HDD są stosunkowo niedrogie, ale Microsoft nie określa dla nich jasno umów SLA dotyczących dostępności. Konfiguracje Premium SSD lub Ultra SSD są znacznie droższe, ale także znacznie szybsze i otrzymują lepsze wsparcie, a Microsoft określa SLA na poziomie 99,99%. Jeśli nie masz aż tak dużych wymagań w kwestii opóźnień, możesz alternatywnie skorzystać z opcji Microsoft Blob Storage.

Amazon Web Services

W przypadku usługi AWS, jeśli warunkiem są bardzo małe opóźnienia, wymagane mogą być instancje zoptymalizowane pod kątem operacji we-wy, posiadające lokalną pamięć masową SSD. W przeciwnym razie wystarczająca może być efemeryczna pamięć masowa, taka jak Amazon Elastic Block Store.

Częstym wyborem w AWS jest typ instancji *m4* lub *r3*. Instancja *m4* zapewnia dłuższe okresy przechowywania, ale przepustowość dysku jest mniejsza, ponieważ znajduje się on w elastycznej blokowej pamięci masowej. Instancja *r3* będzie miała znacznie lepszą przepustowość dzięki lokalnym dyskom SSD, ale dyski te ograniczą ilość przechowywanych danych. Aby uzyskać wszystko co najlepsze z obu światów, konieczne może być przejście na typy instancji *i2* lub *d2*, ale są one znacznie droższe.

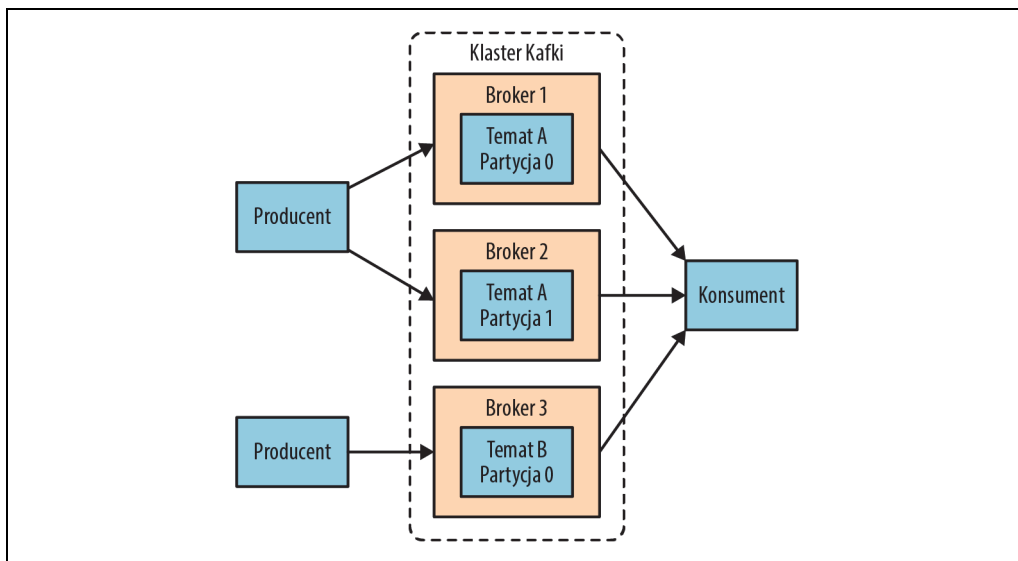
Konfiguracja klastrów Kafki

Pojedynczy broker Kafki dobrze sprawdza się w przypadku lokalnych prac programistycznych lub systemu weryfikacji koncepcji, ale posiadanie wielu brokerów skonfigurowanych jako klaster zapewnia znaczne korzyści, jak pokazaliśmy na rysunku 2.2. Największą korzyścią jest możliwość skalowania obciążenia na wiele serwerów. Drugą istotną zaletą jest używanie replikacji do ochrony przed utratą danych z powodu awarii pojedynczego systemu. Replikacja pozwala również wykonywać prace konserwacyjne w Kafce lub systemach bazowych przy zachowaniu dostępności dla klientów. W tym podrozdziale skoncentrujemy się na krokach konfigurowania podstawowego klastra Kafki. Więcej informacji na temat replikacji danych i trwałości znajdziesz w rozdziale 7.

Ile brokerów?

O odpowiedniej wielkości klastra Kafki decyduje kilka czynników. Zazwyczaj rozmiar klastra będzie powiązany z paroma kluczowymi obszarami, do których zaliczamy:

- pojemność dysku;
- pojemność replik na broker;
- moc procesora;
- przepustowość sieci.



Rysunek 2.2. Prosty klaster Kafki

Pierwszym czynnikiem, który należy wziąć pod uwagę, jest pojemność dysku wymagana do retencji komunikatów oraz ilość pamięci masowej dostępnej w pojedynczym brokerze. Jeżeli klaster ma przechowywać 10 TB danych, a jeden broker może przechowywać 2 TB, to minimalny rozmiar klastra wynosi 5 brokerów. Ponadto zwiększenie współczynnika replikacji spowoduje wzrost wymagań dotyczących pamięci masowej o co najmniej 100%, w zależności od wybranego ustawienia współczynnika replikacji (zob. rozdział 7.). W tym przypadku repliki odnoszą się do liczby różnych brokerów, do których kopiowana jest pojedyncza partycja. Oznacza to, że ten sam klaster skonfigurowany ze współczynnikiem replikacji 2 będzie musiał zawierać co najmniej 10 brokerów.

Innym czynnikiem do rozważenia jest zdolność klastra do obsługi żądań. Może się to przejawiać w pozostałych trzech wspomnianych wcześniej wąskich gardłach.

Jeśli masz klaster Kafki z 10 brokerami, ale z ponad milionem replik (tj. 500 000 partycji przy współczynniku replikacji 2), to w scenariuszu równomiernego rozłożenia każdy broker przyjmuje ok. 100 000 replik. Może to prowadzić do powstawania wąskich gardeł w kolejkach produkowania, konsumowania i kontrolerów. W przeszłości oficjalne zalecenia przewidywały nie więcej niż 4000 replik partycji na broker i nie więcej niż 200 000 replik partycji na klaster. Jednak postępy w zwiększaniu wydajności klastrów pozwoliły Kafce na znacznie większe skalowanie. Obecnie w dobrze skonfigurowanym środowisku zaleca się posiadanie nie więcej niż 14 000 replik partycji na broker oraz nie więcej niż milion *replik* na klaster.

Jak wspomnieliśmy wcześniej w tym rozdziale, w większości przypadków użycia procesor zwykle nie jest głównym wąskim gardłem, ale może się nim stać, jeśli będziemy mieli do czynienia z nadmierną liczbą połączeń klientów i żądań na broker. Monitorowanie ogólnego wykorzystania procesora na podstawie liczby unikatowych klientów i grup konsumentów oraz zapewnianie dodatkowej mocy obliczeniowej w celu zaspokojenia tych potrzeb może pomóc w uzyskiwaniu lepszej

ogólnej wydajności w dużych klastrach. Jeśli chodzi o przepustowość sieci, należy pamiętać o przepustowości interfejsów sieciowych i o tym, czy mogą one obsłużyć ruch klientów, jeżeli istnieje wielu konsumentów danych lub ruch nie jest spójny w całym okresie retencji danych (np. występują gwałtowne wzrosty natężenia ruchu w godzinach szczytu). Jeśli w okresie największego natężenia ruchu interfejs sieciowy pojedynczego brokera jest nasycony w 80% i są dwa konsumenci danych, nie będą one w stanie nadążyć za ruchem szczytowym, chyba że będą dwa brokery. Gdy w klastrze używana jest replikacja, mamy do czynienia z dodatkowym konsumentem danych, który należy wziąć pod uwagę. Aby poradzić sobie z problemami z wydajnością spowodowanymi mniejszą przepustowością dysku lub mniejszą ilością dostępnej pamięci systemowej, można również przeprowadzić skalowanie do większej liczby brokerów w klastrze.

Konfiguracja brokera

Konfiguracja brokera ma tylko dwa wymagania, których spełnienie umożliwia połączenie wielu brokerów Kafki w pojedynczy klaster. Po pierwsze, wszystkie brokery muszą mieć taką samą konfigurację dla parametru `zookeeper.connect`. Określa on ansambl ZooKeepera oraz ścieżkę, w której klaster przechowuje metadane. Drugim wymaganiem jest, aby wszystkie brokery w klastrze miały unikatową wartość parametru `broker.id`. Jeśli do tego samego klastra spróbują dołączyć dwa brokery z tym samym identyfikatorem `broker.id`, drugi broker zarejestruje błąd i nie uruchomi się. Istnieją jeszcze inne parametry konfiguracyjne używane podczas uruchamiania klastra, w szczególności parametry sterujące replikacją, ale omówimy je dalej w książce.

Dostosowywanie systemu operacyjnego

Chociaż w większości dystrybucji Linuksa parametry jądra są od ręki skonfigurowane w taki sposób, że będzie to działać całkiem dobrze z lwią częścią aplikacji, dla brokera Kafki można wprowadzić kilka zmian, które poprawią wydajność. Zmiany te dotyczą głównie pamięci wirtualnej i podsystemów sieciowych, a także konkretnych problemów związanych z punktem montowania dysku wykorzystywanym do przechowywania segmentów dziennika. Wspomniane parametry konfiguruje się zwykle w pliku `/etc/sysctl.conf`, ale szczegółowe informacje dotyczące dostosowywania konfiguracji jądra znajdziesz w dokumentacji swojej dystrybucji Linuksa.

Pamięć wirtualna

Generalnie system pamięci wirtualnej Linuksa automatycznie dostosowuje się do obciążenia roboczego systemu operacyjnego. Możemy wprowadzić pewne poprawki w sposobie obsługi przestrzeni wymiany, a także w brudnych stronach pamięci, by dostosować je do obciążenia roboczego Kafki.

Podobnie jak w przypadku większości aplikacji, szczególnie tych, w których problemem jest przepustowość, najlepiej jest za (niemal) wszelką cenę unikać stosowania mechanizmu wymiany. Wymiana stron pamięci na dysk będzie miała zauważalny wpływ na wszystkie aspekty wydajności w Kafce. Ponadto Kafka intensywnie wykorzystuje systemową pamięć podręczną stron, a jeśli system VM przeprowadza wymianę na dysk, oznacza to, że do pamięci podręcznej stron nie ma przydzielonej wystarczającej ilości pamięci.

Jednym ze sposobów uniknięcia wymiany jest po prostu niekonfigurowanie żadnej przestrzeni wymiany. Posiadanie przestrzeni wymiany nie jest wymagane, ale zapewnia siatkę bezpieczeństwa, jeżeli w systemie wydarzy się coś katastrofalnego. Mechanizm wymiany może zapobiec nagłemu zamykaniu procesów przez system operacyjny z powodu braku pamięci. Dlatego zaleca się ustawienie dla parametru `vm.swappiness` bardzo małej wartości, np. 1. Parametr ten określa procent prawdopodobieństwa, że podsystem maszyny wirtualnej użyje przestrzeni wymiany, zamiast usuwać strony z pamięci podręcznej stron. Lepiej jest zmniejszyć ilość pamięci dostępnej dla pamięci podręcznej stron, niż wykorzystywać jakąkolwiek ilość pamięci wymiany.



Dlaczego nie ustawiać `vm.swappiness` na 0?

Niegdyś zaleceniem dla `vm.swappiness` było zawsze ustawienie wartości 0. Wartość ta oznaczała „nie dokonuj wymiany, chyba że zabrakło pamięci”. Jednak znaczenie tej wartości zmieniło się od wersji 3.5-rc1 jądra Linuksa i ta zmiana została przeniesiona do wielu starszych dystrybucji, w tym do jąder Linuksa Red Hat Enterprise od wersji 2.6.32-303. Spowodowało to zmianę znaczenia wartości 0 na „nigdy nie stosuj wymiany w żadnych okolicznościach”. Dlatego teraz zalecana jest wartość 1.

Korzystne jest również dostosowanie sposobu, w jaki jądro obsługuje brudne strony, które muszą zostać zrzucone na dysk. Aby zapewnić producentom dobre czasy odpowiedzi, Kafka opiera się na wydajności operacji we-wy dysku. Jest to również powód, dla którego segmenty dziennika są zwykle umieszczane na szybkim dysku, niezależnie od tego, czy jest to pojedynczy dysk z krótkim czasem odpowiedzi (np. SSD), czy podsystem dyskowy ze znaczną ilością pamięci NVRAM dla buforowania (np. RAID). W rezultacie można zmniejszyć liczbę tworzonych brudnych stron, zanim działający w tle proces spłukiwania (ang. *flushing*) rozpocznie zapisywanie ich na dysku. W tym celu należy ustawić dla parametru `vm.dirty_background_ratio` wartość niższą niż domyślna 10. Parametr ten określa procent całkowitej ilości pamięci systemowej, a w wielu sytuacjach odpowiednie jest ustawienie dla niego wartości 5. Nie należy jednak ustawiać wartości 0, ponieważ spowodowałoby to ciągle spłukiwanie stron przez jądro, co z kolei eliminowałoby zdolność jądra do buforowania zapisów na dysku dla chwilowych skoków wydajności bazowego urządzenia.

Całkowitą liczbę brudnych stron dozwolonych przed wymuszeniem przez jądro synchronicznych operacji spłukiwania ich na dysk można zwiększyć także przez zmianę wartości `vm.dirty_ratio` (która również określa procent całkowitej pamięci systemowej) na większą niż domyślna 20. Istnieje szeroki zakres możliwych wartości tego ustawienia, ale rozsądna liczba to między 60 a 80. Takie ustawienie wprowadza niewielkie ryzyko, zarówno w odniesieniu do ilości niespłukanej aktywności dysku, jak i możliwości długich przerw dla operacji we-wy w przypadku wymuszonego spłukiwania synchronicznego. Jeśli wybrana zostanie wyższa wartość dla ustawienia `vm.dirty_ratio`, zdecydowanie zaleca się użycie replikacji w klastrze Kafki w celu ochrony przed awariami systemu.

Wybierając wartości dla tych parametrów, warto przejrzeć liczbę brudnych stron w czasie, gdy klastr Kafki działa pod obciążeniem, zarówno w środowisku produkcyjnym, jak i symulowanym. Aktualną liczbę brudnych stron można określić, sprawdzając plik `/proc/vmstat`:

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 21845
nr_writeback 0
nr_writeback_temp 0
```

```
nr_dirty_threshold 32715981
nr_dirty_background_threshold 2726331
#
```

Dla segmentów dziennika i otwartych połączeń Kafka używa deskryptorów plików. Jeśli broker ma wiele partycji, wtedy do śledzenia wszystkich segmentów dziennika potrzebuje co najmniej $(liczba\_partycji) \cdot (rozmiar\_partycji / rozmiar\_segmentu)$, nie licząc połączeń, które wykonuje broker. Dlatego zaleca się aktualizację parametru `vm.max_map_count` na podstawie tych obliczeń i ustawienie dla niego bardzo dużej liczby. W zależności od środowiska korzystna może być zmiana tej wartości na 400 000 lub 600 000. Zaleca się również ustawienie `vm.overcommit_memory` na 0. Ustawienie domyślnej wartości 0 wskazuje, że jądro określa ilość wolnej pamięci na podstawie działania aplikacji. Jeżeli właściwość zostanie ustawiona na wartość inną niż 0, może to spowodować, że system operacyjny pobierze zbyt dużo pamięci, uniemożliwiając Kafce optymalne działanie. Jest to powszechne w przypadku aplikacji o dużej szybkości przetwarzania.

Dysk

Poza wyborem sprzętu dyskowego, a także konfiguracją RAID, jeśli jest używana, istotny wpływ na wydajność może mieć także wybór systemu plików dla tego dysku. Dostępnych jest wiele różnych systemów plików, ale najczęstszymi wyborami dla lokalnych systemów plików są Ext4 (ang. *Fourth Extended File System*) lub XFS (ang. *Extents File System*). XFS stał się domyślnym systemem plików dla wielu dystrybucji Linuksa nie bez powodu: w przypadku większości obciążeń roboczych ma lepszą wydajność niż Ext4, i to przy minimalnym dostrojeniu. Ext4 może się sprawdzać, ale wymaga użycia parametrów dostrajania, które są uważane za mniej bezpieczne. Obejmuje to ustawienie dłuższego interwału zatwierdzania niż domyślna wartość 5, aby wymusić rzadsze spłukiwanie. Ponadto Ext4 wprowadza opóźnioną alokację bloków, co niesie ze sobą większe prawdopodobieństwo utraty danych i uszkodzenia systemu plików w przypadku awarii systemu. System plików XFS również wykorzystuje algorytm opóźnionej alokacji, ale generalnie jest on bezpieczniejszy niż ten używany przez Ext4. XFS ma także lepszą wydajność dla obciążenia roboczego Kafki, nie wymagając dostrajania poza automatycznym dostrajaniem wykonywanym przez system plików. Jest również bardziej wydajny podczas grupowania zapisów na dysku, co razem daje lepszą ogólną przepustowość operacji we-wy.

Niezależnie od tego, który system plików zostanie wybrany dla punktu montowania przechowującego segmenty dziennika, zaleca się ustawienie dla punktu montowania opcji `noatime`. Metadane pliku zawierają trzy znaczniki czasu: czas utworzenia (`ctime`), czas ostatniej modyfikacji (`mtime`) i czas ostatniego uzyskania dostępu (`atime`). Domyślnie `atime` jest aktualizowany za każdym razem, gdy plik jest odczytywany. Generuje to dużą liczbę zapisów na dysku. Atrybut `atime` jest generalnie uważany za mało przydatny, chyba że aplikacja musi wiedzieć, czy uzyskiwano dostęp do pliku od czasu jego ostatniej modyfikacji (w takim przypadku można użyć opcji `relatime`). Kafka w ogóle nie używa `atime`, więc wyłączenie go jest bezpieczne. Ustawienie `noatime` dla punktu montowania zapobiegnie wykonywaniu aktualizacji tych znaczników czasu, ale nie wpłynie na prawidłową obsługę atrybutów `ctime` i `mtime`. W przypadku większych zapisów na dysku wydajność Kafki może również poprawić użycie opcji `largeio`.

Konfiguracja sieciowa

Dostosowywanie domyślnej konfiguracji stosu sieciowego Linuksa jest powszechne dla każdej aplikacji, która generuje duży ruch sieciowy, ponieważ jądro nie jest domyślnie dostrójone dla dużych, szybkich transferów danych. Tak naprawdę zalecane zmiany dla Kafki są takie same, jak te sugerowane dla większości serwerów WWW oraz innych aplikacji sieciowych. Pierwszą regulacją jest zmiana domyślnej i maksymalnej ilości pamięci alokowanej dla buforów wysyłania i odbierania dla każdego gniazda. Zwiększy to znacznie wydajność przy dużych transferach. Odpowiednie parametry domyślnego rozmiaru bufora wysyłania i odbierania dla każdego gniazda to `net.core.wmem_default` i `net.core.rmem_default`, a rozsądnym ustawieniem dla tych parametrów jest 131072, czyli 128 kB. Parametry maksymalnych rozmiarów bufora wysyłania i odbierania to `net.core.wmem_max` i `net.core.rmem_max`, a rozsądnym ustawieniem jest 2097152, czyli 2 MB. Należy pamiętać, że maksymalny rozmiar nie oznacza, że każde gniazdo będzie miało przydzieloną tak dużą ilość miejsca w buforze; pozwala tylko na tyle, ile jest potrzebne.

Oprócz skonfigurowania ustawień gniazd osobno należy ustawić rozmiary buforów wysyłania i odbierania dla gniazd TCP za pomocą parametrów `net.ipv4.tcp_wmem` i `net.ipv4.tcp_rmem`. Są one ustawiane przy użyciu trzech oddzielonych spacjami liczb całkowitych, które określają odpowiednio rozmiar minimalny, domyślny i maksymalny. Rozmiar maksymalny nie może być większy niż wartości określone dla wszystkich gniazd przy użyciu parametrów `net.core.wmem_max` i `net.core.rmem_max`. Przykładowe ustawienie dla każdego z tych parametrów to 4096 65536 2048000, co oznacza 4 kB (bufora minimalnego), 64 kB (domyślnego) i 2 MB (maksymalnego). Opierając się na rzeczywistym obciążeniu roboczym brokerów Kafki, możesz zwiększyć rozmiary maksymalne, aby umożliwić buforowanie większej liczby połączeń sieciowych.

Przydatne jest ustawienie również kilku innych parametrów dostrajania sieci. Włączenie skalowania okna TCP przez ustawienie dla parametru `net.ipv4.tcp_window_scaling` wartości 1 umożliwi klientom wydajniejsze przesyłanie danych i pozwoli na buforowanie tych danych po stronie brokera. Zwiększenie wartości `net.ipv4.tcp_max_syn_backlog` powyżej domyślnej liczby 1024 pozwoli na akceptowanie większej liczby jednoczesnych połączeń. Zwiększenie wartości `net.core.netdev_max_backlog` powyżej domyślnej liczby 1000 może być pomocne w nagłych wzrostach natężenia ruchu sieciowego, szczególnie przy korzystaniu z wielogigabitowych szybkości połączeń sieciowych, umożliwiając umieszczanie większej liczby pakietów w kolejce do przetworzenia przez jądro.

Zagadnienia produkcyjne

Gdy będziesz gotowy, aby przenieść swoją konfigurację Kafki ze środowiska testowego do operacji produkcyjnych, będziesz musiał przemyśleć jeszcze kilka kwestii, które pomogą w skonfigurowaniu niezawodnej usługi wymiany komunikatów.

Opcje mechanizmu odzyskiwania pamięci

Dostrajanie opcji mechanizmu odzyskiwania pamięci (ang. *garbage collector*) Javy dla aplikacji zawsze było sztuką wymagającą szczegółowych informacji o sposobie wykorzystywania pamięci przez daną aplikację oraz znacznej ilości obserwacji. Wymagało też zastosowania techniki prób i błędów. Na

szczęście zmieniło się to wraz z wydaniem Javy 7 i wprowadzeniem mechanizmu odzyskiwania pamięci G1GC (ang. *Garbage-First Garbage Collector*). Chociaż G1GC był początkowo uważany za niestabilny, znaczną poprawę odnotowano w JDK8 i JDK11. Obecnie zaleca się, by Kafka używała G1GC jako domyślnego mechanizmu odzyskiwania pamięci. G1GC został zaprojektowany w taki sposób, by automatycznie dostosowywać się do różnych obciążeń roboczych i zapewniać spójne okresy wstrzymywania działania mechanizmu odzyskiwania pamięci przez cały okres życia aplikacji. Z łatwością radzi sobie również z dużymi rozmiarami sterty, dzieląc stertę na mniejsze strefy i nie odzyskując całej sterty przy każdym wstrzymaniu.

G1GC robi to wszystko podczas normalnej pracy przy minimalnej ilości konfiguracji. Dostępne są dwie opcje konfiguracji G1GC służące do dostosowania jego wydajności:

MaxGCPauseMillis

Ta opcja określa preferowany czas wstrzymania dla każdego cyklu mechanizmu odzyskiwania pamięci. Nie jest to stałe maksimum — G1GC może przedłużyć ten czas, jeśli zajdzie potrzeba. Domyślna wartość dla tej opcji wynosi 200 ms. Oznacza to, że G1GC spróbuje zaplanować częstotliwość cykli mechanizmu odzyskiwania pamięci, a także liczbę stref odzyskiwanych w każdym cyklu, aby każdy cykl zajmował ok. 200 ms.

InitiatingHeapOccupancyPercent

Ta opcja określa procent całkowitej sterty, który może być używany, zanim G1GC rozpocznie cykl odzyskiwania pamięci. Wartość domyślna wynosi 45. Oznacza to, że G1GC nie rozpocznie cyklu odzyskiwania pamięci, dopóki w użyciu nie będzie 45% sterty. Obejmuje to łączne wykorzystanie zarówno nowej strefy (Edenu), jak i starej.

Broker Kafki jest dość wydajny z uwagi na sposób, w jaki wykorzystuje pamięć sterty i tworzy obiekty możliwe do usunięcia przez odzyskiwanie pamięci, dlatego można ustawić dla tych opcji niższe wartości. Uznaje się, że przedstawione w tym punkcie opcje dostrajania mechanizmu odzyskiwania pamięci są odpowiednie dla serwera z 64 GB pamięci, na którym działa Kafka w stercie o rozmiarze 5 GB. Dla parametru `MaxGCPauseMillis` tego brokera można skonfigurować wartość 20 ms. Wartość `InitiatingHeapOccupancyPercent` ustawia się na 35, co powoduje, że cykl mechanizmu odzyskiwania pamięci jest uruchamiany nieco wcześniej niż przy wartości domyślnej.

Kafka została pierwotnie wydana, zanim kolektor G1GC stał się dostępny i został uznany za stabilny. Dlatego w Kafce domyślnie dla mechanizmu odzyskiwania pamięci używany jest współbieżny algorytm *mark and sweep*, aby zapewnić zgodność ze wszystkimi maszynami wirtualnymi Javy. Nową najlepszą praktyką jest używanie do wszystkiego mechanizmu G1GC dla Javy w wersji co najmniej 1.8. Zmiana jest łatwa do wprowadzenia dzięki zmiennym środowiskowym. Używając polecenia start z wcześniejszej części rozdziału, możesz zmodyfikować to ustawienie w następujący sposób:

```
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -Xmx6g -Xms6g
-XX:MetaspaceSize=96m -XX:+UseG1GC -XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
-XX:+ExplicitGCInvokesConcurrent"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/
server.properties
#
```

Układ centrum danych

W przypadku środowisk testowych i programistycznych fizyczna lokalizacja brokerów Kafki w centrum danych nie jest dużym problemem, ponieważ częściowa lub całkowita niedostępność klastra przez krótkie przedziały czasu nie ma tak poważnego wpływu. Jednak w przypadku obsługi ruchu produkcyjnego przestój zwykle oznacza utratę pieniędzy z powodu przerwy w świadczeniu usług dla użytkowników lub utratę danych telemetrycznych dotyczących działań podejmowanych przez użytkowników. Wtedy kluczowe staje się skonfigurowanie replikacji w klastrze Kafki (zob. rozdział 7.), co jest również ważne, gdy należy brać pod uwagę fizyczną lokalizację brokerów w szafach rackowych w centrum danych. Preferowane jest środowisko centrum danych z koncepcją stref awarii. Jeśli nie zostanie to rozwiązane przed wdrożeniem Kafki, potrzebna może być kosztowna konserwacja w celu przeniesienia serwerów.

Kafka może przypisywać nowe partycje do brokerów w sposób uwzględniający rozmieszczenie w szafach rackowych, zapewniając, że repliki dla pojedynczej partycji nie będą współdzielić jednego racku. W tym celu należy odpowiednio ustawić konfigurację `broker.rack` dla każdego brokera. Ta konfiguracja może być ustawiona z uwzględnieniem domen awarii w środowiskach chmurowych lub na podstawie innych czynników. Dotyczy to jednak tylko nowo tworzonych partycji. Klaster Kafki nie monitoruje partycji, które przestają zapewniać informacje o lokalizacji w szafie rackowej (np. w wyniku ponownego przypisania partycji), ani nie koryguje automatycznie tej sytuacji. Aby utrzymywać informacje o lokalizacji rackowej, zaleca się używanie narzędzi takich jak Cruise Control (zob. dodatek B), które pomagają zapewnić odpowiednie wyważenie klastra. Właściwe skonfigurowanie tych ustawień pomoże zapewnić ciągłość danych o fizycznej lokalizacji w miarę upływu czasu.

Generalnie najlepszą praktyką jest instalowanie każdego brokera Kafki w klastrze w innym racku lub przynajmniej unikanie współdzielenia pojedynczych punktów awarii dla usług infrastrukturalnych, takich jak zasilanie i sieć. Zwykle oznacza to co najmniej wdrożenie serwerów, które będą uruchamiać brokery z podwójnymi połączeniami do zasilania (do dwóch różnych obwodów) i podwójnymi przełącznikami sieciowymi (z połączonym interfejsem na samych serwerach w celu bezproblemowego przełączania awaryjnego). Nawet przy podwójnych połączeniach korzystne jest posiadanie brokerów w całkowicie oddzielonych rackach. Od czasu do czasu konieczne może być wykonanie fizycznej konserwacji danego racku lub szafy rackowej, wymagające ich odłączenia (np. w przypadku przenoszenia serwerów lub zmiany podłączeń zasilania).

Współlokowanie aplikacji na ZooKeeperze

Kafka wykorzystuje serwer ZooKeeper do przechowywania metadanych o brokerach, tematach i partycjach. Operacje zapisywania w ZooKeeperze są wykonywane tylko w przypadku zmian członkostwa grup konsumentów lub zmian w samym klastrze Kafki. Ten ruch jest na ogół minimalny i nie uzasadnia użycia dedykowanego ansamblu ZooKeepera dla pojedynczego klastra Kafki. Tak naprawdę w wielu wdrożeniach korzysta się z pojedynczego ansamblu ZooKeepera dla wielu klastrów Kafki (przy zastosowaniu ścieżki `chroot` ZooKeepera dla każdego klastra, jak opisaliśmy wcześniej w tym rozdziale).



Konsumenty Kafki, narzędzia, ZooKeeper i Ty

W miarę upływu czasu zależność od ZooKeepera maleje. W wersji 2.8.0 Kafki wprowadzono podgląd funkcjonalności platformy całkowicie pozbawionej ZooKeepera, ale wciąż nie jest ona gotowa do zastosowania w środowisku produkcyjnym. Nadal można jednak zaobserwować to zmniejszone poleganie na ZooKeeperze w kolejnych wersjach Kafki. W starszych wersjach tej platformy konsumenci (tak jak brokerzy) wykorzystywały np. ZooKeeper do bezpośredniego przechowywania informacji o składzie grupy konsumentów i konsumowanych przez nią tematach oraz do okresowego zatwierdzania przesunięć dla każdej konsumowanej partycji (aby umożliwić przełączanie awaryjne między konsumentami w grupie). W wersji 0.9.0.0 zmieniono interfejs konsumenta, umożliwiając zarządzanie nim bezpośrednio za pomocą brokerów Kafki. W każdym kolejnym wydaniu 2.x Kafki widzimy podejmowanie pewnych kroków w celu usunięcia ZooKeepera z innych wymaganych ścieżek Kafki. Narzędzia administracyjne łączą się teraz bezpośrednio z klastrem i zarzucono konieczność bezpośredniego łączenia się z ZooKeeperem w przypadku takich operacji jak tworzenie tematów, dynamiczne zmiany konfiguracji itp. W związku z tym wiele narzędzi wiersza poleceń, które wcześniej używały opcji `--zookeeper`, zostało zaktualizowanych do korzystania z opcji `--bootstrap-server`. Opcje `--zookeeper` mogą być nadal używane, ale są przestarzałe i w przyszłości zostaną usunięte, gdy Kafka nie będzie już musiała łączyć się z ZooKeeperem w celu tworzenia tematów, zarządzania nimi lub ich konsumowania.

Jednak w niektórych konfiguracjach istnieje problem z konsumentami i ZooKeeperem. Chociaż używanie serwera ZooKeeper do takich celów jest przestarzałe, konsumenci mają konfigurowalną opcję używania ZooKeepera lub Kafki do zatwierdzania przesunięć, a ponadto mogą konfigurować interwały między zatwierdzeniami. Jeśli konsument używa do przesunięć ZooKeepera, to każdy konsument w każdym interwale dla wszystkich konsumowanych partycji będzie wykonywał zapis w ZooKeeperze. Rozsądny interwał dla zatwierdzeń przesunięć wynosi minutę, ponieważ właśnie tak długo grupa konsumentów będzie odczytywała zduplikowane komunikaty w przypadku awarii któregoś konsumenta. Zatwierdzenia te mogą stanowić znaczną część ruchu ZooKeepera, zwłaszcza w klastrze z wieloma konsumentami, i należy je wziąć pod uwagę. Jeżeli ansambl ZooKeepera nie będzie w stanie obsłużyć ruchu, konieczne może być użycie dłuższego interwału zatwierdzania. Zaleca się jednak, aby konsumenci korzystające z najnowszych bibliotek Kafki używały właśnie Kafki do zatwierdzania przesunięć, usuwając zależność od ZooKeepera.

Poza kwestią używania pojedynczego ansamblu dla wielu klastrów Kafki, nie zaleca się współdzielenia tego ansamblu z innymi aplikacjami, jeśli można tego uniknąć. Kafka jest wrażliwa na opóźnienia i limity czasu ZooKeepera, a przerwa w komunikacji z ansamblem będzie powodowała nieprzewidywalne zachowanie brokerów. Może to łatwo doprowadzić do jednoczesnego wyłączenia wielu brokerów, jeżeli stracą one połączenia z ZooKeeperem, co z kolei spowoduje wyłączenie partycji. Obciąża to również kontroler klastra, który może wykazywać subtelne błędy długo po przywróceniu sprawności po przestoju, np. podczas próby wykonania kontrolowanego zamknięcia brokera. Osobny ansambl należy przydzielić wszelkim aplikacjom, które mogą obciążać ansambl ZooKeepera z powodu intensywnego użytkowania lub przeprowadzania nieodpowiednich operacji.

Podsumowanie

W tym rozdziale dowiedziałeś się, jak uruchomić platformę Apache Kafka. Omówiliśmy również wybór odpowiedniego sprzętu dla brokerów oraz konkretne problemy związane z konfiguracją w środowisku produkcyjnym. Skoro masz już klaster Kafki, omówimy podstawy aplikacji klienckich Kafki. Kolejne dwa rozdziały poświęcimy tworzeniu klientów zarówno do produkowania komunikatów do Kafki (rozdział 3.), jak i ponownego konsumowania tych komunikatów (rozdział 4.).

Producenci Kafki

— zapisywanie komunikatów w Kafce

Niezależnie od tego, czy używasz Kafki jako kolejki, magistrali komunikatów, czy platformy do przechowywania danych, zawsze będzie to polegało na utworzeniu producenta zapisującego dane w Kafce, konsumenta odczytującego dane z Kafki lub aplikacji pełniącej obie te funkcje.

W systemie przetwarzania transakcji kartą kredytową będzie istniała np. aplikacja kliencka, powiedzmy: sklep internetowy, odpowiedzialna za wysyłanie do Kafki każdej transakcji natychmiast po dokonaniu płatności. Kolejna aplikacja będzie odpowiedzialna za natychmiastowe sprawdzenie tych transakcji względem silnika reguł i określenie, czy transakcja zostanie zatwierdzona lub odrzucona. Odpowiedź zatwierdzenia lub odrzucenia transakcji może być następnie zapisywana w Kafce i propagowana z powrotem do sklepu internetowego, w którym zainicjowano transakcję. Trzecia aplikacja może odczytywać z Kafki zarówno transakcje, jak i status zatwierdzenia oraz zapisywać je w bazie danych, gdzie analitycy będą mogli później przejrzeć podejmowane decyzje i być może ulepszyć silnik reguł.

Platforma Apache Kafka jest dostarczana z wbudowanymi interfejsami API klienta, których deweloperzy mogą używać podczas tworzenia aplikacji współpracujących z Kafką.

W tym rozdziale dowiesz się, jak korzystać z producenta Kafki. Zaczniemy od przeglądu jego konstrukcji i komponentów. Pokażemy, jak tworzyć obiekty `KafkaProducer` i `ProducerRecord`, wysyłać rekordy do Kafki oraz obsługiwać błędy, które Kafka może zwrócić. Następnie przyjrzymy się najważniejszym opcjom konfiguracyjnym używanym do kontrolowania zachowania producenta. Zakończymy omówieniem różnych metod partycjonowania i serializatorów oraz pisania własnych serializatorów i partycjonerów.

W rozdziale 4. przyjrzymy się klientowi konsumenckiemu Kafki i odczytywaniu danych z tej platformy.



Klienty zewnętrzne

Oprócz wbudowanych klientów Kafka posiada binarny protokół połączeniowy (ang. *wire protocol*). Oznacza to, że aplikacje mogą odczytywać komunikaty z Kafki lub zapisywać komunikaty w Kafce, wysyłając po prostu odpowiednie sekwencje bajtów do portu sieciowego platformy. Dostępnych jest wiele klientów implementujących protokół podłączania Kafki w różnych językach programowania i zapewniających proste sposoby korzystania z Kafki nie tylko w aplikacjach Javy, ale także C++, Pythona, Go itd. Klienci te nie są częścią projektu Apache Kafka, ale lista klientów napisanych w językach innych niż Java jest utrzymywana na stronie wiki projektu (<https://oreil.ly/9SbJr>). Omówienie protokołu podłączania i zewnętrznych klientów wykracza poza zakres tego rozdziału.

Omówienie producenta

Aplikacja może potrzebować zapisywania komunikatów w Kafce z wielu powodów, takich jak: rejestrowanie aktywności użytkownika do celów audytu lub analizy, rejestrowanie wskaźników, zapisywanie komunikatów dziennika, rejestrowanie informacji z urządzeń typu smart, asynchroniczna komunikacja z innymi aplikacjami, buforowanie informacji przed zapisaniem w bazie danych itd.

Te różnorodne przypadki użycia mają również odmienne wymagania. Czy każdy komunikat jest istotny, czy też możemy tolerować utratę komunikatów? Czy przeszkadza nam przypadkowe duplikowanie komunikatów? Czy są jakieś ściśle wymagania dotyczące opóźnień lub przepustowości, które musimy obsługiwać?

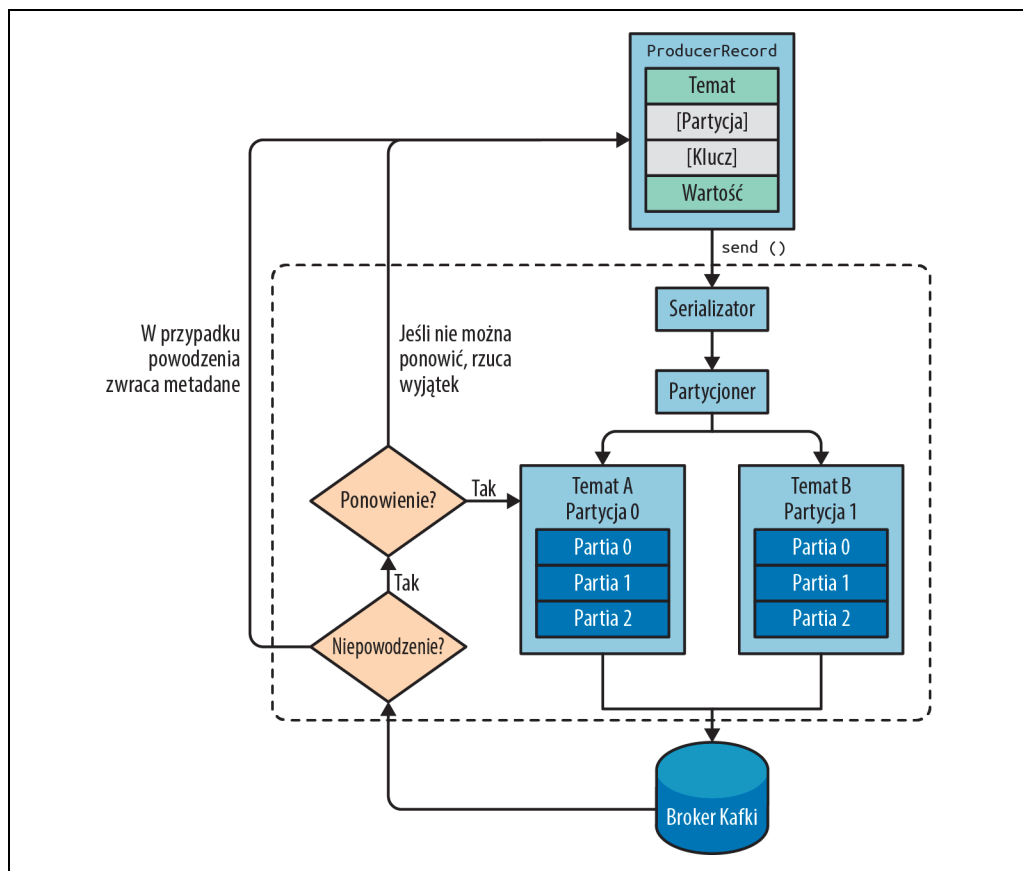
W przedstawionym wcześniej przykładzie przetwarzania transakcji kartą kredytową widzimy, że bardzo ważne jest, by nigdy nie utracić żadnego komunikatu ani nie powielić komunikatów. Opóźnienia powinny być małe, ale opóźnienia do 500 milisekund mogą być tolerowane, a przepływność powinna być bardzo wysoka — oczekujemy przetwarzania do miliona komunikatów na sekundę.

Innym przypadkiem użycia może być zapisywanie informacji o kliknięciach na stronie internetowej. W takiej sytuacji można tolerować pewną utratę komunikatów lub niewielką duplikację; opóźnienie może być duże, o ile nie będzie miało wpływu na doświadczenie użytkownika. Innymi słowy: nie przeszkadza nam, że komunikat dotrze do Kafki po kilku sekundach, pod warunkiem że następna strona będzie ładowała się natychmiast po kliknięciu linku przez użytkownika. Przepustowość będzie zależeć od oczekiwanego przez nas poziomu aktywności na stronie internetowej.

Te różne wymagania będą miały wpływ na sposób wykorzystywania interfejsu API producenta do zapisywania komunikatów w Kafce oraz na używaną konfigurację.

Chociaż API producenta jest bardzo proste, gdy wysyłamy dane, za kulisami trochę się dzieje. Na rysunku 3.1 pokazaliśmy główne kroki związane z wysyłaniem danych do Kafki.

Produkowanie komunikatów do Kafki zaczynamy od utworzenia obiektu `ProducerRecord` — musi on zawierać temat, do którego chcemy wysłać rekord, oraz wartość. Opcjonalnie możemy określić również klucz, partycję, znacznik czasu i (lub) zbiór nagłówków. Gdy wyślesmy obiekt `ProducerRecord`, producent w pierwszej kolejności serializuje obiekty klucza i wartości do postaci tablic bajtów, aby można je było przesłać przez sieć.



Rysunek 3.1. Ogólny przegląd komponentów producenta Kafki

Jeśli nie określiliśmy bezpośrednio partycji, dane są następnie wysyłane do partycjonera, który wybiera dla nas partycję, zwykle na podstawie klucza z obiektu `ProducerRecord`. Po wybraniu partycji producent otrzymuje informacje o temacie i partycji, do których trafi rekord. Dodaje wtedy rekord do partii rekordów, które również zostaną wysłane do tego samego tematu i tej samej partycji. Za wysłanie tych partii rekordów do odpowiednich brokerów Kafki odpowiada osobny wątek.

Gdy broker otrzymuje komunikaty, odsyła odpowiedź. Jeżeli komunikaty zostaną pomyślnie zapisane w Kafce, zwraca obiekt `RecordMetadata` z tematem, partycją i przesunięciem rekordu w obrębie danej partycji. Jeśli broker nie zapisze komunikatów, zwraca błąd. Gdy producent otrzyma błąd, może spróbować wysłać określony komunikat jeszcze kilka razy, zanim zrezygnuje i zwróci błąd.

Konstruowanie producenta Kafki

Pierwszym krokiem w zapisywaniu komunikatów w Kafce jest utworzenie obiektu producenta z właściwościami, które chcemy przekazać producentowi. Producent Kafki ma trzy obowiązkowe właściwości:

`bootstrap.servers`

Lista par `host:port` brokerów, których producent będzie używał do nawiązania początkowego połączenia z klastrem Kafka. Ta lista nie musi zawierać wszystkich brokerów, ponieważ producent uzyska więcej informacji po nawiązaniu początkowego połączenia. Zaleca się jednak uwzględnienie co najmniej dwóch, by w przypadku awarii jednego brokera producent nadal mógł połączyć się z klastrem.

`key.serializer`

Nazwa klasy, która będzie używana do serializacji kluczy rekordów produkowanych do Kafka. Brokery Kafka jako kluczy i wartości komunikatów oczekują tablic bajtów. Interfejs producenta umożliwia jednak przesyłanie kluczy i wartości w postaci dowolnych obiektów Javy dzięki użyciu typów sparametryzowanych. Zwiększa to czytelność kodu, ale oznacza również, że producent musi wiedzieć, jak przekonwertować te obiekty na tablice bajtów. Jako wartość właściwości `key.serializer` należy ustawić nazwę klasy, która implementuje interfejs `org.apache.↪kafka.common.serialization.Serializer`. Producent będzie używał tej klasy do serializowania obiektów klucza do postaci tablicy bajtów. Pakiet klienta Kafka zawiera m.in.: `ByteArray↪Serializer` (który niewiele robi), `StringSerializer` oraz `IntegerSerializer`, dlatego jeżeli używasz standardowych typów, nie musisz implementować własnych serializatorów. Ustawienie wartości `key.serializer` jest wymagane, nawet jeśli zamierzasz wysyłać tylko wartości, możesz jednak użyć dla klucza typu `Void` oraz serializatora `VoidSerializer`.

`value.serializer`

Nazwa klasy, która będzie używana do serializacji wartości rekordów produkowanych do Kafka. Podobnie jak ustawiasz dla właściwości `key.serializer` nazwę klasy, która będzie serializowała obiekt klucza komunikatu do postaci tablicy bajtów, dla `value.serializer` ustawiasz klasę, która będzie serializowała obiekt wartości komunikatu.

W poniższym fragmencie kodu pokazaliśmy, jak utworzyć nowy producent, ustawiając tylko obowiązkowe parametry, a dla wszystkich innych używając wartości domyślnych:

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

- ❶ Zaczynamy od obiektu `Properties`.
- ❷ Ponieważ dla klucza i wartości komunikatów planujemy używać łańcuchów znaków, korzystamy z wbudowanego serializatora `StringSerializer`.
- ❸ Tutaj tworzymy nowy producent, ustawiając odpowiednie typy kluczy i wartości oraz przekazując obiekt `Properties`.

Przy tak prostym interfejsie widać wyraźnie, że większość zachowań producenta kontroluje się przez ustawianie odpowiednich właściwości konfiguracyjnych. Opis wszystkich opcji konfiguracyjnych

znajdziesz w dokumentacji platformy Apache Kafka (<http://bit.ly/2sMu1c8>), a najważniejsze omówimy dalej w tym rozdziale.

Gdy utworzymy już instancję producenta, możemy zacząć wysyłać komunikaty. Istnieją trzy podstawowe metody wysyłania komunikatów:

Odpal i zapomnij

Wysyłamy komunikat do serwera i nie przejmujemy się, czy dotrze pomyślnie. W większości przypadków komunikat dociera, ponieważ Kafka oferuje wysoką dostępność, a producent automatycznie ponawia próbę wysłania komunikatu. Jednak w przypadku błędów nieodwracalnych lub przekroczenia limitu czasu komunikaty zostają utracone, a aplikacja nie otrzymuje żadnych informacji na ten temat ani rzuconych wyjątków.

Wysyłanie synchroniczne

Z technicznego punktu widzenia producent Kafki jest zawsze asynchroniczny — wysyłamy komunikat, a metoda `send()` zwraca obiekt `Future`. Używamy jednak metody `get()`, aby poczekać na obiekt `Future` i przed wysłaniem następnego rekordu sprawdzić, czy wykonywanie metody `send()` zakończyło się powodzeniem.

Wysyłanie asynchroniczne

Wywołujemy metodę `send()` z funkcją wywołania zwrotnego, która jest uruchamiana po otrzymaniu odpowiedzi od brokera Kafki.

W poniższych przykładach pokażemy, jak wysyłać komunikaty przy użyciu tych metod i radzić sobie z różnymi typami błędów, które mogą wystąpić.

Chociaż wszystkie przykłady przedstawione w tym rozdziale są jednowątkowe, obiekt producenta może być używany do wysyłania komunikatów przez wiele wątków.

Wysyłanie komunikatów do Kafki

Komunikat najprościej wysłać w taki sposób:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Produkty precyzyjne", "Francja"); ❶  
try {  
    producer.send(record); ❷  
} catch (Exception e) {  
    e.printStackTrace(); ❸  
}
```

❶ Producent przyjmuje obiekty `ProducerRecord`, zaczynamy więc od utworzenia takiego obiektu. `ProducerRecord` ma wiele konstruktorów, które omówimy później. Tutaj używamy konstruktora wymagającego nazwy tematu (będącego zawsze łańcuchem znaków), do którego wysyłamy dane, oraz klucza i wartości wysyłanych do Kafki, które w tym przypadku również są łańcuchami znaków. Typy klucza i wartości muszą być zgodne z naszymi obiektami `key serializer` oraz `value serializer`.

❷ Do wysłania obiektu `ProducerRecord` używamy metody `send()` obiektu producenta. Jak widziałeś na diagramie architektury producenta na rysunku 3.1, komunikat zostaje umieszczony

w buforze i wysłany do brokera w osobnym wątku. Metoda `send()` zwraca obiekt `Future Javy` (<http://bit.ly/2rG7Cg6>) z obiektem `RecordMetadata`, ponieważ jednak wartość zwracaną po prostu ignorujemy, nie mamy możliwości sprawdzenia, czy komunikat został wysłany pomyślnie. Ta metoda wysyłania komunikatów może być używana, gdy dopuszczalne jest porzucanie komunikatów po cichu. W aplikacjach produkcyjnych zazwyczaj nie ma to zastosowania.

- ❸ Chociaż ignorujemy błędy, które mogą wystąpić podczas wysyłania komunikatów do brokerów Kafki lub w samych brokerach, nadal możemy otrzymać wyjątek, jeśli producent napotka błąd przed wysłaniem komunikatu do Kafki. Tym wyjątkiem może być np.: `SerializationException`, gdy nie powiedzie się serializacja komunikatu, `BufferExhaustedException` lub `TimeoutException`, jeżeli bufor będzie pełny, lub `InterruptedException`, kiedy zakłócone zostanie działanie wątku wysyłającego.

Synchroniczne wysyłanie komunikatów

Synchroniczne wysyłanie komunikatu jest proste, ale producent i tak może przechwytywać wyjątki, gdy Kafka na żądanie produkcji odpowie błędem lub wyczerpie się limit kolejnych prób ponownego wysłania komunikatu. W tym obszarze głównym kompromisem jest wydajność. W zależności od obciążenia klastra Kafki brokery mogą odpowiadać na żądania w czasie od 2 ms do kilku sekund. Jeżeli wysyłasz komunikaty synchronicznie, wątek wysyłający poświęci ten czas na całkowicie bezczynne czekanie, nie wysyłając nawet kolejnych komunikatów. Prowadzi to do znaczącego obniżenia wydajności, przez co wysyłanie synchroniczne zwykle nie jest używane w aplikacjach produkcyjnych (ale jest bardzo powszechne w przykładach kodu).

Najprostszy sposób synchronicznego wysłania komunikatu wygląda następująco:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Produkty precyzyjne", "Francja");  
try {  
    producer.send(record).get(); ❶  
} catch (Exception e) {  
    e.printStackTrace(); ❷  
}
```

- ❶ Używamy tutaj metody `Future.get()`, aby poczekać na odpowiedź od Kafki. Ta metoda rzuci wyjątek, jeśli rekord nie zostanie pomyślnie wysłany do Kafki. Jeżeli nie będzie błędów, otrzymamy obiekt `RecordMetadata`, którego możemy użyć do pobrania przesunięcia, w którym został zapisany dany komunikat, oraz innych metadanych.
- ❷ Jeżeli wystąpią jakieś błędy przed wysłaniem lub w trakcie wysyłania rekordu do Kafki, napotkamy wyjątek. W takim przypadku wypisujemy po prostu każdy wyjątek, na który natrafiliśmy.

`KafkaProducer` ma dwa rodzaje błędów. Błędy **odwracalne** (ang. *retriable*) można usunąć przez ponowne wysłanie komunikatu. Usunąć można np. błąd połączenia, ponieważ połączenie może zostać ponownie nawiązane. Błędu „braku lidera dla partycji” można się pozbyć po wybraniu nowego lidera dla partycji i odświeżeniu metadanych klienta. `KafkaProducer` można skonfigurować w taki sposób, aby automatycznie ponawiał próby usuwania tych błędów, by kod aplikacji otrzymywał nieodwracalne wyjątki tylko wtedy, gdy wyczerpana zostanie liczba prób ponawiania, a błąd

nie zostanie wyeliminowany. Niektórych błędów nie można usunąć przez próbę ponawiania — należy do nich np. błąd „zbyt duży rozmiar komunikatu”. W takich przypadkach KafkaProducer nie podejmuje ponownej próby i natychmiast zwraca wyjątek.

Asynchroniczne wysyłanie komunikatów

Założmy, że czas obiegu pakietu w obie strony między naszą aplikacją a klastrem Kafki wynosi 10 ms. Jeżeli po wysłaniu każdego komunikatu będziemy czekać na odpowiedź, wysłanie 100 komunikatów zajmie ok. sekundy. Natomiast jeśli po prostu wyślemy wszystkie komunikaty i nie będziemy czekać na żadne odpowiedzi, wysłanie 100 komunikatów nie zajmie zbyt wiele czasu. W większości przypadków tak naprawdę nie potrzebujemy odpowiedzi — Kafka odsyła temat, partycję i przesunięcie rekordu po jego zapisaniu, czego aplikacja wysyłająca z reguły nie wymaga. Z drugiej strony, musimy wiedzieć, kiedy nie udało się wysłać komunikatu, byśmy mogli rzucić wyjątek, zarejestrować błąd lub być może zapisać komunikat w pliku „błędów” w celu dokonania późniejszej analizy.

Aby wysyłać komunikaty asynchronicznie i nadal obsługiwać scenariusze błędów, producent obsługuje dodawanie wywołania zwrotnego podczas wysyłania rekordu. Oto przykład użycia wywołania zwrotnego:

```
private class DemoProducerCallback implements Callback { ❶
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Materiały biomedyczne", "USA"); ❸
producer.send(record, new DemoProducerCallback()); ❹
```

- ❶ W celu skorzystania z wywołań zwrotnych potrzebujesz klasy implementującej interfejs `org.apache.kafka.clients.producer.Callback`, który ma jedną funkcję: `onCompletion()`.
- ❷ Jeśli Kafka zwróci błąd, funkcja `onCompletion()` będzie miała wyjątek inny niż `null`. Tutaj „obsługujemy” to przez wypisywanie, ale kod produkcyjny prawdopodobnie będzie miał solidniejsze funkcje obsługi błędów.
- ❸ Rekordy są takie same jak wcześniej.
- ❹ Podczas wysyłania rekordu przekazujemy obiekt `Callback`.



Wywołania zwrotne są wykonywane w głównym wątku producenta. Gwarantuje to, że gdy do tej samej partycji wyślemy dwa komunikaty jeden po drugim, ich wywołania zwrotne zostaną wykonane w tej samej kolejności, w jakiej wysłaliśmy komunikaty. Jednak oznacza to również, że wywołanie zwrotne powinno być wykonywane dość szybko, aby uniknąć opóźnień producenta i uniemożliwienia wysyłania innych komunikatów. Nie zaleca się wykonywania operacji blokowania w ramach wywołania zwrotnego. Zamiast tego do wspólnego wykonywania jakiejkolwiek operacji blokowania należy użyć innego wątku.

Konfiguracja producentów

Jak dotąd poznałeś bardzo niewiele parametrów konfiguracyjnych dla producentów — jedynie obowiązkowy identyfikator URI `bootstrap.servers` i serializatory.

Producent ma dużą liczbę parametrów konfiguracyjnych, które zostały opisane w dokumentacji platformy Apache Kafka (<https://oreil.ly/RkxSS>), a wiele z nich ma rozsądne wartości domyślne, więc nie ma powodu majstrować przy każdym parametrze. Jednak niektóre parametry mają istotny wpływ na wykorzystanie pamięci, wydajność i niezawodność producentów. Przejrzymy je w tym podrozdziale.

`client.id`

`client.id` to logiczny identyfikator klienta i aplikacji, w której jest używany. Może to być dowolny łańcuch znaków i będzie on wykorzystywany przez brokery do identyfikowania komunikatów wysyłanych przez klienta. Jest stosowany także do celów rejestrowania i zbierania wskaźników oraz w przypadku kwot. Wybór dobrej nazwy klienta znacznie ułatwia rozwiązywanie problemów — to jak różnica między sytuacją „widzimy wysoki wskaźnik niepowodzeń uwierzytelniania z adresu IP 104.27.155.134” a „wygląda na to, że usługa weryfikacji zamówień nie może się uwierzytelnić — czy możesz poprosić Laurę, by się temu przyjrzała?”.

`acks`

Parametr `acks` kontroluje liczbę replik partycji, które muszą otrzymać rekord, aby producent mógł uznać zapis za udany. Domyślnie Kafka będzie odpowiadać, że rekord został zapisany pomyślnie, po otrzymaniu tego rekordu przez lidera (przewiduje się, że w wersji 3.0 platformy Apache Kafka to ustawienie domyślne zostanie zmienione). Ta opcja ma znaczący wpływ na trwałość zapisanych komunikatów i w zależności od przypadku użycia ustawienie domyślne może nie być najlepszym wyborem. Gwarancje niezawodności Kafki omówiliśmy szczegółowo w rozdziale 7., a na razie przyjmiemy się trzem dozwolonym wartościom parametru `acks`:

`acks=0`

Nie czekając na odpowiedź od brokera, producent uzna, że komunikat został wysłany pomyślnie. Oznacza to, że jeśli coś pójdzie nie tak i broker nie otrzyma komunikatu, producent nie będzie o tym wiedział, a komunikat zostanie utracony. Ponieważ jednak producent nie czeka na żadną odpowiedź z serwera, może wysyłać komunikaty z taką szybkością, na jaką pozwala sieć, dlatego to ustawienie można wykorzystać do uzyskania bardzo dużej przepustowości.

`acks=1`

Producent otrzyma od brokera pomyślną odpowiedź w momencie, gdy replika główna odbierze komunikat. Jeśli nie można zapisać komunikatu w replice głównej (np. gdy lider uległ awarii, a nowy lider nie został jeszcze wybrany), producent otrzyma odpowiedź o błędzie i będzie mógł ponowić próbę wysłania komunikatu, unikając potencjalnej utraty danych. Nadal istnieje jednak pewne ryzyko utraty komunikatu, jeżeli replika główna ulegnie awarii, a najnowsze komunikaty nie zostały jeszcze zreplikowane do nowego lidera.

acks=all

Producent otrzyma od brokera pomyślną odpowiedź, gdy komunikat otrzymają wszystkie repliki pozostające w synchronizacji. Jest to najbezpieczniejszy tryb, gdyż można mieć pewność, że komunikat jest w posiadaniu więcej niż jednego brokera i przetrwa nawet w razie awarii (więcej informacji na ten temat znajdziesz w rozdziale 6.). Jednak opóźnienie, o którym wspomnieliśmy w przypadku ustawienia `acks=1`, będzie jeszcze większe, ponieważ będziemy czekać, aż komunikat otrzyma więcej niż jeden broker.



Przekonasz się, że przy niższej i bardziej zawodnej konfiguracji `acks` producent będzie mógł szybciej wysłać rekordy. Oznacza to, że rezygnujesz z niezawodności na rzecz zmniejszenia **opóźnień producenta** (ang. *producer latency*). **Opóźnienie całkowite**, zwane też opóźnieniem od końca do końca (ang. *end-to-end latency*), jest jednak mierzone od chwili utworzenia rekordu do momentu, gdy staje się on dostępny do odczytu przez konsumenty, i jest identyczne dla wszystkich trzech opcji. Powodem jest to, że w celu zachowania spójności Kafka nie pozwala konsumentom na odczytywanie rekordów, dopóki nie zostaną one zapisane we wszystkich zsynchronizowanych replikach. Dlatego jeśli zależy Ci na opóźnieniu całkowitym, a nie tylko na opóźnieniu producenta, nie musisz brać pod uwagę żadnych kompromisów: uzyskasz takie samo opóźnienie całkowite, nawet jeśli wybierzesz najbardziej niezawodną opcję.

Czas dostarczania komunikatów

Producent ma wiele parametrów konfiguracyjnych, za pomocą których można kontrolować jedno z najbardziej interesujących dla programistów zachowań: ile czasu minie od chwili wywołania metody `send()` do momentu pomyślnego lub niepomyślnego zakończenia jej wykonywania. Tyle czasu jesteśmy gotowi poświęcić w oczekiwaniu na pozytywną odpowiedź Kafki lub na informację, że musimy się poddać i przyznać do porażki.

Na przestrzeni lat te konfiguracje i ich zachowania były wielokrotnie modyfikowane. Opiszemy tutaj najnowszą implementację wprowadzoną w platformie Apache Kafka 2.1.

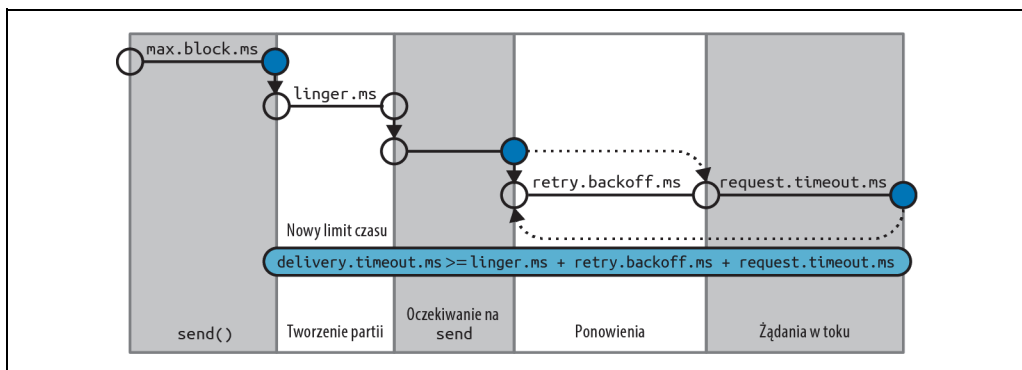
Od wersji 2.1 Kafki czas poświęcony na wysyłanie obiektu `ProducerRecord` dzielimy na dwa obsługiwane osobno przedziały:

- Czas oczekiwania na zakończenie wykonywania asynchronicznego wywołania `send()`. W tym przedziale czasu wątek, który wywołał `send()`, zostaje zablokowany.
- Czas liczony od momentu pomyślnego zakończenia asynchronicznego wywołania `send()` do momentu wyzwolenia wywołania zwrotnego (z powodzeniem lub niepowodzeniem). Jest równoznaczny z przedziałem czasu liczonym od chwili umieszczenia obiektu `ProducerRecord` w partii do wysłania aż do momentu, gdy Kafka odeśle pozytywną odpowiedź, nieodwracalny błąd lub upłynie limit czasu na wysłanie.



Jeśli używasz metody `send()` synchronicznie, wątek wysyłający będzie w sposób ciągły blokował wykonywanie w obu przedziałach czasowych i nie będziesz w stanie określić, ile czasu spędził w każdym z nich. Omówimy częsty i zalecany przypadek, w którym metoda `send()` jest używana asynchronicznie z wywołaniem zwrotnym.

Przepływ danych w obrębie producenta oraz wzajemny wpływ różnych parametrów konfiguracyjnych pokazaliśmy na rysunku 3.2¹.



Rysunek 3.2. Diagram sekwencji dla podziału czasu dostarczania wewnątrz producenta Kafka

Przyjrzymy się różnym parametrom konfiguracyjnym, które są używane do kontrolowania czasu poświęconego na czekanie w tych dwóch interwałach, oraz ich wzajemnym interakcjom.

max.block.ms

Ten parametr kontroluje, jak długo producent może blokować wykonywanie podczas wywoływania metody `send()` oraz bezpośredniego żądania metadanych za pośrednictwem metody `partitionsFor()`. Metody te mogą blokować, gdy bufor wysyłania producenta jest pełny lub gdy metadane nie są dostępne. Po osiągnięciu progu `max.block.ms` rzucany jest wyjątek upłynięcia limitu czasu.

delivery.timeout.ms

Ta konfiguracja ogranicza, ile czasu może upłynąć od momentu, w którym rekord jest gotowy do wysłania (wykonanie metody `send()` zakończyło się pomyślnie, a rekord został umieszczony w określonej partii), do momentu uzyskania odpowiedzi brokera lub rezygnacji klienta, włącznie z czasem spędzonym na ponawianiu prób. Jak pokazaliśmy na rysunku 3.2, ten czas powinien być dłuższy niż suma czasów `linger.ms` i `request.timeout.ms`. Jeśli spróbujesz utworzyć producent z niespójną konfiguracją limitu czasu, otrzymasz wyjątek. Komunikaty mogą być z powodzeniem wysyłane znacznie szybciej, niż określa `delivery.timeout.ms`, i zazwyczaj tak się dzieje.

Jeżeli podczas ponawiania próby producent przekroczy limit czasu dostarczania określony przez `delivery.timeout.ms`, wywołanie zwrotne zostanie wywołane z wyjątkiem odpowiadającym błędowi zwróconemu przez broker przed ponowieniem próby. Jeśli `delivery.timeout.ms` zostanie przekroczony, gdy partia rekordów będzie nadal czekała na wysłanie, wywołanie zwrotne zostanie wywołane z wyjątkiem przekroczenia limitu czasu.

¹ Diagram autorstwa Sumanty Tambego udostępniony projektowi Apache Kafka na licencji ASLv2.



Dla limitu czasu dostarczania możesz skonfigurować maksymalny czas oczekiwania na wysłanie wiadomości, zwykle kilka minut, i pozostawić domyślną liczbę prób ponawiania (praktycznie nieskończoną). W tej konfiguracji producent będzie ponawiał próby tak długo, jak będzie mógł (lub do skutku). Jest to bardzo rozsądny sposób potraktowania kwestii prób ponawiania. Nasz normalny proces dostrajania prób ponawiania jest następujący: w przypadku awarii brokera ukończenie wyboru lidera trwa zwykle 30 sekund, więc dla bezpieczeństwa ponawiamy próby przez 120 sekund. Zamiast przekładać to na liczbę ponownych prób i interwał między próbami, skonfigurujmy po prostu `delivery.timeout.ms` na 120.

request.timeout.ms

Ten parametr kontroluje, jak długo producent podczas wysyłania danych będzie czekał na odpowiedź z serwera. Zwróć uwagę, że jest to czas oczekiwania na każde żądanie producenta, a po jego upływie producent rezygnuje; ten czas nie uwzględnia prób ponawiania, czasu przed wysłaniem itd. Jeśli limit czasu zostanie osiągnięty bez otrzymania odpowiedzi, producent ponowi próbę wysłania lub zakończy wywołanie zwrotne z wyjątkiem `TimeoutException`.

retries i retry.backoff.ms

Gdy producent otrzyma z serwera komunikat o błędzie, może chodzić o jakiś przejściowy błąd (np. brak lidera dla partycji). W takim przypadku o tym, ile razy producent ponawia próbę wysłania komunikatu przed zrezygnowaniem i powiadomieniem klienta o problemie, decyduje wartość parametru `retries`. Domyślnie producent między kolejnymi próbami odczekuje 100 ms, ale można to kontrolować za pomocą parametru `retry.backoff.ms`.

Odradzamy używanie tych parametrów w aktualnej wersji Kafki. Zamiast tego należy sprawdzić, ile czasu zajmuje odzyskanie sprawności po awarii brokera (tj. wybranie liderów dla wszystkich partycji), i ustawić `delivery.timeout.ms` w taki sposób, aby całkowity czas poświęcony na ponawianie prób był dłuższy niż czas potrzebny klastrowi Kafki na odzyskanie sprawności po awarii — w przeciwnym razie producent zbyt wcześnie rezygnuje.

Nie wszystkie błędy będą skutkowały ponawianiem prób przez producenta. Niektóre błędy nie są przejściowe i nie będą powodować ponawiania prób (np. błąd „zbyt duża wiadomość”). Ponieważ producent sam obsługuje próby ponawiania, generalnie nie ma sensu wprowadzanie tej kwestii do logiki aplikacji. Swoje wysiłki powinieneś skoncentrować na obsłudze błędów nieodwracalnych lub przypadków, w których liczba prób ponawiania zostanie wyczerpana.



Jeśli chcesz całkowicie wyłączyć próby ponawiania, jedynym sposobem jest ustawienie `retries=0`.

linger.ms

Parametr `linger.ms` kontroluje czas oczekiwania na dodatkowe komunikaty przed wysłaniem bieżącej partii. `KafkaProducer` wysyła partię komunikatów, kiedy bieżąca partia zostanie zapełniona lub osiągnięty zostanie limit `linger.ms`. Domyślnie producent wyśle komunikaty, gdy tylko dostępny

będzie wątek nadawcy, który może je wysłać, nawet jeśli w partii jest tylko jeden komunikat. Przez ustawienie dla `linger.ms` wartości większej niż 0 instruujemy producent, by odczekał kilka milisekund w celu dołączenia do danej partii dodatkowych komunikatów przed wysłaniem jej do brokerów. Zwiększa to nieco opóźnienie i istotnie zwiększa przepustowość — narzut generowany przez pojedynczy komunikat jest dużo mniejszy, a kompresja jest znacznie lepsza (jeśli została włączona).

buffer.memory

Ta konfiguracja określa ilość pamięci, której producent będzie używał do buforowania komunikatów oczekujących na wysłanie do brokerów. Jeżeli aplikacja wysyła komunikaty szybciej, niż można je dostarczyć do serwera, producentowi może zabraknąć miejsca, a kolejne wywołania `send()` przed rzuceniem wyjątku będą blokowały wykonywanie przez `max.block.ms` i czekały na zwolnienie miejsca. Zwróć uwagę, że inaczej niż w przypadku większości wyjątków producenta, ten limit czasu jest rzucany przez metodę `send()`, a nie przez wynikowy obiekt `Future`.

compression.type

Domyślnie komunikaty są wysyłane w postaci nieskompresowanej. Ten parametr może mieć wartość `snappy`, `gzip`, `lz4` lub `zstd`, określając odpowiedni algorytm kompresji, który ma zostać użyty do skompresowania danych przed wysłaniem ich do brokerów. Kompresja Snappy została opracowana przez firmę Google, by zapewnić przyzwoite współczynniki kompresji przy niskim obciążeniu procesora i dobrej wydajności, dlatego jest zalecana w przypadkach, w których problemem jest zarówno wydajność, jak i przepustowość. Kompresja Gzip zwykle zużywa więcej procesora i czasu, ale zapewnia lepsze współczynniki kompresji, stąd jest zalecana, gdy przepustowość sieci jest bardziej ograniczona. Włączając kompresję, zmniejszasz wykorzystanie sieci i pamięci masowej, które często stanowią wąskie gardło przy wysyłaniu komunikatów do Kafki.

batch.size

Gdy do tej samej partycji wysyłanych jest wiele rekordów, producent łączy je w partię. Ten parametr kontroluje ilość pamięci mierzoną w bajtach (a nie w komunikatach!), która będzie używana dla każdej partii. Gdy partia jest pełna, wysyłane są wszystkie znajdujące się w niej komunikaty. Nie oznacza to jednak, że producent będzie czekał na wypełnienie partii. Będzie wysyłał także niepełne partie, a nawet partie zawierające tylko jeden komunikat. Dlatego ustawienie zbyt dużego rozmiaru partii nie spowoduje opóźnień w wysyłaniu komunikatów; będzie jednak skutkowało używaniem większej ilości pamięci dla partii. Ustawienie zbyt małego rozmiaru partii będzie generowało dodatkowe obciążenie, gdyż producent będzie musiał częściej wysyłać komunikaty.

max.in.flight.requests.per.connection

To ustawienie kontroluje, ile partii komunikatów producent wyśle do serwera bez otrzymywania odpowiedzi. Wyższe ustawienia mogą zwiększyć użycie pamięci, poprawiając jednocześnie przepustowość. Eksperymenty opisane na stronie wiki Apache (<https://oreil.ly/NZmJ0>) pokazują, że

w środowisku z jednym centrum danych przepustowość jest maksymalizowana przy zaledwie 2 żądaniach w toku, jednak wartość domyślna 5 wykazuje podobną wydajność.



Gwarancje kolejności

Apache Kafka zachowuje kolejność komunikatów w ramach partycji. Oznacza to, że jeśli komunikaty są wysyłane przez producenta w określonej kolejności, broker zapisuje je na partycji w tej samej kolejności i wszystkie konsumenty również w tej kolejności będą je odczytywać. W pewnych przypadkach kolejność jest bardzo ważna. Istnieje duża różnica, jeśli najpierw wpłacimy 100 zł na konto, a później je wypłacimy, niż gdybyśmy zrobili na odwrót! Niektóre przypadki użycia są jednak mniej wrażliwe na tę kwestię.

Ustawiając dla parametru `retries` wartość niezerową, a dla parametru `max.in.flight.requests.per.connection` wartość większą niż 1, ryzykujemy, że jeśli brokerowi nie uda się zapisać pierwszej partii komunikatów, ale uda się zapisać drugą (która była już w toku), a następnie ponowna próba zapisania pierwszej partii zakończy się powodzeniem, kolejność zostanie w ten sposób odwrócona.

Ponieważ ze względu na wydajność chcemy mieć co najmniej dwa żądania w toku, a ze względu na niezawodność — dużą liczbę ponownych prób, najlepszym rozwiązaniem jest ustawienie `enable.idempotence=true`. Gwarantuje to porządkowanie wiadomości z maksymalnie pięcioma żądaniem w toku, a ponadto zapewnia, że próby ponawiania nie będą wprowadzały duplikatów. Zagadnienie idempotentnego producenta omówiliśmy szczegółowo w rozdziale 8.

max.request.size

Parametr ten kontroluje rozmiar żądania produkcji wysyłanego przez producenta. Ogranicza nie tylko rozmiar największego komunikatu, jaki można wysłać, ale także liczbę komunikatów, które producent może wysłać w pojedynczym żądaniu. Dla przykładu przy domyślnym maksymalnym rozmiarze żądania 1 MB największy komunikat, jaki można wysłać, będzie miał rozmiar 1 MB. Ewentualnie producent może w jednym żądaniu wysłać partię 1024 komunikatów o rozmiarach 1 kB. Ponadto broker ma swój własny limit rozmiaru największego akceptowanego komunikatu (`message.max.bytes`). Zwykle dobrym pomysłem jest dopasowanie tych konfiguracji, by producent nie próbował wysłać komunikatów o rozmiarach, które będą odrzucane przez broker.

receive.buffer.bytes i send.buffer.bytes

Są to rozmiary buforów wysyłania i odbierania TCP używanych przez gniazda podczas zapisywania i odczytywania danych. Jeśli są ustawione na `-1`, użyte zostaną domyślne ustawienia systemu operacyjnego. Dobrym pomysłem jest zwiększenie wartości tych parametrów, gdy producenci lub konsumenci komunikują się z brokerami w innym centrum danych, ponieważ te łącza sieciowe mają zwykle większe opóźnienia i niższą przepustowość.

enable.idempotence

Od wersji 0.11 Kafka obsługuje semantykę „**dokładnie raz**” (ang. *exactly once*). „Dokładnie raz” to dość obszerny temat i poświęcimy mu cały rozdział, ale idempotentny producent to prosty i bardzo korzystny jego element.

Założmy, że konfigurujesz swój producent w taki sposób, aby zmaksymalizować niezawodność, czyli ustawiasz `acks=all`, a ponadto w celu zapewnienia wystarczającej liczby prób ponawiania ustawiasz przyzwoicie duży limit `delivery.timeout.ms`. Dzięki temu każdy komunikat będzie zapisywany w Kafce przynajmniej raz. W niektórych przypadkach oznacza to, że komunikaty będą zapisywane w Kafce więcej niż raz. Wyobraźmy sobie np., że broker otrzymał od producenta rekord, zapisał go na dysku lokalnym i rekord został pomyślnie zreplikowany do pozostałych brokerów, ale potem ten pierwszy broker uległ awarii, zanim zdążył wysłać odpowiedź do producenta. Producent będzie czekał, dopóki nie osiągnie limitu czasu żądania `request.timeout.ms`, a następnie spróbuje ponownie wysłać rekord. Ponowna próba zostanie skierowana do nowego lidera, który ma już kopię tego rekordu, gdyż poprzedni zapis został pomyślnie zreplikowany. W ten sposób otrzymasz zduplikowany rekord.

Aby tego uniknąć, możesz ustawić `enable.idempotence=true`. Gdy włączona jest opcja producenta idempotentnego, producent do każdego wysyłanego rekordu dołącza numer sekwencyjny. Jeżeli broker otrzyma rekordy z tym samym numerem sekwencyjnym, odrzuci drugą kopię, a producentowi odeśle nieszkodliwy wyjątek `DuplicateSequenceException`.



Włączenie idempotencji wymaga, aby wartość `max.in.flight.requests.per.connection` była mniejsza lub równa 5, wartość `retries` była większa niż 0, oraz ustawienia `acks=all`. W przypadku skonfigurowania niezgodnych wartości rzucony zostanie wyjątek `ConfigException`.

Serializatory

Jak widziałeś w poprzednich przykładach, konfiguracja producenta obejmuje obowiązkowe serializatory. Pokazaliśmy, jak używać domyślnego serializatora `String`. Kafka zawiera również serializatory dla liczb całkowitych, tablic bajtów itd., ale nie obejmuje to większości przypadków użycia. W końcu będziesz potrzebował serializować bardziej ogólne rekordy.

Zacniemy od pokazania, jak napisać własny serializator, a następnie przedstawimy serializator Avro jako zalecaną alternatywę.

Serializatory niestandardowe

Gdy obiekt, który potrzebujesz wysłać do Kafki, nie jest prostym łańcuchem znaków ani liczbą całkowitą, możesz do tworzenia rekordów wykorzystać ogólną bibliotekę serializacji, taką jak Avro, Thrift czy Protobuf, albo utworzyć niestandardową serializację dla używanych przez Ciebie obiektów. Zdecydowanie zalecamy jednak korzystanie z ogólnej biblioteki serializacji. Aby zrozumieć, jak działają serializatory i dlaczego zastosowanie biblioteki serializacji jest dobrym pomysłem, zobaczymy, na czym polega pisanie własnego niestandardowego serializatora.

Założmy, że zamiast rejestrować jedynie nazwę klienta, tworzysz prostą klasę do reprezentowania klientów:

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

Przyjmijmy teraz, że chcesz utworzyć niestandardowy serializator dla tej klasy. Będzie to wyglądało mniej więcej tak:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

    @Override
    public void configure(Map configs, boolean isKey) {
        // Nic do konfigurowania
    }

    @Override
    /**
     * Serializujemy Customer jako:
     * 4 bajty int reprezentujące customerId
     * 4 bajty int reprezentujące długość nazwy customerName w bajtach UTF-8 (0, jeśli nazwa jest Null)
     * N bajtów reprezentujących customerName w UTF-8
     */
    public byte[] serialize(String topic, Customer data) {
        try {
            byte[] serializedName;
            int stringSize;
            if (data == null)
                return null;
            else {
                if (data.getName() != null) {
                    serializedName = data.getName().getBytes("UTF-8");
                    stringSize = serializedName.length;
                } else {
                    serializedName = new byte[0];
                    stringSize = 0;
                }
            }
        }
    }
}
```

```

        ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
        buffer.putInt(data.getID());
        buffer.putInt(stringSize);
        buffer.put(serializedName);

        return buffer.array();
    } catch (Exception e) {
        throw new SerializationException(
            "Błąd podczas serializacji Customer na byte[] " + e);
    }
}

@Override
public void close() {
    // Nic do zamknięcia
}
}

```

Skonfigurowanie producenta przy użyciu tego niestandardowego serializatora `CustomerSerializer` umożliwi zdefiniowanie `ProducerRecord<String, Customer>` oraz wysyłanie danych `Customer` i przekazywanie obiektów `Customer` bezpośrednio do producenta. Ten przykład jest dość prosty, ale możesz się przekonać, jak kruchy jest kod. Jeżeli kiedyś będziesz mieć np. zbyt wielu klientów i będziesz musiał zmienić `customerID` na `Long` albo zdecydujesz się dodać do `Customer` pole `StartDate`, napotkasz poważny problem z utrzymywaniem zgodności między starymi i nowymi komunikatami. Debugowanie problemów ze zgodnością między różnymi wersjami serializatorów i deserializatorów jest dość trudne: musisz porównywać tablice nieprzetworzonych bajtów. Co gorsza, jeśli wiele zespołów w tej samej firmie będzie zapisywało w Kafce dane `Customer`, wszystkie będą musiały używać tych samych serializatorów i modyfikować kod dokładnie w tym samym czasie.

Z tych powodów zalecamy używanie istniejących serializatorów i deserializatorów, takich jak JSON, Apache Avro, Thrift lub Protobuf. W następnym punkcie opiszemy Apache Avro, a następnie pokażemy, jak serializować rekordy Avry i wysyłać je do Kafki.

Serializacja przy użyciu Apache Avro

Apache Avro to neutralny językowo format serializacji danych. Projekt ten został utworzony przez Douga Cuttinga, aby umożliwić udostępnianie plików danych dużej grupie odbiorców.

Dane Avry są opisywane w schemacie niezależnym od języka programowania. Schemat wykorzystuje zwykle notację JSON, a serializacja jest przeprowadzana najczęściej do postaci plików binarnych, chociaż obsługiwana jest również serializacja do formatu JSON. Avro zakłada obecność schematu podczas odczytywania i zapisywania plików, co jest z reguły realizowane przez osadzanie schematu w samych plikach.

Jedną z najciekawszych cech Avry, dzięki której serializator ten dobrze nadaje się do wykorzystania w systemie wymiany komunikatów takim jak Kafka, polega na tym, że gdy aplikacja zapisująca komunikaty przełącza się na nowy, ale zgodny schemat, aplikacje odczytujące dane mogą kontynuować przetwarzanie komunikatów bez konieczności wprowadzania zmian lub aktualizacji.

Przyjmijmy, że oryginalny schemat wyglądał tak:

```
{ "namespace": "customerManagement.avro",  
  "type": "record",  
  "name": "Customer",  
  "fields": [  
    { "name": "id", "type": "int" },  
    { "name": "name", "type": "string" },  
    { "name": "faxNumber", "type": ["null", "string"], "default": "null" } ❶  
  ]  
}
```

- ❶ Pola `id` i `name` są obowiązkowe, natomiast pole `faxNumber` jest opcjonalne i domyślnie ma wartość `null`.

Używaliśmy tego schematu przez kilka miesięcy i wygenerowaliśmy kilka terabajtów danych w tym formacie. Załóżmy teraz, że wraz z nową wersją decydujemy się wkroczyć w XXI w. i nie będziemy już uwzględniać pola numeru faksu, a zamiast tego użyjemy pola adresu e-mailowego.

Nowy schemat byłby następujący:

```
{ "namespace": "customerManagement.avro",  
  "type": "record",  
  "name": "Customer",  
  "fields": [  
    { "name": "id", "type": "int" },  
    { "name": "name", "type": "string" },  
    { "name": "email", "type": ["null", "string"], "default": "null" }  
  ]  
}
```

Po uaktualnieniu do nowej wersji stare rekordy będą zawierać pole `faxNumber`, a nowe rekordy pole `email`. W wielu organizacjach uaktualnienia są wykonywane powoli i przez wiele miesięcy. Musimy więc zastanowić się, w jaki sposób aplikacje sprzed uaktualnienia, które nadal używają numerów faksów, oraz aplikacje po uaktualnieniu, używające adresu e-mailowego, będą w stanie obsłużyć wszystkie zdarzenia w Kafce.

Aplikacja odczytująca będzie zawierała wywołania metod takich, jak np.: `getName()`, `getId()` czy `getFaxNumber()`. Jeżeli napotka komunikat napisany w nowym schemacie, metody `getName()` i `getId()` będą nadal działały bez modyfikacji, ale metoda `getFaxNumber()` zwróci `null`, gdyż komunikat nie będzie zawierał numeru faksu.

Przyjmijmy, że uaktualnimy naszą aplikację odczytującą i nie będzie już miała metody `getFaxNumber()`, a zamiast niej otrzyma metodę `getEmail()`. Jeśli napotka komunikat napisany w starym schemacie, `getEmail()` zwróci `null`, ponieważ starsze komunikaty nie zawierają adresu e-mailowego.

Ten przykład ilustruje korzyści płynące ze stosowania platformy Avro: nawet jeśli zmienimy schemat w komunikatach bez zmieniania wszystkich aplikacji odczytujących dane, nie będzie wyjątków ani przerywających działanie błędów, a ponadto nie trzeba będzie przeprowadzać kosztownych aktualizacji istniejących danych.

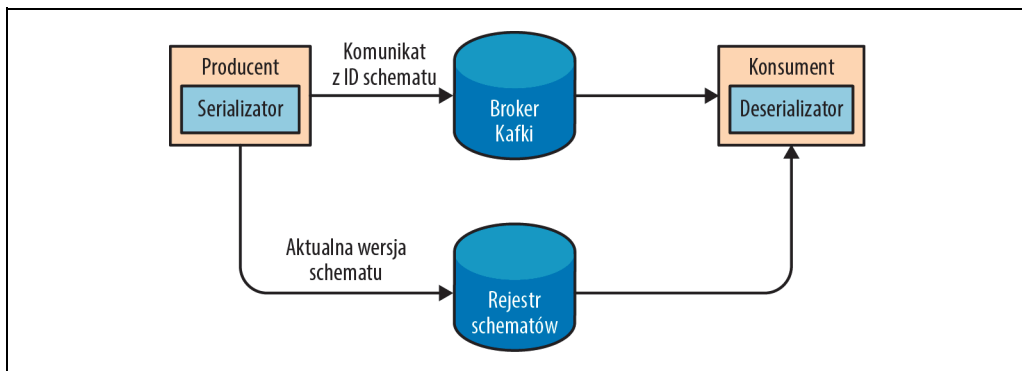
Ten scenariusz ma jednak dwa zastrzeżenia:

- Schemat używany do zapisywania danych i schemat oczekiwany przez aplikację odczytującą muszą być zgodne. Reguły zgodności zostały opisane w dokumentacji Avry (<http://bit.ly/2t9FmEb>).
- Deserializator będzie potrzebował dostępu do schematu, który został użyty podczas zapisywania danych, nawet jeżeli różni się on od schematu oczekiwanego przez aplikację uzyskującą dostęp do danych. W Avrze schemat zapisu jest zawarty w pliku, ale w przypadku komunikatów Kafki można to obsłużyć w lepszy sposób. Przyjrzymy się temu w następnym punkcie.

Używanie rekordów Avry z Kafką

Inaczej niż w przypadku plików Avry, gdzie przechowywanie całego schematu w pliku danych generuje stosunkowo duży narzut, przechowywanie całego schematu w każdym rekordzie zazwyczaj co najmniej podwaja rozmiar rekordu. Mimo to Avra wymaga, żeby podczas odczytu rekordu obecny był cały schemat, musimy więc umieścić schemat w innym miejscu. W tym celu stosujemy powszechny wzorzec architektoniczny i używamy tzw. **rejestrów schematów** (ang. *Schema Registry*). Rejestr schematów nie jest częścią platformy Apache Kafka, ale dostępnych jest do wyboru kilka opcji na licencji *open source*. W tym przykładzie użyjemy Confluent Schema Registry, którego kod znajdziesz w GitHubie (<https://oreil.ly/htoZK>) albo możesz go zainstalować jako część platformy Confluent (<https://oreil.ly/n2V71>). Jeżeli zdecydujesz się skorzystać z tego rejestru schematów, powinieneś zapoznać się z dokumentacją dostępną na stronie Confluenta (<https://oreil.ly/yFkTX>).

Pomysł polega na przechowywaniu w jednym rejestrze wszystkich schematów używanych do zapisywania danych w Kafce. W rekordzie tworzonym dla Kafki wystarczy wtedy zapisać jedynie identyfikator schematu. Konsumenty będą mogły używać tego identyfikatora do pobierania rekordu z rejestru schematów i deserializowania danych. Kluczową kwestią jest to, że cała ta praca — zapisywanie schematu w rejestrze i pobieranie go, gdy jest potrzebny — jest wykonywana w serializatorach i deserializatorach. Kod produkujący dane dla Kafki używa w tej sytuacji serializatora Avro w taki sam sposób, jak każdego innego. Proces ten przedstawiliśmy na rysunku 3.3.



Rysunek 3.3. Schemat blokowy serializacji i deserializacji rekordów Avry

Poniżej zamieściliśmy przykład produkowania do Kafki generowanych przez Avrę obiektów (więcej informacji na temat generowania obiektów na podstawie schematów Avro znajdziesz w dokumentacji tego serializatora na stronie <https://oreil.ly/klcjK>):

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";

Producer<String, Customer> producer = new KafkaProducer<>(props); ❸

// Aby przerwać produkowanie nowych zdarzeń, wciśnij Ctrl+C
while (true) {
    Customer customer = CustomerGenerator.getNext(); ❹
    System.out.println("Wygenerowano konsument " + customer.toString());
    ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getName(), customer); ❺
    producer.send(record); ❻
}
```

- ❶ Do serializacji obiektów za pomocą Avry używamy `KafkaAvroSerializer`. Zwróć uwagę, że `KafkaAvroSerializer` może obsługiwać również typy proste, dlatego później możemy użyć typu `String` jako klucza rekordu i naszego obiektu `Customer` jako wartości.
- ❷ `schema.registry.url` to konfiguracja serializatora Avro, która zostanie przekazana do niego przez producent. Wskazuje, gdzie przechowujemy schematy.
- ❸ `Customer` jest naszym wygenerowanym obiektem. Instruujemy producent, że rekordy jako wartość będą zawierały obiekt `Customer`.
- ❹ Klasa `Customer` nie jest standardową klasą Javy (tzw. zwykłym starym obiektem Javy; ang. *Plain Old Java Object* — POJO), lecz raczej wyspecjalizowanym obiektem platformy Avro, utworzonym na bazie schematu przy użyciu funkcjonalności generowania kodu tej platformy. Serializator Avro może serializować tylko obiekty Avry, a nie POJO. Klasy Avry można generować za pomocą `avro-tools.jar` lub wtyczki Avro Maven, które są częścią Apache Avro. Więcej informacji na temat generowania klas Avry znajdziesz w dokumentacji *Apache Avro Getting Started (Java)* (<https://oreil.ly/sHGEe>).
- ❺ Tworzymy również instancję `ProducerRecord` z typem wartości `Customer` i przekazujemy obiekt `Customer` podczas tworzenia nowego rekordu.
- ❻ To wszystko. Wysyłamy rekord z naszym obiektem `Customer`, a resztą zajmie się `KafkaAvroSerializer`.

Avro pozwala również na używanie ogólnych obiektów Avry jako map klucz-wartość zamiast wygenerowanych obiektów Avry z metodami pobierającymi i ustawiającymi zgodnymi ze schematem użytym do ich wygenerowania. Aby skorzystać z ogólnych obiektów Avry, wystarczy podać schemat:

```

Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer",
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

String schemaString =
    "{ \"namespace\": \"customerManagement.avro\",
      \"type\": \"record\", \" + ❸
      \"name\": \"Customer\", \" +
      \"fields\": [\" +
        { \"name\": \"id\", \"type\": \"int\" }, \" +
        { \"name\": \"name\", \"type\": \"string\" }, \" +
        { \"name\": \"email\", \"type\": \" + \"[\"null\", \"string\"]\", \" +
        \"default\": \"null\" }\" +
      ] }";
Producer<String, GenericRecord> producer =
    new KafkaProducer<String, GenericRecord>(props); ❹

Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(schemaString);

for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
    String name = "exampleCustomer" + nCustomers;
    String email = "example " + nCustomers + "@example.com";

    GenericRecord customer = new GenericData.Record(schema); ❺
    customer.put("id", nCustomers);
    customer.put("name", name);
    customer.put("email", email);

    ProducerRecord<String, GenericRecord> data =
        new ProducerRecord<>("customerContacts", name, customer);
    producer.send(data);
}

```

- ❶ Nadal używamy tego samego serializatora `KafkaAvroSerializer`.
- ❷ Podajemy identyfikator URI tego samego rejestru schematów.
- ❸ Teraz musimy jednak podać także schemat Avry, ponieważ nie jest on zapewniany przez obiekt generowany przez Avrę.
- ❹ Nasz typ obiektu to `GenericRecord` Avry, inicjowany przy użyciu schematu i danych, które chcemy zapisywać.
- ❺ Wartością `ProducerRecord` staje się po prostu `GenericRecord`, która zawiera schemat i dane. Serializator będzie wiedział, jak uzyskać schemat z tego rekordu, przechować go w rejestrze schematów i przeprowadzić serializację danych obiektu.

Partycje

W poprzednich przykładach tworzone przez nas obiekty `ProducerRecord` zawierały nazwę tematu, klucz i wartość. Komunikaty Kafki są parami klucz-wartość i chociaż można tworzyć rekordy `ProducerRecord` jedynie z tematem i wartością przy domyślnie ustawionej dla klucza wartości `null`,

większość aplikacji produkuje rekordy z kluczami. Klucze służą dwóm celom: stanowią dodatkowe informacje zapisywane wraz z komunikatem, a ponadto są zwykle wykorzystywane do określania partycji tematu, w której ma zostać zapisany dany komunikat (klucze odgrywają również ważną rolę w skompaktowanych tematach — omówimy je w rozdziale 6.). Wszystkie komunikaty z tym samym kluczem trafiają do jednej partycji. Oznacza to, że jeśli proces odczytuje tylko podzbiór partycji z tematu (więcej informacji na ten temat znajdziesz w rozdziale 4.), wszystkie rekordy dla pojedynczego klucza będą odczytywane przez ten sam proces. Aby utworzyć rekord klucz-wartość, wystarczy wyprodukować rekord `ProducerRecord` w następujący sposób:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Sprzęt laboratoryjny", "USA");
```

Przy tworzeniu komunikatów z pustym kluczem można go po prostu pominąć:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ W tym przypadku dla klucza zostanie ustawiona wartość `null`.

Gdy klucz ma wartość `null` i używany jest domyślny partycjoner, rekord jest wysyłany losowo do jednej z dostępnych partycji tematu. Do równomiernego dystrybuowania komunikatów między partycjami wykorzystywany jest algorytm karuzelowy (ang. *round-robin*). Od wersji 2.4 producenta platformy Apache Kafka algorytm karuzelowy jest stosowany w domyślnym partycjonerze, gdy obsługa pustych kluczy jest lekka. Oznacza to, że przed przejściem do następnej partycji algorytm ten wypełnia partię komunikatów wysyłanych do jednej partycji. Pozwala to wysłać do Kafki tę samą liczbę komunikatów w mniejszej liczbie żądań, co zapewnia mniejsze opóźnienia i zmniejszenie wykorzystania procesora w brokerze.

Jeżeli klucz istnieje i używany jest domyślny partycjoner, Kafka generuje skróty kluczy (stosując własny algorytm mieszający, dzięki czemu wartości skrótów nie zmieniają się po uaktualnieniu Javy) i używa wyniku do mapowania komunikatów na określone partycje. Ponieważ ważne jest, aby dany klucz był zawsze mapowany na tę samą partycję, do określania mapowania wykorzystujemy wszystkie partycje z danego tematu, a nie tylko partycje dostępne. Oznacza to, że jeśli określona partycja będzie niedostępna podczas zapisywania w niej danych, może wystąpić błąd. Jest to dość rzadkie, o czym przekonasz się podczas lektury rozdziału 7., gdy będziemy omawiać replikację i dostępność Kafki.

Oprócz partycjonera domyślnego klienty Apache Kafki zapewniają również partycjonery `RoundRobinPartitioner` i `UniformStickyPartitioner`. Oferują one odpowiednio losowe przypisywanie partycji oraz lepsze losowe przypisywanie partycji, nawet jeśli komunikaty mają klucze. Partycjonery te są przydatne, gdy dla aplikacji konsumującej komunikaty istotne są klucze (niektóre aplikacje ETL używają klucza z rekordów Kafki jako klucza podstawowego podczas ładowania danych z Kafki do relacyjnej bazy danych), ale `RoundRobinPartitioner` wypacza rozkład obciążenia roboczego, ponieważ pojedynczy klucz może generować nieproporcjonalnie duże obciążenie. Równomierne rozłożenie obciążenia roboczego na wszystkie partycje można uzyskać przy użyciu partycjonera `UniformStickyPartitioner`.

Gdy używany jest domyślny partycjoner, mapowanie kluczy na partycje jest spójne tylko pod warunkiem, że liczba partycji w temacie nie ulega zmianie. Dopóki liczba partycji jest stała, możesz być

pewien, że rekordy dotyczące np. użytkownika 045189 będą zawsze zapisywane na partycji 34. Umożliwia to stosowanie rodzaju optymalizacji podczas odczytu danych z partycji. Jednak w momencie dodania do tematu nowych partycji przestaje to być gwarantowane — stare rekordy pozostaną w partycji 34, podczas gdy nowe rekordy mogą zostać zapisane na innej partycji. Gdy klucze partycjonowania są ważne, najłatwiejszym rozwiązaniem jest tworzenie tematów z odpowiednio dużą liczbą partycji i nigdy niedodawanie nowych (sugestie dotyczące ustalania liczby partycji znajdziesz na blogu Confluenta: <https://oreil.ly/ortRk>).

Wdrażanie niestandardowej strategii partycjonowania

Do tej pory omówiliśmy cechy najczęściej używanego partycjonera domyślnego. Kafka nie ogranicza nas jednak tylko do partycjonowania mieszającego, a czasami istnieją dobre powody, aby partycjonować dane w inny sposób. Załóżmy, że jesteś dostawcą B2B, a Twoim największym klientem jest firma produkująca urządzenia przenośne zwane Bananami. Przyjmijmy, że klient „Banan” generuje ponad 10% Twoich codziennych transakcji. Jeżeli używasz domyślnego partycjonowania mieszającego, rekordy Banana są alokowane na tej samej partycji co rekordy innych kont, w wyniku czego jedna partycja staje się znacznie większa niż reszta. Może to spowodować wyczerpanie miejsca na serwerach, spowolnienie przetwarzania itp. Tak naprawdę potrzebowalibyś przydzielić Bananowi osobną partycję, a partycjonowania mieszającego użyć do mapowania pozostałych kont na wszystkie inne partycje.

Oto przykład niestandardowego partycjonera:

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ❶

    public int partition(String topic, Object key, byte[] keyBytes,
                        Object value, byte[] valueBytes,
                        Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceof String))) ❷
            throw new InvalidRecordException("Oczekujemy, że wszystkie komunikaty " +
                "będą miały jako klucz nazwę klienta");

        if (((String) key).equals("Banan"))
            return numPartitions - 1; // Banan będzie zawsze przydzielany do ostatniej partycji

        // Pozostałe rekordy będą mieszane do reszty partycji
        return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
    }

    public void close() {}
}
```


- ❶ Interfejs partycjonera obejmuje metody `configure`, `partition` i `close`. Tutaj implementujemy tylko metodę `partition`, chociaż tak naprawdę zamiast kodować specjalną nazwę klienta na sztywno w `partition`, powinniśmy przekazać ją za pośrednictwem metody `configure`.
- ❷ Oczekujemy tylko kluczy typu `String`, więc jeśli klucz jest innego typu, rzucamy wyjątek.

Nagłówki

Oprócz klucza i wartości rekordy mogą zawierać również nagłówki. Nagłówki rekordów umożliwiają dodawanie metadanych dotyczących rekordu Kafki bez dołączania dodatkowych informacji do pary klucz-wartość samego rekordu. Nagłówki są często używane do określania pochodzenia danych w rekordzie, aby wskazać ich źródło, oraz do celów routingu lub śledzenia komunikatów na podstawie informacji z nagłówka bez konieczności parsowania samego komunikatu (który jest np. zaszyfrowany, a router nie ma uprawnień dostępu do danych).

Nagłówki są implementowane jako uporządkowana kolekcja par klucz-wartość. Klucze są zawsze typu `String`, a wartości mogą być dowolnymi obiektami serializowanymi, podobnie jak wartości komunikatów.

Oto prosty przykład pokazujący, jak dodać nagłówki do obiektu `ProducerRecord`:

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Produkty precyzyjne", "Francja");  
  
record.headers().add("privacy-level", "YOLO".getBytes(StandardCharsets.UTF_8));
```

Przechwytywacze

Czasami zachodzi potrzeba zmodyfikowania zachowania aplikacji klienckiej Kafki bez modyfikowania jej kodu, np. w celu dodania identycznego zachowania do wszystkich aplikacji w organizacji albo z powodu dostępu do oryginalnego kodu.

Przechwytywacz `ProducerInterceptor` Kafki obejmuje dwie kluczowe metody:

```
ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)
```

Metoda wywołana przed wysłaniem do Kafki utworzonego rekordu (a tak właściwie przed jego serializacją). Nadpisując tę metodę, można przechwytywać informacje o wysłanym rekordzie, a nawet je modyfikować. Należy tylko pamiętać, aby zwrócić z tej metody prawidłowy `ProducerRecord`. Zwracany rekord jest serializowany i wysyłany do Kafki.

```
void onAcknowledgement(RecordMetadata metadata, Exception exception)
```

Metoda wywołana pod warunkiem, że Kafka odpowie potwierdzeniem wysłania. Nie pozwala na modyfikację odpowiedzi Kafki, ale można przechwytywać informacje dotyczące odpowiedzi.

Do typowych przypadków użycia przechwytywaczy producentów należą: przechwytywanie informacji z monitorowania i śledzenia, wzbogacanie komunikatów o standardowe nagłówki, zwłaszcza w celu śledzenia pochodzenia, oraz redagowanie poufnych informacji.

Oto przykład bardzo prostego przechwytywacza producenta. Zlicza on wysłane komunikaty i potwierdzenia otrzymane w określonych oknach czasowych:

```
public class CountingProducerInterceptor implements ProducerInterceptor {

    ScheduledExecutorService executorService =
        Executors.newSingleThreadScheduledExecutor();
    static AtomicLong numSent = new AtomicLong(0);
    static AtomicLong numAcked = new AtomicLong(0);

    public void configure(Map<String, ?> map) {
        Long windowSize = Long.valueOf(
            (String) map.get("counting.interceptor.window.size.ms")); ❶
        executorService.scheduleAtFixedRate(CountingProducerInterceptor::run,
            windowSize, windowSize, TimeUnit.MILLISECONDS);
    }

    public ProducerRecord onSend(ProducerRecord producerRecord) {
        numSent.incrementAndGet();
        return producerRecord; ❷
    }

    public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
        numAcked.incrementAndGet(); ❸
    }

    public void close() {
        executorService.shutdownNow(); ❹
    }

    public static void run() {
        System.out.println(numSent.getAndSet(0));
        System.out.println(numAcked.getAndSet(0));
    }
}
```

- ❶ `ProducerInterceptor` to interfejs `Configurable`. Metodę `configure` można nadpisać i ustawić przed wywołaniem jakiejkolwiek innej metody. Ta metoda odbiera całą konfigurację producenta i można uzyskać dostęp do dowolnego parametru konfiguracyjnego. W tym przypadku dodaliśmy własną konfigurację, do której się tutaj odwołujemy.
- ❷ Po wysłaniu rekordu zwiększamy liczbę rekordów i zwracamy rekord bez jego modyfikacji.
- ❸ Kiedy Kafka odpowiada potwierdzeniem, zwiększamy liczbę potwierdzeń i nie musimy niczego zwracać.
- ❹ Ta metoda jest wywoływana, gdy producent kończy działanie, co daje nam szansę na wyczyszczenie stanu przechwytywacza. W tym przypadku zamykamy utworzony przez nas wątek. Gdybyśmy otworzyli uchwyt plików, połączenia ze zdalnymi magazynami danych itp., w tym miejscu moglibyśmy to wszystko zamknąć i uniknąć wycieków.

Jak wspomnieliśmy wcześniej, przechwytywacze producenta mogą być stosowane bez żadnych zmian w kodzie klienta. Aby użyć powyższego przechwytywacza z producentem `kafka-console-producer`, przykładową aplikacją dostarczaną z platformą Apache Kafka, wykonaj następujące trzy proste kroki:

1. Dodaj swój *jar* do zmiennej środowiskowej CLASSPATH:

```
export CLASSPATH=$CLASSPATH:~/target/CountProducerInterceptor-1.0-SNAPSHOT.jar
```

2. Utwórz plik konfiguracyjny, który zawiera poniższe ustawienia:

```
interceptor.classes=com.shapira.examples.interceptors.CountProducerInterceptor
counting.interceptor.window.size.ms=10000
```

3. Uruchom aplikację w normalny sposób, ale pamiętaj o uwzględnieniu konfiguracji utworzonej w poprzednim kroku:

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic interceptor-test --
producer.config producer.config
```

Kwoty i dławienie przepływności

Brokery Kafki mają możliwość ograniczenia szybkości produkowania i konsumowania komunikatów. Odbywa się to za pomocą mechanizmu kwot. Kafka ma trzy rodzaje kwot: produkcji, konsumpcji i żądań. Kwoty produkcji i konsumpcji ograniczają szybkość (mierzoną w bajtach na sekundę), z jaką klienci mogą wysyłać i odbierać dane. Kwoty żądań określają, jaki procent czasu broker ma poświęcać na przetwarzanie żądań klientów.

Kwoty można zastosować do wszystkich klientów, ustawiając domyślne wartości, określone identyfikatory klientów, określonych użytkowników lub oba te elementy. Kwoty charakterystyczne dla poszczególnych użytkowników są brane pod uwagę tylko w klastrach, w których skonfigurowano zabezpieczenia, a klienci się uwierzytelniają.

Domyślne kwoty produkcji i konsumpcji stosowane do wszystkich klientów są częścią pliku konfiguracyjnego brokera Kafki. Aby ograniczyć np. każdemu producentowi wysyłanie do 2 MB/s, należy do pliku konfiguracyjnego brokera dodać ustawienie `quota.producer.default=2M`.

Chociaż nie jest to zalecane, można również skonfigurować dla konkretnych klientów określone kwoty, nadpisujące wartości domyślne z pliku konfiguracyjnego brokera. W celu umożliwienia klientowi `klientA` produkowania 4 MB/s, a klientowi `klientB` produkowania 10 MB/s można użyć konfiguracji `quota.producer.override="klientA:4M,klientB:10M"`.

Kwoty określone w pliku konfiguracyjnym Kafki są statyczne i można je modyfikować tylko przez zmianę tej konfiguracji, a następnie zrestartowanie wszystkich brokerów. Ponieważ w dowolnym momencie mogą się pojawić nowe klienty, jest to bardzo niewygodne. Dlatego zwyczajową metodą stosowania kwot dla określonych klientów jest konfiguracja dynamiczna, którą można ustawić za pomocą skryptu `kafka-config.sh` lub interfejsu AdminClient API.

Przyjrzyjmy się kilku przykładom:

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config
'producer_byte_rate=1024' --entity-name klientC --entity-type clients ❶
```

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config
'producer_byte_rate=1024,consumer_byte_rate=2048' --entity-name użytkownik1
--entity-type users ❷
```

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config
'consumer_byte_rate=2048' --entity-type users
```

- ❶ Ograniczenie klienta `client` (identyfikowanego za pośrednictwem `id` klienta) do produkowania maksymalnie 1024 bajtów na sekundę.
- ❷ Ograniczenie użytkownika `userId` (identyfikowanego za pośrednictwem uwierzytelnionego pryncypała) do wytwarzania maksymalnie 1024 bajtów na sekundę i konsumowania maksymalnie 2048 bajtów na sekundę.
- ❸ Ograniczenie wszystkich użytkowników do konsumowania 2048 bajtów na sekundę. Nie będzie to dotyczyło użytkowników ze szczegółowym nadpisaniem. Jest to sposób na dynamiczną modyfikację domyślnych kwot.

Gdy klient wyczerpie kwotę, broker zaczyna dławić przepływność żądań klienta, aby zapobiec przekroczeniu limitu. Oznacza to, że broker będzie opóźniał odpowiedzi na żądania klientów; u większości klientów spowoduje to automatyczne zmniejszenie liczby żądań (ponieważ liczba żądań w toku będzie ograniczona) i obniżenie ruchu klienta do poziomu dozwolonego przez kwotę. W celu ochrony przed niewłaściwie zachowującymi się klientami wysyłającymi dodatkowe żądania podczas dławienia przepływności broker będzie również wyciszał kanał komunikacji z klientem na czas potrzebny do osiągnięcia zgodności z kwotą.

Zachowanie dławienia jest przekazywane klientom za pośrednictwem atrybutów `produce-time-avg`, `produce-time-max` i `fetch-time-max`, określających średni i maksymalny czas opóźnienia żądań produkcji i żądań pobierania z powodu ograniczania przepływności. Należy pamiętać, że ten czas może reprezentować ograniczenia przepływności związane z kwotami produkcji i konsumpcji, kwotami czasu żądania lub obu tych limitów. Inne typy żądań klientów można ograniczać tylko ze względu na kwoty czasu żądania, które również są przekazywane za pomocą podobnych wskaźników.



Jeśli będziesz używać asynchronicznej metody `Producer.send()` i kontynuować wysyłanie komunikatów z szybkością wyższą niż akceptowana przez broker (z powodu ustawionych kwot lub po prostu zwykłych ograniczeń pojemności), komunikaty będą najpierw umieszczane w kolejce w pamięci klienta. Jeżeli szybkość wysyłania będzie pozostawać wyższa niż szybkość przyjmowania komunikatów, klientowi w końcu zabraknie miejsca w buforze do przechowywania nadmiarowych komunikatów i zablokuje on kolejne wywołanie `Producer.send()`. Gdy zwiększenie limitu czasu będzie niewystarczające, aby broker mógł dogonić producenta i zwolnić trochę miejsca w buforze, ostatecznie `Producer.send()` rzuci wyjątek `TimeoutException`. Ewentualnie niektóre rekordy, które zostały już umieszczone w partiach, będą czekać dłużej, niż określa `delivery.timeout.ms`, i wygasną, co spowoduje wykonanie wywołania zwrotnego `send()` z wyjątkiem `TimeoutException`. Dlatego ważne jest planowanie i monitorowanie, by mieć pewność, że wydajność brokera wraz z upływem czasu będzie odpowiadać szybkości, z jaką producenci będą wysyłać dane.

Podsumowanie

Ten rozdział rozpoczęliśmy od prostego przykładu producenta — zaledwie 10 linii kodu, które wysyłają zdarzenia do Kafki. Rozwinęliśmy ten przykład, dodając obsługę błędów i eksperymentując z produkcją synchroniczną i asynchroniczną. Następnie zbadaliśmy najważniejsze parametry konfiguracji producenta i pokazaliśmy, jak modyfikują one zachowanie producentów. Omówiliśmy

serializatory, które pozwalają nam kontrolować format zdarzeń zapisywanych w Kafce. Przyjrzelśmy się Avrze, jednemu z wielu sposobów serializacji zdarzeń, bardzo często używanemu z Kafką. Rozdział zakończyliśmy omówieniem partycjonowania w Kafce i przedstawieniem przykładu zaawansowanej niestandardowej techniki partycjonowania.

Skoro potrafisz już zapisywać zdarzenia w Kafce, z rozdziału 4. dowiesz się wszystkiego o konsumowaniu zdarzeń z Kafki.

Konsumenty Kafki

— odczytywanie danych z Kafki

Aplikacje, które muszą odczytywać dane z Kafki, używają konsumenta `KafkaConsumer`, by subskrybować tematy Kafki i otrzymywać komunikaty z tych tematów. Odczytywanie danych z Kafki różni się nieco od odczytywania danych z innych systemów wymiany komunikatów i wiąże się z tym kilka unikatowych koncepcji i pomysłów. Zrozumienie sposobu korzystania z interfejsu API konsumenta (Consumer API) bez uprzedniego zrozumienia tych pojęć może być trudne. Zaczniemy od wyjaśnienia kilku ważnych koncepcji, a następnie przeanalizujemy przykłady, które pokażą różne sposoby wykorzystania interfejsów API konsumenta do implementowania aplikacji o odmiennych wymaganiach.

Koncepcje konsumentenckie Kafki

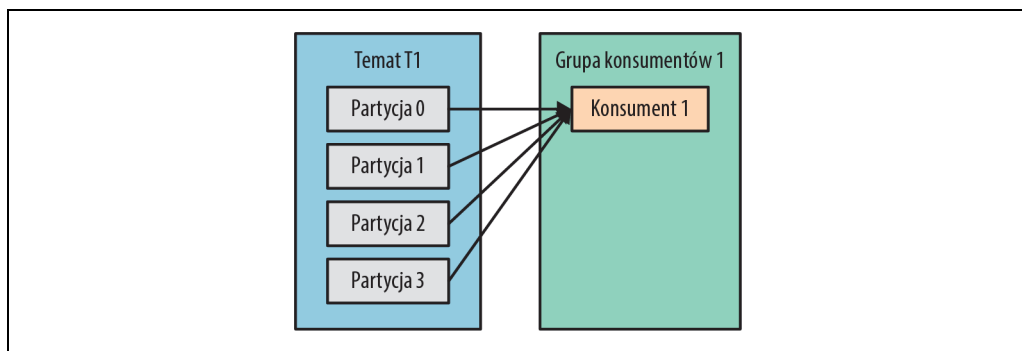
Aby zrozumieć, jak należy odczytywać dane z Kafki, najpierw trzeba poznać działanie jej konsumentów i grup konsumentów. Te koncepcje omówimy w kolejnych punktach.

Konsumenty i grupy konsumentów

Żałujemy, że masz aplikację, która musi odczytywać komunikaty z tematu Kafki, przeprowadzać ich walidację i zapisywać wyniki w innym magazynie danych. W takim przypadku aplikacja tworzy obiekt konsumenta, subskrybuje odpowiedni temat i zaczyna odbierać komunikaty, sprawdzać ich poprawność i zapisywać wyniki. Może to działać przez jakiś czas, lecz co się stanie, jeśli szybkość zapisywania komunikatów w temacie przez producenci zacznie przekraczać szybkość, z jaką Twoja aplikacja będzie mogła je weryfikować? Jeżeli jesteś ograniczony do pojedynczego konsumenta odczytującego i przetwarzającego dane, aplikacja może zacząć pozostawać coraz bardziej w tyle, nie nadążając za szybkością przychodzących komunikatów. Oczywiście istnieje potrzeba skalowania konsumpcji z tematów. Podobnie jak wielu producentów może zapisywać w tym samym temacie, musisz umożliwić odczytywanie z tego samego tematu wielu konsumentom, dzieląc dane między nich.

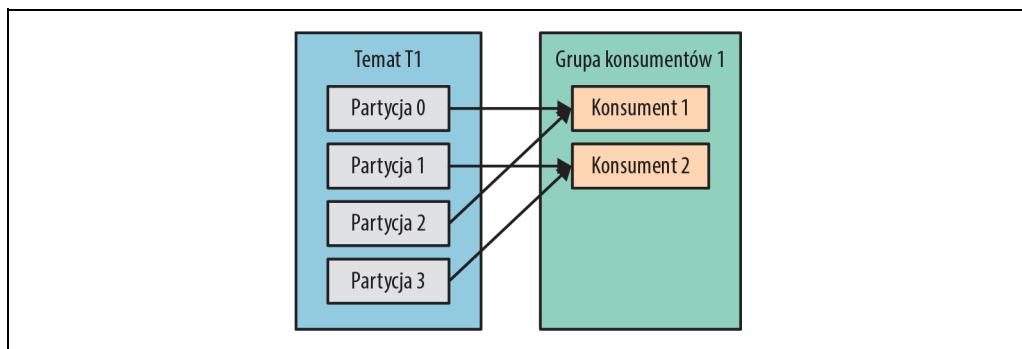
Konsumenty Kafki są zazwyczaj częścią **grupy konsumentów** (ang. *consumer group*). Gdy wiele konsumentów subskrybuje określony temat i należy do tej samej grupy konsumentów, każdy konsument z grupy będzie otrzymywał komunikaty z innego podzbioru partycji z tematu.

Przyjmijmy, że mamy temat T1 z czterema partycjami. Załóżmy również, że utworzyliśmy nowego konsumenta K1, będącego jedynym konsumentem w grupie G1, i używamy go do subskrybowania tematu T1. Konsument K1 będzie otrzymywał wszystkie komunikaty ze wszystkich czterech partycji T1, jak pokazaliśmy na rysunku 4.1.



Rysunek 4.1. Jedna grupa konsumentów z czterema partycjami

Jeśli do grupy G1 dodamy kolejnego konsumenta K2, każdy z nich będzie otrzymywał komunikaty tylko z dwóch partycji. Być może komunikaty z partycji 0 i 2 będą trafiać do K1, a komunikaty z partycji 1 i 3 do K2, jak pokazaliśmy na rysunku 4.2.

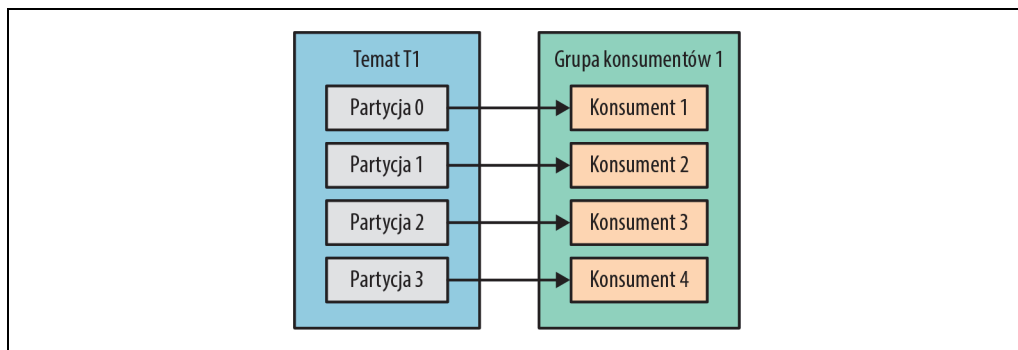


Rysunek 4.2. Cztery partycje podzielone na dwa konsumenty w grupie

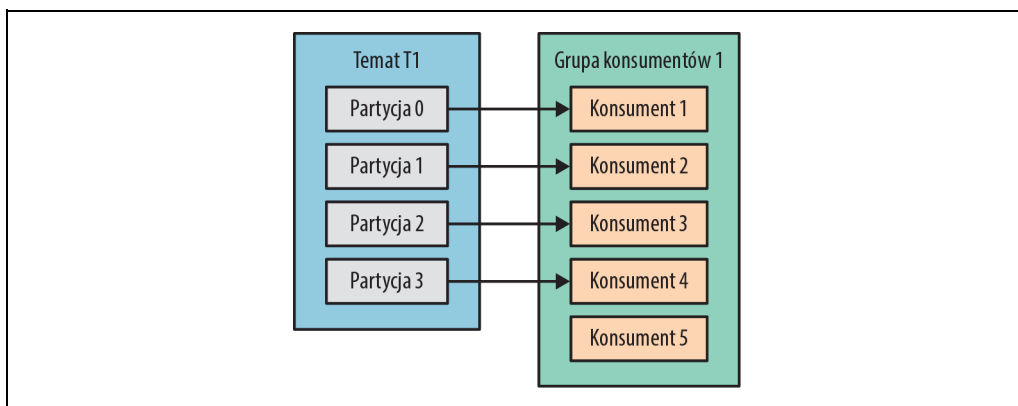
Gdyby grupa G1 miała czterech konsumentów, każdy z nich odczytywałby komunikaty z jednej partycji. Pokazaliśmy to na rysunku 4.3.

Jeżeli dodamy do pojedynczej grupy z jednym tematem więcej konsumentów, niż mamy partycji, niektóre konsumenty będą bezczynne i nie będą otrzymywać żadnych komunikatów. Pokazaliśmy to na rysunku 4.4.

Głównym sposobem skalowania konsumowania danych z tematu Kafki jest dodanie do grupy większej liczby konsumentów. Konsumenty Kafki często wykonują operacje charakteryzujące się dużym opóźnieniem, np. zapisy w bazie danych lub czasochłonne obliczenia na danych. W takich przypadkach pojedynczy konsument prawdopodobnie nie będzie w stanie nadążyć za prędkością przepływu



Rysunek 4.3. Cztery konsumenci w grupie; każdy ma po jednej partycji



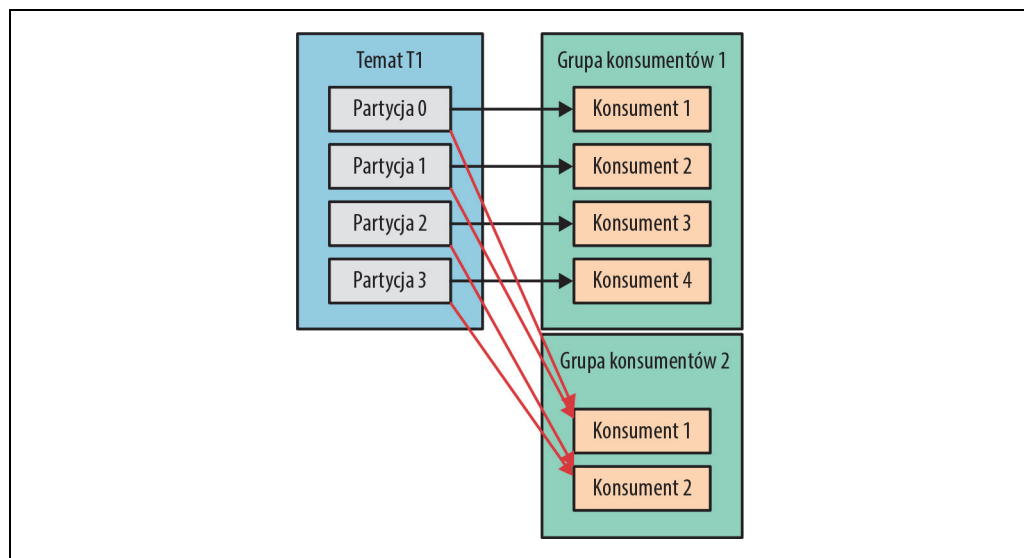
Rysunek 4.4. Gdy w grupie jest więcej konsumentów niż partycji, mamy bezczynne konsumenty

danych do tematu, więc główną metodą skalowania jest dodanie większej liczby konsumentów współdzielących obciążenie w taki sposób, że każdy konsument jest właścicielem tylko podzbioru partycji i komunikatów. To dobry powód, aby tworzyć tematy z dużą liczbą partycji, ponieważ gdy obciążenie wzrośnie, można dodawać konsumenty. Pamiętaj, że nie ma sensu dodawanie więcej konsumentów, niż jest partycji w temacie — niektóre konsumenci byłyby po prostu bezczynne. Kilka sugestii dotyczących ustalania liczby partycji w temacie znajdziesz w rozdziale 2.

Oprócz dodawania konsumentów w celu skalowania pojedynczej aplikacji bardzo często zdarza się, że wiele aplikacji musi odczytywać dane z tego samego tematu. Tak naprawdę jednym z głównych celów projektowych w Kafce było umożliwienie wykorzystania danych produkowanych do tematów Kafki w wielu przypadkach użycia w całej organizacji. W takich sytuacjach chcemy, aby każda aplikacja otrzymywała wszystkie komunikaty, a nie tylko ich podzbiór. By mieć pewność, że aplikacja będzie otrzymywać wszystkie komunikaty z tematu, trzeba jej zagwarantować własną grupę konsumentów. Inaczej niż wiele tradycyjnych systemów wymiany komunikatów Kafka skaluje się do dużej liczby konsumentów i grup konsumentów bez zmniejszania wydajności.

Wróćmy do naszego przykładu. Jeśli dodamy nową grupę konsumentów (G2) z pojedynczym konsumentem, będzie on otrzymywał wszystkie komunikaty z tematu T1 bez względu na to, co będzie

robić G1. G2 może mieć więcej niż jeden konsument, a wtedy każdy z nich otrzyma podzbiór partycji, tak jak pokazaliśmy dla G1, ale G2 jako całość nadal będzie otrzymywać wszystkie komunikaty niezależnie od innych grup konsumentów. Pokazaliśmy to na rysunku 4.5.



Rysunek 4.5. Gdy dodamy nową grupę konsumentów, obie grupy będą otrzymywać wszystkie komunikaty

Możemy podsumować, że dla każdej aplikacji, która potrzebuje wszystkich komunikatów z co najmniej jednego tematu, tworzymy nową grupę konsumentów. Aby skalować odczytywanie i przetwarzanie komunikatów z tematów, do istniejącej grupy dodajemy kolejne konsumenty, dzięki czemu każdy dodatkowy konsument w grupie będzie otrzymywał tylko podzbiór komunikatów.

Grupy konsumentów i równoważenie partycji

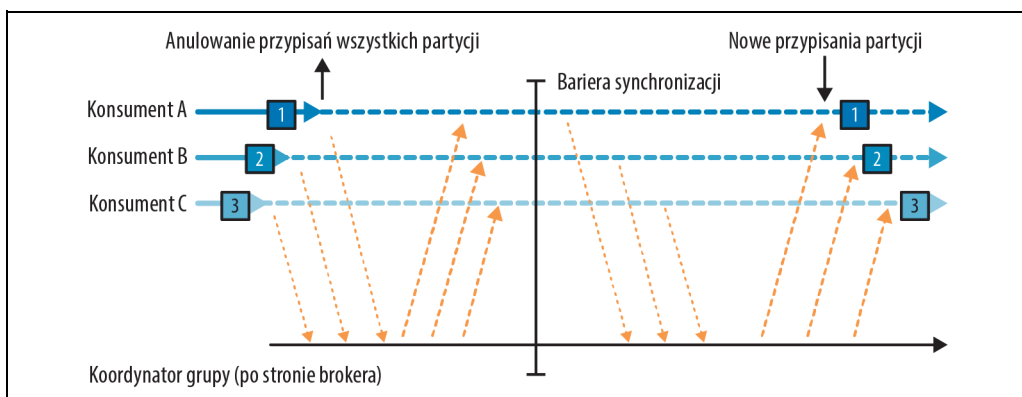
Jak dowiedziałeś się z poprzedniego punktu, konsumenci w grupie konsumentów mają udział we własności partycji w tematach, które subskrybują. Po dodaniu do grupy nowy konsument zaczyna konsumować komunikaty z partycji przetwarzanej wcześniej przez inny konsument. To samo dzieje się, gdy jakiś konsument się wyłącza lub ulega awarii, przez co opuszcza grupę, a przetwarzane przez niego partycje zaczynają być konsumowane przez jeden z pozostałych konsumentów. Ponowne przypisanie partycji do konsumentów ma miejsce również wtedy, gdy modyfikowane są tematy przetwarzane przez daną grupę konsumentów, np. kiedy administrator dodaje nowe partycje.

Przenoszenie własności partycji z jednego konsumenta na drugi nazywa się **równoważeniem obciążenia** lub **rebalansowaniem** (ang. *rebalance*). Równoważenie obciążenia jest istotne, ponieważ zapewnia grupie konsumentów wysoką dostępność i skalowalność (umożliwiając nam łatwe i bezpieczne dodawanie i usuwanie konsumentów), ale w normalnym toku zdarzeń może być niepożądane.

W zależności od strategii przypisywania partycji używanej przez daną grupę konsumentów dostępne są dwa rodzaje równoważenia obciążenia¹:

Gorliwe równoważenie obciążenia

Podczas gorliwego równoważenia obciążenia (ang. *eager rebalance*) wszystkie konsumenty wstrzymują przetwarzanie, rezygnują z własności wszelkich partycji, ponownie dołączają do danej grupy konsumentów i otrzymują zupełnie nowe przypisania partycji. Generalnie jest to krótkie okno niedostępności całej grupy konsumentów. Czas trwania okna niedostępności zależy od wielkości grupy konsumentów oraz od kilku parametrów konfiguracyjnych. Na rysunku 4.6 pokazaliśmy, że gorliwe równoważenie obciążenia ma dwie odrębne fazy: po pierwsze, wszystkie konsumenty rezygnują ze swoich przypisań partycji, a po drugie, po zakończeniu tego procesu i ponownym dołączeniu do grupy otrzymują nowe przypisania partycji i mogą wznowić konsumowanie komunikatów.



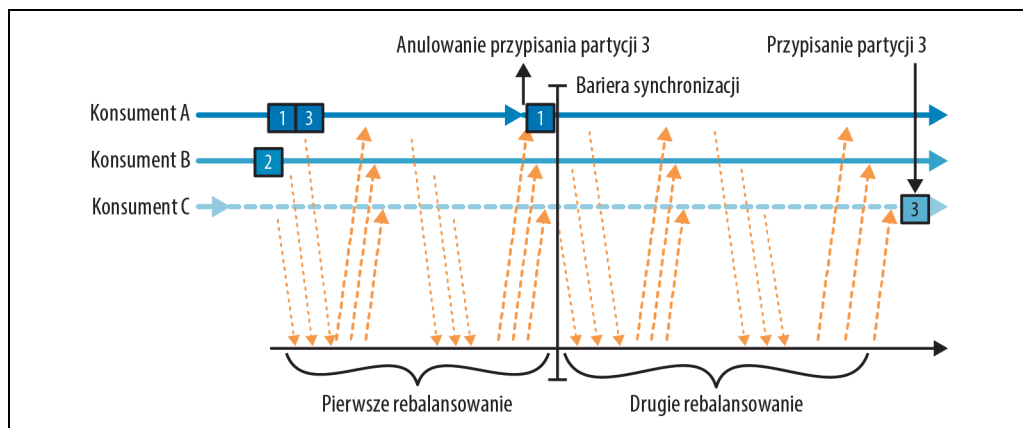
Rysunek 4.6. Gorliwe równoważenie obciążenia anuluje wszystkie przypisania partycji, wstrzymuje przetwarzanie i ponownie przypisuje partycje

Kooperacyjne równoważenie obciążenia

Kooperacyjne równoważenie obciążenia (nazywane również **równoważeniem przyrostowym**; ang. *incremental rebalance*) zazwyczaj polega na ponownym przypisywaniu tylko niewielkiego podzbioru partycji po kolei dla poszczególnych konsumentów i umożliwieniu konsumentom kontynuowania przetwarzania rekordów ze wszystkich partycji, które nie zostały ponownie przypisane. Osiąga się to przez rebalansowanie w co najmniej dwóch fazach. Na początku lider grupy konsumentów informuje wszystkie konsumenty, że tracą własność podzbioru swoich partycji, następnie konsumenty przestają przetwarzać komunikaty z tych partycji i rezygnują z ich własności. W drugiej fazie lider grupy konsumentów przypisuje te osierocone partycje nowym właścicielom. Takie przyrostowe podejście może wymagać wykonania kilku iteracji, dopóki nie zostanie osiągnięte stabilne przypisanie partycji, ale pozwala uniknąć całkowitej niedostępności („zatrzymania świata”), która występuje przy podejściu gorliwym. Jest to szczególnie ważne w dużych grupach konsumentów, w których równoważenie obciążenia może

¹ Diagramy Sophie Blee-Goldman z jej postu *From Eager to Smarter in Apache Kafka Consumer Rebalances* (<https://oreil.ly/fZzac>) na blogu Confluenta z maja 2020 r.

zając sporo czasu. Na rysunku 4.7 pokazaliśmy, na czym polega przyrostowość kooperacyjnego równoważenia obciążenia i że w dowolnym momencie zaangażowany jest tylko podzbiór konsumentów i partycji.



Rysunek 4.7. Kooperacyjne równoważenie obciążenia wstrzymuje przetwarzanie tylko dla podzbioru partycji, które są ponownie przypisywane

Konsumenci zachowują członkostwo w grupie konsumentów i własność przypisanych im partycji, wysyłając do brokera Kafka wyznaczonego jako **koordynator grupy** (ten broker może być różny dla różnych grup konsumentów) **sygnały heartbeat**. Sygnały są wysyłane przez działający w tle wątek konsumenta i dopóki konsument wysyła heartbeaty w regularnych odstępach czasu, zakłada się, że jest aktywny.

Jeśli konsument nie będzie wysyłał sygnału heartbeat przez dłuższy czas, to jego sesja przekroczy limit czasu, a koordynator grupy uzna go za nieaktywny i uruchomi rebalansowanie. Gdy konsument ulegnie awarii i przestanie przetwarzać komunikaty, upłynie kilka sekund bez wysyłania sygnału heartbeat, zanim koordynator grupy stwierdzi, że konsument przestał działać, i wywoła rebalansowanie. W ciągu tych kilku sekund nie będą przetwarzane żadne komunikaty z partycji należących do nieaktywnego konsumenta. W przypadku czystego zakończenia działania konsument powiadomi o tym koordynatora grupy, który natychmiast uruchamia rebalansowanie, skracając przerwę w przetwarzaniu komunikatów. Dalej w tym rozdziale omówimy opcje konfiguracyjne, które kontrolują częstotliwość sygnału heartbeat, limity czasu sesji i inne parametry, których można użyć w celu dostosowania zachowania konsumenta.



Jak działa proces przypisywania partycji konsumentom?

Gdy konsument chce dołączyć do grupy, wysyła do koordynatora grupy żądanie `JoinGroup`. Pierwszy konsument, który dołączy do grupy, zostaje jej **liderem**. Lider otrzymuje od koordynatora grupy listę wszystkich należących do niej konsumentów (lista obejmuje wszystkie konsumenty, które ostatnio wysłały sygnał heartbeat i w związku z tym są uznawane za aktywne) i staje się odpowiedzialny za przypisywanie podzbioru partycji do każdego konsumenta. Do decydowania, które partycje powinny być obsługiwane przez poszczególne konsumenty, używa implementacji klasy `PartitionAssignor`.

Kafka ma kilka wbudowanych reguł przypisywania partycji, które omówimy szerzej dalej w tym rozdziale. Po podjęciu decyzji o przypisaniu partycji lider grupy konsumentów wysyła listę przypisań do koordynatora grupy (GroupCoordinator), który przekazuje te informacje do wszystkich konsumentów. Każdy konsument zna tylko własne przypisanie — lider jest jedynym procesem klienckim, który ma pełną listę konsumentów w grupie i ich przypisań. Proces ten powtarza się za każdym razem, gdy uruchamiane jest równoważenie obciążenia.

Statyczne członkostwo w grupie

Domyślnie tożsamość konsumenta jako członka grupy konsumentów jest przejściowa. Gdy konsumenci opuszczają grupę, ich przypisanie partycji są anulowane, a po ponownym dołączeniu do grupy konsument otrzymuje nowy identyfikator członka i nowy zbiór partycji za pośrednictwem protokołu równoważenia obciążenia.

Nie ma to zastosowania, jeżeli skonfiguruje się konsument z unikatowym identyfikatorem `group.instance.id`, co czyni go **statycznym** członkiem grupy. Gdy konsument po raz pierwszy łączy do grupy konsumentów jako jej statyczny członek, przypisywany jest do niego zbiór partycji zgodnie ze strategią przypisywania partycji, z której w normalnym trybie korzysta grupa. Jednak po zakończeniu działania konsumenta nie opuszcza on automatycznie grupy — pozostaje członkiem grupy do czasu wygaśnięcia jego sesji. Gdy konsument ponownie łączy do grupy, jest rozpoznawany na podstawie swojej statycznej tożsamości i jest ponownie przypisywany do tych samych partycji, które miał wcześniej, bez wywoływania równoważenia obciążenia. Koordynator grupy buforujący przypisanie dla wszystkich członków grupy nie musi uruchamiać równoważenia obciążenia, a jedynie wysłać przypisanie z pamięci podręcznej do ponownie dołączającego statycznego członka.

Jeśli do tej samej grupy dołączają dwa konsumenci o tym samym identyfikatorze `group.instance.id`, drugi w kolejności konsument otrzyma komunikat błędu informujący, że konsument o tym identyfikatorze już istnieje.

Statyczne członkostwo w grupie jest przydatne, gdy aplikacja utrzymuje lokalny stan lub lokalną pamięć podręczną zapełnianą przez partycje przypisywane do poszczególnych konsumentów. Jeżeli odtworzenie tej pamięci podręcznej okaże się czasochłonne, nie jest pożądane, by ten proces następował przy każdym restarcie konsumenta. Z drugiej strony należy pamiętać, że przy restarcie jednego konsumenta partycje należące do poszczególnych konsumentów nie są ponownie przypisywane. Przez pewien czas żaden konsument nie będzie konsumował z komunikatów z tych partycji, a gdy dany konsument w końcu uruchomi kopię zapasową, będzie w tyle za najnowszymi komunikatami z tych partycji. Należy mieć pewność, że konsument będący właścicielem tych partycji będzie w stanie nadrobić opóźnienie powstałe po restarcie.

Trzeba zwrócić uwagę, że statyczni członkowie grup konsumentów przy zamykaniu nie opuszczają grupy proaktywnie, a wykrywanie, że „naprawdę ich nie ma”, zależy od konfiguracji parametru `session.timeout.ms`. Powinno się ustawić dla niego na tyle dużą wartość, by ustrzec się wyzwalania równoważenia obciążenia po zwykłym restarcie aplikacji, ale jednocześnie wartość ta powinna umożliwiać automatyczne ponowne przypisywanie danych partycji w przypadku dłuższego przestoju w celu uniknięcia dłuższych przerw w przetwarzaniu tych partycji.

Tworzenie konsumenta Kafki

Pierwszym krokiem do rozpoczęcia konsumowania rekordów jest utworzenie instancji klasy `KafkaConsumer`. Tworzenie obiektu `KafkaConsumer` bardzo przypomina konstruowanie obiektu `KafkaProducer` — stworzysz instancję klasy `Properties` Javy z właściwościami, które chcesz przekazać konsumentowi. Wszystkie właściwości omówimy szczegółowo dalej w tym rozdziale. Na początek wystarczy użyć trzech obowiązkowych właściwości: `bootstrap.servers`, `key.deserializer` i `value.deserializer`.

Właściwość `bootstrap.servers` to łańcuch znaków dla połączenia z klastrem Kafki. Używa się go dokładnie w taki sam sposób, jak w przypadku instancji `KafkaProducer` (szczegółowe informacje na temat jego definiowania znajdziesz w rozdziale 3.). Pozostałe dwie właściwości, `key.deserializer` i `value.deserializer`, są podobne do serializerów definiowanych dla producenta, ale zamiast określania klas przekształcających obiekty Javy w tablice bajtów należy określić klasy przekształcające tablice bajtów w obiekty Javy.

Istnieje też czwarta właściwość, która nie jest bezwzględnie obowiązkowa, ale bardzo powszechnie stosowana. Jest to właściwość `group.id` określająca grupę konsumentów, do której należy dana instancja `KafkaConsumer`. Chociaż możliwe jest tworzenie konsumentów, które nie należą do żadnej grupy konsumentów, jest to stosunkowo rzadkie, dlatego dalej w tym rozdziale będziemy przyjmować założenie, że konsument jest częścią jakiejś grupy.

Poniższy fragment kodu pokazuje, jak utworzyć obiekt `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

Jeśli czytałeś rozdział 3. poświęcony tworzeniu producentów, większość tego kodu powinna wyglądać znajomo. Zakładamy, że konsumowane rekordy będą miały obiekty typu `String` zarówno jako klucz, jak i wartość rekordu. Jedyną nową właściwością jest tutaj `group.id`, czyli nazwa grupy konsumentów, do której należy dany konsument.

Subskrybowanie tematów

Następnym krokiem po utworzeniu konsumenta jest zasubskrybowanie co najmniej jednego tematu. Metoda `subscribe()` przyjmuje jako parametr listę tematów, więc jest całkiem prosta w użyciu:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Tworzymy listę z jednym elementem: nazwą tematu `customerCountries`.

Możliwe jest również wywoływanie `subscribe` z wyrażeniem regularnym. Wyrażenie może dopasowywać wiele nazw tematów, a jeśli zostanie utworzony nowy temat o pasującej nazwie, niemal

natychmiast przeprowadzone zostanie rebalansowanie, a konsumenci zaczną go przetwarzać. Jest to przydatne w aplikacjach, które muszą korzystać z wielu tematów i mogą obsługiwać różne typy danych, które będą zawierały tematy. Subskrybowanie wielu tematów za pomocą wyrażenia regularnego jest najczęściej używane w aplikacjach replikujących dane między Kafką a innym systemem lub w aplikacjach przetwarzania strumieniowego.

Aby zasubskrybować np. wszystkie tematy testowe, możemy wykonać takie wywołanie:

```
consumer.subscribe(Pattern.compile("test.*"));
```



Jeśli Twój klaster Kafki ma dużą liczbę partycji, np. 30 000 lub więcej, powinieneś mieć świadomość, że filtrowanie tematów pod kątem subskrypcji odbywa się po stronie klienta. Oznacza to, że gdy zasubskrybujesz podzbiór tematów za pomocą wyrażenia regularnego, a nie za pomocą bezpośredniej listy, konsument będzie w regularnych odstępach czasu żądał od brokera listy wszystkich tematów i ich partycji. Klient będzie następnie używał tej listy do wykrywania nowych tematów, które powinien uwzględnić i zasubskrybować. Gdy lista tematów jest długa i masz wiele konsumentów, wówczas rozmiar listy tematów i partycji jest duży, a subskrybowanie przy użyciu wyrażenia regularnego generuje znaczne obciążenie brokera, klienta i sieci. W niektórych przypadkach przepustowość wykorzystywana przez metadane tematu jest większa niż przepustowość wykorzystywana do wysyłania danych. Oznacza to również, że w celu subskrybowania za pomocą wyrażenia regularnego klient potrzebuje uprawnień do opisywania wszystkich tematów w klastrze — czyli pełnych uprawnień `describe` dla całego klastra.

Pętla odpytywania

Sercem interfejsu Consumer API jest prosta pętla do odpytywania serwera w celu pozyskiwania kolejnych danych. Główne ciało konsumenta będzie wyglądało następująco:

```
Duration timeout = Duration.ofMillis(100);

while (true) { ❶
    ConsumerRecords<String, String> records = consumer.poll(timeout); ❷

    for (ConsumerRecord<String, String> record : records) { ❸
        System.out.printf("temat = %s, partycja = %d, przesunięcie = %d, " +
            "klient = %s, kraj = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);

        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()); ❹
    }
}
```

- ❶ Jest to w rzeczywistości pętla nieskończona. Konsumenty to zwykle długo działające aplikacje, które nieustannie odpytują Kafkę, by pozyskiwać kolejne dane. Dalej w tym rozdziale pokażemy, jak czysto wyjść z pętli i zamknąć konsument.
- ❷ To najważniejsza linia kodu w rozdziale. Podobnie jak rekiny muszą się poruszać albo umrzeć, konsumenci muszą stale odpytywać Kafkę, gdyż inaczej zostaną uznane za nieaktywne, a konsumowane przez nie partycje zostaną przekazane innemu konsumentowi z grupy, by kontynuował ich przetwarzanie. Parametrem przekazywanym do metody `poll()` jest interwał limitu czasu, który kontroluje, jak długo metoda będzie blokować, jeśli dane nie będą dostępne w buforze konsumenta. Gdy ustawiona jest wartość 0 lub istnieją już dostępne rekordy, metoda `poll()` natychmiast kończy wykonywanie; w przeciwnym razie będzie czekać przez określoną liczbę milisekund.
- ❸ Metoda `poll()` zwraca listę rekordów. Każdy rekord zawiera temat i partycję swojego pochodzenia, przesunięcie rekordu w obrębie partycji oraz rzecz jasna klucz i wartość rekordu. Zazwyczaj iterujemy przez listę i przetwarzamy poszczególne rekordy indywidualnie.
- ❹ Przetwarzanie zwykle kończy się zapisaniem wyniku w magazynie danych lub aktualizacją przechowywanego rekordu. W tym przypadku celem jest utrzymywanie bieżącej liczby klientów z poszczególnych krajów, dlatego aktualizujemy tablicę mieszającą i wypisujemy wynik w formacie JSON. Bardziej realistycznym przykładem byłoby zapisywanie wyników aktualizacji w magazynie danych.

Pętla `poll` nie zajmuje się jedynie pobieraniem danych. Przy pierwszym wywołaniu z nowym konsumentem metoda `poll()` jest odpowiedzialna za znalezienie instancji `GroupCoordinator`, dołączenie konsumenta do grupy i odebranie przypisania partycji. Jeśli uruchomione zostanie równoważenie obciążenia, również zostanie ono obsłużone w pętli odpytywania, łącznie z powiązanymi wywołaniami zwrotnymi. Oznacza to, że prawie wszystko, co może pójść nie tak z konsumentem lub w wywołaniach zwrotnych używanych przez jego nasłuchiwalce, prawdopodobnie wypłynie jako wyjątek rzucony przez `poll()`.

Należy pamiętać, że jeśli metoda `poll()` nie będzie wywoływana przez okres dłuższy niż `max.poll.interval.ms`, konsument zostanie uznany za nie działający i usunięty z grupy konsumentów, unikaj więc robienia wszystkiego, co może blokować wykonywanie w pętli odpytywania przez nieprzewidywalne przedziały czasu.

Bezpieczeństwo wątków

W jednym wątku nie możesz mieć wielu konsumentów należących do tej samej grupy i nie możesz mieć wielu wątków bezpiecznie korzystających z tego samego konsumenta. Regułą jest jeden konsument na wątek. Aby uruchomić wielu konsumentów w tej samej grupie w jednej aplikacji, każdego musisz uruchomić w osobnym wątku. Przydatne jest opakowanie logiki konsumenta we własny obiekt, a następnie użycie usługi `ExecutorService` Javy do uruchomienia wielu wątków, z których każdy będzie miał własnego konsumenta. Na blogu Confluenta znajdziesz samouczek (<https://oreil.ly/8YOVe>), który pokazuje, jak to zrobić.



W starszych wersjach Kafki pełną sygnaturą metody było `poll(long)`; ta sygnatura jest już przestarzała, a nowy interfejs API to `poll(Duration)`. Oprócz zmiany typu argumentu nieznacznie zmieniła się semantyka blokowania wykonywana przez tę metodę. Oryginalna metoda `poll(long)` będzie blokować, dopóki nie uzyska z Kafki potrzebnych metadanych, nawet jeśli przekroczy limit czasu. Nowa metoda `poll(Duration)` będzie przestrzegać ograniczeń limitu czasu i nie będzie czekać na metadane. Jeżeli masz kod konsumenta wykorzystujący `poll(0)` jako metodę zmuszającą Kafkę do pobrania metadanych bez konsumowania jakichkolwiek rekordów (dość powszechna sztuczka), nie możesz zmienić go po prostu na `poll(Duration.ofMillis(0))` i oczekiwać tego samego zachowania. Musisz wymyślić nowy sposób na osiągnięcie swoich celów. Częstym rozwiązaniem jest umieszczenie logiki w metodzie `rebalancerListener.onPartitionAssignment()`, której wywołanie jest gwarantowane po uzyskaniu metadanych dla przypisanych partycji, ale przed rozpoczęciem napływania rekordów. Kolejne rozwiązanie zostało udokumentowane przez Jessego Andersona na blogu w poście *Kafka's Got a Brand-New Poll* (<https://oreil.ly/zN6ek>).

Innym możliwym podejściem może być zapełnianie kolejki zdarzeń przez jednego konsumenta i wykonywanie pracy z tej kolejki przez wiele wątków roboczych. Przykład tego wzorca znajdziesz na blogu w poście Igora Buzatovicia (<https://oreil.ly/uMzj1>).

Konfiguracja konsumentów

Do tej pory koncentrowaliśmy się na nauce interfejsu Consumer API i przyjrzelśmy się tylko kilku właściwościom konfiguracyjnym — obowiązkowym `bootstrap.servers`, `key.deserializer` i `value.deserializer` oraz powszechnie stosowanej `group.id`. Cała konfiguracja konsumenta została opisana w dokumentacji Apache Kafki (<https://oreil.ly/Y00GI>). Większość parametrów ma rozsądne wartości domyślne i nie wymaga modyfikacji, ale niektóre mają wpływ na wydajność i dostępność konsumentów. Przyjrzyjmy się kilku ważniejszym właściwościom.

`fetch.min.bytes`

Ta właściwość umożliwia konsumentowi określenie minimalnej ilości danych, które chce otrzymywać od brokera podczas pobierania rekordów — domyślnie jeden bajt. Jeżeli broker otrzyma od konsumenta żądanie rekordów, ale nowe rekordy będą miały mniej bajtów niż `fetch.min.bytes`, to przed wysłaniem rekordów do konsumenta broker poczeka, aż dostępnych będzie więcej komunikatów. Zmniejsza to obciążenie zarówno konsumenta, jak i brokera, ponieważ muszą wymieniać między sobą mniejszą liczbę komunikatów w przypadkach, gdy w tematach nie będzie zbyt dużo nowej aktywności (lub w godzinach zmniejszonej aktywności w ciągu dnia). Jeżeli konsument zużywa zbyt dużo procesora, gdy nie ma zbyt wielu dostępnych danych, lub potrzebujesz zmniejszyć obciążenie brokerów, gdy masz dużą liczbę konsumentów, możesz ustawić dla tego parametru wartość wyższą niż domyślna — pamiętaj jednak, że zwiększenie tej wartości może zwiększyć opóźnienia w przypadkach niskiej przepustowości.

fetch.max.wait.ms

Ustawienie `fetch.min.bytes` instruuje Kafkę, by przed przekazaniem konsumentowi odpowiedzi poczekała, aż będzie miała wystarczającą ilość danych do wysłania. Właściwość `fetch.max.wait.ms` pozwala kontrolować czas oczekiwania. Domyślnie Kafka czeka do 500 ms. Powoduje to do 500 ms dodatkowego opóźnienia w przypadku, gdy do tematu Kafki nie przepływa wystarczająca ilość danych, by spełnić wymagania minimalnej ilości danych do zwrócenia. Jeśli chcesz ograniczyć potencjalne opóźnienie (zwykle ze względu na umowy SLA kontrolujące maksymalne opóźnienia aplikacji), możesz ustawić niższą wartość `fetch.max.wait.ms`. Jeżeli dla `fetch.max.wait.ms` ustawisz wartość 100 ms, a dla `fetch.min.bytes` wartość 1 MB, to po otrzymaniu od konsumenta żądania pobrania Kafka odpowie danymi, gdy będzie miała 1 MB danych do zwrócenia lub po upływie 100 ms, w zależności od tego, co nastąpi wcześniej.

fetch.max.bytes

Ta właściwość pozwala określić maksymalną liczbę bajtów, które Kafka będzie zwracać przy każdym odpytywaniu brokera przez konsument (domyślnie 50 MB). Służy do ograniczania rozmiaru pamięci używanej przez konsument do przechowywania danych zwracanych z serwera, niezależnie od liczby partycji lub komunikatów. Zwróć uwagę, że rekordy są wysyłane do klienta w partiach, a jeśli pierwsza partia rekordów, którą ma wysłać broker, przekroczy ten rozmiar, partia zostanie wysłana, a limit zignorowany. Gwarantuje to konsumentowi robienie ciągłych postępów. Warto zauważyć, że istnieje analogiczna konfiguracja brokera, która pozwala administratorowi Kafki ograniczyć także maksymalny rozmiar pobierania. Konfiguracja brokera może być przydatna, ponieważ żądania dużych ilości danych mogą się wiązać z dużymi odczytami z dysku i długimi wysyłkami przez sieć, co może powodować rywalizację i zwiększać obciążenie brokera.

max.poll.records

Ta właściwość kontroluje maksymalną liczbę rekordów, które zwróci pojedyncze wywołanie `poll()`. Używaj jej do kontrolowania ilości danych (ale nie rozmiaru danych), które aplikacja będzie musiała przetworzyć w jednej iteracji pętli odpytywania.

max.partition.fetch.bytes

Właściwość kontroluje maksymalną liczbę bajtów, które serwer będzie zwracał na partycję (domyślnie 1 MB). Gdy wywołanie `KafkaConsumer.poll()` zwraca obiekt `ConsumerRecords`, używa on co najwyżej `max.partition.fetch.bytes` na partycję przypisaną do konsumenta. Zwróć uwagę, że kontrolowanie wykorzystania pamięci za pomocą tej konfiguracji może być dość skomplikowane, ponieważ nie masz kontroli nad liczbą partycji, które zostaną uwzględnione w odpowiedzi brokera. Dlatego zdecydowanie zalecamy zamiast tego użycie właściwości `fetch.max.bytes`, chyba że masz specjalne powody, by próbować przetwarzać podobne ilości danych z każdej partycji.

session.timeout.ms i heartbeat.interval.ms

Czas, przez który konsument może nie mieć kontaktu z brokerami i nadal być uznawany za aktywny, wynosi domyślnie 10 sekund. Jeśli upłynie więcej czasu, niż określa `session.timeout.ms`, a konsument nie wyśle do koordynatora grupy sygnału `heartbeat`, zostanie uznany za nieaktywny i koordynator grupy uruchomi rebalansowanie grupy konsumentów w celu przydzielenia partycji nieaktywnego konsumenta innym konsumentom z grupy. Ta właściwość jest ściśle powiązana z `heartbeat.interval.ms`, która kontroluje częstotliwość wysyłania przez konsumenta Kafka sygnału `heartbeat` do koordynatora grupy, podczas gdy `session.timeout.ms` kontroluje, jak długo konsument może działać bez wysyłania tego sygnału. Z tego powodu te dwie właściwości są zwykle modyfikowane razem — wartość `heartbeat.interval.ms` musi być mniejsza niż wartość `session.timeout.ms` i z reguły ustawia się dla niej jedną trzecią limitu czasu. Stąd jeśli `session.timeout.ms` wynosi 3 sekundy, wartość `heartbeat.interval.ms` powinna wynosić 1 sekundę. Ustawienie `session.timeout.ms` poniżej wartości domyślnej umożliwi grupom konsumentów szybsze wykrywanie awarii i odzyskiwanie sprawności po awarii, ale może również spowodować niepożądane rebalansowanie. Ustawienie wyższej wartości `session.timeout.ms` zmniejsza ryzyko przypadkowego rebalansowania, ale oznacza również, że wykrywanie rzeczywistych awarii będzie zajmowało więcej czasu.

max.poll.interval.ms

Właściwość pozwala ustawić przedział czasu, przez który konsument może działać bez odpytywania i nie zostać uznany za nieaktywny. Jak wspomnieliśmy wcześniej, sygnał `heartbeat` i limit czasu sesji stanowią główny mechanizm, dzięki któremu Kafka wykrywa nieaktywne konsumenty i usuwa ich przydziały partycji. Wspomnieliśmy jednak również, że sygnały `heartbeat` są wysyłane przez wątek działający w tle. Istnieje ryzyko, że główny wątek konsumujący z Kafka zostanie zakleszczony, ale wątek w tle nadal będzie wysyłał sygnały `heartbeat`. Będzie to oznaczać, że rekordy z partycji należących do tego konsumenta nie będą przetwarzane. Najłatwiejszym sposobem zweryfikowania, czy konsument nadal przetwarza rekordy, jest sprawdzenie, czy prosi o kolejne rekordy. Interwały między żądaniami kolejnych rekordów są jednak trudne do przewidzenia i zależą od ilości dostępnych danych, rodzaju przetwarzania wykonywanego przez konsumenta, a czasem od opóźnienia dodatkowych usług. W aplikacjach wymagających czasochłonnego przetwarzania każdego zwróconego rekordu parametr `max.poll.records` pozwala ograniczać ilości zwracanych danych, a tym samym skracać czas, w którym aplikacja nie jest przejściowo dostępna dla wywołań metody `poll()`. Nawet po zdefiniowaniu `max.poll.records` interwał między wywołaniami `poll()` jest trudny do przewidzenia, a parametr `max.poll.interval.ms` jest stosowany jako zabezpieczenie przed awarią. Musi to być na tyle duży interwał, by zdrowy konsument rzadko w pełni go wykorzystywał, ale jednocześnie na tyle mały, aby uniknąć znaczącego wpływu wywieranego przez zawieszony konsument. Domyślną wartością jest 5. Po przekroczeniu tego limitu czasu wątek działający w tle wysyła żądanie „opuść grupę”, aby poinformować broker, że konsument jest nieaktywny i obciążenie dla grupy musi zostać zrównoważone, a następnie przestaje wysyłać sygnały `heartbeat`.

default.api.timeout.ms

Jest to limit czasu, który będzie miał zastosowanie do (niemal) wszystkich wywołań interfejsu API wykonanych przez konsument, gdy nie określisz dla nich bezpośredniego limitu czasu. Wartość domyślna to 1 minuta, a ponieważ jest ona większa niż domyślny limit czasu żądania, uwzględnia ponowienie próby, jeśli zachodzi taka potrzeba. Godnym uwagi wyjątkiem od interfejsów API używających tej wartości domyślnej jest metoda `poll()`, która zawsze wymaga bezpośredniego limitu czasu.

request.timeout.ms

Jest to maksymalny czas oczekiwania konsumenta na odpowiedź od brokera. Jeżeli broker nie odpowie w tym czasie, klient założy, że broker nie odpowie w ogóle, zamknie połączenie i spróbuje połączyć się ponownie. Wartość domyślna tej konfiguracji wynosi 30 sekund i nie zaleca się jej obniżania. Przed rozłączeniem się trzeba pozostawić brokerowi wystarczająco dużo czasu na przetworzenie żądania — ponawianie wysyłania żądań do już przeciążonego brokera niewiele daje, a czynność rozłączania i ponownego łączenia generuje dodatkowe obciążenie.

auto.offset.reset

Ta właściwość steruje zachowaniem konsumenta, gdy zaczyna on odczytywać partycję, dla której nie ma zatwierdzonego przesunięcia, lub jeśli zatwierdzone przesunięcie jest nieprawidłowe (zwykle jest to spowodowane nieaktywnością konsumenta przez tak długi czas, że rekord z danym przesunięciem został już zdezaktualizowany przez broker). Wartość domyślna `latest` (najnowsze) oznacza, że przy braku prawidłowego przesunięcia konsument zacznie odczytywanie od najnowszych rekordów (czyli tych, które zostały zapisane po tym, jak konsument zaczął działać). Dostępna jest również wartość `earliest` (najstarsze), która oznacza, że w przypadku braku prawidłowego przesunięcia konsument będzie odczytywał wszystkie dane z partycji, zaczynając od samego początku. Ustawienie dla `auto.offset.reset` wartości `none` spowoduje rzucenie wyjątku podczas próby konsumowania do nieprawidłowego przesunięcia.

enable.auto.commit

Ten parametr kontroluje, czy konsument będzie zatwierdzał przesunięcia automatycznie, a jego domyślna wartość to `true`. Ustaw wartość `false`, jeśli wolisz kontrolować moment zatwierdzania przesunięć, co jest konieczne, by zminimalizować liczbę duplikatów i uniknąć wybrakowania danych. Jeżeli ustawisz `enable.auto.commit` na `true`, możesz również zechcieć kontrolować częstotliwość zatwierdzania przesunięć za pomocą właściwości `auto.commit.interval.ms`. Różne opcje zatwierdzania przesunięć omówimy szerzej dalej w tym rozdziale.

partition.assignment.strategy

Dowiedziałeś się, że partycje są przypisane do konsumentów w grupie konsumentów. `PartitionAssignor` to klasa, która na podstawie danych konsumentów i subskrybowanych przez nie tematów przypisuje określone partycje do poszczególnych konsumentów. Oto domyślne strategie przypisywania Kafki:

Przypisywanie zakresowe

Przypisuje poszczególnym konsumentom kolejne podzbiory partycji z każdego subskrybowanego tematu. Jeśli konsumenci K1 i K2 subskrybują np. dwa tematy, T1 i T2, a każdy z tematów ma trzy partycje, to K1 otrzyma partycje 0 i 1 z tematów T1 i T2, podczas gdy K2 otrzyma partycję 2 z tych tematów. Ponieważ każdy temat ma nieparzystą liczbę partycji, a przypisywanie odbywa się niezależnie dla każdego tematu, pierwszy konsument otrzymuje więcej partycji niż drugi. Dzieje się tak za każdym razem, gdy używane jest przypisywanie zakresowe (Range ➔ Assignor), a liczba partycji w każdym temacie nie dzieli się równo przez liczbę konsumentów.

Przypisywanie karuzelowe

Przypisuje wszystkie partycje ze wszystkich subskrybowanych tematów kolejno poszczególnym konsumentom, jednemu po drugim. Gdyby opisane wcześniej konsumenci K1 i K2 używały przypisywania karuzelowego (RoundRobinAssignor), K1 otrzymałby partycje 0 i 2 z tematu T1 i partycję 1 z tematu T2. K2 dostałby partycję 1 z tematu T1 oraz partycje 0 i 2 z tematu T2. Generalnie jeśli wszystkie konsumenci subskrybują te same tematy (co jest bardzo częstym scenariuszem), przypisywanie karuzelowe powoduje, że wszystkie będą miały taką samą liczbę partycji (lub będą różnić się co najwyżej jedną partycją).

Przypisywanie lepkie

Przypisywanie lepkie (StickyAssignor) ma dwa cele: po pierwsze, utrzymywać jak najbardziej zrównoważony przydział, a po drugie, po rebalansowaniu zachowywać możliwie najwięcej niezmiennych przydziałów, minimalizując obciążenie związane z przenoszeniem przypisań partycji z jednego konsumenta na drugi. W typowym przypadku, gdy wszystkie konsumenci subskrybują ten sam temat, początkowe przypisania będą tak samo zrównoważone jak w przypisywaniu karuzelowym, podobnie jak kolejne, ale partycje będą rzadziej przenoszone. Gdy konsumenci z tej samej grupy subskrybują różne tematy, przypisania lepkie będą bardziej zrównoważone niż przypisania karuzelowe.

Przypisywanie kooperacyjne lepkie

Ta strategia przypisywania (CooperativeStickyAssignor) jest taka sama jak w przypisywaniu lepkim, ale obsługuje kooperacyjne równoważenie obciążenia, gdzie konsumenci mogą nadal konsumować partycje, które nie zostały ponownie przypisane. Więcej informacji na temat kooperacyjnego równoważenia obciążenia znajdziesz wcześniej w tym rozdziale w punkcie „Grupy konsumentów i równoważenie partycji”. Pamiętaj też, że jeśli przeprowadzasz uaktualnienie z wersji starszej niż 2.3, wówczas w celu udostępnienia strategii przypisywania kooperacyjnego lepkiego musisz podążać określoną ścieżką aktualizacji, zapoznaj się więc uważnie z przewodnikiem aktualizacji (<https://oreil.ly/kLMI6>).

Strategię przypisywania partycji możesz wybrać przez ustawienie właściwości `partition.assignment.strategy`. Wartość domyślna to `org.apache.kafka.clients.consumer.RangeAssignor`, która implementuje opisaną wcześniej strategię przypisywania zakresowego. Możesz ją zastąpić ustawieniem `org.apache.kafka.clients.consumer.RoundRobinAssignor`, `org.apache.kafka.clients.consumer.StickyAssignor` lub `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`. Bardziej zaawansowaną opcją jest zaimplementowanie własnej strategii przypisywania, w której wartość `partition.assignment.strategy` powinna wskazywać na nazwę Twojej klasy.

client.id

Może to być dowolny łańcuch znaków i będzie używany przez brokery do identyfikowania żądań wysyłanych przez klienty, np. żądań pobierania. Jest używany do rejestrowania komunikatów dzienników i wskaźników oraz do kwot.

client.rack

Domyślnie konsumenci pobierają komunikaty z repliki głównej każdej partycji. Jednak gdy klastr obejmuje wiele centrów danych lub wiele stref dostępności w chmurze, pobieranie komunikatów z repliki znajdującej się w tej samej strefie co konsument ma swoje zalety zarówno pod względem wydajności, jak i kosztów. Aby umożliwić pobieranie z najbliższej repliki, musisz ustawić konfigurację `client.rack` i określić strefę, w której znajduje się dany klient. Następnie możesz skonfigurować brokery, aby zastąpiły domyślną klasę `replica.selector.class` klasą `org.apache.kafka.com.mon.replica.RackAwareReplicaSelector`.

Możesz również zaimplementować własną klasę `replica.selector.class` z niestandardową logiką, aby najlepszą replikę do konsumowania wybierać na podstawie metadanych klienta i partycji.

group.instance.id

Może to być dowolny unikatowy łańcuch znaków i służy do zapewniania konsumentowi statycznego członkostwa w grupie.

receive.buffer.bytes i send.buffer.bytes

Są to rozmiary buforów wysyłania i odbierania TCP używanych przez gniazda podczas zapisywania i odczytywania danych. Jeśli mają ustawioną wartość `-1`, używane są domyślne ustawienia systemu operacyjnego. Dobrym pomysłem może być zwiększenie wartości tych parametrów, gdy producenci lub konsumenci komunikują się z brokerami w innym centrum danych, ponieważ te łącza sieciowe zwykle mają większe opóźnienia i niższą przepustowość.

offsets.retention.minutes

Jest to konfiguracja brokera, ale trzeba ją znać ze względu na jej wpływ na zachowanie konsumentów. Dopóki grupa konsumentów ma aktywnych członków (tj. członków, którzy aktywnie utrzymują członkostwo w grupie przez wysyłanie sygnałów heartbeat), Kafka będzie zachowywać ostatnie przesunięcia zatwierdzone przez grupę dla poszczególnych partycji, będzie więc można je odzyskać w przypadku zmiany przypisań lub restartu. Jednak gdy grupa stanie się pusta, Kafka zachowa swoje zatwierdzone przesunięcia tylko do czasu ustawionego w tej konfiguracji — domyślnie przez 7 dni. Gdy przesunięcia zostaną usunięte, a dana grupa stanie się ponownie aktywna, będzie się zachowywać jak zupełnie nowa grupa konsumentów bez informacji o tym, co konsumowała w przeszłości. Zwróć uwagę, że to zachowanie zmieniało się kilka razy, dlatego jeżeli używasz wersji starszych niż 2.1.0, sprawdź dokumentację swojej wersji pod kątem oczekiwanego zachowania.

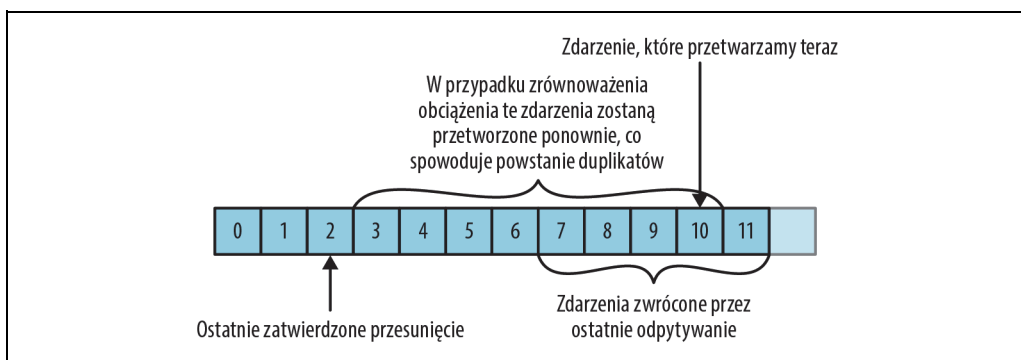
Zatwierdzenia i przesunięcia

Przy każdym wywołaniu metoda `poll()` zwraca zapisane w Kafce rekordy, których konsumenci z naszej grupy jeszcze nie odczytali. Oznacza to, że mamy sposób na śledzenie, które rekordy zostały odczytane przez konsumenta z danej grupy. Jak wspomnieliśmy wcześniej, jedną z unikatowych cech Kafki jest to, że nie śledzi potwierdzeń od konsumentów w taki sposób, jak robi to wiele kolejek JMS (ang. *Java Message Service*). Zamiast tego konsumenci mogą używać Kafki do śledzenia swoich pozycji (przesunięć) na poszczególnych partycjach.

Zaktualizowanie bieżącej pozycji na partycji nazywamy **zatwierdzeniem przesunięcia** (ang. *offset commit*). Inaczej niż tradycyjne kolejki komunikatów Kafka nie zatwierdza indywidualnych rekordów. Zamiast tego każdy konsument zatwierdza ostatni komunikat, który pomyślnie przetworzył z danej partycji, i domyślnie przyjmuje, że wszystkie wcześniejsze komunikaty również zostały przetworzone pomyślnie.

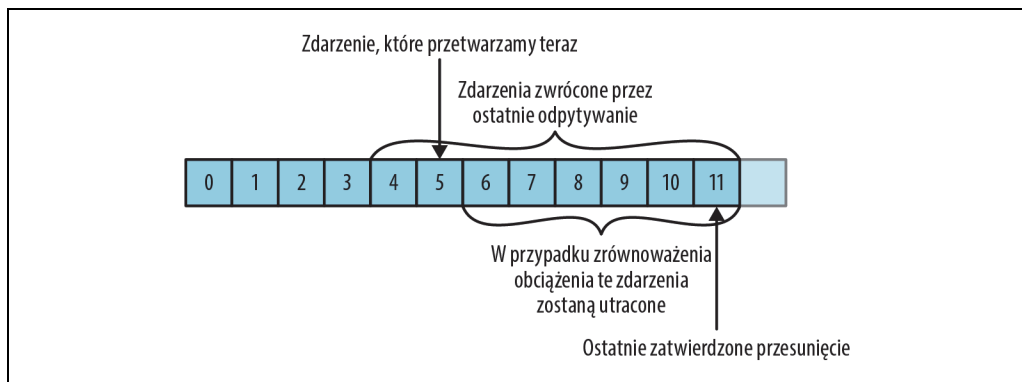
W jaki sposób konsument zatwierdza przesunięcie? Wysła do Kafki komunikat, który aktualizuje specjalny temat `__consumer_offsets` zatwierdzonym przesunięciem dla poszczególnych partycji. Dopóki wszystkie konsumenci będą bezproblemowo działać, nie będzie to miało żadnych konsekwencji. Jeśli jednak jakiś konsument ulegnie awarii lub do grupy konsumentów dołączy nowy konsument, **uruchomione zostanie równoważenie obciążenia**. Po wykonaniu równoważenia obciążenia każdemu konsumentowi może zostać przypisany nowy zbiór partycji, inny od przetwarzanego wcześniej. Aby wiedzieć, od czego wznowić pracę, konsument odczytuje wtedy ostatnie zatwierdzone przesunięcie dla każdej partycji i od tego miejsca kontynuuje przetwarzanie komunikatów.

Jeśli zatwierdzone przesunięcie będzie mniejsze niż przesunięcie ostatniego komunikatu przetworzonego przez klient, komunikaty między ostatnim przetworzonym przesunięciem a zatwierdzonym przesunięciem zostaną przetworzone dwukrotnie. Pokazaliśmy to na rysunku 4.8.



Rysunek 4.8. Ponownie przetwarzane komunikaty

Jeżeli zatwierdzone przesunięcie będzie większe niż przesunięcie ostatniego komunikatu faktycznie przetworzonego przez klient, wówczas wszystkie komunikaty między ostatnim przetworzonym przesunięciem a zatwierdzonym przesunięciem zostaną przez daną grupę konsumentów pominięte. Pokazaliśmy to na rysunku 4.9.



Rysunek 4.9. Komunikaty pominięte między przesunięciami

Oczywiście zarządzanie przesunięciami ma duży wpływ na aplikację kliencką. Interfejs API Kafka
 ↪ Consumer zapewnia wiele sposobów zatwierdzania przesunięć.



Które przesunięcie jest zatwierdzone?

Domyślnym zachowaniem przy zatwierdzaniu automatycznym lub bez określania zamierzonych przesunięć jest zatwierdzanie po ostatnim przesunięciu zwróconym przez `poll()`. Należy o tym pamiętać przy próbie ręcznego zatwierdzania określonych przesunięć lub przy dążeniu do ich zatwierdzenia. Żmudne jest również jednak wielokrotne czytanie komunikatu: „Zatwierdź przesunięcie, które jest o jeden większe niż ostatnie przesunięcie, które klient otrzymał z wywołania `poll()`” i w 99% przypadków nie ma to znaczenia. Dlatego gdy będziemy odnosić się do zachowania domyślnego, będziemy pisać: „Zatwierdź ostatnie przesunięcie”. Pamiętaj o tej uwadze, jeżeli potrzebujesz ręcznie manipulować przesunięciami.

Zatwierdzanie automatyczne

Najłatwiejszym sposobem na zatwierdzenie przesunięć jest delegowanie tego zadania konsumentowi. Jeśli skonfigurujesz parametr `enable.auto.commit=true`, konsument co 5 sekund będzie zatwierdzał najnowsze przesunięcie, które klient otrzymał z wywołania metody `poll()`. Ustawieniem domyślnym jest interwał 5-sekundowy, który można kontrolować za pomocą właściwości `auto.commit.interval.ms`. Podobnie jak wszystkie inne kwestie dotyczące klienta, automatyczne zatwierdzenia są sterowane przez pętlę odpytywania. Przy każdym odpytywaniu konsument sprawdza, czy nadszedł czas na zatwierdzenie, a jeśli tak, zatwierdza przesunięcia zwrócone w ostatnim odpytywaniu.

Zanim jednak skorzystasz z tej wygodnej opcji, powinieneś zrozumieć jej konsekwencje.

Przypomnijmy, że domyślnie automatyczne zatwierdzenia odbywają się co 5 sekund. Załóżmy, że 3 sekundy po ostatnim zatwierdzeniu konsument uległ awarii. Po zrównoważeniu obciążenia pozostałe aktywne konsumenty zaczynają przetwarzać należące wcześniej do tego brokera partycje. Rozpoczynają jednak od ostatniego zatwierdzonego przesunięcia. W tym przypadku jest to przesunięcie sprzed 3 sekund, stąd wszystkie zdarzenia, które pojawiły się w ciągu tych 3 sekund, zostaną

przetworzone dwukrotnie. Możliwe jest skonfigurowanie krótszego interwału zatwierdzania i zmniejszenie okna, w którym rekordy będą duplikowane, ale nie można tego całkowicie wyeliminować.

Gdy przy włączonym automatycznym zatwierdzaniu nadchodzi czas na zatwierdzenie przesunięć, następne odpytywanie zatwierdza ostatnie przesunięcie zwrócone przez poprzednie odpytywanie. Nie wiadomo, które zdarzenia zostały rzeczywiście przetworzone, dlatego ważne jest, by przed ponownym wywołaniem `poll()` zawsze przetwarzać wszystkie zdarzenia zwrócone przez ostatnie wywołanie tej metody. (Metoda `close()` również automatycznie zatwierdza przesunięcia, podobnie jak `poll()`). Zwykle nie stanowi to problemu, ale musisz być ostrożny, gdy obsługujesz wyjątki lub przedwcześnie wychodzisz z pętli odpytywania.

Automatyczne zatwierdzenia są wygodne, lecz nie dają programistom wystarczającej kontroli, aby uniknąć duplikowania komunikatów.

Zatwierdzanie bieżących przesunięć

Programiści zazwyczaj starają się bardziej kontrolować momenty zatwierdzania przesunięć — w celu zarówno wyeliminowania ryzyka pominięcia komunikatów, jak i zmniejszenia liczby komunikatów duplikowanych podczas równoważenia obciążenia. Consumer API ma opcję zatwierdzania bieżącego przesunięcia w punkcie, który jest sensowny dla programisty aplikacji, a nie na podstawie timera.

Ustawienie `enable.auto.commit=false` powoduje, że przesunięcia są zatwierdzane tylko na wyraźne żądanie aplikacji. Najprostszy i najbardziej niezawodny interfejs API zatwierdzania jest `commitSync()`. Ten interfejs API zatwierdza ostatnie przesunięcie zwrócone przez `poll()` i kończy działanie po zatwierdzeniu przesunięcia, zgłaszając wyjątek, jeżeli zatwierdzenie z jakiegoś powodu się nie powiedzie.

Należy pamiętać, że `commitSync()` zatwierdza ostatnie przesunięcie zwrócone przez `poll()`, dlatego jeśli wywołasz `commitSync()` przed zakończeniem przetwarzania wszystkich rekordów ze zbioru, w przypadku awarii aplikacji ryzykujesz pominięcie komunikatów, które zostały zatwierdzone, ale nie przetworzone. Jeśli aplikacja ulegnie awarii podczas przetwarzania rekordów z danego zbioru, wszystkie komunikaty od początku ostatniej partii do momentu uruchomienia równoważenia obciążenia zostaną przetworzone dwukrotnie — być może lepsze to niż brakujące komunikaty.

W poniższym listingu pokazaliśmy sposób użycia `commitSync` do zatwierdzenia przesunięć po zakończeniu przetwarzania ostatniej partii komunikatów:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %d, przesunięcie = %d, klient = %s, kraj = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
    }
    try {
        consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
```



```

        log.error("zatwierdzenie się nie powiodło", e) ❸
    }
}

```

- ❶ Załóżmy, że wypisując zawartość rekordu, zakończyliśmy jego przetwarzanie. Twoja aplikacja prawdopodobnie będzie robić z rekordami znacznie więcej — modyfikować je, wzbogacać, agregować, wyświetlać na dashboardzie lub powiadamiać użytkowników o ważnych zdarzeniach. Moment „kończenia” z rekordem powinieneś określać zgodnie ze swoim przypadkiem użycia.
- ❷ Gdy skończymy „przetwarzanie” wszystkich rekordów z bieżącej partii, przed odpytywaniem o kolejne komunikaty wywołujemy metodę `commitSync` w celu zatwierdzenia ostatniego przesunięcia w partii.
- ❸ Metoda `commitSync` ponawia próby zatwierdzenia, o ile nie ma błędu, którego nie można naprawić. Jeżeli taki błąd wystąpi, będziemy mogli jedynie go zarejestrować.

Zatwierdzanie asynchroniczne

Jedną z wad ręcznego zatwierdzania jest to, że aplikacja jest blokowana, dopóki broker nie odpowie na żądanie zatwierdzenia. Ogranicza to przepływność aplikacji. Przepływność można poprawić przez rzadsze zatwierdzanie, ale wtedy zwiększy się liczba potencjalnych duplikatów, które może wygenerować równoważenie obciążenia.

Kolejną opcją jest API zatwierdzania asynchronicznego. Zamiast czekać na odpowiedź brokera na zatwierdzenie, wysyłamy żądanie i kontynuujemy pracę:

```

Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s,
            przesunięcie = %d, klient = %s, kraj = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}

```

- ❶ Zatwierdzamy ostatnie przesunięcie i kontynuujemy pracę.

Minusem jest to, że o ile metoda `commitSync()` ponawia próbę zatwierdzenia, dopóki nie zakończy wykonywania pomyślnie lub nie napotka nieodwracalnego błędu, o tyle metoda `commitAsync()` nie podejmuje ponownej próby. Nieponawianie próby jest spowodowane tym, że zanim `commitAsync()` otrzyma odpowiedź z serwera, może nastąpić późniejsze zatwierdzenie, które już się powiodło. Wyobraź sobie, że wysłaliśmy żądanie zatwierdzenia przesunięcia o wartości 2000. Wystąpił tymczasowy problem z komunikacją, więc broker nigdy nie otrzymał żądania i dlatego nigdy nie odpowie. Tymczasem przetworzyliśmy kolejną partię i pomyślnie zatwierdziliśmy przesunięcie 3000. Jeżeli `commitAsync()` ponowi teraz poprzednie nieudane zatwierdzenie, może powieść się zatwierdzenie przesunięcia 2000 *po tym*, jak przesunięcie 3000 zostało już przetworzone i zatwierdzone. W przypadku uruchomienia równoważenia obciążenia spowoduje to wygenerowanie większej liczby duplikatów.

Wspominamy o tej komplikacji i znaczeniu prawidłowej kolejności zatwierdzeń, ponieważ metoda `commitAsync()` ma również opcję przekazania wywołania zwrotnego, które zostanie wywołane, gdy broker odpowie. Tego wywołania zwrotnego często używa się do rejestrowania błędów zatwierdzeń lub zliczania ich jako wskaźnika, ale jeśli chcesz wykorzystać wywołanie zwrotne do ponawiania prób, musisz mieć świadomość problemu z kolejnością zatwierdzeń:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s,
            przesunięcie = %d, klient = %s, kraj = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
            OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Nie powiodło się zatwierdzenie dla przesunięć {}", offsets, e);
        }
    }); ❶
}
```

- ❶ Wysyłamy zatwierdzenie i kontynuujemy pracę, ale jeśli zatwierdzenie się nie powiedzie, niepowodzenie i przesunięcia zostaną zarejestrowane.



Ponawianie zatwierdzania asynchronicznego

Prostym wzorcem umożliwiającym uzyskanie prawidłowej kolejności zatwierdzeń w przypadku asynchronicznego ponawiania prób jest użycie monotonicznie rosnącego numeru sekwencyjnego. Przy każdym zatwierdzeniu zwiększamy numer sekwencyjny i w momencie zatwierdzania dodajemy do wywołania zwrotnego `commitAsync`. Kiedy przygotowujemy się do wysłania ponownej próby, sprawdzamy, czy numer sekwencyjny zatwierdzenia otrzymany przez wywołanie zwrotne jest równy zmiennej instancyjnej. Jeśli tak, nie było nowszego zatwierdzenia i można bezpiecznie spróbować ponownie. Jeżeli numer sekwencyjny instancji jest wyższy, nie próbujemy ponownie, ponieważ wysłane zostało już nowsze zatwierdzenie.

Łączenie zatwierdzeń synchronicznych i asynchronicznych

Zwykle sporadyczne niepowodzenia zatwierdzania bez ponawiania prób nie stanowią dużego problemu, bo gdy problem jest tymczasowy, kolejne zatwierdzenie zakończy się sukcesem. Jeśli jednak wiemy, że jest to ostatnie zatwierdzenie przed zakończeniem działania konsumenta lub przed równoważeniem obciążenia, powinniśmy się upewnić, że zatwierdzenie się powiedzie.

Dlatego powszechnym wzorcem jest połączenie `commitAsync()` z `commitSync()` tuż przed zakończeniem działania. W poniższym listingu pokazaliśmy, jak to działa (zatwierdzanie tuż przed równoważeniem obciążenia omówimy, gdy przejdziemy do tematu nasłuchiwanego rebalansowania):

```

Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                               klient = %s, kraj = %s\n",
                               record.topic(), record.partition(),
                               record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
    consumer.commitSync(); ❷
} catch (Exception e) {
    log.error("Nieoczekiwany błąd", e);
} finally {
    consumer.close();
}

```

- ❶ Gdy wszystko jest w porządku, używamy metody `commitAsync()`. Jest szybsza, a jeżeli jedno zatwierdzenie się nie powiedzie, następne posłuży jako ponowienie próby.
- ❷ Jeśli jednak kończymy działanie, nie będzie „następnego zatwierdzenia”. Wywołujemy metodę `commitSync()`, ponieważ będzie ona ponawiać próby, aż zatwierdzenie się powiedzie lub wystąpi nieodwracalny błąd.

Zatwierdzanie określonego przesunięcia

Zatwierdzanie ostatniego przesunięcia pozwala zatwierdzać jedynie po zakończeniu przetwarzania każdej partii. Co jednak zrobić, jeżeli wymagane jest częstsze zatwierdzanie? A jeśli wywołanie `poll()` zwróci dużą partię i pożądane będzie zatwierdzanie przesunięcia w środku partii, by uniknąć konieczności ponownego przetwarzania wszystkich tych rekordów w przypadku wystąpienia równoważenia obciążenia? Nie można wywołać po prostu `commitSync()` lub `commitAsync()` — spowoduje to zatwierdzenie ostatniego zwróconego przesunięcia, którego nie udało się jeszcze przetworzyć.

Na szczęście Consumer API pozwala wywołać `commitSync()` i `commitAsync()` oraz przekazać mapę partycji i przesunięć, które mają zostać zatwierdzone. Jeżeli jesteś w trakcie przetwarzania partii rekordów, a ostatni komunikat otrzymany z partycji 3 w temacie `customers` ma offset 5000, możesz wywołać `commitSync()`, aby zatwierdzić offset 5001 dla partycji 3 we wspomnianym temacie. Ponieważ konsument może przetwarzać więcej niż jedną partycję, będziesz musiał śledzić przesunięcia na wszystkich z nich, co zwiększa złożoność kodu.

Tak wygląda zatwierdzanie określonych przesunięć:

```

private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....
Duration timeout = Duration.ofMillis(100);

```

```

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
            klient = %s, kraj = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value()); ❷
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset()+1, "brak metadanych")); ❸
        if (count % 1000 == 0) ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}

```

- ❶ To jest mapa, której użyjemy do ręcznego śledzenia przesunięć.
- ❷ Pamiętaj, że w miejsce printf możesz wstawić każde przetwarzanie, które będziesz wykonywał dla konsumowanych rekordów.
- ❸ Po odczytaniu każdego rekordu aktualizujemy mapę przesunięć o przesunięcie następnego komunikatu, który spodziewamy się przetworzyć. Zatwierdzone przesunięcie powinno być zawsze przesunięciem następnego komunikatu, który aplikacja będzie odczytywać. Od tego miejsca zaczniemy odczytywać następnym razem.
- ❹ Tutaj decydujemy się zatwierdzać bieżące przesunięcia co 1000 rekordów. W swojej aplikacji możesz zatwierdzać np. na podstawie czasu lub zawartości rekordów.
- ❺ Zdecydowaliśmy się wywołać `commitAsync()` (bez wywołania zwrótnego, dlatego drugi parametr ma wartość `null`), ale wywołanie `commitSync()` również będzie tu całkowicie poprawne. Oczywiście kiedy zatwierdzasz określone przesunięcia, nadal musisz wykonać całą obsługę błędów, którą pokazaliśmy w poprzednich przykładach.

Nasłuchiwanie równoważenia obciążenia

Jak wspomnieliśmy w poprzednim podrozdziale poświęconym zatwierdzaniu przesunięć, przed zakończeniem działania oraz przed równoważeniem obciążenia partycji konsument będzie chciał wykonywać pewne prace porządkowe.

Jeśli wiesz, że konsument wkrótce straci własność partycji, będziesz chciał zatwierdzić przesunięcia ostatniego przetworzonego zdarzenia. Prawdopodobnie będziesz musiał zamknąć także uchwyty plików, połączenia z bazą danych itp.

Consumer API umożliwia uruchamianie własnego kodu, gdy partycje są dodawane do konsumenta lub usuwane z niego. Wymaga to przekazania obiektu nasłuchiacza `ConsumerRebalanceListener` podczas wywoływania metody `subscribe()`, którą omówiliśmy wcześniej. Klasa `ConsumerRebalanceListener` ma trzy metody, które możesz zaimplementować:

```
public void onPartitionsAssigned(Collection<TopicPartition> partitions)
```

Wywoływana po ponownym przypisaniu partycji do konsumenta, ale zanim konsument rozpocznie konsumowanie komunikatów. W tym miejscu przygotowujesz lub ładujesz dowolny

stan, którego chcesz użyć z partycją, poszukujesz poprawnych przesunięć, jeśli zachodzi taka potrzeba, itp. Wszelkie wykonywane tu przygotowania powinny zakończyć się przed upływem limitu czasu `max.poll.timeout.ms`, by konsument mógł pomyślnie dołączyć do grupy.

```
public void onPartitionsRevoked(Collection<TopicPartition> partitions)
```

Wywoływana, gdy konsument musi zrezygnować z posiadanych partycji w wyniku równoważenia obciążenia lub kończenia działania konsumenta. W typowym przypadku używania algorytmu gorliwego równoważenia obciążenia metoda ta jest wywoływana przed rozpoczęciem równoważenia i po zakończeniu konsumowania komunikatów przez konsument. Jeśli używany jest algorytm kooperacyjnego równoważenia obciążenia, metoda ta jest wywoływana na końcu równoważenia obciążenia i jedynie z podzbiorem partycji, z których konsument musi zrezygnować. W tym miejscu powinieneś zatwierdzać przesunięcia, by następny właściciel partycji wiedział, od czego zacząć.

```
public void onPartitionsLost(Collection<TopicPartition> partitions)
```

Wywoływana tylko przy zastosowaniu algorytmu kooperacyjnego równoważenia obciążenia i jedynie w wyjątkowych przypadkach, gdy partycje zostały przypisane innym konsumentom bez uprzedniego anulowania przypisań przez algorytm równoważenia (w normalnych sytuacjach wywoływana jest metoda `onPartitionsRevoked()`). W tym miejscu czyścisz stan lub zasoby używane z tymi partycjami. Zwróć uwagę, że należy to robić ostrożnie — nowy właściciel partycji mógł już zapisać własny stan i lepiej unikać konfliktów. Jeśli nie zaimplementujesz tej metody, zamiast niej wywoływana będzie metoda `onPartitionsRevoked()`.



Jeżeli używasz algorytmu kooperacyjnego równoważenia obciążenia, powinieneś pamiętać o kilku kwestiach:

- Metoda `onPartitionsAssigned()` będzie wywoływana przy każdym równoważeniu obciążenia jako sposób powiadamiania o nim konsumenta. Jeśli jednak danemu konsumentowi nie będą przypisywane żadne nowe partycje, metoda będzie wywoływana z pustą kolekcją.
- Metoda `onPartitionsRevoked()` będzie wywołana w normalnych warunkach równoważenia obciążenia, ale tylko wtedy, gdy dany konsument zrezygnuje z własności partycji. Nie będzie wywoływana z pustą kolekcją.
- Metoda `onPartitionsLost()` będzie wywoływana w wyjątkowych warunkach równoważenia obciążenia, a partycje z kolekcji będą już wtedy miały nowych właścicieli.

Jeśli zaimplementujesz wszystkie trzy metody, będziesz miał gwarancję, że podczas normalnego równoważenia metoda `onPartitionsAssigned()` będzie wywoływana przez nowego właściciela ponownie przypisywanych partycji dopiero po tym, jak poprzedni właściciel wywoła `onPartitionsRevoked()` i zrezygnuje z własności.

W poniższym przykładzie pokazaliśmy, jak używać `onPartitionsRevoked()` do zatwierdzania przesunięć przed utratą własności partycji:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
Duration timeout = Duration.ofMillis(100);
```

```
private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
        partitions) { ❷
    }
}
```

```
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
```

```

        System.out.println("Podczas rebalansowania utracono partycje. " +
            "Zatwierdzanie bieżących przesunięć:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                klient = %s, kraj = %s\n",
                    record.topic(), record.partition(), record.offset(),
                    record.key(), record.value());
            currentOffsets.put(
                new TopicPartition(record.topic(), record.partition()),
                new OffsetAndMetadata(record.offset()+1, null));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // Ignorujemy, ponieważ kończymy działanie
} catch (Exception e) {
    log.error("Nieoczekiwany błąd", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Zamknęliśmy konsument i zakończyliśmy");
    }
}
}

```

- ❶ Zaczynamy od zaimplementowania nasłuchiacza `ConsumerRebalanceListener`.
- ❷ W tym przykładzie nie musimy nic robić, gdy otrzymamy nową partycję; zaczynamy po prostu konsumować komunikaty.
- ❸ Jednak gdy grozi nam utrata partycji z powodu równoważenia obciążenia, musimy zatwierdzić przesunięcia. Robimy to dla wszystkich partycji, a nie tylko dla tych, które zaraz utracimy — przesunięcia dotyczą zdarzeń, które zostały już przetworzone, więc nie ma w tym nic złego. Ponadto używamy `commitSync()`, aby przed kontynuowaniem równoważenia obciążenia upewnić się, że przesunięcia zostały zatwierdzone.
- ❹ Najważniejsza część: przekazujemy `ConsumerRebalanceListener` do metody `subscribe()`, aby została wywołana przez konsument.

Konsumowanie rekordów z określonymi przesunięciami

Nauczyłeś się już, jak używać metody `poll()` do rozpoczęcia konsumowania komunikatów od ostatniego zatwierdzonego przesunięcia dla każdej partycji i kontynuowania przetwarzania po kolei wszystkich komunikatów. Czasami jednak potrzebujesz zacząć odczytywanie z innym przesunięciem.

Kafka oferuje różne metody, które powodują, że następne wywołanie `poll()` rozpoczyna konsumowanie z innym przesunięciem.

Jeśli chcesz zacząć odczytywanie wszystkich komunikatów od początku partycji albo przeskoczyć do końca partycji i zacząć przetwarzać tylko nowe komunikaty, możesz skorzystać ze specjalnie do tego celu przeznaczonych interfejsów API: `seekToBeginning(Collection<TopicPartition> tp)` i `seekToEnd(Collection<TopicPartition> tp)`.

API Kafki pozwala również wyszukać określone przesunięcie. Z tej funkcjonalności można korzystać na wiele sposobów, np. pozostająca w tyle aplikacja, w której istotny jest czas, może przeskoczyć do przodu o kilka rekordów, albo konsument zapisujący dane w pliku może zostać zresetowany do określonego punktu w czasie w celu odzyskania danych w przypadku utraty pliku.

Oto przykład, jak ustawić bieżące przesunięcie na wszystkich partycjach na rekordy, które zostały utworzone w określonym momencie:

```
Long oneHourEarlier = Instant.now().atZone(ZoneId.systemDefault())
    .minusHours(1).toEpochSecond();
Map<TopicPartition, Long> partitionTimestampMap = consumer.assignment()
    .stream()
    .collect(Collectors.toMap(tp -> tp, tp -> oneHourEarlier)); ❶
Map<TopicPartition, OffsetAndTimestamp> offsetMap
    = consumer.offsetsForTimes(partitionTimestampMap); ❷

for(Map.Entry<TopicPartition, OffsetAndTimestamp> entry: offsetMap.entrySet()) {
    consumer.seek(entry.getKey(), entry.getValue().offset()); ❸
}
```

- ❶ Tworzymy mapowanie wszystkich partycji przypisanych do danego konsumenta (za pośrednictwem `consumer.assignment()`) na znacznik czasu, do którego chcemy przywrócić konsumenty.
- ❷ Następnie pobieramy przesunięcia, które były aktualne w określonych momentach. Ta metoda wysyła do brokera żądanie, w którym do zwrócenia odpowiednich przesunięć używany jest indeks znacznika czasu.
- ❸ Na koniec resetujemy przesunięcie na każdej partycji do tego, które zostało zwrócone w poprzednim kroku.

Wychodzenie z pętli

Wcześniej w tym rozdziale, gdy pisaliśmy o pętli odpytywania, stwierdziliśmy, że nie należy się przejmować tym, iż konsumenty odpytują w pętli nieskończonej, a omówienie czystego wychodzenia z pętli odłożyliśmy na później. Porozmawiajmy więc teraz o tym, jak czysto wyjść z pętli odpytywania.

Gdy zdecydujesz się zamknąć konsument i będziesz chciał natychmiast wyjść z pętli, nawet jeśli konsument będzie czekał na zakończenie długiego wywołania `poll()`, będziesz potrzebować kolejnego wątku do wywołania metody `consumer.wakeup()`. Jeżeli uruchamiasz pętlę konsumenta w głównym

wątku, możesz to zrobić za pomocą zaczepu zamknięcia (ShutdownHook). Zwróć uwagę, że `consumer.wakeup()` jest jedyną metodą konsumenta, którą można bezpiecznie wywoływać z innego wątku. Wywołanie `wakeup` spowoduje zakończenie działania metody `poll()` z wyjątkiem `WakeupException`, a jeśli metoda `consumer.wakeup()` zostanie wywołana, gdy wątek nie oczekuje na rezultaty odpytowania, wyjątek zostanie zgłoszony przy wywołaniu `poll()` w następnej iteracji. `WakeupException` nie musi być obsługiwany, ale przed wyjściem z wątku należy wywołać metodę `consumer.close()`. Zamknięcie konsumenta spowoduje zatwierdzenie przesunięć, jeżeli zachodzi taka potrzeba, oraz wysłanie koordynatorowi grupy komunikatu, że konsument opuszcza grupę. Koordynator konsumentów natychmiast uruchomi równoważenie obciążenia i nie trzeba będzie czekać na upłynięcie limitu czasu sesji, by partycje zamykanego konsumenta zostały przypisane do innego konsumenta w grupie.

W poniższym listingu pokazaliśmy, jak będzie wyglądał kod wyjścia, jeśli konsument działa w głównym wątku aplikacji. Ten przykład jest nieco skrócony, ale możesz zobaczyć go w całości na GitHubie (<http://bit.ly/2u47e9A>):

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Rozpoczęcie procesu wyjścia...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...
Duration timeout = Duration.ofMillis(10000); ❷

try {
    // Zapętlamy do wcisnięcia Ctrl+C, a przy wyjściu z pętli zaczep zamknięcia wykona czyszczenie
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "-- oczekiwanie na dane...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("przesunięcie = %d, klucz = %s, wartość = %s\n",
                record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Zatwierdzanie przesunięcia na pozycji:" +
                consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // Przy zamykaniu ignorujemy ❸
} finally {
    consumer.close(); ❹
    System.out.println("Konsument zamknięty - gotowe");
}
```


- ❶ ShutdownHook jest uruchamiany w osobnym wątku, stąd jedyną bezpieczną czynnością, jaką możesz wykonać, jest wywołanie wakeup, by wyjść z pętli poll.
- ❷ Wyjątkowo długi limit czasu odpytywania. Jeśli pętla odpytywania jest odpowiednio krótka i możesz trochę poczekać przed wyjściem z niej, nie musisz wywoływać wakeup — wystarczy sprawdzanie niepodzielnej wartości logicznej w każdej iteracji. Długie limity czasu odpytywania są przydatne podczas konsumowania tematów o niskiej przepływności; dzięki temu klient zużywa mniej procesora do ciągłego zapętlenia, gdy broker nie ma żadnych nowych danych do zwrócenia.
- ❸ Kolejny wątek wywołujący wakeup spowoduje, że poll zgłosi wyjątek WakeupException. Powinieneś wychwytywać ten wyjątek, by mieć pewność, że aplikacja nie zostanie nieoczekiwanie zamknięta, nie musisz jednak niczego z nim robić.
- ❹ Na koniec upewnij się, że konsument został czysto zamknięty.

Deserializatory

Jak pisaliśmy w poprzednim rozdziale, producenci Kafki wymagają **serializatorów** (ang. *serializer*) do konwersji obiektów na tablice bajtów, które są następnie wysyłane do Kafki. Analogicznie konsumenci Kafki wymagają **deserializatorów** (ang. *deserializer*) do konwersji otrzymanych z Kafki tablic bajtów na obiekty Javy. W poprzednich przykładach założyliśmy, że zarówno klucz, jak i wartość każdego komunikatu są łańcuchami znaków, i w konfiguracji konsumenta korzystaliśmy z domyślnego deserializatora `StringDeserializer`.

W rozdziale 3. poświęconym producentowi Kafki pokazaliśmy, jak serializować typy niestandardowe. Dowiedziałeś się ponadto, jak używać platformy Avro i serializatorów `AvroSerializer` do generowania obiektów Avry na podstawie definicji schematów i serializowania tych obiektów podczas produkowania komunikatów do Kafki. Przyjrzymy się teraz tworzeniu niestandardowych deserializatorów dla własnych obiektów oraz korzystaniu z Avry i jego deserializatorów.

Powinno być oczywiste, że serializator stosowany do produkowania zdarzeń do Kafki musi odpowiadać deserializatorowi, który będzie używany podczas konsumowania tych zdarzeń. Serializacja za pomocą `IntSerializer`, a następnie deserializacja za pomocą `StringDeserializer` nie skończy się dobrze. Oznacza to, że jako programista musisz śledzić, które serializatory zostały wykorzystane do zapisywania w poszczególnych tematach, i upewnić się, że każdy temat zawiera tylko dane, które mogą być interpretowane przez używane przez Ciebie deserializatory. Jedną z korzyści serializacji i deserializacji za pomocą Avry i rejestru schematów polega na tym, że `AvroSerializer` może zagwarantować zgodność ze schematem tematu wszystkich zapisywanych w nim danych, co oznacza, że będzie można deserializować te dane odpowiednim deserializatorem i schematem. Wszelkie błędy w zgodności — po stronie producenta lub konsumenta — zostaną łatwo wychwycone z wygenerowaniem odpowiedniego komunikatu o błędzie i nie trzeba będzie debugować tablic bajtów pod kątem błędów serializacji.

Zacniemy od pokazania, jak napisać niestandardowy deserializator, mimo że jest to mniej powszechna metoda, a potem przejdziemy do przykładu użycia Avry do deserializacji kluczy i wartości komunikatów.

Niestandardowe deserializatory

Wykorzystajmy ten sam niestandardowy obiekt, który serializowaliśmy w rozdziale 3., i napiszmy dla niego deserializator. Oto ten obiekt:

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

Niestandardowy deserializator będzie wyglądał następująco:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> { ❶

    @Override
    public void configure(Map configs, boolean isKey) {
        // Nie trzeba niczego konfigurować
    }

    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
        String name;

        try {
            if (data == null)
                return null;
            if (data.length < 8)
                throw new SerializationException("Rozmiar danych otrzymanych " +
                    "przez deserializator jest mniejszy niż oczekiwany");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, "UTF-8");
        }
```

```

        return new Customer(id, name); ❷
    } catch (Exception e) {
        throw new SerializationException("Błąd podczas deserializacji " +
            "byte[] do klienta " + e);
    }
}

@Override
public void close() {
    // Nie trzeba niczego zamykać
}
}

```

- ❶ Konsument również potrzebuje implementacji klasy `Customer`, a zarówno klasa, jak i serializer muszą być zgodne w aplikacjach produkujących i konsumujących. W dużej organizacji, w której wiele konsumentów i producentów współdzieli dostęp do danych, może to być problematyczne.
- ❷ Odwracamy logikę serializatora — pobieramy z tablicy bajtów identyfikator klienta i nazwę, a potem używamy ich do skonstruowania potrzebnego obiektu.

Kod konsumenta korzystającego z tego deserializatora będzie wyglądał mniej więcej tak:

```

Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    CustomerDeserializer.class.getName());

KafkaConsumer<String, Customer> consumer =
    new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout);
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("Id bieżącego klienta: " +
            record.value().getID() + " oraz"
            nazwa bieżącego klienta: " + record.value().getName());
    }
    consumer.commitSync();
}

```

Jeszcze raz podkreślamy, że implementowanie niestandardowego serializatora i deserializatora nie jest zalecane. Powoduje to ściśle powiązanie producentów i konsumentów, a ponadto jest kruche i podatne na błędy. Lepszym rozwiązaniem jest użycie standardowego formatu komunikatów, takiego jak JSON, Thrift, Protobuf lub Avro.

Zobaczmy teraz, jak stosować deserializatory Avry z konsumentem Kafki. Więcej informacji na temat platformy Apache Avro, jej schematów oraz zapewniania zgodności ze schematami znajdziesz w rozdziale 3.

Stosowanie deserializacji Avry z konsumentem Kafki

Założmy, że korzystamy w Avrze z implementacji klasy `Customer`, która została pokazana w rozdziale 3. Aby konsumować te obiekty z Kafki, należy zaimplementować aplikację konsumującą:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader", "true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts";
KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Czytanie tematu:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Nazwa bieżącego klienta to: " +
            record.value().getName());
    }
    consumer.commitSync();
}
```

- ❶ Do deserializacji komunikatów Avry wykorzystujemy `KafkaAvroDeserializer`.
- ❷ `schema.registry.url` to nowy parametr. Wskazuje lokalizację przechowywania schematów. W ten sposób konsument może użyć schematu zarejestrowanego przez producenta do deserializacji wiadomości.
- ❸ Jako typ wartości rekordu podajemy wygenerowaną klasę `Customer`.
- ❹ `record.value()` jest instancją `Customer` i możemy używać jej zgodnie z definicją tej klasy.

Samodzielny konsument — dlaczego i jak korzystać z konsumenta bez grupy?

Do tej pory pisaliśmy o grupach konsumentów, w których partycje są automatycznie przypisywane do konsumentów i automatycznie rebalansowane przy dodawaniu konsumentów do grupy lub usuwaniu ich z niej. Zazwyczaj właśnie takie zachowanie jest wymagane, ale w niektórych przypadkach potrzebujemy czegoś znacznie prostszego. Czasami wiemy, że mamy jednego konsumenta, który zawsze musi odczytywać dane ze wszystkich partycji z tematu lub z określonej partycji z tematu. W takiej sytuacji nie ma powodu do korzystania z grup lub przeprowadzania równoważenia obciążenia — wystarczy przypisać temat i (lub) partycje charakterystyczne dla danego konsumenta, konsumować komunikaty i od czasu do czasu zatwierdzać przesunięcia (choć do

zatwierdzania przesunięć i tak trzeba skonfigurować `group.id`, bez wywoływania `subscribe` konsument nie dołączy do żadnej grupy).

Kiedy dokładnie wiesz, które partycje powinien odczytywać konsument, nie *subskrybujesz* tematu — zamiast tego *przypisujesz* sobie kilka partycji. Konsument może subskrybować tematy (i należeć do grupy konsumentów) lub przypisywać sobie partycje, ale nie może robić obu rzeczy naraz.

Poniżej pokazaliśmy, w jaki sposób konsument może przypisać sobie wszystkie partycje określonego tematu i konsumować z nich:

```
Duration timeout = Duration.ofMillis(100);
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                klient = %s, kraj = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

- ❶ Zaczynamy od zapytania klastra o partycje dostępne w danym temacie. Jeżeli planujesz przetwarzać tylko określoną partycję, możesz pominąć tę część.
- ❷ Gdy już wiemy, których partycji potrzebujemy, wywołujemy metodę `assign()` z ich listą.

Poza brakiem równoważenia obciążenia i koniecznością ręcznego wyszukania partycji, wszystko inne działa jak zwykle. Pamiętaj, że jeśli ktoś doda do tematu nowe partycje, konsument nie zostanie o tym powiadomiony. Będziesz musiał to obsłużyć przez okresowe sprawdzanie `consumer.partitionsFor()` lub restartowanie aplikacji po każdorazowym dodaniu partycji.

Podsumowanie

Ten rozdział rozpoczęliśmy od szczegółowego omówienia grup konsumentów Kafki i sposobu, w jaki umożliwiają one wielu konsumentom dzielenie się pracą polegającą na odczytywaniu zdarzeń z tematów. Teoretyczne rozważania poparliśmy praktycznym przykładem konsumenta subskrybującego określony temat i nieustannie odczytującego zdarzenia. Następnie przyjrzelśmy się najważniejszym parametrom konfiguracji konsumenta i ich wpływowi na jego zachowanie. Dużą część rozdziału poświęciliśmy na omówienie przesunięć i metod ich śledzenia przez konsumenty. Zrozumienie, jak konsumenty zatwierdzają przesunięcia, ma kluczowe znaczenie przy pisaniu

niezawodnych konsumentów, dlatego dogłębnie wyjaśniliśmy to zagadnienie. Potem opisaliśmy dodatkowe elementy interfejsów API konsumentów, obsługę równoważenia obciążenia i zamykanie konsumenta.

Na koniec omówiliśmy deserializatory używane przez konsumenty do przekształcania bajtów przechowywanych w Kafce w obiekty Javy, które aplikacje mogą przetwarzać. Nieco szerzej przyjrzelśmy się deserializatorom Avry i chociaż to tylko jeden z możliwych do zastosowania typów deserializatorów, są one najczęściej używane z Kafką.

Programowe zarządzanie Kafką

Dostępnych jest wiele narzędzi CLI i GUI do zarządzania Kafką (omówimy je w rozdziale 9.), ale czasami zachodzi potrzeba wykonywania niektórych poleceń administracyjnych z poziomu aplikacji klienckiej. Powszechnym przypadkiem użycia jest np. tworzenie nowych tematów na żądanie na podstawie danych wejściowych lub danych użytkownika: aplikacje internetu rzeczy (ang. *Internet of Things* — IoT) często odbierają zdarzenia z urządzeń użytkowników i zapisują je w tematach według typu urządzenia. Jeśli producent wprowadza nowy typ urządzenia, musisz pamiętać, aby utworzyć również odpowiedni temat, albo aplikacja może dynamicznie tworzyć nowy temat, gdy otrzyma zdarzenie z nierozpoznanym typem urządzenia. Ta druga metoda ma pewne wady, ale we właściwych scenariuszach unikanie zależności od dodatkowego procesu generowania tematów jest atrakcyjną funkcjonalnością.

W wersji 0.11 platformy Apache Kafka dodany został interfejs AdminClient API w celu programistycznego sterowania funkcjonalnościami administracyjnymi, takimi jak wyświetlanie, tworzenie i usuwanie tematów, generowanie opisu klastra, zarządzanie listami ACL i modyfikowanie konfiguracji, które wcześniej były obsługiwane z poziomu wiersza poleceń.

Oto przykład. Twoja aplikacja ma produkować zdarzenia do określonego tematu. Oznacza to, że temat musi istnieć przed wyprodukowaniem pierwszego zdarzenia. Zanim w Kafce dodano AdminClient API, dostępnych było kilka opcji, a żadna z nich nie była szczególnie przyjazna dla użytkownika. Należały do nich: przechwytywanie wyjątku `UNKNOWN_TOPIC_OR_PARTITION` rzucanego z metody `producer.send()` oraz informowanie użytkownika, że musi utworzyć temat, optymistyczne zakładanie, że klaster Kafki, w którym się zapisuje, ma włączone automatyczne tworzenie tematów, lub poleganie na wewnętrznych interfejsach API i radzenie sobie z konsekwencjami braku gwarancji zgodności. Teraz w Kafce dostępne jest znacznie lepsze rozwiązanie, które polega na użyciu interfejsu AdminClient API, za pomocą którego sprawdzamy, czy dany temat istnieje, a jeśli nie, tworzymy go od ręki.

W tym rozdziale najpierw przedstawimy przegląd AdminClient API, a potem przejdziemy do szczegółów dotyczących stosowania go w aplikacjach. Skoncentrujemy się na najczęściej wykorzystywanych funkcjonalnościach: zarządzaniu tematami, grupach konsumentów i konfiguracji encji.

Przegląd interfejsu AdminClient

Jeśli dopiero zaczynasz korzystać z interfejsu AdminClient Kafki, warto poznać jego podstawowe zasady projektowe. Gdy zrozumiesz, jak zaprojektowano AdminClient i jak należy go używać, specyfika poszczególnych metod stanie się znacznie bardziej intuicyjna.

Asynchroniczny i ostatecznie spójny interfejs API

Być może najważniejszą cechą interfejsu AdminClient Kafki jest jego asynchroniczność. Każda metoda kończy wykonywanie natychmiast po dostarczeniu żądania do kontrolera klastra i zwraca co najmniej jeden obiekt Future. Obiekty Future są wynikiem operacji asynchronicznych i mają metody do sprawdzania statusu operacji asynchronicznej, anulowania jej, oczekiwania na jej zakończenie i wykonywania funkcji po jej zakończeniu. AdminClient Kafki opakowuje obiekty Future w obiekty Result, które zapewniają metody oczekiwania na zakończenie operacji oraz metody pomocnicze dla typowych operacji uzupełniających. Metoda KafkaAdminClient.createTopics zwraca np. obiekt CreateTopicsResult, który pozwala poczekać na utworzenie wszystkich tematów, sprawdzić indywidualnie stan każdego tematu i pobrać konfigurację określonego tematu po jego utworzeniu.

Ponieważ propagacja metadanych Kafki z kontrolera do brokerów jest asynchroniczna, obiekty Future zwracane przez interfejsy AdminClient API są uznawane za zakończone, gdy stan kontrolera zostanie w pełni zaktualizowany. Na tym etapie nie każdy broker może mieć informacje o nowym stanie, stąd żądanie listTopics może zostać obsłużone przez broker, który nie jest zaktualizowany i nie zawiera ostatnio utworzonego tematu. Ta właściwość jest nazywana **spójnością ostateczną** (ang. *eventual consistency*): ostatecznie każdy broker będzie wiedział o każdym temacie, ale nie możemy zagwarantować, kiedy dokładnie to nastąpi.

Opcje

Każda metoda interfejsu AdminClient przyjmuje jako argument obiekt Options, który jest charakterystyczny dla danej metody. Metoda listTopics przyjmuje jako argument np. obiekt ListTopicsOptions, a describeCluster obiekt DescribeClusterOptions. Obiekty te zawierają różne ustawienia dotyczące sposobu obsługi żądania przez broker. Jedynym ustawieniem wspólnym dla wszystkich metod AdminClient jest timeoutMs, które kontroluje, jak długo klient będzie czekał na odpowiedź z klastra, zanim zgłosi wyjątek TimeoutException. Ogranicza to czas, w którym aplikacja może być blokowana przez operację interfejsu AdminClient. Do innych opcji należą m.in. ustawienia, czy metoda listTopics powinna zwracać również tematy wewnętrzne albo czy metoda descriptionOptions powinna zwracać także operacje, które klient może wykonywać w klastrze.

Hierarchia płaska

Wszystkie operacje administracyjne obsługiwane przez protokół Apache Kafki są zaimplementowane bezpośrednio w klasie KafkaAdminClient. Nie ma hierarchii obiektów ani przestrzeni nazw. Jest to nieco kontrowersyjne, ponieważ interfejs może być dość duży i przytłaczający, ale główną zaletą jest to, że jeśli chcesz się dowiedzieć, jak programowo wykonać dowolną operację administracyjną w Kafce, musisz przeszukać tylko jeden dokument JavaDoc, a autouzupełnianie w Twoim

IDE będzie bardzo przydatne. Nie musisz się zastanawiać, czy szukasz nie tam, gdzie trzeba. Jeżeli tej operacji nie ma w interfejsie AdminClient, nie została jeszcze zaimplementowana (wkład własny mile widziany!).



Jeśli jesteś zainteresowany współtworzeniem projektu Apache Kafka, zapoznaj się z naszym przewodnikiem *How to contribute* (<https://oreil.ly/8zFsj>). Zanim zajmiesz się wprowadzaniem bardziej znaczących zmian w architekturze lub protokole, zacznij od mniejszych, niekonwersyjnych poprawek błędów i ulepszeń. Zachęcamy również do wnoszenia wkładów niekodowych, takich jak raporty o błędach, ulepszenia w dokumentacji, odpowiedzi na pytania i wpisy na blogu.

Dodatkowe uwagi

Wszystkie operacje modyfikujące stan klastra — tworzenie, usuwanie i zmienianie — są obsługiwane przez kontroler. Operacje, które odczytują stan klastra, takie jak `list` i `describe`, mogą być obsługiwane przez dowolny broker i są kierowane do najmniej obciążonego brokera (na podstawie informacji posiadanych przez klient). Nie powinno to mieć wpływu na Ciebie jako użytkownika interfejsu API, ale dobrze o tym wiedzieć na wypadek zauważenia nieoczekiwanego zachowania, dostrzeżenia, że niektóre operacje kończą się powodzeniem, podczas gdy inne zawiodą, lub próby zrozumienia, dlaczego wykonywanie jakiejś operacji trwa zbyt długo.

W chwili, gdy piszemy ten rozdział (na krótko przed wydaniem platformy Apache Kafka w wersji 2.5), większość operacji administracyjnych można wykonać za pośrednictwem interfejsu AdminClient lub bezpośrednio poprzez modyfikację metadanych klastra w ZooKeeperze. Zalecamy, byś nigdy nie używał ZooKeepera bezpośrednio, a jeśli absolutnie musisz, zgłoś to programistom Kafki jako błąd. Wynika to z faktu, że w niedalekiej przyszłości społeczność Apache Kafki usunie zależność od ZooKeepera, a każda aplikacja, która używa tego serwera bezpośrednio do operacji administracyjnych, będzie musiała zostać zmodyfikowana. Z drugiej strony, interfejs AdminClient API pozostanie dokładnie taki sam, tylko z inną implementacją w klastrze Kafki.

Cykl życia AdminClient API — tworzenie, konfiguracja i zamykanie

Aby użyć interfejsu AdminClient Kafki, najpierw trzeba skonstruować instancję klasy AdminClient. Jest to całkiem proste:

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
AdminClient admin = AdminClient.create(props);
// TODO: zrobić coś użytecznego z interfejsem AdminClient
admin.close(Duration.ofSeconds(30));
```

Statyczna metoda `create` przyjmuje jako argument obiekt `Properties` z konfiguracją. Jedyną obowiązkową konfiguracją jest identyfikator URI klastra: lista rozdzielonych przecinkami brokerów, z którymi należy się połączyć. Jak zwykle w środowiskach produkcyjnych należy określić co najmniej

trzy brokery na wypadek, gdyby jeden był w danym momencie niedostępny. Sposób konfigurowania bezpiecznego i uwiarygodnionego połączenia omówimy osobno w rozdziale 11.

Jeżeli uruchomisz interfejs `AdminClient`, w odpowiednim momencie powinieneś go również zamknąć. Należy pamiętać, że po wywołaniu `close` nadal mogą być wykonywane pewne operacje `AdminClient`. Dlatego metoda `close` przyjmuje parametr limitu czasu. Po wywołaniu `close` nie można wywoływać żadnych innych metod i wysyłać kolejnych żądań, ale klient będzie czekał na odpowiedzi, aż upłynie limit czasu. Gdy to nastąpi, klient przerwie wszystkie trwające operacje, zgłaszając wyjątek limitu czasu, i zwolni wszystkie zasoby. Wywołanie `close` bez określenia limitu czasu oznacza, że klient będzie czekał tak długo, aż zakończone zostaną wszystkie trwające operacje.

Prawdopodobnie pamiętasz z rozdziałów 3. i 4., że `KafkaProducer` i `KafkaConsumer` mają sporo ważnych parametrów konfiguracyjnych. Na szczęście interfejs `AdminClient` jest znacznie prostszy i nie ma w nim zbyt wiele do skonfigurowania. O wszystkich parametrach konfiguracyjnych możesz przeczytać w dokumentacji Kafki (<https://oreil.ly/0kjKE>). Istotne według nas parametry konfiguracyjne opisaliśmy w kolejnych punktach.

client.dns.lookup

Ta konfiguracja została wprowadzona w wydaniu Apache Kafka 2.1.0.

Domyślnie Kafka weryfikuje, rozwiązuje i tworzy połączenia na podstawie nazwy hosta podanej w konfiguracji serwera ładowania początkowego (a później w nazwach zwracanych przez brokery określone w konfiguracji `advertised.listeners`). Ten prosty model z reguły się sprawdza, ale nie obejmuje dwóch ważnych przypadków użycia: stosowania aliasów DNS, zwłaszcza w konfiguracji ładowania początkowego, oraz korzystania z pojedynczego serwera DNS, który jest mapowany na wiele adresów IP. Choć te dwa przypadki użycia wydają się podobne, nieco się różnią. Przyjrzyjmy się szerzej każdemu z tych wzajemnie wykluczających się scenariuszy.

Korzystanie z aliasu DNS

Załóżmy, że mamy wiele brokerów z następującą konwencją nazewnictwa: *broker1.hostname.com*, *broker2.hostname.com* itd. Zamiast określać je wszystkie w konfiguracji serwera ładowania początkowego, co może być trudne w utrzymywaniu, można utworzyć pojedynczy alias DNS, który będzie mapowany na wszystkie z nich. Do ładowania początkowego użyjemy *all-brokers.hostname.com*, ponieważ tak naprawdę nie obchodzi nas, który broker otrzyma początkowe połączenie od klientów. To wszystko jest bardzo wygodne, chyba że do uwiarygodniania używamy SASL. W takim przypadku klient spróbuje uwierzytelnić *all-brokers.hostname.com*, ale pryncypałem serwera będzie *broker2.hostname.com*. Jeżeli nazwy nie będą się zgadzały, SASL odmówi uwierzytelnienia (certyfikat brokera mógłby być atakiem typu *man-in-the-middle*), a próba nawiązania połączenia zakończy się niepowodzeniem.

W tym scenariuszu powinieneś użyć ustawienia `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. W przypadku tej konfiguracji klient „rozszerza” alias DNS, a rezultat jest taki sam, jakbyś na oryginalnej liście ładowania początkowego uwzględnił wszystkie nazwy brokerów, z którymi łączy się alias DNS.

Nazwa DNS z wieloma adresami IP

W nowoczesnych architekturach sieciowych powszechne jest umieszczanie wszystkich brokerów za serwerem proxy lub systemem równoważenia obciążenia. Jest to szczególnie powszechne, gdy korzysta się z Kubernetesa, gdzie mechanizmy równoważenia obciążenia są niezbędne, by możliwe były połączenia spoza klastra Kubernetesa. W takich przypadkach moduły równoważenia obciążenia nie powinny stać się pojedynczym punktem awarii. Dlatego bardzo często zdarza się, że *broker1.hostname.com* wskazuje listę adresów IP, z których wszystkie są rozwiązywane na moduły równoważenia obciążenia i wszystkie kierują ruch do tego samego brokera. Te adresy IP również mogą się zmieniać wraz z upływem czasu. Domyślnie klient Kafki będzie próbował połączyć się po prostu z pierwszym adresem IP, na który rozwiązuje się nazwa hosta. Oznacza to, że jeśli ten adres IP stanie się niedostępny, klient nie nawiąże połączenia, mimo że broker będzie w pełni dostępny. Dlatego zdecydowanie zaleca się użycie konfiguracji `client.dns.lookup=use_all_dns_ips`, by mieć pewność, że klient nie przegapi korzyści płynących z wysoce dostępnej warstwy równoważenia obciążenia.

request.timeout.ms

Ta konfiguracja ogranicza czas, który aplikacja może poświęcić na oczekiwanie na odpowiedź interfejsu AdminClient. Obejmuje to czas przeznaczony na ponawianie prób, jeżeli klient otrzyma nieodwracalny błąd.

Wartość domyślna to 120 sekund, co jest dość długim okresem, ale w przypadku niektórych operacji AdminClient, zwłaszcza poleceń zarządzania grupami konsumentów, oczekiwanie na odpowiedź może chwilę potrwać. Jak wspomnieliśmy w podrozdziale „Przegląd interfejsu AdminClient”, każda metoda AdminClient API przyjmuje obiekt `Options`, który może zawierać wartość limitu czasu odnoszącą się konkretnie do danego wywołania. Jeśli operacja AdminClient znajduje się na kluczowej ścieżce wykonywania Twojej aplikacji, powinieneś użyć niższej wartości limitu czasu, a brak terminowej odpowiedzi od Kafki obsłużyć w inny sposób. Typowym przykładem są usługi próbujące przy pierwszym uruchomieniu zweryfikować istnienie określonych tematów, ale jeśli Kafka nie odpowie w ciągu 30 sekund, powinieneś kontynuować uruchamianie serwera, a walidację istnienia tematów odłożyć na później (lub pominąć to sprawdzanie całkowicie).

Podstawowe zarządzanie tematami

Skoro utworzyliśmy i skonfigurowaliśmy AdminClient, nadszedł czas, aby zobaczyć, co możemy z nim zrobić. Najczęstszym przypadkiem użycia interfejsu AdminClient Kafki jest zarządzanie tematami. Obejmuje to wyświetlanie tematów, generowanie ich opisów oraz tworzenie i usuwanie tematów.

Zacznijmy od wyświetlenia listy wszystkich tematów w klastrze:

```
ListTopicsResult topics = admin.listTopics();
topics.names().get().forEach(System.out::println);
```

Zwróć uwagę, że `admin.listTopics()` zwraca obiekt `ListTopicsResult`, który jest cienką warstwą opakowującą kolekcję obiektów `Future`. Zauważ także, że metoda `topics.names()` zwraca zbiór

właściwości `name` obiektów `Future`. Kiedy w danym obiekcie `Future` wywołamy `get()`, wątek wykonujący będzie czekał, aż serwer odpowie zbiorem nazw tematów, lub otrzymamy wyjątek limitu czasu. Gdy mamy już listę, iterujemy przez nią, aby wypisać wszystkie nazwy tematów.

Teraz spróbujmy czegoś bardziej ambitnego: sprawdzimy, czy istnieje określony temat, a jeśli nie, utworzymy go. Jednym ze sposobów zweryfikowania, czy dany temat istnieje, jest pobranie listy wszystkich tematów i sprawdzenie, czy żądany temat znajduje się na liście. W dużym klastrze może to być nieefektywne. Ponadto czasami potrzebujemy sprawdzić coś więcej niż tylko istnienie tematu — chcemy się upewnić, że temat ma odpowiednią liczbę partycji i replik. Kafka Connect i rejestr tematów Confluenta używają tematu Kafki do przechowywania konfiguracji. Po uruchomieniu weryfikują, czy temat konfiguracji istnieje, czy ma tylko jedną partycję, by zagwarantować, że zmiany w konfiguracji będą nadchodzić w określonej kolejności, czy ma trzy repliki zapewniające dostępność oraz czy temat jest skompaktowany, co umożliwia bezterminowe przechowywanie starej konfiguracji:

```
DescribeTopicsResult demoTopic = admin.describeTopics(TOPIC_LIST); ❶

try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get(); ❷
    System.out.println("Opis tematu demo:" + topicDescription);

    if (topicDescription.partitions().size() != NUM_PARTITIONS) { ❸
        System.out.println("Temat ma nieodpowiednią liczbę partycji. Zakończenie.");
        System.exit(-1);
    }
} catch (ExecutionException e) { ❹
    // Kończymy wykonywanie wcześniej w przypadku niemal wszystkich wyjątków
    if (! (e.getCause() instanceof UnknownTopicOrPartitionException)) {
        e.printStackTrace();
        throw e;
    }

    // Jeżeli dotarliśmy tutaj, temat nie istnieje
    System.out.println("Temat " + TOPIC_NAME +
        " nie istnieje. Teraz go utworzymy");
    // Zwróć uwagę, że liczba partycji i replik jest opcjonalna. Jeżeli te wartości nie zostaną określone,
    // użyte zostaną wartości domyślne skonfigurowane w brokerze Kafki
    CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(new
    NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❺

    // Sprawdzamy, czy temat został utworzony pomyślnie:
    if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ❻
        System.out.println("Temat ma nieodpowiednią liczbę partycji.");
        System.exit(-1);
    }
}
```

- ❶ Aby sprawdzić, czy istnieje określony temat z poprawną konfiguracją, wywołujemy funkcję `describeTopics()` z listą nazw tematów, które chcemy zweryfikować. Spowoduje to zwrócenie obiektu `DescribeTopicResult`, który opakowuje mapowanie nazw tematów na opisy obiektów `Future`.

- 2 Dowiedziałeś się już, że jeśli czekamy na zakończenie `Future`, używając metody `get()`, możemy uzyskać pożądany rezultat, w tym przypadku `TopicDescription`. Istnieje jednak również ryzyko, że serwer nie będzie mógł poprawnie ukończyć żądania — jeżeli dany temat nie istnieje, serwer nie będzie mógł odpowiedzieć jego opisem. W takiej sytuacji serwer odeśle błąd, a `Future` zakończy się rzuceniem wyjątku `ExecutionException`. Przyczyną (cause) rzucenia wyjątku będzie rzeczywisty błąd przesłany przez serwer. Ponieważ chcemy obsłużyć ewentualność, że temat nie istnieje, obsługujemy te wyjątki.
- 3 Jeśli temat istnieje, `Future` kończy się zwróceniem obiektu `TopicDescription`, który zawiera listę wszystkich partycji danego tematu oraz dla każdej partycji, w której broker jest liderem, listy replik i listy zsynchronizowanych replik. Zwróć uwagę, że nie obejmuje to konfiguracji tematu. Konfigurację omówimy dalej w tym rozdziale.
- 4 Zwróć uwagę, że wszystkie obiekty wynikowe `AdminClient` API zgłaszają `ExecutionException`, gdy Kafka odpowiada błędem. Dzieje się tak, ponieważ wyniki `AdminClient` są opakowanymi obiektami `Future`, a te opakowują wyjątki. Zawsze musisz sprawdzić przyczynę rzucenia wyjątku `ExecutionException`, by uzyskać błąd zwrócony przez Kafkę.
- 5 Jeżeli temat nie istnieje, tworzymy nowy temat. Podczas tworzenia tematu możesz określić samą nazwę i dla wszystkich szczegółów użyć wartości domyślnych. Możesz także określić liczbę partycji, liczbę replik i konfigurację.
- 6 Na koniec należy poczekać na zakończenie tworzenia tematu i ewentualnie zweryfikować wynik. W tym przykładzie sprawdzamy liczbę partycji. Ponieważ podczas tworzenia tematu określiliśmy ich liczbę, jesteśmy prawie pewni, że wynik jest poprawny. Sprawdzanie wyniku jest bardziej powszechne, jeżeli podczas tworzenia tematu polega się na domyślnych ustawieniach brokera. Zwróć uwagę, że skoro ponownie wywołujemy `get()`, aby sprawdzić wyniki `CreateTopic`, ta metoda może zgłosić wyjątek. W tym scenariuszu powszechny jest wyjątek `TopicExistsException` i powinien go obsługiwać (np. przez generowanie opisu tematu w celu zweryfikowania poprawności konfiguracji).

Skoro mamy temat, usuńmy go:

```
admin.deleteTopics(TOPIC_LIST).all().get();

// Sprawdzamy, czy został usunięty. Zauważ, że z powodu asynchronicznej natury operacji usuwania
// możliwe jest, iż na tym etapie temat wciąż istnieje
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get();
    System.out.println("Temat " + TOPIC_NAME + " nadal istnieje");
} catch (ExecutionException e) {
    System.out.println("Temat " + TOPIC_NAME + " został usunięty");
}
```

Ten kod powinien być dość znajomy. Wywołujemy metodę `deleteTopics` z listą nazw tematów do usunięcia i używamy metody `get()`, aby poczekać na ukończenie tej operacji.



Chociaż ten kod jest prosty, pamiętaj, że w Kafce usuwanie tematów jest ostateczne — nie ma kosza, z którego można by uratować usunięty temat, ani weryfikacji, czy temat jest pusty i naprawdę chcemy go usunąć. Usunięcie niewłaściwego tematu może oznaczać nieodwracalną utratę danych, więc używaj tej metody ze szczególną ostrożnością.

We wszystkich dotychczasowych przykładach stosowaliśmy blokujące wywołanie `get()` w obiekcie `Future` zwracanym przez różne metody `AdminClient` API. W większości przypadków jest to wystarczające — operacje administracyjne są rzadkie, a czekanie, aż operacja się powiedzie lub przekroczy limit czasu, jest zwykle akceptowalne. Istnieje jeden wyjątek: gdy zapisujesz na serwerze, który z założenia ma przetwarzać dużą liczbę żądań administracyjnych. W takim przypadku podczas oczekiwania na odpowiedź Kafki nie powinieneś blokować wątków serwera. Powinieneś kontynuować przyjmowanie żądań od użytkowników i wysyłać je do Kafki, a gdy Kafka odpowie, wysłać odpowiedź do klienta. W takich scenariuszach wszechstronność `KafkaFuture` staje się całkiem przydatna. Oto bardzo prosty przykład:

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);

    DescribeTopicsResult demoTopic = admin.describeTopics( ❸
        Collections.singletonList(topic),
        new DescribeTopicsOptions().timeoutMs(timeoutMs));

    demoTopic.values().get(topic).whenComplete( ❹
        new KafkaFuture.BiConsumer<TopicDescription, Throwable>() {
            @Override
            public void accept(final TopicDescription topicDescription,
                               final Throwable throwable) {
                if (throwable != null) {
                    request.response().end("Błąd podczas próby opisanie tematu " +
                        topic + " z uwagi na " + throwable.getMessage()); ❺
                } else {
                    request.response().end(topicDescription.toString()); ❻
                }
            }
        })
    });
}).listen(8080);
```

- ❶ Używamy komponentu `Vert.x` do utworzenia prostego serwera HTTP. Za każdym razem, gdy ten serwer odbiera żądanie, wywołuje procedurę obsługi żądań `requestHandler`, którą tutaj definiujemy.
- ❷ Żądanie zawiera jako parametr nazwę tematu, a my odpowiemy opisem tego tematu.
- ❸ Wywołujemy jak zwykle metodę `AdminClient.describeTopics` i otrzymujemy w odpowiedzi opakowany `Future`.
- ❹ Zamiast używać blokującego wywołania `get()`, konstruujemy funkcję, która zostanie wywołana po zakończeniu `Future`.
- ❺ Gdy `Future` zakończy się z wyjątkiem, wysyłamy błąd do klienta HTTP.
- ❻ Jeśli `Future` zakończy się pomyślnie, odpowiadamy klientowi opisem tematu.

W tym przypadku istotne jest, że nie czekamy na odpowiedź Kafki. `DescribeTopicResult` wyśle odpowiedź do klienta HTTP, gdy nadejdzie odpowiedź z Kafki. W tym czasie serwer HTTP może kontynuować przetwarzanie innych żądań. Możesz sprawdzić to zachowanie, używając `SIGSTOP` do wstrzymania Kafki (nie próbuj tego w środowisku produkcyjnym!) i wysyłając dwa żądania HTTP

do Vert.x: jedno z długim limitem czasu, a drugie z krótką wartością. Nawet jeśli drugie żądanie wyśle się po pierwszym, odpowie wcześniej z uwagi na niższą wartość limitu czasu i nie zablokuje wykonywania po pierwszym żądaniu.

Zarządzanie konfiguracją

Zarządzanie konfiguracją odbywa się poprzez opisywanie i aktualizowanie kolekcji zasobów konfiguracyjnych (`ConfigResource`). Zasobami konfiguracyjnymi mogą być brokery, rejestratory brokerów i tematy. Sprawdzanie i modyfikowanie brokerów oraz konfiguracji rejestrowania brokerów zwykle odbywa się za pomocą takich narzędzi jak `kafka-config.sh` lub innych do zarządzania Kafką, ale dość powszechne jest sprawdzanie i aktualizowanie konfiguracji tematów z poziomu aplikacji, które ich używają.

Wiele aplikacji w celu zapewnienia poprawnego działania opiera się na skompaktowanych tematach. Aplikacje te zazwyczaj okresowo (na wszelki wypadek częściej niż trwa domyślny okres retencji) będą sprawdzać, czy temat jest rzeczywiście skompaktowany, i podejmować działania w celu poprawienia konfiguracji tematu, jeśli tak nie jest.

Oto przykład, jak to się robi:

```
ConfigResource configResource =
    new ConfigResource(ConfigResource.Type.TOPIC, TOPIC_NAME); ❶
DescribeConfigsResult configsResult =
    admin.describeConfigs(Collections.singleton(configResource));
Config configs = configsResult.all().get().get(configResource);

// Wypisujemy konfiguracje inne niż domyślne
configs.entries().stream().filter(
    entry -> !entry.isDefault()).forEach(System.out::println); ❷

// Sprawdzamy, czy temat jest skompaktowany
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
    TopicConfig.CLEANUP_POLICY_COMPACT);
if (!configs.entries().contains(compaction)) {
    // Jeżeli temat nie jest skompaktowany, kompaktujemy go
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET)); ❸
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Temat " + TOPIC_NAME + " jest skompaktowany");
}
```

- ❶ Jak wspomnieliśmy wcześniej, istnieje kilka typów `ConfigResource`; tutaj sprawdzamy konfigurację dla konkretnego tematu. W jednym żądaniu można określić wiele różnych zasobów różnych typów.
- ❷ Wynikiem `defineConfigs` jest mapowanie poszczególnych zasobów `ConfigResource` na kolekcje konfiguracji. Każdy wpis konfiguracji ma metodę `isDefault()`, która informuje, które konfiguracje zostały zmodyfikowane. Konfiguracja tematu jest uważana za niedomyślną, jeżeli użytkownik skonfigurował temat w taki sposób, aby miał wartość inną niż domyślna, lub jeśli

konfiguracja na poziomie brokera została zmodyfikowana, a utworzony temat odziedziczył tę niedomyślną wartość z brokera.

- 3 Aby zmodyfikować konfigurację, określamy mapę `ConfigResource`, którą chcemy zmodyfikować, oraz kolekcję operacji. Każda operacja modyfikacji konfiguracji składa się z wpisu konfiguracji (nazwy i wartości konfiguracji; w tym przypadku `cleanup.policy` to nazwa konfiguracji, a `compact` to wartość) oraz typu operacji. W Kafce mamy cztery typy operacji modyfikujących konfigurację: `SET`, który ustawia wartość konfiguracyjną, `DELETE` usuwającą wartość i przywracającą wartość domyślną, `APPEND` oraz `SUBTRACT`. Dwie ostatnie dotyczą tylko konfiguracji z typem `List` i pozwalają na dodawanie wartości do listy i usuwanie ich z niej bez konieczności przesyłania za każdym razem do Kafki całej listy.

Generowanie opisu konfiguracji może być zaskakująco przydatne w sytuacji awaryjnej. Zdarzyło nam się, że podczas uaktualniania plik konfiguracyjny dla brokerów został przypadkowo podmieniony na uszkodzoną kopię. Zostało to wykryte po ponownym uruchomieniu pierwszego brokera i zauważeniu, że uruchomienie się nie powiodło. Zespół nie miał sposobu na odzyskanie oryginału i przygotowaliśmy się na wiele prób i błędów, próbując zrekonstruować poprawną konfigurację i przywrócić broker do życia. Sytuację uratował inżynier SRE, który połączył się z jednym z pozostałych brokerów i zrzucił jego konfigurację za pomocą interfejsu `AdminClient`.

Zarządzanie grupą konsumentów

Wspomnieliśmy wcześniej, że w przeciwieństwie do większości kolejek komunikatów, Kafka umożliwia ponowne przetwarzanie danych w dokładnie tej samej kolejności, w jakiej zostały one skonsurowane i przetworzone wcześniej. W rozdziale 4., w którym omawialiśmy grupy konsumentów, wyjaśniliśmy, jak używać interfejsów API konsumentów do cofania się i ponownego odczytywania starszych komunikatów z danego tematu. Jednak korzystanie z tych API oznacza, że wcześniej musisz zaprogramować możliwość ponownego przetwarzania danych w swojej aplikacji. Twoja aplikacja sama w sobie musi udostępniać funkcjonalność ponownego przetwarzania danych.

Istnieje kilka scenariuszy, w których aplikacja powinna przetwarzać komunikaty ponownie, nawet jeśli ta funkcjonalność nie została wcześniej wbudowana w aplikację. Jednym z takich przypadków jest rozwiązywanie problemów z nieprawidłowo działającą aplikacją podczas incydentu. Innym jest przygotowanie aplikacji do uruchomienia w nowym klastrze podczas scenariusza przełączania awaryjnego i odzyskiwania sprawności po katastrofie (omówimy to szerzej w rozdziale 9., gdzie będziemy zajmować się technikami odzyskiwania po katastrofie).

W tej sekcji przyjrzymy się, w jaki sposób można użyć interfejsu `AdminClient` do programowego eksplorowania i modyfikowania grup konsumentów oraz przesunięć, które zostały zatwierdzone przez te grupy. W rozdziale 10. omówimy dostępne zewnętrzne narzędzia do wykonywania tych samych operacji.

Eksploracja grup konsumentów

Jeśli chcesz eksplorować i modyfikować grupy konsumentów, pierwszym krokiem jest wypisanie ich listy:


```
admin.listConsumerGroups().valid().get().forEach(System.out::println);
```

Zwróć uwagę, że w wyniku użycia metody `valid()` kolekcja zwracana przez `get()` będzie zawierać tylko te grupy konsumentów, które klaster zwrócił bez błędów, jeśli takie istnieją. Wszelkie błędy będą całkowicie ignorowane i nie będą zgłaszane jako wyjątki. Do pobrania wszystkich wyjątków można użyć metody `errors()`. Jeżeli użyjesz `all()`, tak jak robiliśmy w innych przykładach, to jako wyjątek zgłoszony zostanie tylko pierwszy błąd zwrócony przez klaster. Do prawdopodobnych przyczyn takich błędów należy m.in. autoryzacja, gdy nie masz uprawnień do przeglądania grupy, lub niedostępność koordynatora dla niektórych grup konsumentów.

Jeśli chcemy uzyskać więcej informacji o niektórych grupach, możemy wygenerować ich opisy:

```
ConsumerGroupDescription groupDescription = admin
    .describeConsumerGroups(CONSUMER_GRP_LIST)
    .describedGroups().get(CONSUMER_GROUP).get();
System.out.println("Opis grupy " + CONSUMER_GROUP + ":" + groupDescription);
```

Opis zawiera mnóstwo informacji o grupie. Obejmuje to członków grupy, ich identyfikatory i hosty, przypisane do nich partycje, algorytm użyty do przypisania partycji oraz host koordynatora grupy. Taki opis jest bardzo przydatny podczas rozwiązywania problemów z grupami konsumentów. Brakuje w nim jednak jednej z najważniejszych informacji o grupie konsumentów — nie wskazuje, jakie były ostatnie przesunięcia zatwierdzone przez grupę dla poszczególnych konsumowanych partycji i jak bardzo pozostaje ona w tyle za najnowszymi komunikatami z dziennika.

Niegdyś jedynym sposobem uzyskania tych informacji było parsowanie komunikatów zatwierdzenia, które grupy konsumentów zapisywały w wewnętrznym temacie Kafki. Chociaż ta stara metoda spełniała swoje zadanie, Kafka nie gwarantuje zgodności formatów komunikatów wewnętrznych, dlatego nie jest ona już zalecana. Przyjrzymy się, jak te informacje pozwala pobrać `AdminClient` API Kafki:

```
Map<TopicPartition, OffsetAndMetadata> offsets =
    admin.listConsumerGroupOffsets(CONSUMER_GROUP)
        .partitionsToOffsetAndMetadata().get(); ❶

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new HashMap<>();

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ❷
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> latestOffsets =
    admin.listOffsets(requestLatestOffsets).all().get();

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ❸
    String topic = e.getKey().topic();
    int partition = e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offset();

    System.out.println("Grupa konsumentów " + CONSUMER_GROUP
        + " zatwierdziła przesunięcie " + committedOffset
        + " dla tematu " + topic + " partycji " + partition
        + ". Ostatnim przesunięciem w partycji jest "
```

```
+ latestOffset + ", stąd grupa konsumentów pozostaje o następującą liczbę
  rekordów w tyle: "
+ (latestOffset - committedOffset));
}
```

- ❶ Pobieramy mapę wszystkich tematów i partycji obsługiwanych przez daną grupę konsumentów oraz najnowsze zatwierdzone dla nich przesunięcia. Zwróć uwagę, że inaczej niż `describeConsumerGroups`, metoda `listConsumerGroupOffsets` przyjmuje tylko pojedynczą grupę odbiorców, a nie kolekcję.
- ❷ Chcemy uzyskać przesunięcie ostatniego komunikatu w partycji dla wszystkich tematów i partycji z wyników. `OffsetSpec` ma trzy bardzo wygodne implementacje: `earliest()`, `latest()` i `forTimestamp()`, które pozwalają pobrać najstarsze i najnowsze przesunięcia w partycji, a także przesunięcie rekordu zapisanego w określonym czasie lub bezpośrednio po nim.
- ❸ Na koniec iterujemy przez wszystkie partycje i dla każdej z nich wypisujemy ostatnie zatwierdzone przesunięcie, ostatnie przesunięcie w partycji i opóźnienie między nimi.

Modyfikowanie grup konsumentów

Do tej pory badaliśmy jedynie dostępne informacje. `AdminClient` ma również metody modyfikowania grup konsumentów: usuwania grup, usuwania członków, usuwania zatwierdzonych przesunięć i modyfikowania przesunięć. Są one powszechnie używane przez inżynierów SRE do tworzenia na poczekaniu narzędzi do odzyskiwania sprawności po awarii.

Z wymienionych działań najbardziej przydatne jest modyfikowanie przesunięć. Usuwanie przesunięć może wydawać się prostym sposobem na zmuszenie konsumenta do „zaczynania od zera”, ale tak naprawdę zależy to od konfiguracji danego konsumenta. Czy jeśli konsument rozpocznie przetwarzanie i nie znajdzie żadnych przesunięć, to zacznie od początku, czy może przeskoczy do najnowszego komunikatu? Nie dowiemy się, dopóki nie będziemy mieć wartości `auto.offset.reset`. Bezpośrednie modyfikowanie zatwierdzonych przesunięć przez ustawianie wartości najstarszych dostępnych przesunięć zmusi konsument do rozpoczęcia przetwarzania od początku tematu i zasadniczo spowoduje „zresetowanie” konsumenta.

Należy pamiętać, że grupy konsumentów nie otrzymują aktualizacji, gdy zmieniają się przesunięcia w temacie przesunięć. Odczytują przesunięcia tylko wtedy, gdy konsumentowi przypisywana jest nowa partycja, lub podczas uruchamiania. Aby uniemożliwić wprowadzanie w przesunięciach zmian, o których konsumenci nie będą wiedzieć (i w związku z tym je nadpisać), Kafka uniemożliwia modyfikowanie przesunięć, gdy grupa konsumentów jest aktywna.

Należy również pamiętać, że jeśli aplikacja konsumentencka utrzymuje stan (a robi to większość aplikacji przetwarzania strumieniowego), wówczas resetowanie przesunięć i powodowanie, że grupa konsumentów rozpoczyna konsumowanie od początku tematu, może mieć dziwny wpływ na przechowywany stan. Załóżmy, że masz aplikację do przetwarzania strumieniowego, która stale liczy buty sprzedawane w sklepie. O godzinie 8:00 odkrywasz błąd w danych wejściowych i chcesz od nowa przeliczyć stan od godziny 3:00. Jeśli zresetujesz przesunięcia na 3:00 bez odpowiedniego zmodyfikowania zapisanej agregacji, to każdą sprzedaną tego dnia parę butów policzysz dwukrotnie (przetworzysz również wszystkie dane między 3:00 a 8:00, ale założmy, że to jest konieczne do usunięcia

błędu). Musisz zadbać o odpowiednią aktualizację zapisanego stanu. W środowisku programistycznym przed zresetowaniem przesunąć do początku tematu wejściowego zwykle całkowicie usuwamy magazyn stanów.

Mając na uwadze wszystkie te ostrzeżenia, przyjrzyjmy się poniższemu przykładowi:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> earliestOffsets =  
    admin.listOffsets(requestEarliestOffsets).all().get(); ❶  
  
Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();  
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:  
    earliestOffsets.entrySet()) {  
    resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ❷  
}  
  
try {  
    admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ❸  
} catch (ExecutionException e) {  
    System.out.println("Niepowodzenie aktualizacji przesunąć zatwierdzonych przez grupę "  
        + CONSUMER_GROUP + " z błędem " + e.getMessage());  
    if (e.getCause() instanceof UnknownMemberIdException)  
        System.out.println("Sprawdź, czy grupa konsumentów jest nadal aktywna."); ❹  
}
```

- ❶ W celu zresetowania grupy konsumentów, aby zaczęła przetwarzanie od najstarszego przesunięcia, najpierw musimy pobrać najstarsze przesunięcia. Przypomina to pobieranie najnowszych przesunąć, które pokazaliśmy w poprzednim przykładzie.
- ❷ W tej pętli mapę z wartościami `ListOffsetsResultInfo` zwróconymi przez `listOffsets` przekształcamy w mapę z wartościami `OffsetAndMetadata`, które są wymagane przez metodę `alterConsumerGroupOffsets`.
- ❸ Po wywołaniu `alterConsumerGroupOffsets` czekamy na zakończenie wykonywania `Future`, byśmy mogli sprawdzić, czy zakończyło się ono pomyślnie.
- ❹ Jednym z najczęstszych powodów niepowodzenia wykonywania `alterConsumerGroupOffsets` jest to, że nie zatrzymaliśmy najpierw grupy konsumentów (należy to zrobić przez bezpośrednie zamknięcie aplikacji konsumującej; nie ma polecenia administracyjnego do zamykania grupy konsumentów). Jeśli grupa pozostanie aktywna, koordynator konsumentów będzie postrzegał naszą próbę modyfikacji przesunąć jako zatwierdzanie przesunięcia dla danej grupy przez klient, który nie jest członkiem. W takim przypadku otrzymamy wyjątek `UnknownMemberIdException`.

Metadane klastra

Rzadko zdarza się, by aplikacja musiała bezpośrednio wykrywać jakiekolwiek informacje dotyczące klastra, z którym jest połączona. Komunikaty możesz produkować i konsumować, nie znając w ogóle liczby istniejących brokerów i nie wiedząc, który z nich jest kontrolerem. Klienci Kafki nie potrzebują tych informacji — muszą zajmować się wyłącznie tematami i partycjami.

Na wypadek gdybyś był ciekaw, Twoją żądę wiedzy powinien zaspokoić ten niewielki fragment kodu:

```
DescribeClusterResult cluster = admin.describeCluster();  
  
System.out.println("Podłączono do klastra " + cluster.clusterId().get()); ❶  
System.out.println("W klastrze są następujące brokery:");  
cluster.nodes().get().forEach(node -> System.out.println("    " + node));  
System.out.println("Kontroler to: " + cluster.controller().get());
```

- ❶ Identyfikator klastra jest identyfikatorem GUID i dlatego nie jest czytelny dla człowieka. Tak czy inaczej warto sprawdzić, czy klient połączył się z właściwym klastrem.

Zaawansowane operacje administracyjne

W tym podrozdziale omówimy kilka metod, które są rzadko stosowane i mogą być ryzykowne w użyciu, ale są niezwykle przydatne, jeśli zajdzie taka potrzeba. Są one ważne przede wszystkim dla inżynierów SRE podczas reagowania na incydenty, ale nie odwracaj nauki korzystania z ich do chwili wystąpienia incydentu. Czytaj i ćwicz, zanim będzie za późno. Zwróć uwagę, że te metody mają ze sobą niewiele wspólnego poza tym, że wszystkie pasują do tej kategorii.

Dodawanie partycji do tematu

Zwykle liczba partycji w temacie jest ustawiana podczas tworzenia tematu. Ponieważ każda partycja może mieć bardzo wysoką przepływność, dochodzenie do limitów pojemności tematu jest stosunkowo rzadkie. Ponadto jeśli komunikaty w temacie mają klucze, konsumenci mogą założyć, że wszystkie komunikaty z określonym kluczem będą zawsze trafiały do tej samej partycji i będą przetwarzane w tej samej kolejności przez ten sam konsument.

Z tych powodów dodawanie partycji do tematu jest rzadko potrzebne i potencjalnie ryzykowne. Trzeba sprawdzić, czy taka operacja nie doprowadzi do awarii jakiegokolwiek aplikacji korzystającej z danego tematu. Czasami jednak zdarza się osiągnąć górny pułap przepływności przetwarzania z istniejącymi partycjami i jedynym wyjściem jest dodanie kilku partycji.

Partycje można dodawać do kolekcji tematów za pomocą metody `createPartitions`. Pamiętaj, że jeśli spróbujesz rozszerzyć wiele tematów naraz, możliwe, że ta operacja nie zakończy się równie pomyślnie dla wszystkich.

```
Map<String, NewPartitions> newPartitions = new HashMap<>();  
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_PARTITIONS+2)); ❶  
admin.createPartitions(newPartitions).all().get();
```

- ❶ Podczas rozszerzania tematów musisz określić docelową całkowitą liczbę partycji tematu, a nie liczbę nowych partycji.



Ponieważ metoda `createPartitions` przyjmuje jako parametr całkowitą liczbę partycji, które temat będzie miał po jej wykonaniu, przed rozpoczęciem rozszerzania konieczne może być wygenerowanie opisu tematu i sprawdzenie aktualnej liczby partycji.

Usuwanie rekordów z tematu

Obecne przepisy dotyczące prywatności szczegółowo określają zasady przechowywania danych. Choć Kafka ma reguły retencji dla tematów, niestety nie zostały one zaimplementowane w sposób gwarantujący zapewnienie zgodności z tymi regulacjami. Temat z regułą 30-dniowej retencji może przechowywać starsze dane, jeśli wszystkie dane zmieszczą się w pojedynczym segmencie każdej partycji.

Metoda `deleteRecords` oznacza jako usunięte wszystkie rekordy z przesunięciami starszymi niż te określone podczas wywoływania metody i sprawi, że przestaną one być dostępne dla konsumentów Kafki. Metoda zwraca usunięte przesunięcia z największą wartością, można więc sprawdzić, czy usunięcie dokonało się zgodnie z oczekiwaniami. Pełne czyszczenie z dysku będzie odbywać się asynchronicznie. Pamiętaj, że do uzyskania przesunięć dla rekordów, które zostały zapisane w określonym czasie lub bezpośrednio po nim, można użyć metody `listOffsets`. Łącząc te dwie metody, można usuwać rekordy, które są starsze niż dowolny określony moment w czasie:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> olderOffsets =
    admin.listOffsets(requestOlderOffsets).all().get();
Map<TopicPartition, RecordsToDelete> recordsToDelete = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e :
    olderOffsets.entrySet())
    recordsToDelete.put(e.getKey(),
        RecordsToDelete.beforeOffset(e.getValue().offset()));
admin.deleteRecords(recordsToDelete).all().get();
```

Wybór lidera

Metoda `electLeader()` pozwala na uruchamianie dwóch różnych typów wyboru lidera:

Wybór preferowanego lidera

Każda partycja ma replikę główną wyznaczoną jako **preferowany lider**. Jest ona liderem preferowanym, ponieważ jeśli wszystkie partycje będą używać jako lidera preferowanej repliki, liczba liderów w każdym brokerze powinna być zrównoważona. Domyślnie Kafka co 5 minut sprawdza, czy liderem jest preferowana replika; jeśli tak nie jest, ale ta replika się kwalifikuje, wybiera ją na lidera. Jeżeli `auto.leader.rebalance.enable` ma wartość `false` lub chcesz, by odbywało się to szybciej, ten proces może wywoływać metoda `electLeader()`.

Wybór nieczystego lidera

Jeśli będąca liderem replika partycji stanie się niedostępna, a pozostałe repliki nie będą kwalifikować się do zostania liderami (zwykle z powodu braku pewnych danych), partycja zostanie pozbawiona lidera i dlatego nie będzie dostępna. Jednym ze sposobów rozwiązania tego problemu jest zainicjowanie wyborów **nieczystego lidera**, co oznacza wybranie na lidera repliki, która inaczej by się nie kwalifikowała. Spowoduje to utratę danych — utracone zostaną wszystkie zdarzenia, które zostały zapisane w starym liderze i nie zostały zreplikowane do nowego. Do inicjowania wyboru nieczystego lidera również można używać metody `electLeader()`.

Jest to metoda asynchroniczna, co oznacza, że nawet po pomyślnym zakończeniu wykonywania potrzeba trochę czasu, zanim wszystkie brokery dowiedzą się o nowym stanie, a wywołania `describeTopics()` mogą zwracać niespójne wyniki. Jeżeli uruchomisz wybór lidera dla wielu partycji, istnieje ryzyko, że operacja zakończy się sukcesem tylko dla niektórych partycji:

```

Set<TopicPartition> electableTopics = new HashSet<>();
electableTopics.add(new TopicPartition(TOPIC_NAME, 0));
try {
    admin.electLeaders(ElectionType.PREFERRED, electableTopics).all().get(); ❶
} catch (ExecutionException e) {
    if (e.getCause() instanceof ElectionNotNeededException) {
        System.out.println("Wszyscy liderzy są już liderami preferowanymi"); ❷
    }
}

```

- ❶ Wybieramy preferowanego lidera dla pojedynczej partycji określonego tematu. Możemy podać dowolną liczbę partycji i tematów. Jeśli zamiast z kolekcją partycji wywołasz to polecenie z wartością null, spowoduje to przeprowadzenie danego typu wyboru lidera dla wszystkich partycji.
- ❷ Jeżeli klastery jest w dobrym stanie, wywołanie polecenia nie spowoduje wykonania żadnych działań. Wybory preferowanego i nieczystego lidera odbywają się tylko wtedy, gdy aktualnym liderem jest replika inna niż preferowany lider.

Ponowne przypisywanie replik

Czasami nie podoba Ci się aktualna lokalizacja niektórych replik. Być może broker jest przeciążony i potrzebujesz przenieść pewne repliki, może chcesz dodać więcej replik lub przenieść wszystkie repliki z określonego brokera, aby usunąć daną maszynę, a może kilka tematów generuje tyle szumu, że trzeba je odizolować od reszty obciążenia roboczego. We wszystkich tych scenariuszach precyzyjną kontrolę nad rozmieszczeniem każdej repliki partycji zapewnia metoda `alterPartitionReassignments`. Powinieneś pamiętać, że ponowne przypisywanie replik z jednego brokera do drugiego może się wiązać z kopiowaniem dużych ilości danych pomiędzy tymi brokerami. Zwróć uwagę na dostępną przepustowość sieci i jeśli zajdzie taka potrzeba, ogranicz przepływność replikacji za pomocą kwot. Są one konfiguracją brokera, dlatego możesz wygenerować ich opis i zaktualizować za pomocą interfejsu `AdminClient`.

W tym przykładzie założmy, że mamy pojedynczy broker o identyfikatorze 0. Nasz temat ma kilka partycji, wszystkie z jedną repliką na tym brokerze. Po dodaniu nowego brokera chcemy go wykorzystać do przechowywania niektórych replik tematu. Każdą partycję z tematu przypiszemy w nieco inny sposób:

```

Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new HashMap<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
    Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
    Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹

admin.alterPartitionReassignments(reassignment).all().get();

System.out.println("Aktualnie na nowo przypisywane: " +
    admin.listPartitionReassignments().reassignments().get()); ❺
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Opis tematu demo:" + topicDescription); ❻

```

- ❶ Dodaliśmy kolejną replikę do partycji 0, umieściliśmy tę nową replikę na nowym brokerze o identyfikatorze 1, ale pozostawiliśmy lidera bez zmian.
- ❷ Nie dodaliśmy żadnych replik do partycji 1; przenieśliśmy po prostu jedną istniejącą replikę do nowego brokera. Ponieważ posiadamy tylko jedną replikę, jest ona również liderem.
- ❸ Dodaliśmy kolejną replikę do partycji 2 i uczyniliśmy ją preferowanym liderem. Następne wybory preferowanego lidera uczynią liderem nową replikę na nowym brokerze. Istniejąca replika stanie się wtedy repliką podrzędną.
- ❹ Nie toczy się żaden proces ponownego przypisywania dla partycji 3, ale gdyby tak było, spowodowałoby to jego anulowanie i przywrócenie stanu sprzed rozpoczęcia operacji ponownego przypisywania.
- ❺ Możemy wypisać trwające zmiany przypisań.
- ❻ Możemy wypisać również nowy stan, ale pamiętaj, że może minąć trochę czasu, zanim zostaną wyświetlone spójne wyniki.

Testowanie

Apache Kafka udostępnia klasę testową `MockAdminClient`, którą można zainicjować z dowolną liczbą brokerów i wykorzystać do przetestowania prawidłowego działania aplikacji bez konieczności uruchamiania rzeczywistego klastra Kafki i wykonywania na nim operacji administracyjnych. Choć `MockAdminClient` nie jest częścią interfejsu API Kafki i dlatego może ulec zmianie bez ostrzeżenia, tworzy atrapy metody, które są publiczne, a zatem sygnatury metod pozostaną zgodne. Pewnym kompromisem stosowania tej wygodnej klasy jest konieczność liczenia się z ryzykiem, że zmieni się ona i pospuje testy, dlatego miej to na uwadze.

Ta klasa testowa jest szczególnie atrakcyjna z tego względu, że niektóre popularne metody mają bardzo wszechstronne atrapy: za pomocą `MockAdminClient` możesz utworzyć tematy, a kolejne wywołanie `listTopics()` spowoduje wyświetlenie tematów, które „utworzyłeś”.

Nie wszystkie metody mają jednak zapewnione atrapy. Jeśli będziesz używać `AdminClient` z wersją 2.5 lub wcześniejszą i wywołasz metodę `incrementalAlterConfigs()` klasy `MockAdminClient`, otrzymasz wyjątek `UnsupportedOperationException`, ale możesz sobie z tym poradzić, wstrzykując własną implementację.

Aby zademonstrować, jak testować za pomocą `MockAdminClient`, zacznijmy od zaimplementowania klasy, której instancja jest tworzona z klientem administracyjnym i używa tego klienta do tworzenia tematów:

```
public TopicCreator(AdminClient admin) {
    this.admin = admin;
}

// Przykład metody, która będzie tworzyć temat, jeśli jego nazwa będzie zaczynać się od "test"
public void maybeCreateTopic(String topicName)
    throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
}
```

```

if (topicName.toLowerCase().startsWith("test")) {
    admin.createTopics(topics);

    // Zmienia konfigurację jedynie dla celów demonstracyjnych
    ConfigResource configResource =
        new ConfigResource(ConfigResource.Type.TOPIC, topicName);
    ConfigEntry compaction =
        new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
            TopicConfig.CLEANUP_POLICY_COMPACT);
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET));
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
}
}

```

Logika nie jest wyrafinowana: `maybeCreateTopic` utworzy temat, jeżeli nazwa tematu będzie zaczynać się od `test`. Modyfikujemy również konfigurację tematu, dzięki czemu możemy pokazać, jak postępować w przypadku, gdy używana przez nas metoda nie jest zaimplementowana w klasie `MockAdminClient`.



Do sprawdzania, czy metody `MockAdminClient` są wywoływane zgodnie z oczekiwaniami, oraz do wypełniania luk dla niezaimplementowanych metod używamy frameworku testowego Mockito (<https://site.mockito.org>). Mockito jest dość prostym frameworkiem do tworzenia atrap i ma przyjemne interfejsy API, dzięki czemu dobrze nadaje się do niewielkiego przykładu testu jednostkowego.

Testy zaczniemy od utworzenia instancji atrapy klienta:

```

@Before
public void setUp() {
    Node broker = new Node(0, "localhost", 9092);
    this.admin = spy(new MockAdminClient(Collections.singletonList(broker), broker)); ❶

    // Bez tego test rzuci wyjątek `java.lang.UnsupportedOperationException: Not implemented yet`
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}

```

- ❶ Instancja klasy `MockAdminClient` jest tworzona z listą brokerów (tu używamy tylko jednego) i jednym brokerem, który będzie kontrolerem. Brokery to jedynie identyfikator brokera, nazwa hosta i port — oczywiście wszystko to atrapy. Podczas wykonywania tych testów nie będzie uruchamiany żaden broker. Wykorzystamy wstrzykiwanie `spy` Mockito, abyśmy mogli później sprawdzić, czy `TopicCreator` zadziałał poprawnie.
- ❷ Tutaj używamy metod `doReturn` frameworku Mockito, by upewnić się, że atrapa klienta administratora nie zgłasza wyjątków. Testowana metoda oczekuje obiektu `AlterConfigsResult` z metodą `all()`, która zwraca obiekt `KafkaFuture`. Upewniliśmy się, że właśnie to będzie zwracać atrapa `incrementalAlterConfigs`.

Teraz, gdy mamy odpowiednią atrapę `AdminClient`, możemy użyć jej do sprawdzenia, czy metoda `maybeCreateTopic()` działa poprawnie:


```

@Test
public void testCreateTestTopic()
    throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}

@Test
public void testNotTopic() throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ❷
}

```

- ❶ Nazwa tematu zaczyna się od `test`, spodziewamy się więc, że `maybeCreateTopic()` utworzy temat. Sprawdzamy, czy metoda `createTopics()` została wywołana raz.
- ❷ Jeżeli nazwa tematu nie zaczyna się od słowa `test`, to sprawdzamy, czy metoda `createTopics()` nie została w ogóle wywołana.

Ostatnia uwaga: programiści projektu Apache Kafka opublikowali klasę `MockAdminClient` w pliku `test.jar`, dlatego upewnij się, że `pom.xml` zawiera zależność testową:

```

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.5.0</version>
  <classifier>test</classifier>
  <scope>test</scope>
</dependency>

```

Podsumowanie

`AdminClient` to użyteczne narzędzie, które warto mieć w swoim pakiecie programistycznym Kafki. Przydaje się ono programistom, którzy chcą tworzyć tematy w locie i sprawdzać, czy używane przez nich tematy są poprawnie skonfigurowane dla ich aplikacji. Jest przydatne dla również dla operatorów i inżynierów SRE, którzy chcą tworzyć oprzyrządowanie i automatyzację związane z Kafką lub muszą odzyskać sprawność po incydencie. Interfejs `AdminClient` ma tak wiele użytecznych metod, że inżynierowie SRE mogą traktować go jako szwajcarski szczyroryk dla operacji Kafki.

W tym rozdziale omówiliśmy podstawy korzystania z interfejsu `AdminClient` Kafki: zarządzanie tematami, zarządzanie konfiguracją i grupami konsumentów, a także kilka innych przydatnych metod, które warto mieć pod ręką — nigdy nie wiadomo, kiedy będą potrzebne.

Wewnętrzne mechanizmy działania Kafki

Aby uruchomić Kafkę w środowisku produkcyjnym lub pisać korzystające z niej aplikacje, nie musisz koniecznie rozumieć wewnętrznych mechanizmów działania tej platformy. Jednak wiedza o tym, jak funkcjonuje Kafka, zapewnia kontekst podczas rozwiązywania problemów lub podejmowania prób zrozumienia jej zachowania. Ponieważ omówienie poszczególnych szczegółów implementacji i decyzji projektowych wykracza poza zakres tej książki, w tym rozdziale skupimy się na kilku tematach, które są szczególnie istotne dla praktyków Kafki. Oto one:

- kontroler Kafki;
- sposób działania replikacji w Kafce;
- obsługa przez Kafkę żądań od producentów i konsumentów;
- obsługa przez Kafkę pamięci masowej, takiej jak format plików i indeksy.

Zrozumienie tych zagadnień będzie przydatne zwłaszcza podczas dostosowywania konfiguracji Kafki — poznanie mechanizmów kontrolowanych za pomocą konkretnych ustawień pozwala na korzystanie z nich świadomie i precyzyjnie, zamiast bawienia się nimi losowo.

Przynależność do klastra

Kafka wykorzystuje serwer Apache ZooKeeper do utrzymywania listy brokerów, które są aktualnie członkami klastra. Każdy broker ma unikatowy identyfikator ustawiany w pliku konfiguracyjnym brokera lub generowany automatycznie. Gdy broker uruchamia swój proces, rejestruje się za pomocą tego identyfikatora w ZooKeeperze, tworząc **węzeł efemeryczny** (<http://bit.ly/2s3MYHh>). Aby otrzymywać powiadomienia o dodawaniu brokerów lub ich usuwaniu, brokery, kontroler i niektóre narzędzia ekosystemu Kafki subskrybują w ZooKeeperze ścieżkę `/brokers/ids`, gdzie rejestrowane są brokery.

Jeśli spróbujesz uruchomić kolejny broker z tym samym identyfikatorem, otrzymasz błąd — ten nowy broker będzie próbował się zarejestrować bez powodzenia, ponieważ w ZooKeeperze będzie już istniał węzeł dla tego identyfikatora.

Gdy broker utraci połączenie z ZooKeeperem (zwykle w wyniku zatrzymania brokera, ale może to być również związane z partycjonowaniem sieci lub długą przerwą na działanie mechanizmu odzyskiwania pamięci), to węzeł efemeryczny utworzony przez broker podczas uruchamiania

zostanie automatycznie usunięty z ZooKeepera. Komponenty Kafki, które obserwują listę brokerów, zostaną powiadomione, że dany broker został usunięty.

Nawet jeśli po zatrzymaniu brokera zniknie reprezentujący go węzeł, identyfikator brokera nadal będzie istniał w innych strukturach danych. Identyfikatory brokerów dla danej repliki zawierają np. listę replik każdego tematu (zob. podrozdział „Replikacja”). Dlatego jeśli całkowicie utracisz broker i uruchomisz nowy ze starym identyfikatorem, natychmiast dołączy on do klastra w miejscu brakującego brokera i z przypisanymi tymi samymi partycjami i tematami.

Kontroler

Kontroler jest jednym z brokerów Kafki, który oprócz pełnienia zwykłych funkcji odpowiada za wybór liderów partycji. Pierwszy broker uruchomiony w klastrze staje się kontrolerem, tworząc w ZooKeeperze węzeł efemeryczny o nazwie `/controller`. Kolejne uruchamiane brokery również próbują utworzyć ten węzeł, ale otrzymują wyjątek `node already exists` (węzeł już istnieje) i „zdają sobie sprawę”, że węzeł kontrolera został już utworzony, a klaster ma już kontroler. Brokery tworzą na węźle kontrolera **obserwator ZooKeepera** (<http://bit.ly/2sKoTTN>), aby rozpocząć powiadamianie o zmianach na tym węźle. Gwarantuje to, że klaster będzie miał tylko jeden kontroler naraz.

Gdy broker pełniący funkcję kontrolera zostanie zatrzymany lub utraci połączenie z ZooKeeperem, węzeł efemeryczny zniknie. Obejmuje to każdy scenariusz, w którym klient ZooKeepera używany przez kontroler przestaje wysyłać do tego serwera sygnały heartbeat, przekraczając limit czasu `zookeeper.session.timeout.ms`. Gdy węzeł efemeryczny zniknie, pozostałe brokery w klastrze zostają powiadomione za pośrednictwem obserwatora ZooKeepera, że dany kontroler zniknął, i same podejmują próbę utworzenia węzła kontrolera w ZooKeeperze. Pierwszy węzeł, który utworzy nowy kontroler w ZooKeeperze, staje się następnym kontrolerem, podczas gdy pozostałe węzły otrzymują wyjątek „węzeł już istnieje” i odtwarzają obserwator na węźle nowego kontrolera. Każdy kolejny wybierany kontroler otrzymuje nowy, wyższy numer **epoki kontrolera** (ang. *controller epoch*), generowany przez operację warunkowej inkrementacji ZooKeepera. Brokery znają aktualną epokę kontrolera i jeśli otrzymają komunikaty od kontrolera ze starszym numerem, wiedzą, że mają je zignorować. Jest to ważne, ponieważ broker pełniący funkcję kontrolera może odłączyć się od ZooKeepera ze względu na długą przerwę spowodowaną odzyskiwaniem pamięci — podczas tej przerwy zostanie wybrany nowy kontroler. Gdy po przerwie poprzedni lider wznowi działanie, może dalej wysyłać komunikaty do brokerów, ponieważ nie będzie miał informacji o istnieniu nowego kontrolera — w takim przypadku stary kontroler jest uważany za „zombie”. Umieszczana w komunikatach epoka kontrolera, która umożliwia brokerom ignorowanie komunikatów ze starych kontrolerów, jest formą obrony przed zombie.

Zanim kontroler uruchamiany po raz pierwszy będzie mógł zacząć zarządzać metadanymi klastra i przeprowadzać wybory liderów, musi najpierw odczytać z ZooKeepera najnowszą mapę stanu replik. Aby ukryć opóźnienia, w procesie ładowania wykorzystywane są asynchroniczne interfejsy API i potokowanie żądań odczytu wysyłanych do ZooKeepera. Mimo to w klastrach z dużą liczbą partycji proces ładowania może potrwać kilka sekund — kilka testów i porównań znajdziesz we wpisie na blogu Apache Kafki 1.1.0 (<https://oreil.ly/mQpL4>).

Kiedy kontroler (obserwując odpowiednią ścieżkę ZooKeepera lub po otrzymaniu od brokera żądania `ControlledShutdownRequest`) uzna, że jakiś broker opuścił klaster, będzie wiedział, iż wszystkie partycje, które miały w tym brokerze lidera, będą potrzebowały nowego. Kontroler trawersuje przez wszystkie partycje potrzebujące nowego lidera i określa, które repliki powinny być nowym liderem (zwykle jest to następna replika na liście replik danej partycji). Następnie kontroler utrzuca nowy stan w ZooKeeperze (ponownie przy użyciu potokowych żądań asynchronicznych w celu zmniejszenia opóźnień), a potem wysyła żądanie `LeaderAndISR` do wszystkich brokerów, które zawierają repliki dla danych partycji. Żądanie to zawiera informacje o nowym liderze i replikach podrzędnych dla danych partycji. W celu zwiększenia wydajności te żądania są przysyłane partiami, więc każde żądanie zawiera informacje o nowym przywództwie dla wielu partycji, które mają replikę na tym samym brokerze. Każdy nowy lider wie, że musi zacząć obsługiwać żądania producentów i konsumentów od klientów, natomiast repliki podrzędne wiedzą, że muszą zacząć replikować komunikaty nowego lidera. Ponieważ każdy broker w klastrze ma pamięć podręczną metadanych (`Meta` → `dataCache`), która zawiera mapę wszystkich brokerów i replik w danym klastrze, kontroler wysyła wszystkim brokerom informacje o zmianie przywództwa w żądaniu aktualizacji metadanych (`Update` → `Metadata`), by mogły zaktualizować swoją pamięć podręczną. Podobny proces powtarza się, gdy broker zostanie ponownie uruchomiony — główna różnica polega na tym, że wszystkie repliki w brokerze zaczynają jako repliki podrzędne i zanim będą kwalifikować się do wybrania na lidera, muszą dogonić aktualnego lidera.

Możemy podsumować, że Kafka korzysta z funkcjonalności węzłów efemerycznych ZooKeepera do wyboru kontrolera i powiadamiania go, gdy węzły dołączają do klastra lub go opuszczają. Kontroler jest odpowiedzialny za wybór liderów spośród partycji i replik, gdy tylko zauważy, że węzły opuszczają klaster lub do niego dołączają. Aby zapobiec scenariuszowi „podziału mózgu”, w którym dwa węzły uważają, że każdy z nich jest aktualnym kontrolerem, kontroler stosuje numer epoki.

KRaft — nowy kontroler Kafki oparty na algorytmie Raft

W 2019 r. społeczność Apache Kafki rozpoczęła ambitny projekt, którego założeniem było odejście od kontrolera opartego na ZooKeeperze na rzecz kworum kontrolerów opartych na algorytmie Raft. Wersja przeglądowa nowego kontrolera o nazwie KRaft jest częścią wydania Apache Kafka 2.8. Wydanie Apache Kafka 3.0 planowane na połowę roku 2021 będzie zawierało pierwszą produkcyjną wersję KRafta, a klastry Kafki będą mogły działać z tradycyjnym kontrolerem opartym na ZooKeeperze lub z KRaftem.

Dlaczego społeczność Kafki zdecydowała się na wymianę kontrolera? Istniejący kontroler Kafki przeszedł już kilka przeróbek, jednak pomimo ulepszania sposobu, w jaki wykorzystuje ZooKeeper do przechowywania informacji o tematach, partycjach i replikach, stało się jasne, że ten model nie będzie się skalował do tak dużej liczby partycji, jaką powinna obsługiwać Kafka. Zmiana była motywowana kilkoma dobrze znanymi zagadnieniami:

- Aktualizacje metadanych są zapisywane w ZooKeeperze synchronicznie, ale wysyłane do brokerów asynchronicznie. Asynchroniczne jest także otrzymywanie aktualizacji od ZooKeepera. Wszystko to prowadzi do skrajnych przypadków, w których metadane są niespójne między brokerami, kontrolerem i ZooKeeperem. Takie przypadki są trudne do wykrycia.

- Po każdym restarcie kontroler musi odczytać z ZooKeepera całe metadane dla wszystkich brokerów i partycji, a następnie wysłać te metadane do wszystkich brokerów. Pomimo wieloletnich wysiłków pozostaje to głównym wąskim gardłem — wraz ze wzrostem liczby partycji i brokerów restartowanie kontrolera staje się coraz wolniejsze.
- Wewnętrzna architektura związana z własnością metadanych nie jest najlepsza — niektóre operacje są wykonywane za pośrednictwem kontrolera, drugie za pośrednictwem dowolnego brokera, a jeszcze inne bezpośrednio w ZooKeeperze.
- ZooKeeper jest własnym systemem rozproszonym i jego obsługa wymaga pewnej fachowej wiedzy, podobnie jak w przypadku Kafki. Programiści, którzy chcą korzystać z Kafki, muszą więc nauczyć się dwóch systemów rozproszonych zamiast jednego.

Wszystko to skłoniło społeczność projektu Apache Kafka do wymiany używanego kontrolera opartego na ZooKeeperze.

W istniejącej architekturze ZooKeeper pełni dwie ważne funkcje: wybieranie kontrolera i przechowywanie metadanych klastra — zarejestrowanych brokerów, konfiguracji, tematów, partycji i replik. Dodatkowo metadanymi zarządza sam kontroler, jest bowiem wykorzystywany do wybierania liderów, tworzenia i usuwania tematów oraz ponownego przypisywania replik. W nowym kontrolerze wszystkie te funkcjonalności będą musiały zostać zmienione.

Podstawowa idea nowego projektu kontrolera polega na tym, że Kafka ma architekturę opartą na dziennikach, gdzie użytkownicy reprezentują stan jako strumień zdarzeń. Korzyści z takiej reprezentacji są dobrze znane w społeczności — wiele konsumentów może szybko nadrobić zaległości, odtwarzając zdarzenia. Dziennik ustanawia jasne uporządkowanie zdarzeń i zapewnia, że konsumenci zawsze poruszają się wzdłuż jednej osi czasu. Nowa architektura kontrolera przynosi te same korzyści w zarządzaniu metadanymi Kafki.

W nowej architekturze węzły kontrolera to kworum Rafta, które zarządza dziennikiem zdarzeń dotyczących metadanych. Ten dziennik zawiera informacje o każdej zmianie metadanych klastra. Wszystko, co jest obecnie przechowywane w ZooKeeperze, jak tematy, partycje, zsynchronizowane repliki (ang. *In-Sync Replicas* — ISR) czy konfiguracje, będzie przechowywane w tym dzienniku.

Korzystając z algorytmu Raft, węzły kontrolera będą wybierać lidera spośród siebie, bez polegania na zewnętrznym systemie. Lider dziennika metadanych nazywany jest **aktywnym kontrolerem**. Aktywny kontroler obsługuje wszystkie zdalne wywołania procedur (ang. *Remote Procedure Call* — RPC) wykonywane z brokerów. Kontrolery podrzędne replikują dane zapisywane w aktywnym kontrolerze i pozostają w stanie gotowości na wypadek, gdyby aktywny kontroler uległ awarii. Ponieważ wszystkie kontrolery będą teraz śledzić najnowszy stan, przełączanie awaryjne kontrolera nie będzie wymagało długiego okresu przeładowania, w którym cały stan będzie przenoszony do nowego kontrolera.

Zamiast wysyłania aktualizacji przez kontroler do pozostałych brokerów, będziemy teraz mieli brokery otrzymujące aktualizacje z aktywnego kontrolera za pośrednictwem nowego interfejsu API `MetadataFetch`. Podobnie jak w przypadku żądania pobierania brokery będą śledzić przesunięcie ostatniej pobranej zmiany metadanych i żądać od kontrolera jedynie nowszych aktualizacji. Brokery

będą utrzymywać metadane na dysku, co pozwoli im na szybkie uruchamianie się nawet z milionami partycji (<https://oreil.ly/TsU0w>).

Brokery będą rejestrować się w kworum kontrolera i pozostawać zarejestrowane do momentu wyrejestrowania przez administratora, dlatego po zamknięciu broker przechodzi w tryb offline, ale pozostaje zarejestrowany. Brokery, które są online, ale nie są na bieżąco z najnowszymi metadanymi, zostaną odgradzone i nie będą mogły obsługiwać żądań klientów. Ten nowy stan odgradzenia zapobiegnie przypadkom, w których klient produkuje zdarzenia do brokera, który przestał już być liderem, ale jest zbyt nieaktualny, by mieć informacje na ten temat.

W ramach migracji do kworum kontrolera wszystkie operacje, w które wcześniej były zaangażowane klienty lub brokery komunikujące się bezpośrednio z ZooKeeperem, będą teraz routowane za pośrednictwem kontrolera. Umożliwi to bezproblemową migrację przez wymianę kontrolera bez konieczności jakiegokolwiek zmiany w którymś brokerze.

Ogólny projekt nowej architektury został opisany w propozycji KIP-500 (<https://oreil.ly/KAsp9>), natomiast kwestie związane ze sposobem adaptacji protokołu Raft dla Kafki opisano w dokumencie KIP-595 (<https://oreil.ly/XbI8L>). Szczegółowy projekt nowego kworum kontrolera, w tym konfigurację kontrolera i nowe CLI do interakcji z metadanymi klastra, znajdziesz w KIP-631 (<https://oreil.ly/rpOjK>).

Replikacja

Replikacja leży u podstaw architektury Kafki. Platforma ta jest często opisywana jako „rozproszona, podzielona na partycje, zreplikowana usługa dzienników zatwierdzania”. Replikacja ma kluczowe znaczenie, ponieważ właśnie w ten sposób Kafka gwarantuje dostępność i trwałość, gdy poszczególne węzły nieuchronnie ulegają awariom.

Jak już pisaliśmy, dane w Kafce są uporządkowane według tematów. Każdy temat jest podzielony na partycje, a każda partycja może mieć wiele replik. Repliki te przechowywane są w brokerach przechowujących zazwyczaj setki, a nawet tysiące replik należących do różnych tematów i partycji.

Istnieją dwa rodzaje replik:

Replika główna (lider)

W każdej partycji jedna replika jest wyznaczona do roli lidera (repliki głównej). Przez lidera przechodzą wszystkie żądania produkcji, aby zagwarantować spójność. Klienty mogą konsumować zarówno z repliki głównej, jak i z replik podrzędnych.

Replika podrzędna (obserwator)

Niebędące liderami pozostałe repliki partycji są nazywane obserwatorami (replikami podrzędnymi). O ile nie skonfigurowano inaczej, obserwatory nie obsługują żądań klientów; ich głównym zadaniem jest replikowanie komunikatów z lidera i pozostawanie na bieżąco z najnowszymi komunikatami z lidera. W przypadku awarii lidera partycji na nową replikę główną awansowana jest jedna z replik podrzędnych.

Odczytywanie z repliki podrzędnej

Możliwość odczytywania z replik podrzędnych dodano w propozycji KIP-392 (<https://oreil.ly/2xfxa>). Głównym celem tej funkcjonalności jest zmniejszenie kosztów ruchu sieciowego przez umożliwienie klientom konsumowania z najbliższych zsynchronizowanych replik, a nie z lidera. Aby możliwe było skorzystanie z tej opcji, konfiguracja konsumenta powinna zawierać właściwość `client.rack` identyfikującą lokalizację klienta. Konfiguracja brokera powinna zawierać właściwość `replica.selector.class`. Domyślnym ustawieniem tej konfiguracji jest `LeaderSelector` (czyli zawsze konsumuj z lidera), ale można je zmienić na `RackAwareReplicaSelector`, co spowoduje wybieranie repliki, która znajduje się na brokerze z konfiguracją `rack.id` zgodną z `client.rack` klienta. Można wdrożyć również własną logikę wyboru repliki przez zaimplementowanie interfejsu `ReplicaSelector`.

Protokół replikacji został rozszerzony, by zagwarantować, że podczas konsumowania z repliki podrzędnej dostępne będą tylko zatwierdzone komunikaty. Oznacza to, że nawet przy pobieraniu komunikatów z repliki podrzędnej otrzymujemy te same gwarancje niezawodności co zawsze. Aby zapewnić te gwarancje, wszystkie repliki muszą mieć informacje, które komunikaty zostały zatwierdzone przez lidera. W tym celu w danych wysyłanych do repliki podrzędnej lider umieszcza tzw. znak aktualnej wysokiej wody (ang. *high-water mark*), czyli ostatnie zatwierdzone przesunięcie. Propagacja znaku wysokiej wody wprowadza pewne opóźnienie, co oznacza, że dane są dostępne do konsumowania nieco wcześniej z lidera niż z obserwatora. Należy pamiętać o tym dodatkowym opóźnieniu, ponieważ kuszące może być podejmowanie próby zmniejszenia opóźnienia konsumenta przez konsumowanie z repliki głównej.

Kolejnym zadaniem repliki głównie jest śledzenie, która z replik podrzędnych jest z nią na bieżąco. Repliki podrzędne starają się być na bieżąco, replikując wszystkie komunikaty lidera wraz z ich napływaniem, ale utrzymanie synchronizacji może się nie powieść z różnych powodów, np. gdy przeciążenie sieci spowalnia replikację albo dany broker ulega awarii i wszystkie jego repliki zaczynają pozostawać w tyle, aż do zrestartowania brokera, kiedy mogą ponownie rozpocząć replikację.

Aby pozostać w synchronizacji z liderem, repliki wysyłają do niego żądania `Fetch` — dokładnie tego samego typu żądania, które konsumenci wysyłają w celu przetwarzania komunikatów. W odpowiedzi na te żądania lider wysyła replikom komunikaty. Żądania `Fetch` zawierają przesunięcie komunikatu, który replika chce otrzymać jako następny, i zawsze mają właściwą kolejność. Oznacza to, że lider może wiedzieć, iż replika otrzymała wszystkie komunikaty aż do ostatnich pobranych i żądanych, które przysły później. Na podstawie ostatnich przesunięć żądanych przez poszczególne repliki lider może określić, jak daleko w tyle pozostaje każda z nich. Jeśli od ponad 10 sekund replika nie zażądała żadnych komunikatów albo zażądała komunikatów, ale w ciągu ostatnich 10 sekund nie nadrobiła zaległości do najnowszego komunikatu, taka replika jest uznawana za **niezsynchronizowaną**. Jeżeli replika nie nadała za liderem, w wypadku awarii nie może już stać się nowym liderem — w końcu nie zawiera wszystkich komunikatów.

Analogicznie repliki, które konsekwentnie żądają najnowszych komunikatów, są nazywane **replikami zsynchronizowanymi** (ang. *In-Sync Replicas* — ISR). Gdy aktualny lider ulegnie awarii, do wyboru na lidera partycji kwalifikują się tylko repliki zsynchronizowane.

Ilość czasu, przez jaki replika podrzędna może być nieaktywna lub pozostawać w tyle, zanim zostanie uznana za niezsynchronizowaną, jest kontrolowana przez parametr konfiguracyjny `replica.lag.time.max.ms`. To dozwolone opóźnienie ma wpływ na zachowanie klienta i retencję danych

podczas wyboru lidera. Opiszemy to szczegółowo w rozdziale 7., gdy będziemy omawiać gwarancje niezawodności.

Oprócz aktualnego lidera każda partycja zawiera **preferowanego lidera** — replikę, która była liderem, gdy dany temat został utworzony. Jest to lider preferowany, ponieważ gdy partycje są tworzone, liderzy są rozkładani równomiernie między brokery. W rezultacie spodziewamy się, że gdy preferowany lider jest rzeczywiście liderem dla wszystkich partycji w klastrze, obciążenie jest równoważone między brokerami. Domyślnie Kafka ma ustawioną konfigurację `auto.leader.rebalance.enable=true`, która będzie powodować sprawdzanie, czy preferowany lider niebędący aktualnym liderem jest zsynchronizowany. Jeśli tak, uruchomi wybory lidera, aby preferowany lider stał się liderem aktualnym.



Określanie liderów preferowanych

Najlepszym sposobem na identyfikację aktualnego preferowanego lidera jest przyjrzenie się liście replik dla określonej partycji. (Szczegóły dotyczące partycji i replik możesz znaleźć w danych wyjściowych narzędzia *kafka-topics.sh*. To i inne narzędzia administracyjne omówimy w rozdziale 13.). Pierwsza replika na liście jest zawsze preferowanym liderem. Ma to zastosowanie bez względu na to, która replika jest aktualnym liderem, nawet jeśli repliki zostały przypisane ponownie do innych brokerów za pomocą odpowiedniego narzędzia. Jeżeli dokonujesz ręcznego ponownego przypisywania replik, tak naprawdę powinienś pamiętać, że replika, którą określisz jako pierwszą, będzie repliką preferowaną. Dlatego upewnij się, że rozłożysz je na różne brokery, aby uniknąć przeciążenia niektórych brokerów liderami, podczas gdy inne brokery nie będą odpowiednio partycypować w wykonywanej pracy.

Przetwarzanie żądań

Głównym zadaniem brokera Kafki jest przetwarzanie żądań wysyłanych przez klienty do liderów partycji, replik partycji i kontrolera. Kafka ma protokół binarny (działający przez TCP), który określa format żądań i sposób, w jaki brokery na nie odpowiadają — zarówno wtedy, gdy żądanie jest przetwarzane pomyślnie, jak i w przypadku napotkania przez broker błędów podczas przetwarzania żądania.

Projekt Apache Kafka obejmuje klienty Javy, które zostały zaimplementowane i są utrzymywane przez kontrybutorów projektu; dostępne są również klienty w innych językach, takich jak C, Python, Go itd. Pełną listę można zobaczyć na stronie Kafki (<http://bit.ly/2sKvTjx>). Wszystkie te klienty komunikują się z brokerem Kafki przy użyciu wspomnianego wcześniej protokołu.

Klienty zawsze inicjują połączenia i wysyłają żądania, a broker przetwarza je i odsyła odpowiedzi. Wszystkie żądania wysłane do brokera z określonego klienta zostaną przetworzone w kolejności, w jakiej zostały odebrane — ta gwarancja pozwala Kafce zachowywać się jak kolejka komunikatów i zapewnia właściwą kolejność zapisywanych komunikatów.

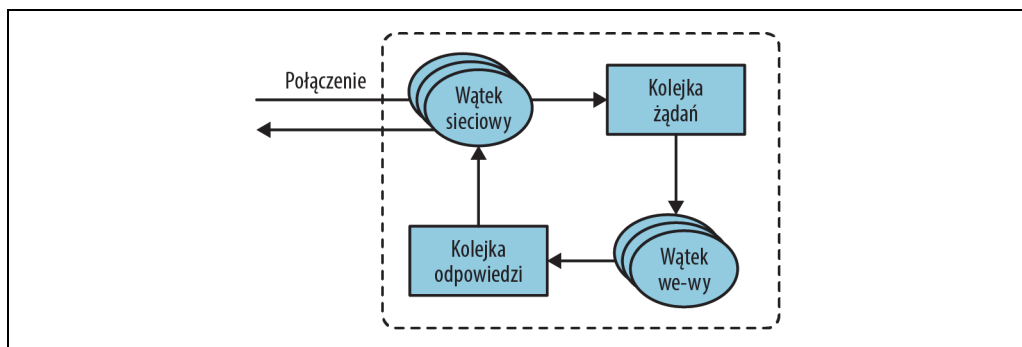
Wszystkie żądania mają standardowy nagłówek, który zawiera następujące elementy:

- typ żądania (zwany również **kluczem API**);
- wersję żądania (aby brokery mogły obsługiwać klienty w różnych wersjach i odpowiednio im odpowiadać);

- identyfikator korelacji — numer, który jednoznacznie identyfikuje żądanie, a ponadto pojawia się w odpowiedzi i w dziennikach błędów (identyfikator ten jest używany do rozwiązywania problemów);
- identyfikator klienta — używany do identyfikacji aplikacji, która wysłała żądanie.

Nie będziemy tutaj omawiać protokołu, ponieważ został on szczegółowo opisany w dokumentacji Kafki (<http://kafka.apache.org/protocol.html>). Pomocne będzie jednak przyjrzenie się sposobowi przetwarzania żądań przez broker — gdy dalej w książce będziemy omawiać monitorowanie Kafki i różne opcje konfiguracji, będziesz miał kontekst, do których kolejek i wątków odnoszą się dane wskaźniki i parametry konfiguracyjne.

Dla każdego nasłuchiwanego portu broker uruchamia wątek **akceptora**, który tworzy połączenie i przekazuje je do wątku **procesora** w celu obsłużenia. Liczbę wątków procesora (zwanych również **wątkami sieciowymi**) można konfigurować. Wątki sieciowe są odpowiedzialne za pobieranie żądań z połączeń klienckich, umieszczanie ich w **kolejce żądań** oraz odbieranie odpowiedzi z **kolejki odpowiedzi** i odsyłanie ich do klientów. Czasami odpowiedzi do klientów muszą być opóźniane — konsumenci otrzymują odpowiedzi tylko wtedy, gdy dane są dostępne, a klienci administracyjne otrzymują odpowiedź na żądanie `DeleteTopic`, gdy usuwanie tematu jest w toku. Opóźniane odpowiedzi są przechowywane w **czyśćcu** (ang. *purgatory*; <https://oreil.ly/2jWos>) do czasu ukończenia ich przetwarzania. Proces ten pokazaliśmy na rysunku 6.1.



Rysunek 6.1. Przetwarzanie żądań w Kafce

Gdy żądania zostają umieszczone w kolejce żądań, za ich pobieranie i przetwarzanie odpowiedzialne są **wątki we-wy** (zwane również **wątkami procedury obsługi żądań**). Do najczęstszych typów żądań wysyłanych przez klienty należą:

Żądania produkcji

Wysyłane przez producenty i zawierające komunikaty, które klienty zapisują w brokerach Kafki.

Żądania pobierania

Wysyłane przez konsumenty i repliki podrzędne, gdy odczytują komunikaty z brokerów Kafki.

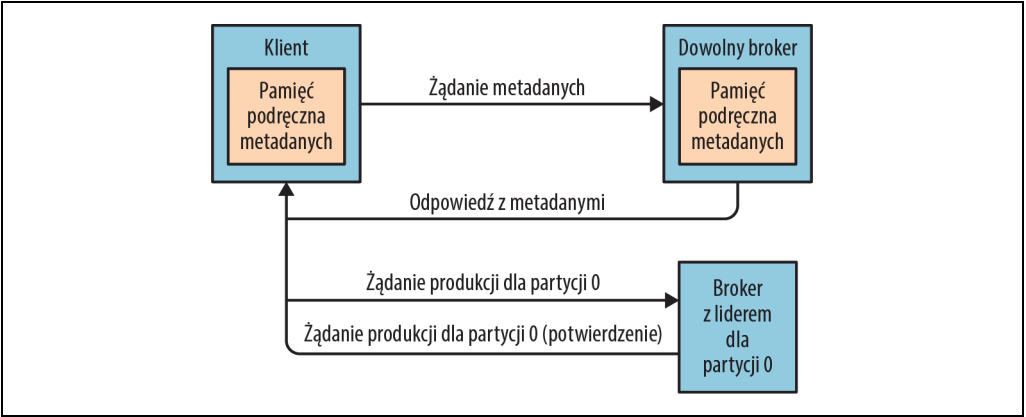
Żądania administratora

Wysyłane przez klienty administratora podczas wykonywania operacji na metadanych, takich jak tworzenie i usuwanie tematu.

Zarówno żądania produkcji, jak i pobierania muszą być wysyłane do głównej repliki partycji. Jeżeli broker otrzyma żądanie produkcji dla określonej partycji, a lider tej partycji będzie się znajdował na innym brokerze, wówczas klient, który wysłał to żądanie produkcji, otrzyma odpowiedź z informacją o błędzie: *Not a Leader for Partition* (nie jest to lider partycji). Ten sam błąd wystąpi, jeśli żądanie pobierania dla określonej partycji dotrze do brokera, który nie ma lidera dla tej partycji. Klienci Kafki są odpowiedzialni za wysyłanie żądań produkcji i pobierania do brokera, który ma lidera dla partycji właściwej dla danego żądania.

Skąd klienci wiedzą, gdzie wysyłać żądania? Klienci Kafki używają kolejnego typu żądania, zwanego **żądaniem metadanych**, które uwzględnia listę tematów interesujących klienta. Odpowiedź serwera określa, które partycje istnieją w danych tematach, jakie są repliki dla poszczególnych partii i która replika jest liderem. Żądania metadanych mogą być wysyłane do dowolnego brokera, ponieważ wszystkie brokery mają pamięć podręczną metadanych zawierającą te informacje.

Klienci zazwyczaj buforują te informacje i wykorzystują je do kierowania żądań produkcji i pobierania do właściwego brokera dla każdej partycji. Muszą również od czasu do czasu odświeżać te informacje (interwały odświeżania są kontrolowane przez parametr konfiguracyjny `metadata.max.age.ms`), wysyłając kolejne żądanie metadanych, by wiedzieć, czy metadane danego tematu nie uległy zmianie, np. czy nie został dodany nowy broker lub jakieś repliki nie zostały przeniesione do nowego brokera (rysunek 6.2). Ponadto jeśli klient dla jednego ze swoich żądań otrzyma błąd *Not a Leader* (nie jest to lider), to przed ponowną próbą wysłania tego żądania odświeży swoje metadane, ponieważ taki błąd wskazuje, że klient korzysta z nieaktualnych informacji i wysłał żądania do niewłaściwego brokera.



Rysunek 6.2. Routing żądań klienta

Żądania produkcji

Jak dowiedziałeś się w rozdziale 3., parametr konfiguracyjny o nazwie `acks` określa liczbę brokerów, które muszą potwierdzić otrzymanie komunikatu, zanim zostanie on uznany za pomyślnie zapisany. Producenty można skonfigurować w taki sposób, aby uznawały komunikaty za „pomyślnie zapisane”, jeśli komunikat zostanie zaakceptowany tylko przez jednego lidera (`acks=1`) lub przez

wszystkie repliki zsynchronizowane (`acks=all`), albo gdy wiadomość zostanie wysłana bez czekania, aż broker w ogóle ją zaakceptuje (`acks=0`).

Gdy broker zawierający replikę główną partycji otrzymuje żądanie produkcji dla tej partycji, rozpoczyna od uruchomienia kilku walidacji:

- Czy użytkownik wysyłający dane ma uprawnienia do zapisu w określonym temacie?
- Czy wartość `acks` określona w żądaniu jest prawidłowa (dozwolone są tylko wartości 0, 1 i `all`)?
- Jeśli parametr `acks` ma ustawioną wartość `all`, czy jest wystarczająco dużo replik zsynchronizowanych, by zapewnić bezpieczne zapisanie danego komunikatu? (Brokery mogą być skonfigurowane do odrzucania nowych komunikatów, jeżeli liczba replik zsynchronizowanych spadnie poniżej dozwolonej wartości; opiszemy to szerzej w rozdziale 7., gdy będziemy omawiać gwarancje trwałości i niezawodności Kafki).

Następnie broker zapisuje nowe komunikaty na dysku lokalnym. W Linuksie komunikaty są zapisywane w pamięci podręcznej systemu plików i nie ma gwarancji dotyczącej tego, kiedy zostaną zapisane na dysku. Kafka nie czeka, aż dane zostaną utrwalone na dysku — w kwestii zapewniania trwałości komunikatów opiera się na replikacji.

Gdy komunikat zostanie zapisany w liderze partycji, broker bada konfigurację parametru `acks`: jeżeli `acks` ma ustawioną wartość 0 lub 1, broker odpowiada natychmiast. Natomiast jeśli `acks` ma ustawioną wartość `all`, żądanie będzie przechowywane w buforze zwanym **czyśćcem**, dopóki lider nie stwierdzi, że repliki podrzędne zreplikowały komunikat, a wtedy odpowiedź zostanie wysłana do klienta.

Żądania pobierania

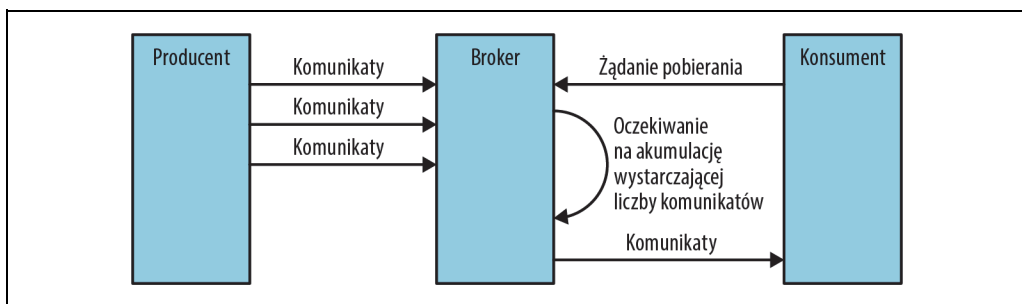
Brokery przetwarzają żądania pobierania bardzo podobnie do tego, w jaki sposób obsługiwane są żądania produkcji. Klient wysyła żądanie, prosząc broker o wysłanie komunikatów z listy tematów, partycji i przesunięć, co można ująć np. tak: „Proszę wysłać mi komunikaty zaczynające się od przesunięcia 53 w partycji 0 tematu *Test* oraz komunikaty rozpoczynające się od przesunięcia 64 w partycji 3 tematu *Test*”. Klienci określają również limit ilości danych, które broker może zwrócić dla każdej partycji. Ten limit jest ważny, ponieważ klienci muszą alokować pamięć, która będzie przechowywała odpowiedź odesłaną przez broker. Bez tego limitu brokery mogłyby odsyłać na tyle duże odpowiedzi, że klientom mogłoby zabraknąć pamięci.

Jak pisaliśmy wcześniej, żądanie musi dotrzeć do określonych w nim liderów partycji, a klient będzie wysyłać niezbędne żądania metadanych, aby się upewnić, że poprawnie routuje żądania pobierania. Gdy lider otrzyma żądanie, najpierw sprawdza, czy jest ono poprawne — czy dla tej konkretnej partycji to przesunięcie w ogóle istnieje? Jeżeli klient prosi o komunikat, który jest tak stary, że został usunięty z partycji, lub określone przesunięcie jeszcze nie istnieje, broker odpowie błędem.

Gdy dane przesunięcie istnieje, broker odczyta z tej partycji komunikaty aż do limitu ustawionego przez klient w żądaniu i wyśle je klientowi. Przy wysyłaniu komunikatów do klientów Kafka robi doskonały użytek z metody **zerowego przepływu** (ang. *zero-copy*), co oznacza, że wysyła komunikaty z pliku (lub, co bardziej prawdopodobne, z pamięci podręcznej systemu plików Linuksa)

bezpośrednio do kanału sieciowego bez żadnych pośrednich buforów. Różni się to od działania większości baz danych, w których dane przed wysłaniem do klientów są zapisywane w lokalnej pamięci podręcznej. Technika zerowego przepływu usuwa narzut związany z kopiowaniem bajtów i zarządzaniem buforami w pamięci, co zapewnia znacznie lepszą wydajność.

Oprócz ustawiania górnej granicy ilości danych, które broker może zwrócić, klienci mogą ustawić również dolną granicę ilości zwracanych danych. Ustawienie dolnej granicy np. na 10 kB instruuje broker, że ma zwracać wyniki tylko wtedy, gdy ma do wysłania co najmniej 10 000 bajtów. Jest to świetny sposób na zmniejszenie wykorzystania procesora i sieci, gdy klienci odczytują tematy, które nie generują dużego ruchu. Zamiast wysyłać żądania do brokerów co kilka milisekund, prosząc o dane i otrzymując w zamian bardzo niewiele komunikatów lub nie otrzymując żadnych, klienci wysyłają żądanie, a broker przed odesłaniem odpowiedzi czeka, aż zbierze się przyzwoita ilość danych, a dopiero potem klient prosi o więcej (rysunek 6.3). W ujęciu całkowitym odczytywana jest ta sama ilość danych, ale przy mniejszej liczbie wymienianych żądań i odpowiedzi, a tym samym z mniejszym narzutem.



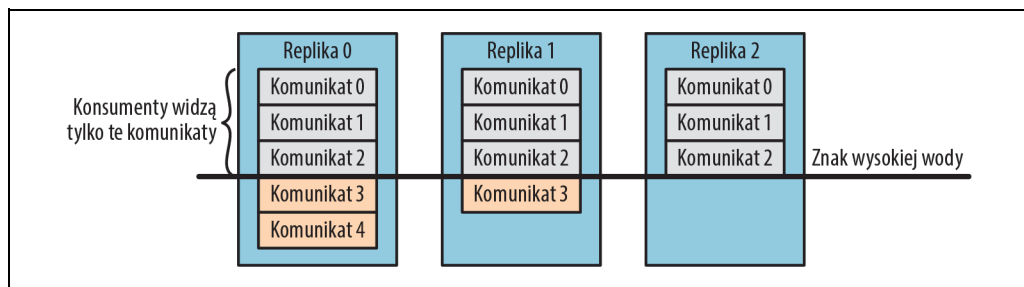
Rysunek 6.3. Broker opóźnia odpowiedź do momentu zgromadzenia wystarczającej ilości danych

Oczywiście nie chcemy, by klienci czekały w nieskończoność, aż broker będzie miał wystarczająco dużą ilość danych. Po pewnym czasie sensowne staje się pobranie zgromadzonych do tej pory danych i ich przetworzenie, zamiast czekać na więcej. Dlatego klienci mogą definiować także limit czasu w celu poinstruowania brokera, że jeśli w ciągu x milisekund nie zbierze minimalnej ilości danych do wysłania, niech wyśle po prostu to, co ma.

Warto zauważyć, że nie wszystkie dane, które znajdują się w liderze partycji, są dostępne do odczytu przez klienty. Większość klientów może odczytywać tylko komunikaty, które zostały zapisane we wszystkich replikach zsynchronizowanych (choćby repliki podrzędne są konsumentami, nie podlegają tej regule, gdyż inaczej replikacja w ogóle by nie działała). Pisaliśmy już, że lider partycji wie, które komunikaty zostały zreplikowane do poszczególnych replik, a dopóki komunikat nie zostanie zapisany we wszystkich replikach zsynchronizowanych, nie zostanie wysłany do konsumenta — próby pobrania takiego komunikatu zakończą się raczej otrzymaniem pustej odpowiedzi niż błędem.

Powodem tego zachowania jest to, że komunikaty niezreplikowane do wystarczającej liczby replik nie są jeszcze uważane za „bezpieczne” — jeśli lider ulegnie awarii, a jego miejsce zajmie inna replika, takie komunikaty zostaną w Kafce utracone. Gdybyśmy pozwolili klientom odczytywać komunikaty istniejące tylko w liderze, moglibyśmy zaobserwować niespójne zachowanie. Jeżeli konsument

odczytuje np. komunikat i w tym momencie lider ulega awarii, a żaden inny broker nie zawiera tego komunikatu, komunikat znika. Żaden inny konsument nie będzie w stanie odczytać tego komunikatu, co może spowodować niespójność w konsumencie, który go odczytał. Dlatego czekamy, aż wszystkie repliki synchroniczne otrzymają komunikat, i dopiero wtedy pozwalamy konsumentom go odczytywać (rysunek 6.4). Takie zachowanie oznacza również, że jeśli z jakiegoś powodu replikacja między brokerami jest spowolniona, nowe komunikaty będą docierać do konsumentów nieco później (ponieważ w pierwszej kolejności czekamy na replikację komunikatów). Opóźnienie to jest ograniczone do wartości parametru `replica.lag.time.max.ms`. Określa on czas, przez jaki replika może pozostawać opóźniona w replikacji nowych komunikatów i nadal być traktowana jako zsynchronizowana.



Rysunek 6.4. Konsumenty widzą tylko komunikaty, które zostały powielone do replik zsynchronizowanych

W niektórych sytuacjach konsument przetwarza zdarzenia z dużej liczby partycji. Wysyłanie do brokera przy każdym żądaniu listy wszystkich partycji, którymi konsument jest zainteresowany, i odsyłanie przez broker wszystkich swoich metadanych może być bardzo nieefektywne — zestaw partycji rzadko się zmienia, podobnie jak ich metadane, a w większości przypadków nie ma zbyt wielu danych do zwracania. W celu zminimalizowania tego narzutu Kafka oferuje pamięć podręczną pobierania sesji (ang. *fetch session cache*). Konsumenty mogą próbować tworzyć sesję buforowaną, przechowującą listę przetwarzanych przez nie partycji oraz ich metadane. Po utworzeniu sesji konsumenty nie muszą już podawać wszystkich partycji w każdym żądaniu i mogą zamiast tego używać przyrostowych żądań pobierania. Brokery w odpowiedzi będą umieszczać metadane tylko w przypadku jakichkolwiek zmian. Pamięć podręczna sesji jest ograniczona, a Kafka nadaje priorytet replikom podrzędnym i konsumentom z dużym zestawem partycji, dlatego w niektórych sytuacjach taka sesja nie zostanie utworzona albo zostanie przerwana. W obu tych przypadkach broker zwróci klientowi odpowiedni błąd, a konsument w przejrzysty sposób ucieknie się do pełnych żądań pobierania, które zawierają wszystkie metadane partycji.

Inne żądania

Omówiliśmy najpopularniejsze typy żądań wykorzystywane przez klienty Kafki: Metadane (żądanie metadanych), Produce (żądanie produkcji) i Fetch (żądanie pobierania). Protokół Kafka obsługuje obecnie 61 różnych typów żądań (<https://oreil.ly/hBmNc>), a z pewnością dodanych zostanie jeszcze więcej. Same konsumenty używają 15 rodzajów żądań, aby tworzyć grupy, koordynować

konsumowanie i umożliwiać programistom zarządzanie grupami konsumentów. Duża liczba żądań jest także związana z zarządzaniem metadanymi i bezpieczeństwem.

Ponadto ten sam protokół jest używany do komunikacji między samymi brokerami Kafki. Te żądania mają charakter wewnętrzny i nie powinny być wykorzystywane przez klienty. Gdy kontroler ogłasza, że partycja ma nowego lidera, wysyła np. żądanie `LeaderAndIsr` do nowego lidera (aby wiedział, że musi zacząć przyjmować żądanie klientów) i do partycji podrzędnych (albowiem muszą wiedzieć, że mają obserwować nowego lidera).

Protokół stale ewoluje — w miarę jak społeczność Kafki dodaje kolejne funkcjonalności klienta, protokół musi się dopasowywać. Niegdyś konsumenci Kafki do śledzenia otrzymywanych przesunięć wykorzystywały np. serwer Apache ZooKeeper. Gdy konsument był uruchamiany, mógł sprawdzić ZooKeeper pod kątem ostatniego przesunięcia, które zostało odczytane z jego partycji, i wiedział, gdzie rozpocząć przetwarzanie. Z różnych powodów społeczność postanowiła zaprzestać stosowania do tego celu ZooKeepera i zamiast tego zapisywać te przesunięcia w specjalnym temacie Kafki. Kontrybutorzy musieli więc dodać do protokołu kilka żądań: `OffsetCommitRequest`, `OffsetFetchRequest` i `ListOffsetsRequest`. Gdy teraz aplikacja wywołuje API klienta w celu zatwierdzenia przesunięć konsumenta, klient nie zapisuje już w ZooKeeperze; zamiast tego wysyła do Kafki żądanie `OffsetCommitRequest`.

Tworzenie tematów było niegdyś obsługiwane przez narzędzia wiersza poleceń, które bezpośrednio aktualizowały listę tematów w ZooKeeperze. Od tego czasu w celu zarządzania metadanymi społeczność Kafki dodała m.in. żądanie `CreateTopicRequest`. Aplikacje Javy wykonują te operacje na metadanych za pośrednictwem klasy `AdminClient` Kafki, co omówiliśmy szczegółowo w rozdziale 5. Ponieważ operacje te są obecnie częścią protokołu Kafki, klienty napisane w językach, które nie mają biblioteki ZooKeepera, mogą teraz tworzyć tematy, zwracając się bezpośrednio do brokerów Kafki.

Poza rozwijaniem protokołu przez dodawanie nowych typów żądań, programiści Kafki czasami decydują się na modyfikację istniejących żądań w celu dodania pewnych funkcjonalności. Między wydaniem wersji 0.9.0 i 0.10.0 postanowili np. informować klienty o aktualnym kontrolerze, dodając informacje do odpowiedzi `Metadata`. W rezultacie do żądania i odpowiedzi `Metadata` dodano nową wersję. Teraz klienty 0.9.0 wysyłają żądania metadanych w wersji 0 (ponieważ wersja 1 nie istniała w klientach 0.9.0), a brokery, niezależnie od tego, czy w wersji 0.9.0 czy 0.10.0, wiedzą, że mają odesłać odpowiedź w wersji 0, która nie ma informacji o kontrolerze. Jest to w porządku, ponieważ klienty 0.9.0 nie oczekują informacji o kontrolerze, albowiem i tak nie wiedziałyby, jak ją sparsować. Klient 0.10.0 będzie wysyłał żądania `Metadata` w wersji 1, a brokery 0.10.0 będą odpowiadać w wersji 1, zawierającej informacje o kontrolerze, z których mogą skorzystać klienty 0.10.0. Jeżeli klient 0.10.0 wyśle brokerowi 0.9.0 wersję 1 żądania `Metadata`, broker nie będzie wiedział, jak obsłużyć nowszą wersję żądania, i odpowie błędem. Właśnie z tego powodu zalecamy uaktualnienie brokerów przed uaktualnieniem jakichkolwiek klientów — w przeciwnieństwie do klientów nowe brokery wiedzą, jak obsługiwać zarówno stare, jak i nowe żądania.

W wydaniu 0.10.0 społeczność Kafki dodała żądanie `ApiVersionRequest`, co pozwala klientom pytać broker, które wersje poszczególnych żądań są obsługiwane, i odpowiednio stosować poprawne wersje. Klienci prawidłowo korzystające z tej nowej funkcjonalności będą mogli komunikować się ze starszymi brokerami za pomocą wersji protokołu obsługiwanej przez broker, z którym się łączą.

Obecnie trwają prace nad dodaniem interfejsów API, które pozwolą klientom wykrywać funkcjonalności obsługiwane przez brokery, a brokerom stosować funkcjonalności odpowiednie dla określonych wersji. Poprawka ta została zaproponowana w dokumencie KIP-584 (<https://oreil.ly/dxg8N>) i obecnie wydaje się, że najprawdopodobniej będzie częścią wersji 3.0.0.

Fizyczna pamięć masowa

Podstawową jednostką pamięci masowej Kafki jest replika partycji. Partycji nie można podzielić pomiędzy wiele brokerów, a nawet między wiele dysków na tym samym brokerze. Wielkość partycji jest więc ograniczona przestrzenią dostępną w jednym punkcie montowania. (Jeśli używana jest konfiguracja JBOD, punktem montowania może być pojedynczy dysk, a jeżeli skonfigurowano RAID, może to być wiele dysków. Więcej informacji na ten temat znajdziesz w rozdziale 2.).

Podczas konfigurowania Kafki administrator definiuje listę katalogów, w których będą przechowywane partycje — jest to parametr `log.dirs` (nie należy go mylić z lokalizacją, w której Kafka przechowuje swój dziennik błędów, a którą konfiguruje się w pliku `log4j.properties`). Zwyczajowa konfiguracja zawiera katalog dla każdego punktu montowania, z którego będzie korzystała Kafka.

Przyjrzyjmy się, jak Kafka używa dostępnych katalogów do przechowywania danych. Przede wszystkim chcemy pokazać, w jaki sposób dane są alokowane brokerom w klastrze i katalogom w brokerze. Potem zobaczymy, jak broker zarządza plikami — w szczególności jak obsługiwane są gwarancje retencji. Następnie zajmiemy się plikami oraz formatami plików i indeksów. Na koniec omówimy kompaktowanie dzienników, czyli zaawansowaną funkcjonalność, która pozwala przekształcić Kafkę w długoterminowy magazyn danych, i opiszemy, jak to działa.

Warstwowy system pamięci masowej

Pod koniec 2018 r. społeczność Apache Kafki rozpoczęła współpracę nad ambitnym projektem mającym na celu dodanie do Kafki funkcjonalności warstwowej pamięci masowej. Prace nad projektem trwają i ma to zostać udostępnione w wydaniu 3.0.

Motywacja jest dość prosta: Kafka jest obecnie wykorzystywana do przechowywania dużej ilości danych ze względu na wysoką przepływność lub długi okres retencji. Trzeba więc uwzględnić następujące kwestie:

- Liczba danych, które można przechowywać w partycji, jest ograniczona. W rezultacie maksymalna retencja i liczba partycji nie zależą jedynie od wymagań produktu, ale także od ograniczenia fizycznych wymiarów dysków.
- Wybór rozmiaru dysku i klastra zależy od wymagań dotyczących pamięci masowej. Klastry często stają się większe, niż gdyby głównym czynnikiem były opóźnienia i przepustowość, co zwiększa koszty.
- Czas potrzebny na przeniesienie partycji z jednego brokera na drugi, np. podczas rozszerzania lub zmniejszania klastra, jest zależny od wielkości partycji. Duże partycje sprawiają, że klastr jest mniej elastyczny. Obecnie architektury są projektowane z myślą o maksymalnej elastyczności, wykorzystując opcje wdrożenia w chmurze, które można łatwo dostosowywać.

W przypadku warstwowej pamięci masowej klastrer Kafki konfiguruje się z dwiema warstwami pamięci masowej: lokalną i zdalną. Warstwa lokalna oznacza to samo, co obecna warstwa magazynu danych Kafki — do przechowywania segmentów dziennika wykorzystuje ona lokalne dyski na brokerach Kafki. Nowa warstwa do przechowywania ukończonych segmentów dziennika używa dedykowanego systemu pamięci masowej, takiego jak HDFS lub S3.

Użytkownicy Kafki mogą ustawić oddzielną regułę retencji pamięci masowej dla każdej warstwy. Ponieważ lokalna pamięć masowa jest zwykle znacznie droższa niż warstwa zdalna, okres retencji dla warstwy lokalnej jest zazwyczaj co najmniej o kilka godzin krótszy, a okres retencji dla warstwy zdalnej może być znacznie dłuższy — o kilka dni lub nawet miesięcy.

Lokalna pamięć masowa ma znacznie mniejsze opóźnienie niż pamięć zdalna. Działa to dobrze, ponieważ aplikacje wrażliwe na opóźnienia wykonują odczyty z ogona i są obsługiwane z warstwy lokalnej, korzystają więc z istniejącego mechanizmu Kafki, który polega na tym, że do serwowania danych wykorzystywana jest efektywnie pamięć podręczna stron. Z warstwy zdalnej obsługiwane jest uzupełnianie danych historycznych i zapewnianie aplikacjom odzyskującym sprawność po awarii danych starszych niż to, co znajduje się w warstwie lokalnej.

Dwupoziomowa architektura stosowana w warstwowej pamięci masowej umożliwia jej skalowanie niezależnie od pamięci i procesorów w klastrze Kafki. Dzięki temu Kafka może być rozwiązaniem do długoterminowego przechowywania danych. Zmniejsza to również ilość danych przechowywanych lokalnie w brokerach Kafki, a tym samym ilość danych, które muszą być kopiowane podczas odzyskiwania sprawności i równoważenia obciążenia. Segmenty dzienników, które są dostępne w warstwie zdalnej, nie muszą być przywracane na brokerze lub przywracane leniwie i są serwowane właśnie z tej warstwy. Ponieważ nie wszystkie dane są przechowywane na brokerach, wydłużenie okresu retencji nie wymaga już skalowania pamięci masowej klastra Kafki i dodawania nowych węzłów. Jednocześnie ogólna retencja danych może być znacznie dłuższa, co eliminuje konieczność stosowania osobnych potoków danych do kopiowania danych z Kafki na zewnętrzne magazyny danych, jak ma to aktualnie miejsce w przypadku wielu wdrożeń.

Projekt warstwowej pamięci masowej został opisany szczegółowo w propozycji KIP-405 (<https://oreil.ly/yZP6w>) i uwzględniono w niej nowy komponent `RemoteLogManager` oraz interakcje z istniejącymi funkcjonalnościami, takimi jak nadrabianie przez repliki zaległości w stosunku do lidera i wybory lidera.

Jednym z interesujących rezultatów omówionych w dokumencie KIP-405 jest wpływ warstwowej pamięci masowej na wydajność. Zespół implementujący warstwową pamięć masową zmierzył wydajność w kilku przypadkach użycia. Pierwszym było wykorzystanie standardowego obciążenia roboczego Kafki o dużej przepływności. W tym przypadku opóźnienie nieco wzrosło (z 21 ms przy p99 do 25 ms), ponieważ brokery musiały wysyłać segmenty również do zdalnej pamięci masowej. Drugi przypadek użycia polegał na tym, że kilka konsumentów musiało odczytywać stare dane. Bez warstwowej pamięci masowej konsumenci odczytujące stare dane miały duży wpływ na opóźnienie (21 ms w porównaniu z 60 ms przy p99), ale przy włączonej warstwowej pamięci masowej wpływ był znacznie mniejszy (25 ms w porównaniu z 42 ms przy p99). Jest to spowodowane tym, że warstwowa pamięć masowa odczytuje z HDFS lub S3 przez ścieżkę sieciową. Odczyty sieciowe

nie konkurują z lokalnymi odczytami z we-wy dysku lub z pamięci podręcznej stron i pozostawiają pamięć podręczną strony nienaruszoną ze świeżymi danymi.

Oznacza to, że poza nieskończoną pojemnością pamięci masowej, niższymi kosztami i elastycznością warstwowa pamięć masowa zapewnia również izolację między odczytami historycznymi a odczytami w czasie rzeczywistym.

Przydzielanie partycji

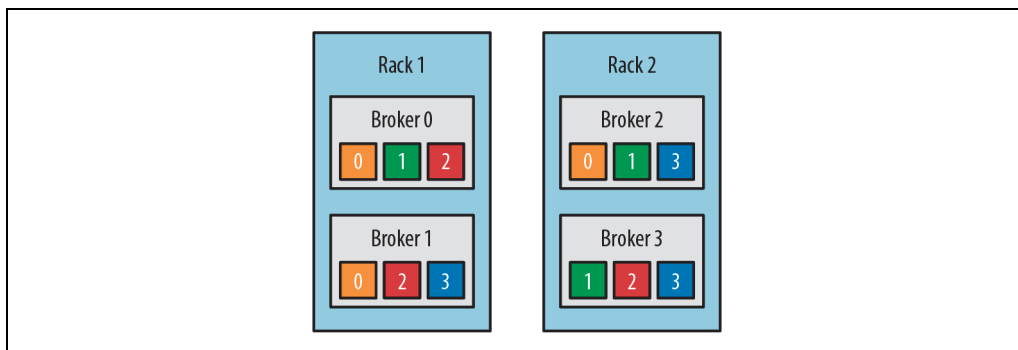
Gdy tworzysz temat, Kafka najpierw decyduje, jak przydzielić partycje brokerom. Załóżmy, że masz 6 brokerów i decydujesz się utworzyć temat z 10 partycjami i współczynnikiem replikacji 3. Kafka ma teraz 30 replik partycji do przydzielenia 6 brokerom. Podczas dokonywania alokacji branych jest pod uwagę kilka kwestii:

- Repliki powinny być równomiernie rozłożone między brokery — w naszym przykładzie będzie to przydzielenie 5 replik na broker.
- Poszczególne repliki każdej partycji powinny znajdować się na innym brokerze. Jeśli partycja 0 ma lidera na brokerze 2, repliki podrzędne mogą zostać umieszczone na brokerach 3 i 4, ale nie na brokerze 2, i nie mogą znajdować się obie na brokerze 3.
- Jeżeli brokery mają informacje o szafach rackowych (ta funkcjonalność jest dostępna w Kafce od wersji 0.10.0), w miarę możliwości repliki poszczególnych partycji powinny zostać przydzielone do różnych racków. Gwarantuje to, że zdarzenie powodujące przestój całej szafy rackowej nie sprawi, że partycje będą całkowicie niedostępne.

Aby to osiągnąć, zaczynamy od losowego brokera (np. brokera 4) i przypisujemy partycje do poszczególnych brokerów z wykorzystaniem algorytmu karuzelowego, by określić lokalizacje dla liderów. Tak więc lider partycji 0 będzie się znajdował na brokerze 4, lider partycji 1 na brokerze 5, lider partycji 2 na brokerze 0 (ponieważ mamy 6 brokerów) itd. Następnie dla każdej partycji rozmieszczamy repliki w zwiększających się przesunięciach w stosunku do lidera. Jeżeli lider dla partycji 0 znajduje się na brokerze 4, pierwsza replika podrzędna zostanie umieszczona na brokerze 5, a druga na brokerze 0. Lider partycji 1 znajduje się na brokerze 5, stąd pierwsza replika będzie na brokerze 0, a druga na brokerze 1.

Gdy uwzględniamy informacje o szafach rackowych, zamiast wybierać brokery w kolejności numerycznej, przygotowujemy listę brokerów zmieniających racki. Załóżmy, że wiemy, iż brokery 0 i 1 zostały przydzielone do tej samej szafy rackowej, a brokery 2 i 3 przydzielono do osobnej szafy. Zamiast wybierać brokery w kolejności od 0 do 3, organizujemy je w porządku 0, 2, 1, 3 — po każdym brokerze następuje broker z innej szafy rackowej (rysunek 6.5). W tym przypadku, jeśli lider dla partycji 0 jest na brokerze 2, pierwsza replika będzie na brokerze 1, czyli w innej szafie rackowej. To świetne rozwiązanie, bo jeśli pierwsza szafa rackowa zostanie wyłączona, nadal będziemy mieli ocalałą replikę, więc partycja pozostanie dostępna. Zastosujemy to do wszystkich naszych replik, dlatego będziemy mieli zagwarantowaną dostępność w przypadku awarii którejś szafy rackowej.

Po wybraniu odpowiednich brokerów dla poszczególnych partycji i replik należy zdecydować, którego katalogu użyć dla tych nowych partycji. Robimy to niezależnie dla każdej partycji, a zasada jest bardzo prosta: liczymy partycje w każdym katalogu, a nowe partycje dodajemy do katalogu, który



Rysunek 6.5. Partycje i repliki przypisane brokerom działającym na różnych szafach rackowych

ma ich najmniej. Oznacza to, że jeśli dodamy nowy dysk, wszystkie nowe partycje zostaną utworzone właśnie na nim. Jest to spowodowane tym, że dopóki sytuacja się nie zrównoważy, nowy dysk będzie zawsze miał najmniejszą liczbę partycji.



Uważaj na miejsce na dysku

Należy pamiętać, że przydzielanie partycji do brokerów nie uwzględnia dostępnej przestrzeni dyskowej ani istniejącego obciążenia, a alokacja partycji na dyskach uwzględnia ich liczbę, ale pomija ich rozmiar. Oznacza to, że jeśli niektóre brokery będą miały więcej miejsca na dysku niż inne (np. dlatego, że klastery jest mieszkanką starszych i nowszych serwerów), to pewne partycje będą nienormalnie duże albo dyski na samym brokerze będą miały różne rozmiary — wtedy podczas przydzielania partycji będziesz musiał zachować ostrożność.

Zarządzanie plikami

W Kafce retencja jest ważną koncepcją, gdyż Kafka nie przechowuje danych w nieskończoność, a przed usunięciem komunikatu nie czeka, aż odczytają go wszystkie konsumenty. Administrator Kafki konfiguruje okres retencji dla każdego tematu — czas przechowywania komunikatów przed ich usunięciem lub maksymalną ilość przechowywanych danych, po przekroczeniu której starsze komunikaty są usuwane.

Ponieważ znalezienie w dużym pliku komunikatów, które wymagają wyczyszczenia, a następnie usunięcie części tego pliku jest zarówno czasochłonne, jak i podatne na błędy, zamiast tego dzielimy każdą partycję na **segmenty**. Domyślnie każdy segment zawiera 1 GB lub tydzień danych, w zależności od tego, co będzie mniejsze. Gdy broker Kafki zapisuje komunikaty w partycji, jeśli limit segmentu zostanie osiągnięty, zamyka dany plik i rozpoczyna nowy.

Segment, w którym przeprowadzane jest bieżące zapisywanie, nazywa się **segmentem aktywnym**. Aktywny segment nigdy nie jest usuwany, więc jeśli ustawisz, by dane dziennika były przechowywane tylko przez jeden dzień, ale każdy segment będzie zawierał pięć dni danych, tak naprawdę będziesz przechowywać dane przez pięć dni, ponieważ nie można usuwać danych przed zamknięciem segmentu. Jeżeli zdecydujesz się przechowywać dane przez tydzień i codziennie wprowadzać

nowy segment, przekonasz się, że każdego dnia rozpoczynany będzie nowy segment, a najstarszy będzie usuwany — tak więc przez większość czasu partycja będzie miała siedem segmentów.

Jak nauczyłeś się z rozdziału 2., broker Kafki będzie utrzymywał uchwyt otwartego pliku dla każdego segmentu w każdej partycji — dotyczy to nawet nieaktywnych segmentów. W rezultacie zazwyczaj jest duża liczba otwartych uchwytów plików, a system operacyjny musi być odpowiednio dostrojony.

Format plików

Każdy segment jest przechowywany w pojedynczym pliku danych. W tym pliku przechowywane są komunikaty Kafki i ich przesunięcia. Format danych na dysku jest taki sam jak format komunikatów wysyłanych z producenta do brokera, a później z brokera do konsumentów. Użycie tego samego formatu komunikatów na dysku i przy przesyłaniu przez sieć pozwala Kafce na wykorzystanie optymalizacji zerowego przepływu podczas wysyłania komunikatów do konsumentów. Umożliwia także uniknięcie dekompresji i ponownej kompresji komunikatów, które już skompresowano. W rezultacie, jeśli zdecydujemy się na zmianę formatu komunikatów, modyfikacji będą wymagać zarówno protokół podłączania, jak i format przechowywania na dysku, a brokery Kafki będą musiały wiedzieć, jak obsługiwać przypadki, w których ze względu na aktualizacje pliki będą zawierały komunikaty w dwóch formatach.

Komunikaty Kafki składają się z ładunku użytkownika i nagłówków systemowych. Ładunek użytkownika zawiera fakultatywny klucz, wartość i opcjonalny zbiór nagłówków, gdzie każdy nagłówek jest parą klucz-wartość.

Od wersji 0.11 (i formatu komunikatów v2) producenci Kafki zawsze wysyłają komunikaty w partiach. Jeżeli wyślesz pojedynczy komunikat, tworzenie partii doda pewien narzut. Jednak w przypadku co najmniej dwóch komunikatów na partię oszczędza to miejsce, co zmniejsza zużycie sieci i dysku. Jest to jeden z powodów, dla których Kafka ma lepszą wydajność przy ustawieniu `linger.ms=10` — to niewielkie opóźnienie zwiększa szanse na to, że w partii zostanie wysłanych więcej komunikatów. Ponieważ Kafka tworzy oddzielną partię na każdą partycję, bardziej wydajne będą również producenci zapisujące w mniejszej liczbie partycji. Należy pamiętać, że pojedyncze żądanie produkcji producentów Kafki może obejmować wiele partycji. Oznacza to, że jeśli w producencie używasz kompresji (co jest zalecane!), wysyłanie większych partii zapewnia większą kompresję zarówno przy transferze sieciowym, jak i zapisywaniu na dyskach brokera.

Oto nagłówki partii komunikatów:

- Magiczna liczba wskazująca aktualną wersję formatu komunikatów (tutaj mówimy o v2).
- Przesunięcie pierwszego komunikatu w partii i różnica w stosunku do przesunięcia ostatniego komunikatu — te wartości są zachowywane, nawet jeśli partia jest później kompaktowana i niektóre komunikaty są usuwane. Gdy producent tworzy i wysyła partię, przesunięcie pierwszego komunikatu jest ustawiane na 0. Broker, który jako pierwszy utrwala tę partię (lider partycji), zastępuje tę wartość rzeczywistym przesunięciem.

- Znacznik czasu pierwszego komunikatu i największy znacznik czasu w partii. Znaczniki czasu mogą być ustawiane przez broker, jeżeli typ znacznika czasu został ustawiony na inkrementację czasu, a nie jego bezpośrednio odzwierciedlanie.
- Wielkość partii w bajtach.
- Epoka lidera, który otrzymał daną partię. Jest to używane podczas obcinania komunikatów po wyborze lidera; szczegółowe wyjaśnienie użycia znajdziesz w propozycjach KIP-101 (<https://oreil.ly/Ffa4D>) i KIP-279 (<https://oreil.ly/LO7nx>).
- Suma kontrolna w celu potwierdzenia, że partia nie jest uszkodzona.
- 16 bitów wskazujących na różne atrybuty: typ kompresji, typ znacznika czasu (znacznik czasu może być ustawiony u klienta lub u brokera) oraz czy partia jest częścią transakcji, czy też jest partią kontrolną.
- Identyfikator producenta, epoka producenta i pierwsza sekwencja w partii — wszystkie te elementy są wykorzystywane do gwarancji „dokładnie raz”.
- I oczywiście zbiór komunikatów, które są częścią partii.

Jak widać, nagłówek partii zawiera wiele informacji. Same rekordy mają również nagłówki systemowe (nie mylić z nagłówkami, które mogą być ustawiane przez użytkowników). Oto nagłówki zawarte w każdym rekordzie:

- Rozmiar rekordu w bajtach.
- Atrybuty — obecnie nie ma atrybutów na poziomie rekordów, więc nie jest to używane.
- Różnica między przesunięciem bieżącego rekordu a pierwszym przesunięciem w partii.
- Różnica w milisekundach między znacznikiem czasu tego rekordu a pierwszym znacznikiem czasu w partii.
- Ładunek użytkownika: klucz, wartość i nagłówki.

Zwróć uwagę, że poszczególne rekordy mają bardzo niewielki narzut, a większość informacji systemowych znajduje się na poziomie partii. Zapisywanie w nagłówku tylko pierwszego przesunięcia i znacznika czasu partii oraz zapisywanie różnicy w każdym rekordzie znacznie zmniejsza narzut każdego rekordu, zwiększając wydajność większych partii.

Poza partiami komunikatów, które zawierają dane użytkownika, Kafka posiada również partie kontrolne, wskazujące np. zatwierdzenie transakcyjne. Są one obsługiwane przez konsumenty i nie są przekazywane do aplikacji użytkownika, a aktualnie zawierają wersję i typ wskaźnika: 0 dla transakcji przerwanej i 1 dla zatwierdzenia.

Jeśli chcesz przekonać się na własne oczy, brokery Kafki są dostarczane z narzędziem `DumpLog` ↪ `Segment`, które pozwala przyrzeć się segmentowi partycji w systemie plików i zbadać jego zawartość. To narzędzie możesz uruchomić w następujący sposób:

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

Gdy zaznaczysz parametr `--deep-iteration`, wyświetlone zostaną tylko informacje o komunikatach skompresowanych wewnątrz komunikatów opakowujących.



Konwersja formatu komunikatów na niższą wersję

Wspomniany wcześniej format komunikatów został wprowadzony w wersji 0.11. Ponieważ Kafka obsługuje uaktualnienie brokerów przed uaktualnieniem wszystkich klientów, musi obsługiwać dowolną kombinację wersji między brokerem, producentem i konsumentem. Większość kombinacji działa bez problemów — nowe brokery będą rozumiały stary format komunikatów od producentów, a nowe producenci będą wiedziały, że do starych brokerów mają wysłać komunikaty w starym formacie. Z trudną sytuacją mamy jednak do czynienia, gdy nowy producent wysyła do nowych brokerów komunikaty v2: taki komunikat jest zapisywany w formacie v2, ale będzie próbował go odczytać stary konsument, który nie obsługuje tego formatu. W tym scenariuszu broker będzie musiał przekonwertować komunikat z formatu v2 na v1, aby konsument mógł go sparsować. Taka konwersja zużywa znacznie więcej zasobów procesora i pamięci niż normalna konsumpcja, dlatego najlepiej jej unikać. Propozycja KIP-188 (<https://oreil.ly/9RwQC>) wprowadziła kilka ważnych wskaźników kondycji, wśród nich `FetchMessageConversionsPerSec` i `MessageConversionsTimeMs`. Jeśli w Twojej organizacji nadal używane są stare klienty, zalecamy sprawdzenie tych wskaźników i jak najszybsze uaktualnienie klientów.

Indeksy

Kafka umożliwia konsumentom rozpoczęcie pobierania komunikatów od dowolnego dostępnego przesunięcia. Oznacza to, że jeśli konsument poprosi o 1 MB komunikatów od przesunięcia 100, broker musi być w stanie szybko zlokalizować komunikat dla tego przesunięcia (który może znajdować się w dowolnym segmencie partycji) i od niego rozpocząć odczytywanie komunikatów. Aby brokery mogły szybciej lokalizować komunikaty dla danego przesunięcia, Kafka utrzymuje indeks dla każdej partycji. Ten indeks mapuje przesunięcia na segmenty plików i pozycje w pliku.

Kafka ma ponadto drugi indeks, który mapuje znaczniki czasu na przesunięcia komunikatów. Indeks ten jest używany podczas wyszukiwania komunikatów według znacznika czasu. Kafka Streams korzysta z tego wyszukiwania w szerokim zakresie; przydaje się ono także w niektórych scenariuszach przełączania awaryjnego.

Indeksy również są podzielone na segmenty, żebyśmy mogli usuwać stare wpisy indeksu, gdy komunikaty są czyszczone. Kafka nie próbuje utrzymywać sum kontrolnych indeksu. Jeżeli indeks ulegnie uszkodzeniu, zostanie odtworzony z odpowiedniego segmentu dziennika przez ponowne odczytanie komunikatów oraz zarejestrowanie przesunięć i lokalizacji. Jeśli zachodzi taka potrzeba, administrator może również całkowicie bezpiecznie (aczkolwiek powodując ewentualnie długotrwałe odtwarzanie) usuwać segmenty indeksów — zostaną one odtworzone automatycznie.

Kompaktowanie

Zwykle Kafka przechowuje komunikaty przez ustalony czas i usuwa starsze, gdy skończy się ich okres retencji. Wyobraź sobie jednak sytuację, w której używasz Kafki do przechowywania adresów wysyłki dla swoich klientów. W takim przypadku bardziej sensowne byłoby przechowywanie ostatniego adresu dla każdego klienta zamiast danych tylko za ostatni tydzień lub rok. W ten sposób nie musisz się przejmować starszymi adresami, a nadal zachowujesz adresy dla klientów, którzy przez jakiś czas nie zmieniali lokalizacji. Innym przypadkiem użycia może być aplikacja wykorzystująca

Kafkę do przechowywania swojego aktualnego stanu. Przy każdej zmianie stanu aplikacja zapisuje nowy stan w Kafce. Podczas odzyskiwania sprawności po awarii aplikacja odczytuje te komunikaty z Kafki, aby przywrócić swój najnowszy stan. W tym przypadku dla aplikacji ważny jest tylko ostatni stan przed awarią, a nie wszystkie zmiany, które zaszły podczas jej działania.

Kafka obsługuje takie przypadki użycia, pozwalając ustawić dla tematu regułę retencji `delete`, czyli **usuwanie**, która powoduje czyszczenie zdarzeń starszych niż okres retencji, albo regułę `compact` (**kompaktowanie**) przechowującą tylko najnowszą wartość dla każdego klucza w temacie. Oczywiście ustawienie reguły `compact` ma sens tylko w przypadku tematów, dla których aplikacje produkują zdarzenia zawierające zarówno klucz, jak i wartość. Jeżeli temat będzie zawierał klucze **zerowe** (`null`), kompaktowanie zakończy się niepowodzeniem.

Tematy mogą mieć również regułę `delete.and.compact`, która łączy kompaktowanie z okresem retencji. Komunikaty starsze niż okres retencji zostaną usunięte, nawet jeśli są to najnowsze wartości klucza. Reguła ta zapobiega nadmiernemu rozrostowi tematów kompaktowanych i jest stosowana także wtedy, gdy prowadzona działalność wymaga usuwania rekordów po określonym czasie.

Jak działa kompaktowanie?

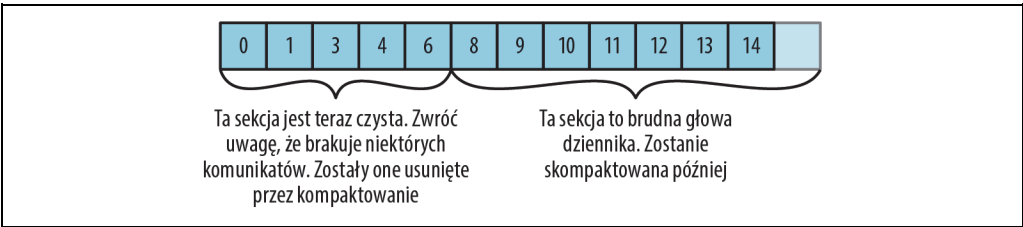
Każdy dziennik jest postrzegany jako podzielony na dwie sekcje (rysunek 6.6). Oto one:

Sekcja czysta

Komunikaty, które zostały wcześniej skompaktowane. Ta sekcja zawiera tylko jedną wartość dla każdego klucza, a jest nią ostatnia wartość z poprzedniego kompaktowania.

Sekcja brudna

Komunikaty, które zostały zapisane po ostatnim kompaktowaniu.



Rysunek 6.6. Partycja z sekcjami czystymi i brudnymi

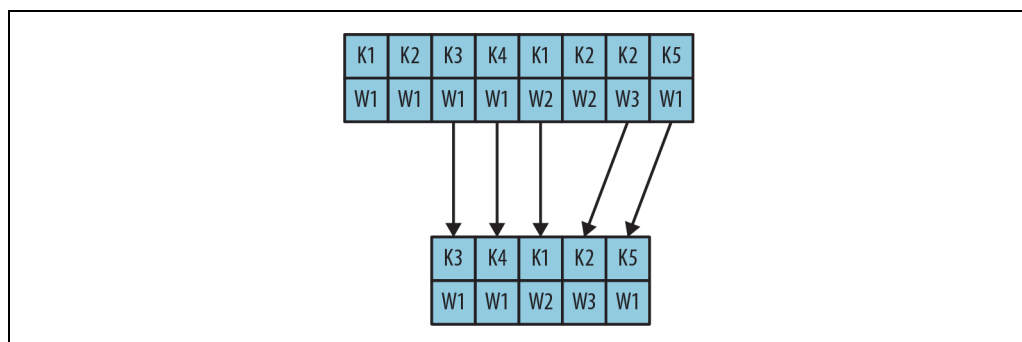
Jeżeli przy uruchamianiu Kafki kompaktowanie jest włączone (co ustawia się przy użyciu niezgrabnie nazwanej konfiguracji `log.cleaner.enabled`), każdy broker uruchamia wątek menedżera kompaktowania oraz wiele wątków kompaktowania. Są one odpowiedzialne za wykonywanie zadań kompaktowania komunikatów. Każdy wątek wybiera partycję o najwyższym współczynniku brudnych komunikatów w stosunku do całkowitego rozmiaru partycji i czyści tę partycję.

W celu skompaktowania partycji wątek odczytuje jej brudną sekcję i tworzy mapę w pamięci. Każdy wpis mapy składa się z 16-bajowego skrótu klucza komunikatu i 8-bajowego przesunięcia poprzedniego komunikatu, który miał ten sam klucz. Oznacza to, że każdy wpis mapy używa tylko 24 bajtów. Jeśli jako przykład weźmiemy segment o rozmiarze 1 GB i założymy, że każdy komunikat w segmencie

zajmuje maksymalnie 1 kB, segment będzie zawierał milion takich komunikatów, a my do skompaktowania segmentu będziemy potrzebowali mapy o rozmiarze zaledwie 24 MB (w rzeczywistości możemy potrzebować znacznie mniejszej — jeżeli klucze się powtarzają, często będziemy wykorzystywać ponownie wpisy z tymi samymi skrótami i używać mniej pamięci). Jest to całkiem wydajne!

Podczas konfigurowania Kafki administrator określa, ile pamięci mogą wykorzystywać wątki kompaktowania dla mapy przesunięć. Chociaż każdy wątek ma własną mapę, konfiguracja dotyczy całej pamięci dla wszystkich wątków. Jeśli ustawimy 1 GB dla map przesunięć kompaktowania i będziemy mieli 5 wątków czyszczenia, każdy z nich otrzyma dla swojej mapy 200 MB. Kafka nie wymaga, aby cała brudna sekcja partycji mieściła się w rozmiarze przydzielonym dla tej mapy, ale pasować musi co najmniej jeden pełny segment. W przeciwnym razie Kafka zarejestruje błąd, a administrator będzie musiał alokować więcej pamięci dla map przesunięć lub użyć mniejszej liczby wątków czyszczenia. Jeżeli pasować będzie tylko kilka segmentów, Kafka rozpocznie kompaktowanie od najstarszych, które odpowiadają rozmiarowi mapy. Reszta pozostanie brudna i będzie czekać na kolejne kompaktowanie.

Gdy wątek czyszczenia zbuduje mapę przesunięć, zainicjuje odczyt czystych segmentów, poczynając od najstarszych, i sprawdzi ich zawartość względem mapy przesunięć. Sprawdzi w mapie przesunięć, czy dla każdego komunikatu istnieje jego klucz. Jeżeli w mapie nie ma danego klucza, wartość odczytanego komunikatu pozostaje najnowsza, a komunikat jest kopiowany do segmentu zastępczego. Jeśli klucz istnieje w mapie, taki komunikat jest pomijany, ponieważ dalej w partycji znajduje się komunikat z identycznym kluczem, ale z nowszą wartością. Po skopiowaniu wszystkich komunikatów, które nadal zawierają najnowszą wartość dla swojego klucza, segment zastępczy jest podmieniany z oryginalnym, a wątek przechodzi do następnego segmentu. Pod koniec procesu zostajemy z jednym komunikatem na klucz — tym z najnowszą wartością. Pokazaliśmy to na rysunku 6.7.



Rysunek 6.7. Segment partycji przed kompaktowaniem i po

Usunięte zdarzenia

Jeżeli zawsze zachowujemy najnowszy komunikat dla każdego klucza, co zrobić, gdy naprawdę chcemy usunąć wszystkie komunikaty dla konkretnego klucza, np. dla użytkownika, który przestał korzystać z naszej usługi i jesteśmy prawnie zobowiązani do usunięcia wszystkich jego śladów z naszego systemu?

Aby całkowicie usunąć klucz z systemu, nie zapisując nawet ostatniego komunikatu, aplikacja musi wyprodukować komunikat zawierający ten klucz i wartość zerową (`null`). Gdy wątek czyszczenia napotka taki komunikat, najpierw wykona normalne kompaktowanie i zachowa tylko komunikat z wartością zerową. Ten specjalny komunikat (zwany **nagrobkiem**) będzie przechowywany przez skonfigurowany okres. W tym czasie konsumenci będą mogli zobaczyć ten komunikat i dowiedzieć się, że wartość została usunięta. Jeśli więc konsument skopiuje dane z Kafki do relacyjnej bazy danych, zobaczy nagrobkowy komunikat i będzie wiedział, że ma usunąć użytkownika z bazy danych. Po ustalonym czasie wątek czyszczenia usunie komunikat nagrobkowy, a klucz zniknie z partycji w Kafce. Ważne jest, aby dać konsumentom wystarczająco dużo czasu na zobaczenie komunikatu nagrobkowego — jeśli nasz konsument był nieaktywny przez kilka godzin i nie widział tego komunikatu, to podczas konsumowania po prostu nie zauważy danego klucza, a więc nie będzie wiedział, że został on usunięty z Kafki lub że musi zostać usunięty z bazy danych.

Warto pamiętać, że klient administracyjny Kafki zawiera również metodę `deleteRecords`, która usuwa wszystkie rekordy przed określonym przesunięciem i wykorzystuje zupełnie inny mechanizm. Po wywołaniu tej metody Kafka przeniesie znak niskiej wody, czyli swój rekord pierwszego przesunięcia partycji, do określonego przesunięcia. Uniemożliwi to konsumentom przetwarzanie rekordów znajdujących się poniżej znaku niskiej wody i w efekcie sprawi, że te rekordy będą niedostępne, aż zostaną usunięte przez wątek czyszczenia. Metoda ta może być stosowana dla tematów z regułą retencji oraz tematów skompaktowanych.

Kiedy tematy są kompaktowane?

Podobnie jak reguła `delete` nie usuwa bieżących aktywnych segmentów, reguła `compact` nigdy nie kompaktuje bieżącego segmentu. Do kompaktowania kwalifikują się tylko komunikaty w nieaktywnych segmentach.

Domyślnie Kafka rozpoczyna kompaktowanie, gdy brudne rekordy zawiera 50% tematu. Chodzi o to, żeby kompaktowanie nie odbywało się zbyt często (ponieważ może mieć wpływ na wydajność odczytu lub zapisu danych tematu), ale również nie pozostawiało zbyt wielu brudnych rekordów (gdyż konsumują one miejsce na dysku). Poświęcanie 50% przestrzeni dyskowej używanej przez temat na brudne rekordy, a następnie kompaktowanie ich za jednym razem wydaje się rozsądnym kompromisem i może być dostrajane przez administratora.

Ponadto administratorzy mogą kontrolować moment przeprowadzania kompaktowania za pomocą dwóch parametrów konfiguracyjnych:

- `min.compaction.lag.ms` może być stosowany do zagwarantowania minimalnej ilości czasu, który musi upłynąć po zapisaniu komunikatu, zanim będzie można go skompaktować.
- `max.compaction.lag.ms` może być stosowany do zagwarantowania maksymalnego opóźnienia między zapisaniem komunikatu a momentem, w którym ten komunikat będzie się kwalifikował do skompaktowania. Taka konfiguracja jest często stosowana w sytuacjach, w których istnieje biznesowy powód, by zagwarantować kompaktowanie w określonym przedziale czasu — np. RODO wymaga, aby niektóre informacje zostały wyczyszczone w ciągu 30 dni po złożeniu wniosku o ich usunięcie.

Podsumowanie

W Kafce jest oczywiście więcej zagadnień, niż moglibyśmy opisać w tym rozdziale. Mamy jednak nadzieję, że dało Ci to wyobrażenie, z jakiego rodzaju decyzjami projektowymi i optymalizacjami borykała się społeczność Kafki podczas pracy nad tym projektem, i być może udało się nam wytłumaczyć niektóre z dość niejasnych zachowań i konfiguracji, które napotkałeś podczas korzystania z Kafki.

Jeśli naprawdę interesują Cię wewnętrzne mechanizmy działania Kafki, nic nie zastąpi czytania kodu. Lista dyskusyjna programistów Kafki (dev@kafka.apache.org) skupia bardzo przyjazną społeczność i zawsze znajdzie się ktoś, kto odpowie na Twoje pytania dotyczące tego, jak naprawdę działa Kafka. A podczas czytania kodu może zdołasz poprawić jakieś błędy — w przypadku projektów *open source* kontrybutorzy są zawsze mile widziani.

Niezawodne dostarczanie danych

Niezawodność jest właściwością systemu, a nie pojedynczego komponentu, więc kiedy mówimy o gwarancjach niezawodności Apache Kafki, musimy mieć na uwadze cały system i jego przypadki użycia. Jeśli chodzi o niezawodność, systemy integrujące się z Kafką są równie ważne jak sama Kafka. A ponieważ niezawodność jest problemem systemowym, nie może być za nią odpowiedzialna tylko jedna osoba. Aby zbudować niezawodny system, współpracować muszą wszyscy: administratorzy Kafki, administratorzy Linuksa, administratorzy sieci i pamięci masowej oraz programiści aplikacji.

W zakresie dostarczania niezawodnych danych Apache Kafka jest bardzo elastyczna. Rozumiemy, że Kafka ma wiele zastosowań, od śledzenia kliknięć na stronach internetowych po płatności kartami kredytowymi. Niektóre przypadki użycia wymagają najwyższej niezawodności, podczas gdy inne stawiają na szybkość i prostotę. Kafka została napisana w sposób zapewniający odpowiednią konfigurowalność, a jej API klienta jest wystarczająco elastyczne, by umożliwić wszelkiego rodzaju kompromisy związane z poziomem niezawodności.

Ze względu na elastyczność Kafki podczas korzystania z niej łatwo jest również przypadkowo strzelić sobie w stopę — wierząc, że nasz system jest niezawodny, gdy w rzeczywistości jest wręcz przeciwnie. W tym rozdziale zaczniemy od omówienia różnych rodzajów niezawodności i ich znaczenia w kontekście platformy Apache Kafka. Następnie opiszemy mechanizm replikacji Kafki i pokażemy, w jaki sposób przyczynia się on do zapewnienia niezawodności systemu. Potem przyjrzymy się brokerom i tematom Kafki oraz sposobom ich konfiguracji dla różnych przypadków użycia. Omówimy klienty, producenci i konsumenci oraz ich zastosowanie w różnych scenariuszach niezawodności. Na koniec zajmiemy się tematem walidacji niezawodności systemu, ponieważ nie wystarczy wierzyć, że system jest niezawodny — to założenie musi zostać dokładnie przetestowane.

Gwarancje niezawodności

O niezawodności zazwyczaj mówimy w kategoriach **gwarancji**, czyli zachowań, które są gwarantowane w systemie w różnych okolicznościach.

Prawdopodobnie najbardziej znana gwarancja niezawodności to ACID (ang. *atomicity, consistency, isolation, durability*), czyli **niepodzielność, spójność, izolacja i trwałość**. Jest to standardowa gwarancja niezawodności obsługiwana powszechnie przez relacyjne bazy danych. Gdy dostawca wyjaśnia,

że jego baza danych jest zgodna z ACID, oznacza to, że gwarantuje ona określone zachowania transakcyjne.

Z powodu tych gwarancji ludzie ufają relacyjnym bazom danych, wykorzystując je w swoich najbardziej krytycznych aplikacjach — dokładnie wiedzą, co dany system oferuje i jak będzie się zachowywał w rozmaitych warunkach. Rozumieją te gwarancje i opierając się na nich, mogą pisać bezpieczne aplikacje.

Zrozumienie gwarancji zapewnianych przez Kafkę ma kluczowe znaczenie dla tych, którzy chcą budować niezawodne aplikacje. Ta wiedza pozwala twórcom systemu określić, jak będzie się on zachowywał w różnych sytuacjach awaryjnych. Co więc gwarantuje Apache Kafka?

- Kafka zapewnia gwarancję kolejności komunikatów w partycji. Jeśli komunikat *B* został zapisany po komunikacie *A* przy użyciu tego samego producenta w tej samej partycji, wówczas Kafka gwarantuje, że przesunięcie komunikatu *B* będzie większe niż komunikatu *A*, a konsumenci będą odczytywać komunikat *B* po komunikacie *A*.
- Wyprodukowane komunikaty są uważane za „zatwierdzone”, gdy zostały zapisane w partycji na wszystkich replikach zsynchronizowanych (ale niekoniecznie przeniesione na dysk). Producenci mogą wybrać otrzymywanie potwierdzeń dla wysłanego komunikatu, gdy zostanie on w pełni zatwierdzony, zapisany w liderze lub wysłany przez sieć.
- Komunikaty, które zostały zatwierdzone, nie zostaną utracone, dopóki aktywna pozostaje co najmniej jedna replika.
- Konsumenci mogą odczytywać tylko zatwierdzone komunikaty.

Te podstawowe gwarancje mogą być wykorzystywane podczas budowania niezawodnego systemu, ale same w sobie nie czynią systemu w pełni niezawodnym. Z budowaniem niezawodnego systemu związane są kompromisy, a Kafka powstała, by umożliwić administratorom i programistom decydowanie o tym, jakiego poziomu niezawodności potrzebują, przez zapewnianie parametrów konfiguracyjnych, które pozwalają kontrolować te kompromisy. Dotyczą one zazwyczaj tego, jak ważne jest niezawodne i konsekwentne przechowywanie komunikatów w porównaniu z innymi istotnymi czynnikami, takimi jak dostępność, wysoka przepływność, małe opóźnienia i koszty sprzętu.

W następnym podrozdziale opiszemy mechanizm replikacji Kafki, wprowadzimy terminologię i przyjrzymy się temu, w jaki sposób niezawodność jest wbudowana w Kafkę. Potem przejdziemy do parametrów konfiguracyjnych, o których wspomnieliśmy przed chwilą.

Replikacja

Mechanizm replikacji Kafki z jego wieloma replikami na partycję jest podstawą wszystkich gwarancji niezawodności oferowanych przez tę platformę. Kafka zapewnia trwałość komunikatów w przypadku awarii przez zapisywanie tych komunikatów w wielu replikach.

Mechanizm replikacji platformy Apache Kafka wyjaśniliśmy szczegółowo w rozdziale 6., ale podsumujmy najważniejsze kwestie.

Każdy temat Kafki jest podzielony na **partycje**, które są podstawowymi elementami konstrukcyjnymi danych. Partycja jest przechowywana na pojedynczym dysku. Kafka gwarantuje kolejność zdarzeń w ramach partycji, a partycja może być online (dostępna) lub offline (nieдоступna). Każda partycja może mieć wiele replik, a jedna z nich jest wyznaczona jako replika główna zwana liderem. Wszystkie zdarzenia są produkowane do repliki głównej i są zwykle również z niej konsumowane. Pozostałe repliki muszą po prostu pozostawać w synchronizacji z liderem i replikować na czas wszystkie najnowsze zdarzenia. Jeśli lider staje się niedostępny, nowym liderem staje się jedna z replik zsynchronizowanych (istnieje wyjątek od tej zasady, który omówiliśmy w rozdziale 6.).

Replika jest uważana za zsynchronizowaną, jeśli jest liderem partycji lub jeśli jest repliką podrzędną (obserwatorem), która spełnia poniższe wymagania:

- Ma aktywną sesję z ZooKeeperem — oznacza to, że w ciągu ostatnich 6 sekund (parametr konfigurowalny) wysłała do tego serwera sygnał heartbeat.
- Pobrała komunikaty z lidera w ciągu ostatnich 10 sekund (parametr konfigurowalny).
- Pobrała najnowsze komunikaty z lidera w ciągu ostatnich 10 sekund. Oznacza to, że samo pobieranie przez replikę podrzędną komunikatów z lidera nie wystarczy; obserwator nie może mieć opóźnień większych niż 10 sekund (parametr konfigurowalny).

Jeżeli replika utraci połączenie z ZooKeeperem, przestanie pobierać nowe komunikaty lub po prostu w tyle i nie będzie mogła nadrobić zaległości w ciągu 10 sekund, zostaje uznana za niesynchronizowaną. Niesynchronizowana replika może z powrotem się zsynchronizować, gdy ponownie połączy się z ZooKeeperem i nadrobi najnowsze komunikaty zapisane w liderze. Zwykle dzieje się to niedługo po ustaniu tymczasowej usterki sieci, ale może chwilę potrwać, jeśli broker, na którym przechowywana jest replika, był wyłączony przez dłuższy czas.



Repliki niesynchronizowane

W starszych wersjach Kafki nierzadko zdarzało się, że co najmniej jedna replika co chwilę przełączała się między statusem zsynchronizowanym i niesynchronizowanym. Była to oczywista oznaka, że coś niedobrego dzieje się z klastrem. Dość częstą przyczyną był duży maksymalny rozmiar żądania i zbyt duża sfera JVM wymagająca dostrojenia, aby zapobiec długim przestojom związanym z działaniem mechanizmu odzyskiwania pamięci, które powodowały tymczasowe rozłączenie się brokera z ZooKeeperem. Obecnie ten problem jest bardzo rzadki, zwłaszcza gdy używa się wersji co najmniej 2.5.0 platformy Apache Kafka z domyślnymi konfiguracjami dla limitu czasu połączenia z ZooKeeperem i maksymalnego opóźnienia repliki. Zastosowanie JVM w wersji przynajmniej 8 (obecnie najstarszej wersji obsługiwanej przez Kafkę) z mechanizmem odzyskiwania pamięci G1 (<https://oreil.ly/oDL86>) pomogło zażegnać ten problem, chociaż w przypadku dużych komunikatów nadal może być wymagane pewne dostrojenie. Generalnie od czasu opublikowania pierwszego wydania książki protokół replikacji Kafki stał się znacznie bardziej niezawodny. Szczegółowe informacje na temat ewolucji tego protokołu znajdziesz w doskonałym wykładzie Jasona Gustafsona *Hardening Apache Kafka Replication* (<https://oreil.ly/Z1R1w>) oraz w przeglądzie ulepszeń Kafki *Please Upgrade Apache Kafka Now* opracowanym przez Gwen Shapirę (<https://oreil.ly/vKnVI>).

Pozostająca nieco w tyle replika zsynchronizowana może spowolnić producenci i konsumenci, ponieważ muszą poczekać, aż komunikat zostanie *zatwierdzony*, co wymaga pobrania go przez wszystkie repliki zsynchronizowane. Kiedy replika przestanie być zsynchronizowana, nie czekamy, aż odbierze komunikaty. Nadal pozostaje w tyle, ale teraz nie ma wpływu na wydajność. Haczyk tkwi w tym, że przy mniejszej liczbie replik zsynchronizowanych efektywny współczynnik replikacji partycji jest niższy, a zatem istnieje większe ryzyko przestoju lub utraty danych.

W następnym podrozdziale przekonamy się, co to oznacza w praktyce.

Konfiguracja brokera

Broker ma trzy parametry konfiguracyjne, które zmieniają zachowanie Kafki w zakresie niezawodnego przechowywania komunikatów. Podobnie jak wiele zmiennych konfiguracyjnych brokera, te parametry mogą mieć zastosowanie na poziomie brokera, kontrolując konfigurację dla wszystkich tematów w systemie, i na poziomie tematu, kontrolując zachowanie konkretnego tematu.

Możliwość kontrolowania kompromisów niezawodności na poziomie tematu oznacza, że ten sam klaster Kafki może być używany do hostowania zarówno niezawodnych, jak i zawodnych tematów. Administrator w banku prawdopodobnie będzie chciał np. ustawić wysoce niezawodne wartości domyślne dla całego klastra, ale zrobi wyjątek od tematu, w którym przechowywane są skargi klientów, w przypadku których dopuszczalna jest pewna utrata danych.

Przyjrzyjmy się po kolei tym parametrom konfiguracyjnym i zobaczmy, jak wpływają na niezawodność przechowywania komunikatów w Kafce i związane z tym kompromisy.

Współczynnik replikacji

Konfiguracja na poziomie tematu to `replication.factor`, czyli współczynnik replikacji. Na poziomie brokera kontrolujemy domyślny współczynnik replikacji (`default.replication.factor`) dla tematów tworzonych automatycznie.

Dotychczas w książce przyjmowaliśmy, że tematy mają współczynnik replikacji o wartości 3, co oznacza, że każda partycja jest replikowana trzy razy na trzech różnych brokerach. Było to rozsądne założenie, ponieważ jest to domyślne ustawienie Kafki, ale tę konfigurację użytkownicy mogą modyfikować. Nawet jeśli temat już istnieje, za pomocą narzędzia Kafki służącego do przypisywania replik możemy dodawać lub usuwać repliki, a tym samym modyfikować współczynnik replikacji.

Współczynnik replikacji N pozwala na utratę $N-1$ brokerów przy zachowaniu możliwości odczytywania danych z tematu i zapisywania w nim. Dlatego wyższy współczynnik replikacji zapewnia większą dostępność, większą niezawodność i mniejszą liczbę ewentualnych katastrof. Z drugiej strony, dla współczynnika replikacji N będziemy potrzebować co najmniej N brokerów i będziemy przechowywać N kopii danych, co oznacza również, że będziemy potrzebować N razy więcej miejsca na dysku. Zasadniczo żonglujemy dostępnością i sprzętem.

Jak więc określić odpowiednią liczbę replik dla tematu? Należy wziąć pod uwagę kilka kluczowych kwestii:

Dostępność

Partycja z tylko jedną repliką stanie się niedostępna nawet podczas rutynowego restartu pojedynczego brokera. Im więcej mamy replik, tym większej dostępności możemy się spodziewać.

Trwałość

Każda replika jest kopią wszystkich danych z partycji. Jeżeli partycja ma pojedynczą replikę i z jakiegoś powodu dysk staje się bezużyteczny, tracimy wszystkie dane z partycji. Przy większej liczbie kopii, zwłaszcza znajdujących się na różnych urządzeniach pamięci masowej, zmniejsza się ryzyko utraty ich wszystkich.

Przepływność

Z każdą dodatkową repliką mnożymy ruch między brokerami. Jeśli produkujemy dane do partycji z szybkością 10 MB/s, pojedyncza replika nie wygeneruje żadnego ruchu replikującego. Gdy będziemy mieli 2 repliki, wygenerujemy ruch replikacyjny 10 MB/s, przy 3 replikach będzie to 20 MB/s, a przy 5 replikach 40 MB/s. Musimy wziąć to pod uwagę przy planowaniu wielkości i przepustowości klastrów.

Opóźnienie całkowite

Zanim każdy wyprodukowany rekord stanie się dostępny dla konsumentów, musi zostać zreplikowany do wszystkich replik zsynchronizowanych. Teoretycznie przy większej liczbie replik istnieje większe ryzyko, że jedna z nich będzie nieco powolna, a zatem spowolni również konsumenty. W praktyce niezależnie od współczynnika replikacji, jeśli jeden broker z jakiegokolwiek powodu stanie się powolny, spowoduje także spowolnienie każdego klienta, który spróbuje go użyć.

Koszty

Jest to najczęstszy powód stosowania dla mniej istotnych danych współczynnika replikacji niższego niż 3. Im więcej mamy replik danych, tym wyższe są koszty przechowywania danych i przesyłania ich w sieci. Ponieważ wiele systemów pamięci masowej domyślnie replikuje każdy blok 3 razy, czasami sensowne jest zmniejszenie kosztów przez skonfigurowanie Kafki ze współczynnikiem replikacji 2. Zwróć uwagę, że zmniejszy to dostępność w porównaniu do współczynnika replikacji 3, ale trwałość zostanie zagwarantowana przez urządzenie pamięci masowej.

Bardzo ważne jest również rozmieszczenie replik. Kafka zawsze upewnia się, że każda replika partycji znajduje się na osobnym brokerze. W niektórych przypadkach nie zapewnia to wystarczającego bezpieczeństwa. Jeśli wszystkie repliki partycji zostaną umieszczone na brokerach, które znajdują się w tej samej szafie rackowej, a najwyższy przełącznik górnej szafy będzie działał niewłaściwie, utracimy dostępność partycji niezależnie od współczynnika replikacji. Aby zabezpieczyć się przed nieszczęściem na poziomie szafy rackowej, zalecamy umieszczenie brokerów w wielu szafach i użycie parametru konfiguracyjnego brokera `broker.rack` w celu skonfigurowania nazwy racku dla każdego brokera. Jeżeli skonfigurowane są nazwy szaf rackowych, Kafka upewni się, że repliki partycji zostaną rozmieszczone w różnych szafach, aby zagwarantować jeszcze wyższą dostępność. Podczas uruchamiania Kafki w środowiskach chmurowych często jako osobne szafy rackowe traktuje się strefy dostępności. Szczegółowe informacje o tym, jak Kafka umieszcza repliki na brokerach i w szafach rackowych, znajdziesz w rozdziale 6.

Wybór nieczystego lidera

Konfiguracja ta jest dostępna tylko na poziomie brokera (i w praktyce całego klastra). Nazwa parametru to `unclean.leader.election.enable` i domyślnie ma on wartość `false`.

Jak wyjaśniliśmy wcześniej, gdy lider partycji przestaje być dostępny, jako nowy lider zostaje wybrana jedna z replik zsynchronizowanych. Ten wybór lidera jest „czysty” w tym sensie, że gwarantuje brak utraty zatwierdzonych danych — według definicji zatwierdzone dane istnieją na wszystkich replikach zsynchronizowanych.

Ale co zrobić, gdy z wyjątkiem lidera, który właśnie stał się niedostępny, nie istnieje żadna replika zsynchronizowana?

Taka sytuacja może wystąpić w jednym z dwóch scenariuszy:

- Partycja ma 3 repliki, a 2 repliki podrzędne stały się niedostępne (załóżmy, że 2 brokery uległy awarii). Ponieważ producenci nadal zapisują w liderze, wszystkie komunikaty są potwierdzane i zatwierdzane (gdyż lider jest jedyną repliką zsynchronizowaną). Załóżmy teraz, że lider staje się niedostępny (ups, kolejna awaria brokera). W tym scenariuszu, jeśli jedna z niesynchronizowanych replik podrzędnych uruchomi się pierwsza, będziemy mieli replikę niesynchronizowaną jako jedyną dostępną replikę dla partycji.
- Partycja ma 3 repliki, a ze względu na problemy z siecią 2 repliki podrzędne pozostały tak daleko w tyle, że chociaż działają i replikują, nie są już zsynchronizowane. Lider nadal przyjmuje komunikaty jako jedyną replikę zsynchronizowaną. Jeśli lider stanie się niedostępny, nowym liderem będzie mogła zostać tylko jedna z 2 dostępnych replik niesynchronizowanych.

W obu tych scenariuszach musimy podjąć trudną decyzję:

- Jeżeli nie pozwolimy, aby replika niesynchronizowana stała się nowym liderem, partycja pozostanie wyłączona, dopóki stary lider (czyli ostatnia replika zsynchronizowana) nie odzyska sprawności i nie powróci do trybu online. W niektórych przypadkach (np. gdy wymiany wymaga chip pamięci) może to potrwać wiele godzin.
- Jeżeli pozwolimy, aby replika niesynchronizowana stała się nowym liderem, utracimy wszystkie komunikaty zapisane w starym liderze, gdy wspomniana replika pozostawała niesynchronizowana. Spowoduje to również pewne niespójności u konsumentów. Dlaczego? Wyobraź sobie, że podczas gdy repliki 0 i 1 nie były dostępne, zapisaliśmy w replice 2 (ówczesnym liderze) komunikaty z przesunięciami 100 – 200. Teraz replika 2 jest niedostępna, a replika 0 jest z powrotem online. Replika 0 zawiera tylko komunikaty 0 – 100, ale nie 100 – 200. Jeśli dopuścimy, by replika 0 stała się nowym liderem, pozwoli to producentom na zapisywanie nowych komunikatów, a konsumentom na ich odczytywanie. Nowy lider będzie miał wtedy zupełnie nowe komunikaty 100 – 200. Przede wszystkim zauważmy, że jedne konsumenty mogły odczytać stare komunikaty 100 – 200, drugie mogły odczytać nowe komunikaty 100 – 200, a jeszcze inne mogły wymieszać stare komunikaty z nowymi. Może to prowadzić do dość złych konsekwencji, jeśli chodzi o takie kwestie jak raporty końcowe. Ponadto replika 2 powróci do trybu online i stanie się obserwatorem nowego lidera. W tym momencie usunie wszystkie otrzymane wcześniej komunikaty, które nie istnieją w aktualnym liderze. W efekcie komunikaty te przestaną być ostatecznie dostępne dla jakiegokolwiek konsumenta.

Możemy podsumować, że jeśli pozwolimy replikom niesynchronizowanym zostać liderami, ryzykujemy utratę danych i niespójność. Natomiast jeżeli nie pozwolimy im zostać liderami, będziemy musieli uporać się z mniejszą dostępnością, bo zanim partycja wróci do trybu online, będziemy musieli poczekać, aż pierwotny lider stanie się dostępny.

Domyślnie parametr `unclean.leader.election.enable` ma ustawioną wartość `false`, co nie pozwala replikom niesynchronizowanym zostać liderami. Jest to najbezpieczniejsza opcja, ponieważ zapewnia największe gwarancje, że dane nie zostaną utracone. Oznacza to, że w opisanych wcześniej ekstremalnych przypadkach niedostępności niektóre partycje pozostaną nieosiągalne do czasu ręcznej interwencji. Administrator zawsze może przyjrzeć się sytuacji, zdecydować o zaakceptowaniu utraty danych w celu udostępnienia partycji i przed uruchomieniem klastra przełączyć tę konfigurację na wartość `true`. Nie należy tylko zapomnieć o przywróceniu wartości `false` po odzyskaniu sprawności klastra.

Minimalna liczba replik zsynchronizowanych

Konfiguracja zarówno na poziomie tematu, jak i brokera nazywa się `min.insync.replicas`.

Jak się dowiedziałeś, w niektórych przypadkach, nawet jeśli temat jest skonfigurowany z 3 replikami, możemy zostać z pojedynczą repliką zsynchronizowaną. Jeżeli ta replika stanie się niedostępna, być może będziemy musieli wybrać pomiędzy dostępnością a spójnością. Nigdy nie jest to łatwe. Zwróć uwagę, że częściowo problem polega na tym, iż zgodnie z gwarancjami niezawodności Kafki dane są uważane za zatwierdzone, gdy zostaną zapisane we wszystkich replikach zsynchronizowanych, nawet gdy „wszystkie” oznacza tylko jedną replikę, a dane mogą zostać utracone, jeśli ta replika stanie się niedostępna.

Gdy chcemy mieć pewność, że dane zostaną zapisane w więcej niż jednej replice, musimy ustawić wyższą wartość dla minimalnej liczby replik zsynchronizowanych. Jeśli temat ma repliki i ustawimy `min.insync.replicas` na 2, to producenci będą mogli zapisywać na partycji w tym temacie tylko wtedy, gdy co najmniej 2 z 3 replik będą zsynchronizowane.

Kiedy wszystkie 3 repliki są zsynchronizowane, wszystko przebiega normalnie. Dotyczy to również sytuacji, gdy jedna z replik staje się niedostępna. Jeśli jednak niedostępne będą 2 z 3 replik, brokery przestaną przyjmować żądania produkcji. Producenci próbujące wysłać dane otrzymają wyjątek `NotEnoughReplicasException`. Natomiast konsumenci będą mogli kontynuować odczytywanie istniejących danych. W efekcie przy takiej konfiguracji pojedyncza replika zsynchronizowana staje się repliką tylko do odczytu. Zapobiega to niepożądanym sytuacji, w której dane są produkowane i konsumowane, a potem zostają utracone w razie wyboru nieczystego lidera. Aby znaleźć wyjście z tej sytuacji „tylko do odczytu”, musimy ponownie udostępnić jedną z dwóch niedostępnych partycji (np. zrestartować broker) i poczekać, aż nadrobi zaległości i zsynchronizuje się.

Utrzymywanie synchronizacji replik

Jak wspomnieliśmy wcześniej, repliki niesynchronizowane zmniejszają ogólną niezawodność, dlatego bardzo ważne jest, by nie doprowadzać do takiego stanu rzeczy. Wyjaśniliśmy również, że replika może stać się zsynchronizowana na jeden z dwóch sposobów: przez utratę łączności

z ZooKeeperem albo przez nienadążanie za liderem i zwiększanie zaległości w replikacji. Kafka posiada dwie konfiguracje brokerów, kontrolujące wrażliwość klastra na te dwie sytuacje.

Parametr `zookeeper.session.timeout.ms` określa przedział czasu, w którym broker Kafki może nie wysłać do ZooKeepera sygnału heartbeat bez uznania przez ten serwer, że broker stał się nieaktywny, i w rezultacie usunąć go z klastra. W wersji 2.5.0 wartość ta została podwyższona z 6 sekund do 18 w celu zwiększenia stabilności klastrów Kafki w środowiskach chmurowych, w których opóźnienia sieciowe są bardziej zróżnicowane. Generalnie ten przedział czasu powinien być na tyle długi, aby uniknąć przypadkowego „trzepotania” spowodowanego działaniem mechanizmu odzyskiwania pamięci lub warunkami sieciowymi, ale jednocześnie na tyle krótki, by mieć pewność, że faktycznie zamrożone brokery zostaną wykryte w odpowiednim czasie.

Jeżeli replika nie pobrała danych z lidera lub nie nadrobiła zaległości do najnowszych komunikatów z lidera przez okres dłuższy niż `replica.lag.time.max.ms`, staje się niezsynchronizowana. W wersji 2.5.0 domyślną wartość tego ustawienia zwiększono z 10 sekund do 30, by poprawić odporność klastra i uniknąć niepotrzebnego trzepotania. Zwróć uwagę, że ta wyższa wartość ma również wpływ na maksymalne opóźnienie dla konsumenta — przy wyższej wartości może to potrwać do 30 sekund, zanim komunikat dotrze do wszystkich replik, a konsumenci będą mogli go przetworzyć.

Utrwalanie na dysku

Kilkakrotnie wspominaliśmy, że Kafka będzie potwierdzać komunikaty, które nie były utrwalone na dysku, w zależności tylko od liczby replik, które otrzymały komunikaty. Kafka będzie przenosić (spłukiwać) komunikaty na dysk podczas rotacji segmentów (domyślnie o rozmiarze 1 GB) i przed restartem, ale w innych wypadkach będzie polegać na pamięci podręcznej stron Linuksa, by spłukiwać komunikaty, gdy pamięć się zapełni. Pomysł polega na tym, że posiadanie trzech maszyn w oddzielnych szafach rackowych lub strefach dostępności, z których każda ma kopię danych, jest bezpieczniejsze niż zapisywanie komunikatów na dysku na partycji głównej, ponieważ jednoczesne awarie w dwóch różnych szafach lub strefach są bardzo mało prawdopodobne. Możliwe jest jednak skonfigurowanie brokerów w taki sposób, by częściej utrwały komunikaty na dysku. Parametr konfiguracyjny `flush.messages` pozwala nam kontrolować maksymalną liczbę komunikatów niezsynchronizowanych z dyskiem, a `flush.ms` umożliwia kontrolowanie częstotliwości synchronizacji z dyskiem. Zanim skorzystasz z tej funkcjonalności, warto przeczytać, jak spłukiwanie (`fsync`) oddziałuje na przepływność Kafki i jak złagodzić jego negatywny wpływ (<https://oreil.ly/Ai1hl>).

Korzystanie z producentów w niezawodnym systemie

Nawet jeśli skonfigurujemy brokery w możliwie najbardziej niezawodny sposób, system jako całość może nadal potencjalnie stracić dane, jeżeli nie skonfigurujemy również niezawodnych producentów.

Zademonstrujemy to za pomocą dwóch przykładowych scenariuszy:

- Skonfigurowaliśmy brokery z 3 replikami, a wybory nieczystych liderów są wyłączone. Nie powinniśmy więc utracić ani jednego komunikatu, który został zatwierdzony w klastrze Kafki. Skonfigurowaliśmy jednak producent do wysyłania komunikatów z `acks=1`. Wysłałiśmy z producenta komunikat, który został zapisany na partycji głównej, ale w replikach

zsynchronizowanych jeszcze nie. Lider odesłał producentowi odpowiedź: „Wiadomość została zapisana pomyślnie” i zaraz po tym uległ awarii, zanim dane zostały replikowane do pozostałych replik. Repliki te są nadal uważane za zsynchronizowane (pamiętaj, że musi upłynąć trochę czasu, nim replika zostanie uznana za niezsynchronizowaną), a jedna z nich stanie się liderem. Ponieważ komunikat nie został zapisany w replikach, został utracony. Ale aplikacja produkcyjna potraktowała go jako zapisany pomyślnie. System jest spójny, gdyż tego komunikatu nie widział żaden konsument (komunikat nigdy nie został zatwierdzony, ponieważ repliki go nie otrzymały), lecz z perspektywy producenta komunikat został utracony.

- Skonfigurowaliśmy brokery z 3 replikami, a nieczyste wybory nieczystych liderów są wyłączane. Nauczylismy się na błędach i zaczęliśmy produkować komunikaty z `acks=all`. Założymy, że próbujemy zapisać w Kafce komunikat, ale lider partycji, na której zapisujemy, właśnie uległ awarii, a wybór nowego wciąż trwa. Kafka odpowie informacją: „Lider niedostępny”. Jeżeli na tym etapie producent nie obsłuży poprawnie błędu i nie będzie ponawiał próby, dopóki zapis się nie powiedzie, komunikat może zostać utracony. Również w tym przypadku nie jest to kwestia niezawodności brokera, ponieważ broker nigdy nie otrzymał komunikatu. Nie jest to też kwestia spójności, gdyż konsumenci również nigdy nie otrzymali komunikatu. Jeśli więc producenci nie będą poprawnie obsługiwać błędów, mogą spowodować utratę komunikatów.

Jak pokazują przykłady, każdy programista aplikacji produkującej komunikaty do Kafki musi zwrócić uwagę na dwie ważne rzeczy:

- użycie prawidłowej konfiguracji `acks`, aby spełnić wymagania niezawodności;
- poprawna obsługa błędów zarówno w konfiguracji, jak i w kodzie.

Konfigurację producenta omówiliśmy szczegółowo w rozdziale 3., ale powtórzmy ważne kwestie.

Wysyłanie potwierdzeń

Producenci mogą wybrać jeden z trzech różnych trybów potwierdzania:

`acks=0`

Oznacza to, że komunikat uważa się za pomyślnie zapisany w Kafce, jeżeli producentowi udało się wysłać go przez sieć. Nadal będziemy otrzymywać błędy, jeśli wysyłany obiekt nie będzie mógł zostać zserializowany lub karta sieciowa ulegnie awarii, ale nie otrzymamy żadnego błędu, gdy partycja będzie offline, wybory lidera będą w toku lub niedostępny będzie nawet cały klaster Kafki. Praca z `acks=0` zapewnia niskie opóźnienia produkcji (dlatego widzimy wiele benchmarków z tą konfiguracją), ale nie poprawia opóźnienia całkowitego (pamiętaj, że konsumenci nie zobaczą komunikatu, dopóki nie zostanie on zreplikowany do wszystkich dostępnych replik).

`acks=1`

Oznacza to, że w momencie otrzymania komunikatu i zapisania go w pliku danych partycji (ale niekoniecznie zsynchronizowania z dyskiem) lider wyśle potwierdzenie lub błąd. Jeśli lider się wyłączy lub ulegnie awarii, a pewne komunikaty, które zostały pomyślnie zapisane w liderze i potwierdzone, nie zostały przed awarią zreplikowane do replik obserwujących, możemy utracić dane. Przy tej konfiguracji możliwe jest również zapisywanie w liderze szybciej, niż może

on replikować komunikaty, przez co partycje nie będą w pełni zreplikowane, ponieważ lider będzie potwierdzał komunikaty od producenta przed ich replikacją.

`acks=all`

Oznacza to, że zanim lider odeśle potwierdzenie lub błąd, poczeka, aż komunikat otrzymają wszystkie repliki zsynchronizowane. W połączeniu z konfiguracją `min.insync.replicas` w brokerze pozwala to kontrolować, ile replik otrzymuje komunikat, przed jego potwierdzeniem. Jest to najbezpieczniejsza opcja — producent nie przestanie ponawiać prób wysłania komunikatu, dopóki nie zostanie on w pełni zatwierdzony. Jest to również opcja z najdłuższym opóźnieniem producenta, gdyż zanim będzie on mógł oznaczyć partię komunikatów jako „gotową” i kontynuować pracę, musi poczekać, aż komunikaty otrzymają wszystkie repliki zsynchronizowane.

Konfigurowanie prób ponawiania przez producenta

Obsługa błędów w producencie składa się z dwóch elementów: błędów, które producent obsługuje automatycznie, i błędów, które musimy obsługiwać my, jako programiści korzystający z biblioteki producentów.

Producent może obsługiwać błędy **odwracalne** (ang. *retriable*). Kiedy producent wysyła do brokera komunikat, broker może zwrócić informacje o powodzeniu lub kod błędu. Te kody błędów należą do dwóch kategorii: możliwych do usunięcia po ponowieniu próby i niemożliwych do usunięcia. Jeśli broker zwróci np. kod błędu `LEADER_NOT_AVAILABLE` (lider jest niedostępny), producent może spróbować wysłać komunikat ponownie — być może wybrany zostanie nowy broker, a druga próba zakończy się powodzeniem. Oznacza to, że `LEADER_NOT_AVAILABLE` jest **błędem odwracalnym**. Natomiast jeżeli broker zwróci wyjątek `INVALID_CONFIG` (nieprawidłowa konfiguracja), ponowna próba wysłania tego samego komunikatu nie zmieni konfiguracji. Jest to przykład **błędu nieodwracalnego**.

Kiedy naszym celem jest, by nigdy nie utracić żadnego komunikatu, generalnie najlepszym podejściem będzie skonfigurowanie producenta do ponawiania wysyłania komunikatu w przypadku napotkania błędu odwracalnego. Jak zaleciliśmy w rozdziale 3., w kwestii ponawiania prób najlepiej jest pozostawić domyślną liczbę ponowień (`MAX_INT`, czyli w efekcie nieskończoną) i użyć parametru `delivery.timeout.ms` w celu skonfigurowania maksymalnego czasu, po którym rezygnujemy z wysłania komunikatu — producent będzie ponawiał próby wysłania komunikatu, aż upłynie wskazany limit czasu.

Ponowna próba wysłania danego komunikatu wiąże się z ryzykiem, że oba komunikaty ostatecznie zostaną pomyślnie zapisane w brokerze, co prowadzi do powstawania duplikatów. Ponawianie próby i staranna obsługa błędów mogą zagwarantować, że każdy komunikat zostanie zapisany **co najmniej raz**, ale nie **dokładnie raz**. Użycie parametru `enable.idempotence=true` spowoduje, że producent będzie umieszczał w swoich rejestrach dodatkowe informacje, których brokery będą używać do pomijania komunikatów zduplikowanych w wyniku ponawiania prób. Omówimy to szczegółowo w rozdziale 8.

Dodatkowa obsługa błędów

Korzystanie z wbudowanego ponawiania prób przez producenty jest łatwym sposobem na prawidłową obsługę wielu różnych błędów bez utraty komunikatów, ale jako programiści musimy być w stanie obsługiwać jeszcze inne rodzaje błędów. Należą do nich:

- Nieodwracalne błędy brokera, takie jak błędy dotyczące rozmiaru komunikatu, błędy autoryzacji itp.
- Błędy, które wystąpiły przed wysłaniem komunikatu do brokera, np. błędy serializacji.
- Błędy, które występują, gdy producent wyczerpie wszystkie próby ponawiania lub gdy zappełni się całkowicie dostępna pamięć używana przez producent ze względu na wykorzystanie jej do przechowywania komunikatów podczas ponawiania.
- Upływanie limitów czasu.

W rozdziale 3. omówiliśmy, jak pisać procedury obsługi błędów zarówno dla synchronicznych, jak i asynchronicznych metod wysyłania komunikatów. Zawartość tych procedur obsługi błędów jest charakterystyczna dla aplikacji i jej celów. Czy odrzucamy „złe komunikaty”? Rejestrujemy błędy? Zatrzymujemy odczytywanie komunikatów z systemu źródłowego? Czy stosujemy mechanizm ciśnienia wstecznego, aby na jakiś czas powstrzymać system źródłowy od wysyłania komunikatów? Przechowujemy komunikaty w katalogu na dysku lokalnym? Wszystkie te decyzje zależą od architektury i wymagań produktu. Należy tylko pamiętać, że jeśli procedura obsługi błędów próbuje ponownie wysłać komunikat, lepiej będzie polegać na funkcjonalności ponawiania prób zapewnianej przez producent.

Korzystanie z konsumentów w niezawodnym systemie

Skoro dowiedziałeś się już, jak produkować dane z uwzględnieniem gwarancji niezawodności Kafki, pokażemy Ci, jak konsumować dane.

Jak przeczytałeś wcześniej w rozdziale, dane są dostępne dla konsumentów dopiero po zatwierdzeniu w Kafce, czyli zapisaniu we wszystkich replikach zsynchronizowanych. Oznacza to, że konsumenci otrzymują dane gwarantujące spójność. Pozostaje im jedynie upewnienie się, że śledzą, które komunikaty odczytały. Jest to klucz do tego, by nie tracić komunikatów podczas ich przetwarzania.

Podczas odczytywania danych z partycji konsument pobiera partię komunikatów, sprawdzając ostatnie przesunięcie w partii, a następnie żąda kolejnej partii komunikatów, rozpoczynającej się od ostatniego odebranego przesunięcia. Gwarantuje to, że konsument Kafki zawsze będzie otrzymywać nowe dane w poprawnej kolejności bez pomijania żadnych komunikatów.

Kiedy konsument przestaje działać, następny konsument musi wiedzieć, w którym miejscu podjąć pracę, czyli jakie było ostatnie przesunięcie, które przetworzył poprzedni konsument, zanim się zatrzymał. Tym następnym konsumentem może być nawet pierwotny konsument po ponownym uruchomieniu. Tak naprawdę nie ma to większego znaczenia — niektóre konsumenci będą przetwarzać komunikaty z tej partycji i muszą wiedzieć, od którego przesunięcia zacząć. Dlatego muszą

„zatwierdzać” swoje przesunięcia. Dla każdej konsumowanej partycji konsument przechowuje swoją aktualną lokalizację, więc ten sam lub inny konsument będzie wiedział, gdzie kontynuować po restarcie. Konsumenty mogą utracić komunikaty przede wszystkim wtedy, gdy zatwierdzają przesunięcia dla zdarzeń, które odczytały, ale których nie zdążyły jeszcze w pełni przetworzyć. W związku z tym gdy inny konsument podejmie przerwana pracę, pominię te komunikaty i nigdy nie zostaną one przetworzone. Dlatego należy zwracać szczególną uwagę na to, kiedy i w jaki sposób zatwierdzane są przesunięcia.



Zatwierdzone komunikaty i zatwierdzone przesunięcia

Zatwierdzone przesunięcie, w odróżnieniu od omówionego wcześniej **zatwierdzonego komunikatu**, polega na tym, że komunikat został zapisany we wszystkich replikach synchronizowanych i jest dostępny dla konsumentów. Zatwierdzone przesunięcia konsument wysłał do Kafki w celu potwierdzenia, że otrzymał i przetworzył wszystkie komunikaty z partycji, aż do danego przesunięcia.

W rozdziale 4. omówiliśmy szczegółowo Consumer API i opisaliśmy wiele metod zatwierdzania przesunięć. W tym podrozdziale zajmiemy się kilkoma istotnymi czynnikami i opcjami, ale szczegółowe informacje na temat korzystania z tych interfejsów API znajdziesz w rozdziale 4.

Ważne właściwości konfiguracji konsumenta w celu niezawodnego przetwarzania

Aby móc skonfigurować konsument pod kątem pożądanego zachowania w zakresie niezawodności, należy zrozumieć cztery istotne właściwości konfiguracyjne konsumenta.

Pierwszą z nich jest `group.id`, wyjaśniona szerzej w rozdziale 4. Zasadniczo chodzi o to, że jeśli dwa konsumenty mają ten sam identyfikator grupy i subskrybują ten sam temat, to każdemu z nich zostanie przypisany podzbiór partycji z tematu, a zatem każdy będzie odczytywał tylko podzbiór komunikatów (ale grupa jako całość odczyta wszystkie komunikaty). Jeśli chcemy, by dany konsument widział wszystkie komunikaty w subskrybowanych tematach, będzie on potrzebował unikatowego identyfikatora `group.id`.

Drugą istotną konfiguracją jest `auto.offset.reset`. Ten parametr kontroluje, co konsument zrobi, gdy nie zostały zatwierdzone żadne przesunięcia (np. przy pierwszym uruchomieniu) lub gdy konsument prosi o zatwierdzenia, które nie istnieją w danym brokerze (w rozdziale 4. wyjaśniliśmy, jak może dojść do takiej sytuacji). Dostępne są tutaj dwie opcje. Jeśli wybierzemy `earliest`, to gdy konsument nie będzie miał poprawnego przesunięcia, zacznie od początku partycji. Może to prowadzić do dwukrotnego przetwarzania przez konsumenta wielu komunikatów, ale gwarantuje minimalizację utraty danych. Jeżeli wybierzemy `latest`, wówczas konsument zacznie od końca partycji. Minimalizuje to przetwarzanie duplikatów przez konsument, ale prawie na pewno prowadzi do pominięcia niektórych komunikatów.

Trzecią konfiguracją jest `enable.auto.commit`. To poważna decyzja: czy pozwolimy konsumentowi zatwierdzać przesunięcia na podstawie harmonogramu, czy też planujemy zatwierdzać przesunięcia ręcznie w naszym kodzie? Główna zaleta automatycznego zatwierdzania przesunięć polega na tym, że mamy o jedno zmartwienie mniej podczas korzystania z konsumentów w naszej aplikacji.

Kiedy całe przetwarzanie konsumowanych rekordów odbywa się w ramach pętli odpytywania, automatyczne zatwierdzanie przesunięć gwarantuje, że nigdy przypadkowo nie zatwierdzimy przesunięcia, którego nie przetworzyliśmy. Najważniejszą wadą automatycznych przesunięć jest to, że nie mamy kontroli nad liczbą zduplikowanych rekordów, które aplikacja może przetwarzać, gdyż została zatrzymana po przetworzeniu niektórych rekordów, ale przed automatycznym zatwierdzeniem. Gdy aplikacja ma bardziej złożone przetwarzanie, np. przekazywanie rekordów do innego wątku w celu przetworzenia w tle, jedynym wyborem jest użycie ręcznego zatwierdzania przesunięć, ponieważ automatyczne zatwierdzenie może spowodować zatwierdzenie przesunięć dla rekordów, które konsument odczytał, ale być może jeszcze ich nie przetworzył.

Czwarta konfiguracja `auto.commit.interval.ms` jest powiązana z trzecią. Jeżeli zdecydujemy się na automatyczne zatwierdzanie przesunięć, ta konfiguracja pozwala ustawić częstotliwość zatwierdzania. Domyślny interwał wynosi 5 sekund. Generalnie częstsze zatwierdzanie zwiększa narzut, ale zmniejsza liczbę duplikatów, które mogą się pojawić w wypadku zatrzymania konsumenta.

Chociaż nie jest to bezpośrednio związane z niezawodnym przetwarzaniem danych, trudno uznać konsumenta za niezawodny, jeśli często przestaje konsumować w celu zrównoważenia obciążenia. Porady dotyczące konfiguracji konsumentów w celu zminimalizowania zbędnego równoważenia obciążenia i skrócenia zwiąanych z tym przestojów znajdziesz w rozdziale 4.

Bezpośrednie zatwierdzanie przesunięć w konsumentach

Jeżeli uznamy, że potrzebujemy większej kontroli, i zdecydujemy się na ręczne zatwierdzanie przesunięć, musimy liczyć się z konsekwencjami dotyczącymi poprawności i wydajności.

Nie będziemy omawiać tutaj mechanizmów działania oraz interfejsów API związanych z zatwierdzaniem przesunięć, ponieważ zostały opisane szczegółowo w rozdziale 4. Zamiast tego dokonamy przeglądu zagadnień istotnych przy opracowywaniu konsumenta w celu zapewnienia niezawodnego przetwarzania danych. Zaczniemy od prostych i prawdopodobnie oczywistych kwestii i przejdziemy do bardziej złożonych wzorców.

Zawsze zatwierdzaj przesunięcia po przetworzeniu komunikatów

Jeżeli całe przetwarzanie wykonujemy w pętli odpytywania i nie utrzymujemy stanu między tymi pętlami (np. w celu agregacji), powinno to być łatwe. Możemy użyć konfiguracji automatycznego zatwierdzania, zatwierdzać przesunięcia na końcu pętli odpytywania lub wewnątrz pętli z częstotliwością, która równoważy wymagania dotyczące zarówno narzutów, jak i nieprzetwarzania duplikatów. Jeśli w grę wchodzi dodatkowe wątki lub przetwarzanie stanowe, staje się to bardziej złożone, zwłaszcza że obiekt konsumenta nie jest wątkowo bezpieczny. W rozdziale 4. opowiedzieliśmy, w jaki sposób można to zrobić, i przedstawiliśmy dodatkowe przykłady.

Częstotliwość zatwierdzania to kompromis między wydajnością a liczbą duplikatów w przypadku awarii

Nawet w najprostszym przypadku, gdy całe przetwarzanie wykonujemy w pętli odpytywania i nie utrzymujemy stanu między tymi pętlami, możemy zdecydować się na wielokrotne zatwierdzanie w pętli lub zatwierdzanie jedynie co kilka pętli. Zatwierdzanie ma istotny wpływ na wydajność. Jest to podobne do produkowania z ustawieniem `acks=all`, ale wszystkie zatwierdzenia przesunięć pojedynczej grupy konsumentów są produkowane do tego samego brokera, który może zostać przeciążony. Częstotliwość zatwierdzania musi równoważyć wymagania dotyczące wydajności i braku duplikatów. Zatwierdzanie po każdym komunikacie należy przeprowadzać zawsze tylko dla tematów o bardzo niskiej przepływności.

Zatwierdzaj odpowiednie przesunięcia we właściwym czasie

Częstą pułapką podczas zatwierdzania w środku pętli odpytywania jest przypadkowe zatwierdzenie ostatniego odczytu przesunięcia podczas odpytywania, a nie po przetworzeniu ostatniego przesunięcia. Pamiętaj, że ważne jest, by zatwierdzać przesunięcia dla komunikatów zawsze po ich przetworzeniu — zatwierdzanie przesunięć dla komunikatów odczytanych, ale nie przetworzonych, może prowadzić do utraty komunikatów przez konsument. Przykłady pokazujące, jak to zrobić, znajdziesz w rozdziale 4.

Równoważenie obciążenia

Projektując aplikację, musimy pamiętać o równoważeniu obciążenia konsumentów i musimy odpowiednio je obsłużyć. Rozdział 4. zawiera kilka przykładów. Zwykle wiąże się to z zatwierdzaniem przesunięć przed anulowaniem przypisań partycji i czyszczeniem jakiegokolwiek stanu aplikacji, gdy przypisywane są jej nowe partycje.

Konsumenty mogą potrzebować ponawiania prób

W niektórych przypadkach po wywołaniu `poll` i przetworzeniu rekordów pewne rekordy nie są w pełni przetworzone i będą musiały zostać przetworzone później. Możemy np. próbować zapisać rekordy z Kafki w bazie danych, ale okazuje się, że baza danych nie jest w tym momencie dostępna i musimy spróbować ponownie później. Należy pamiętać, że inaczej niż w tradycyjnych systemach wymiany komunikatów pub-sub, konsumenci Kafki zatwierdzają przesunięcie i nie potwierdzają (`acks`) poszczególnych komunikatów. Oznacza to, że jeśli nie udało nam się przetworzyć rekordu 30, ale udało się przetworzyć rekord 31, nie powinniśmy zatwierdzać przesunięcia 31 — skutkowałoby to oznaczeniem jako przetworzonych wszystkich rekordów do 31, w tym rekordu 30, a zwykle nie tego chcemy. Aby rozwiązać ten problem, możesz zaimplementować jeden z dwóch opisanych poniżej wzorców.

Gdy napotkamy nieodwracalny błąd, jedną z opcji jest zatwierdzenie ostatniego pomyślnie przetworzonego rekordu. Następnie zapisujemy w buforze rekordy, które nadal muszą zostać przetworzone (aby następne odpytywanie ich nie nadpisało), używamy metody konsumenta `pause()` w celu upewnienia się, że kolejne odpytywania nie zwrócą danych, i próbujemy przetworzyć rekordy.

Drugą opcją przy napotkaniu nieodwracalnego błędu jest zapisanie go w osobnym temacie i kontynuowanie przetwarzania. Do obsługi ponawiania prób z tematu nieprzetworzonych komunikatów może być używana osobna grupa konsumentów lub jeden konsument może subskrybować zarówno główny temat, jak i temat nieprzetworzonych komunikatów, ale pauzować ten ostatni między próbami. Wzorec ten jest podobny do systemu utraconych komunikatów (ang. *dead-letter-queue*) stosowanego w wielu systemach wymiany komunikatów.

Konsumenty mogą potrzebować utrzymywać stan

W niektórych aplikacjach musimy utrzymywać stan między wywołaniami `poll`. Jeśli chcemy obliczyć np. średnią kroczącą, będziemy musieli aktualizować średnią przy każdym odpytywaniu Kafki o nowe komunikaty. Jeżeli nasz proces zostanie zrestartowany, będziemy musieli nie tylko zacząć konsumować od ostatniego przesunięcia, ale także odzyskać odpowiednią wartość średnią kroczącą. Jednym ze sposobów rozwiązania tego problemu jest zapisanie ostatniej skumulowanej wartości w temacie „wyniki” w tym samym czasie, w którym aplikacja zatwierdza przesunięcie. Oznacza to, że uruchamiany wątek może pobrać najnowszą skumulowaną wartość i kontynuować pracę dokładnie tam, gdzie została przerwana. W rozdziale 8. opowiemy, w jaki sposób aplikacja może zapisywać wyniki i zatwierdzać przesunięcia w pojedynczej transakcji. Generalnie jest to dość złożony problem i zalecamy przyjrzenie się bibliotece, takiej jak Kafka Streams lub Flink, która zapewnia wysokopoziomowe interfejsy API typu DSL-owego do agregacji, łączenia, okien i innych złożonych analiz.

Walidacja niezawodności systemu

Po przejściu przez proces ustalania wymagań dotyczących niezawodności, konfigurowania brokerów i klientów oraz korzystania z interfejsów API w najlepszy sposób dla naszego przypadku użycia możemy się po prostu zrelaksować i uruchomić wszystko w środowisku produkcyjnym, mając pewność, że żadne zdarzenie nigdy nie zostanie pominięte, prawda?

Zalecamy najpierw przeprowadzenie walidacji w trzech warstwach: konfiguracji, aplikacji i monitorowania aplikacji w środowisku produkcyjnym. Przyjrzyjmy się każdemu z tych kroków i zobaczmy, co i jak musimy zweryfikować.

Walidacja konfiguracji

Konfigurację brokera i klienta można przetestować dość łatwo w oderwaniu od logiki aplikacji, a zaleca się to z dwóch powodów:

- pomaga to sprawdzić, czy wybrana konfiguracja może spełnić nasze wymagania;
- jest to dobre ćwiczenie w analizowaniu oczekiwanego zachowania systemu.

Do przeprowadzenia tej walidacji można wykorzystać dwa ważne narzędzia oferowane przez Kafkę. Pakiet `org.apache.kafka.tools` zawiera klasy `VerifiableProducer` (weryfikowalny producent) i `VerifiableConsumer` (weryfikowalny konsument). Mogą one być uruchamiane jako narzędzia wiersza poleceń lub zostać osadzone we frameworku zautomatyzowanego testowania.

Chodzi o to, że obiekt `VerifiableProducer` produkuje sekwencję komunikatów zawierających liczby z przedziału od 1 do wybranej przez nas wartości. Weryfikowalny producent możemy skonfigurować zgodnie z konfiguracją własnego producenta, ustawiając odpowiednie wartości parametrów `acks`, `retries` i `delivery.timeout.ms` oraz szybkość produkowania komunikatów. Kiedy go uruchomimy, będzie na podstawie otrzymanych potwierdzeń wypisywać powodzenie lub błąd dla każdego komunikatu wysłanego do brokera. Weryfikowalny konsument przeprowadza kontrolę uzupełniającą. Konsumuje zdarzenia (zwykle te produkowane przez weryfikowalny producent) i wypisuje kolejno informacje o zdarzeniach, które skonsumował. Wypisuje również informacje dotyczące zatwierdzeń i równoważeń obciążenia.

Należy się zastanowić, jakie testy chcemy przeprowadzić. Mogą to być np.:

- Wybór lidera. Co się stanie, jeśli wyłączymy lidera? Ile czasu zajmuje producentowi i konsumentowi wznowienie normalnej pracy?
- Wybór kontrolera. Jak długo trwa wznowienie pracy systemu po zrestartowaniu kontrolera?
- Restart krocący. Czy możemy zrestartować brokery jeden po drugim bez utraty żadnych komunikatów?
- Test wyboru nieczystego lidera. Co się dzieje, gdy po kolei zamykamy wszystkie repliki partycji (aby mieć pewność, że każda straci synchronizację), a następnie uruchamiamy broker, który był niesynchronizowany? Co należy zrobić, by wznowić działanie? Czy jest to akceptowalne?

Następnie wybieramy scenariusz, uruchamiamy weryfikowalny producent i weryfikowalny konsument i realizujemy ten scenariusz, np. wyłączamy lidera partycji, do której produkujemy dane. Jeżeli oczekiwaliśmy krótkiego przestoju, a następnie powrotu wszystkich komponentów do normy bez utraty komunikatów, musimy się upewnić, że liczba komunikatów wyprodukowanych przez producent i liczba komunikatów skonsumowanych przez konsument są zgodne.

Repozytorium źródłowe platformy Apache Kafka zawiera obszerny zestaw testów (<https://oreil.ly/IjJx8>). Wiele testów z tego zestawu opiera się na tej samej zasadzie i wykorzystuje `VerifiableProducer` i `VerifiableConsumer`, aby zweryfikować działanie uaktualnień krocących.

Walidacja aplikacji

Po upewnieniu się, że konfiguracja brokera i klienta spełnia nasze wymagania, należy przetestować, czy aplikacja zapewnia wymagane gwarancje. Obejmuje to sprawdzenie takich elementów, jak niestandardowy kod obsługi błędów, zatwierdzenia przesunąć, nasłuchiwanie równoważenia obciążenia i inne podobne elementy, gdzie logika aplikacji współpracuje z bibliotekami klienckimi Kafki.

Ponieważ logika aplikacji może być bardzo różna, możemy oczywiście podać tylko podstawowe wskazówki dotyczące jej testowania. W ramach każdego procesu rozwoju zalecamy przeprowadzanie testów integracyjnych aplikacji oraz przeprowadzenie testów w warunkach różnych sytuacji awaryjnych. Możemy wymienić kilka podstawowych:

- klienci tracą łączność z jednym z brokerów;
- obserwowane jest duże opóźnienie między klientem a brokerem;
- zapelnienie dysku;

- zawieszenie się dysku (tzw. zaciemnienie dysku; ang. *brown out*);
- wybór lidera;
- kroczące restarty brokerów;
- kroczące restarty konsumentów;
- kroczące restarty producentów.

Do wprowadzania błędów sieciowych i dyskowych dostępnych jest wiele doskonałych narzędzi, więc nie będziemy się starali formułować konkretnych zaleceń. Apache Kafka zawiera framework testowy Trogdor (<https://oreil.ly/P3ai1>) do wstrzykiwania błędów. Dla każdego scenariusza mamy **zachowanie oczekiwane**, które planujemy zaobserwować podczas opracowywania aplikacji. Następnie przeprowadzamy test, aby zobaczyć, co się tak naprawdę stanie. W przypadku kroczących restartów konsumentów możemy oczekiwać np. krótkiego przestoju z powodu rebalansowania konsumentów, a następnie kontynuowania konsumowania z maksymalnie 1000 zduplikowanych wartości. Nasz test pokaże, czy zatwierdzanie przesunięć i obsługiwanie przez aplikację procesów równoważenia obciążenia faktycznie działa w zakładany sposób.

Monitorowanie niezawodności w środowisku produkcyjnym

Testowanie aplikacji jest ważne, ale nie zastępuje potrzeby ciągłego monitorowania systemów produkcyjnych, by mieć pewność, że dane przepływają zgodnie z oczekiwaniami. W rozdziale 12. przedstawimy szczegółowe sugestie dotyczące sposobu monitorowania klastra Kafki, ale oprócz monitorowania kondycji klastra istotne jest również monitorowanie klientów i przepływu danych przez system.

Napisane w Javie klienty Kafki uwzględniają wskaźniki JMX, które umożliwiają monitorowanie statusu i zdarzeń po stronie klienta. Dla producentów dwoma najważniejszymi wskaźnikami niezawodności są współczynniki błędów i ponowień prób na rekord (zagregowane). Powinieneś się im przyglądać, ponieważ rosnące współczynniki błędów lub ponowień prób mogą wskazywać na problem z systemem. Monitoruj również dzienniki producentów pod kątem błędów, które występują podczas wysyłania zdarzeń i są rejestrowane na poziomie WARN, zawierając informacje w stylu *Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (two attempts left). Error: ...* Kiedy widzimy zdarzenia, dla których pozostało 0 prób (*0 attempts left*), producentowi skończyła się możliwość ponawiania prób. W rozdziale 3. powiedzieliśmy, jak skonfigurować parametry `delivery.timeout.ms` i `retries`, aby poprawić obsługę błędów w producencie i uniknąć przedwczesnego wyczerpywania się możliwości ponawiania prób. Oczywiście zawsze lepiej jest od razu rozwiązać problem, który w ogóle powoduje później te błędy. Komunikaty dziennika producenta na poziomie ERROR mogą wskazywać, że wysłanie komunikatów nie powiodło się całkowicie z powodu błędu nieodwracalnego, błędu odwracalnego, który spowodował wyczerpanie się prób, lub przekroczenia limitu czasu. W stosownych przypadkach zarejestrowany zostanie również odpowiedni błąd brokera.

Po stronie konsumentów najważniejszym wskaźnikiem jest zaległość konsumenta. Ten wskaźnik określa, jak daleko konsument znajduje się za ostatnim komunikatem zatwierdzonym w partycji na brokerze. Najlepiej, gdyby zaległość była zawsze zerowa, a konsument odczytywał zawsze

najnowszy komunikat. Ponieważ w praktyce wywołanie `poll()` zwraca wiele komunikatów, a następnie konsument zajmuje się ich przetwarzaniem, zanim pobierze kolejne komunikaty, zaległość zawsze będzie się trochę wahać. Należy się upewnić, że konsumenci będą mogli ostatecznie nadrobić zaległości, a nie pozostawać coraz dalej w tyle. Ze względu na oczekiwane wahania w zaległości konsumenta ustawianie tradycyjnych alertów dla wskaźników może być trudne. Może to ułatwić narzędzie Burrow (<https://oreil.ly/supYI>) od LinkedIna, które jest kontrolerem zaległości konsumenta.

Monitorowanie przepływu danych oznacza również upewnienie się, że wszystkie produkowane dane są konsumowane terminowo (wskaźnik „terminowo” jest zwykle oparty na wymaganiach biznesowych). Aby zapewnić terminowe konsumowanie danych, musimy wiedzieć, kiedy dane zostały wyprodukowane. Pomaga w tym Kafka: od wersji 0.10.0 wszystkie komunikaty zawierają znacznik czasu wskazujący, kiedy określone zdarzenie zostało wyprodukowane (choć należy pamiętać, że może on zostać nadpisany przez aplikację, która wysyła zdarzenia, lub przez same brokery, jeśli zostały tak skonfigurowane).

Aby mieć pewność, że wszystkie produkowane komunikaty będą konsumowane w rozsądnym czasie, aplikacja produkująca komunikaty musi rejestrować liczbę generowanych zdarzeń (zwykle liczbę zdarzeń na sekundę). Konsumenty muszą rejestrować liczbę zdarzeń konsumowanych w wybranej jednostce lub w określonym czasie oraz opóźnienie między produkowaniem a konsumowaniem zdarzeń przy użyciu znacznika czasu zdarzenia. Następnie system musi porównywać liczby zdarzeń na sekundę uzyskane od producenta i konsumenta, by weryfikować, czy po drodze nie zostały utracone żadne komunikaty, oraz upewniać się, że interwał między czasem produkcji a czasem konsumpcji jest rozsądny. Tego typu systemy kompleksowego monitorowania mogą być trudne i czasochłonne w implementacji. Zgodnie z naszą najlepszą wiedzą nie ma żadnej implementacji tego typu systemu udostępnianej na licencji *open source*, ale Confluent zapewnia komercyjną implementację w ramach Confluent Control Center (<https://oreil.ly/KnvVV>).

Oprócz monitorowania klientów i kompleksowego przepływu danych, brokery Kafki zawierają wskaźniki, które określają współczynnik informujący o błędach odpowiedzi wysyłanych przez brokery do klientów. Zalecamy gromadzenie takich wskaźników jak `kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec` i `kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestPerSec`. Czasami pewien poziom informujących o błędach odpowiedzi jest spodziewany — np. jeśli wyłączymy broker w celu konserwacji, a na innym brokerze wybrani zostaną nowi liderzy, oczekuje się, że producenci otrzymają błąd `NOT_LEADER_FOR_PARTITION`, który spowoduje, że przed kontynuowaniem normalnej produkcji zdarzeń zażądają zaktualizowanych metadanych. Niewyjaśniony wzrost liczby żądań kończących się niepowodzeniem powinien być zawsze badany. Pomocne w takich dochodzeniach jest to, że wskaźniki nieudanych żądań są oznaczane wysyłaną przez broker konkretną odpowiedzią z błędem.

Podsumowanie

Jak wspomnieliśmy na początku rozdziału, niezawodność to nie tylko kwestia charakterystycznych funkcjonalności Kafki. Musimy zbudować cały niezawodny system, w tym architekturę aplikacji, sposób wykorzystywania przez aplikacje interfejsów API producenta i konsumenta, konfigurację producenta i konsumenta, konfigurację tematu i konfigurację brokera. Zwiększanie niezawodności

systemu zawsze wiąże się z kompromisami w zakresie złożoności aplikacji, wydajności, dostępności lub wykorzystania przestrzeni dyskowej. Dzięki poznaniu wszystkich tych opcji i powszechnie stosowanych wzorców oraz zrozumieniu wymagań dla każdego przypadku użycia możemy podejmować świadome decyzje dotyczące niezawodności aplikacji i wdrożeń Kafki oraz tego, które kompromisy mają sens.

Semantyka „dokładnie raz”

W rozdziale 7. omówiliśmy parametry konfiguracyjne i najlepsze praktyki, które pozwalają użytkownikom kontrolować gwarancje niezawodności Kafki. Skupiliśmy się na dostarczaniu „co najmniej raz” — gwarancji, że Kafka nie będzie tracić komunikatów, które potwierdziła jako zatwierdzone. Pomimo tego pozostaje jeszcze ryzyko duplikowania komunikatów.

W prostych systemach, w których komunikaty są produkowane, a następnie konsumowane przez różne aplikacje, duplikaty są denerwujące, ale dość łatwe do obsłużenia. Większość rzeczywistych aplikacji zawiera unikatowe identyfikatory, które mogą być używane przez aplikacje konsumujące do deduplikacji komunikatów.

Sytuacja staje się bardziej złożona w przypadku aplikacji do przetwarzania strumieniowego, które agregują zdarzenia. Podczas inspekcji aplikacji, która konsumuje zdarzenia, oblicza średnią i generuje wyniki, często osobom sprawdzającym trudno sobie wyobrazić możliwość odkrycia, że średnia będzie nieprawidłowa, ponieważ podczas jej obliczania zdarzenie zostało przetworzone dwa razy. W takich sytuacjach ważne jest zapewnienie silniejszej gwarancji — semantyki przetwarzania dokładnie raz.

W tym rozdziale powiemy, w jaki sposób używać Kafki z semantyką „dokładnie raz”, przedstawimy zalecane przypadki użycia oraz ograniczenia. Podobnie jak w przypadku gwarancji „co najmniej raz”, przyjrzymy się nieco bliżej sposobowi implementowania tej gwarancji. Te szczegóły możesz pominąć przy pierwszym czytaniu rozdziału, ale warto je zrozumieć przed użyciem tej funkcjonalności — pomogą Ci wyjaśnić znaczenie różnych konfiguracji i interfejsów API, zapewniając wskazówki dotyczące użycia ich w możliwie najlepszy sposób.

W Kafce semantyka „dokładnie raz” to połączenie dwóch kluczowych funkcjonalności: producentów idempotentnych, które pomagają unikać duplikatów powodowanych ponawianiem prób przez producenty, oraz semantyki transakcyjnej, gwarantującej przetwarzanie dokładnie raz w aplikacjach przetwarzania strumieniowego. Omówimy obie, zaczynając od prostszego i generalnie bardziej użytecznego producenta idempotentnego.

Producent idempotentny

Usługa jest nazywana idempotentną, jeśli wykonując wielokrotnie tę samą operację, daje zawsze taki sam wynik, jak za pierwszym razem. W bazach danych zwykle jest to demonstrowane jako różnica między `UPDATE t SET x=x+1 where y=5` i `UPDATE t SET x=18 where y=5`. Pierwsza operacja nie jest idempotentna; jeśli wywołamy ją trzykrotnie, za każdym razem otrzymamy zupełnie inny wynik. Drugi przykład jest idempotentny — bez względu na to, ile razy uruchomimy tę instrukcję, `x` będzie równe 18.

Jaki to ma związek z producentem Kafki? Jeżeli skonfigurujemy producent tak, aby miał semantykę „co najmniej”, a nie idempotentną, będzie to oznaczało, że w przypadkach niepewności producent ponowi próbę wysłania komunikatu, aby dotarł przynajmniej raz. Te powtórki mogą prowadzić do powstawania duplikatów.

Klasyczny przypadek jest wtedy, gdy lider partycji otrzymał od producenta rekord, z powodzeniem zreplikował go do partycji podrzędnych, a następnie broker, na którym znajduje się lider, uległ awarii, zanim zdążył wysłać odpowiedź do producenta. Jeśli producent po określonym czasie nie otrzyma odpowiedzi, ponownie wyśle komunikat. Komunikat dotrze do nowego lidera, który ma już jego kopię z poprzedniej próby — w rezultacie powstanie duplikat.

W niektórych aplikacjach duplikaty nie mają większego znaczenia, ale w innych mogą prowadzić do błędów w inwentaryzacji, nieprawidłowych sprawozdań finansowych lub wysłania komuś dwóch parasoli zamiast jednego, który zamówił.

Idempotentny producent Kafki rozwiązuje ten problem, automatycznie wykrywając i usuwając takie duplikaty.

Jak działa idempotentny producent?

Gdy włączymy idempotentny producent, każdy komunikat będzie zawierał unikatowy identyfikator producenta (ang. *Producer ID* — PID) i numer sekwencyjny. Wraz z tematem docelowym i partycją identyfikują one jednoznacznie każdy komunikat. Brokery używają tych unikatowych identyfikatorów do śledzenia ostatnich pięciu komunikatów generowanych przez każdą partycję brokera. Aby ograniczyć liczbę numerów sekwencyjnych, które muszą być śledzone dla każdej partycji, wymagane jest również, aby producenci używały ustawienia `max.inflight.requests=5` (wartość domyślna).

Gdy broker otrzyma komunikat, który już wcześniej zaakceptował, odrzuca duplikat z odpowiednią informacją o błędzie. Błąd ten jest rejestrowany przez producent i odzwierciedlany w jego wskaźnikach, ale nie powoduje żadnych wyjątków i nie powinien wywoływać alarmów. W przypadku klienta producenta jest on dodawany do wskaźnika `record-error-rate`. W brokerze będzie to część wskaźnika `ErrorsPerSec` typu `RequestMetrics`, zawierającego osobny licznik dla każdego typu błędu.

Co zrobić, jeśli broker otrzyma nieoczekiwanie wysoki numer sekwencyjny? Broker oczekuje, że po komunikacie nr 2 pojawi się komunikat nr 3; co się stanie, jeżeli zamiast tego broker otrzyma komunikat nr 27? W takich przypadkach broker odpowiada błędem *out of order sequence number*

(nie zachowano kolejności numerów sekwencyjnych), ale jeśli użyjemy producenta idempotentnego bez zastosowania transakcji, ten błąd można zignorować.



Chociaż po napotkaniu wyjątku „nie zachowano kolejności numerów sekwencyjnych” producent będzie kontynuował normalną pracę, błąd ten zazwyczaj wskazuje, że między producentem a brokerem utracone zostały jakieś komunikaty — jeśli broker otrzymał komunikat nr 2, a następnie komunikat nr 27, coś musiało się stać z komunikatami od 3 do 26. W razie natknięcia się na taki błąd w dziennikach warto ponownie zapoznać się z konfiguracją producenta i tematu oraz upewnić się, że producent jest skonfigurowany z zalecanymi wartościami w celu zagwarantowania wysokiej niezawodności, i sprawdzić, czy doszło do wyboru nieczystego lidera.

Jak zawsze w przypadku systemów rozproszonych interesujące jest przeanalizowanie zachowania producenta idempotentnego w warunkach awarii. Rozważmy dwa przypadki: restart producenta i awarię brokera.

Restart producenta

Gdy producent ulegnie awarii, zwykle w jego miejsce tworzony jest nowy producent — ręcznie przez zrestartowanie maszyny lub przy użyciu bardziej zaawansowanych frameworków, takich jak Kubernetes, które zapewniają automatyczne odzyskiwanie sprawności po awarii. Kluczową kwestią jest to, że jeżeli po uruchomieniu producenta włączona jest opcja idempotentności, producent zostanie zainicjowany i skomunikuje się z brokerem Kafki w celu wygenerowania identyfikatora producenta (PID). Każde inicjowanie producenta zaowocuje zupełnie nowym ID (przy założeniu, że nie włączyliśmy transakcji). Oznacza to, że jeśli producent ulegnie awarii, a jego zastępca wyśle komunikat, który został wysłany wcześniej przez stary producent, to broker nie wykryje duplikatów — te dwa komunikaty będą miały inne identyfikatory producenta i odmienne numery sekwencyjne, stąd będą traktowane jak dwa różne komunikaty. Zwróć uwagę, że to samo dotyczy sytuacji, w której stary producent zawiesił się, a następnie wznowił działanie już po uruchomieniu jego zastępcy — pierwotnego producenta nie uznaje się za zombie, ponieważ mamy dwa zupełnie różne producenty z unikatowymi PID-ami.

Awaria brokera

Gdy broker ulega awarii, dla partycji, które miały na nim liderów, kontroler wybiera nowych. Powiedzmy, że mamy producent, który wyprodukował komunikaty do tematu A, na partycji 0 z liderem na brokerze 5 i repliką podrzędną na brokerze 3. Po awarii brokera 5 nowym liderem staje się broker 3. Za pośrednictwem protokołu metadanych producent wykrywa, że nowym liderem jest broker 3, i zaczyna produkować do niego komunikaty. Ale skąd broker 3 będzie wiedział, które numery sekwencyjne zostały już wyprodukowane, aby odrzucić duplikaty?

Lider stale aktualizuje w pamięci swój stan producenta o pięć ostatnich numerów sekwencyjnych za każdym razem, gdy generowany jest nowy komunikat. Repliki podrzędne aktualizują własne bufor pamięci przy każdej replikacji nowych komunikatów z lidera. Oznacza to, że gdy replika podrzędna staje się liderem, ma już w pamięci najnowsze numery sekwencyjne, a walidacja nowo generowanych komunikatów może być kontynuowana bez żadnych problemów ani opóźnień.

Co się jednak stanie, gdy wróci stary lider? Po restarcie w pamięci nie będzie już starego stanu producenta. W celu ułatwienia odzyskiwania pamięci brokery wykonują migawkę stanu producenta do pliku za każdym razem, gdy zostają zamknięte lub tworzony jest segment. Kiedy broker zostaje uruchomiony, odczytuje najnowszy stan z pliku. Zrestartowany broker aktualizuje następnie stan producenta w miarę nadrabiania zaległości przez replikację z aktualnego lidera i gdy jest gotowy, aby ponownie stać się liderem, ma w pamięci najnowsze numery sekwencyjne.

A jeśli broker uległ awarii, a ostatnia migawka nie została zaktualizowana? Identyfikator producenta i numer sekwencyjny są również częścią formatu komunikatu zapisywanego w dziennikach Kafki. Podczas odzyskiwania sprawności po awarii stan producenta zostaje przywrócony przez odczytanie starszej migawki oraz komunikatów z najnowszego segmentu każdej partycji. Nowa migawka jest zapisywana zaraz po zakończeniu procesu odzyskiwania.

Ciekawą kwestią jest to, co się stanie, gdy nie będzie żadnych komunikatów. Wyobraź sobie, że dany temat ma dwie godziny retencji, ale w ciągu ostatnich dwóch godzin nie pojawiły się nowe komunikaty — w celu przywrócenia stanu w przypadku awarii brokera nie będzie można wykorzystać żadnych komunikatów. Na szczęście brak komunikatów oznacza również brak duplikatów. W takiej sytuacji natychmiast rozpoczynamy przyjmowanie komunikatów (rejestrując jednocześnie ostrzeżenie o braku stanu) i tworzymy stan producenta na podstawie nowych przychodzących komunikatów.

Ograniczenia producenta idempotentnego

Idempotentny producent Kafki zapobiega duplikatom tylko w przypadku ponawiania prób spowodowanego wewnętrzną logiką producenta. Dwukrotne wywołanie metody `producer.send()` z tym samym komunikatem spowoduje utworzenie duplikatu, a producent idempotentny temu nie zapobiegnie. Jest to spowodowane tym, że producent nie ma możliwości sprawdzenia, czy dwa przesłane rekordy są w rzeczywistości tymi samymi rekordami. Zawsze dobrym pomysłem jest użycie wbudowanego mechanizmu ponawiania prób producenta zamiast wychwytywania wyjątków producenta i ponawiania prób z poziomu samej aplikacji; producent idempotentny sprawia, że ten wzorzec staje się jeszcze bardziej atrakcyjny — jest to najłatwiejszy sposób na uniknięcie duplikatów podczas ponawiania prób.

Dość powszechne jest również, że aplikacje mają wiele instancji lub nawet jedną instancję z wieloma producentami. Jeśli dwa z tych producentów spróbują wysłać identyczne komunikaty, producent idempotentny nie wykryje duplikacji. Ten scenariusz jest dość typowy w aplikacjach, które pobierają dane ze źródła, np. z katalogu z plikami, i produkują je do Kafki. Jeżeli akurat dwie instancje aplikacji odczytywały ten sam plik i produkowały rekordy do Kafki, otrzymamy w tym pliku kilka kopii rekordów.



Producent idempotentny zapobiega jedynie duplikatom generowanym przez mechanizm ponawiania samego producenta, niezależnie od tego, czy ponawianie prób jest powodowane błędami producenta, sieci czy brokera. Ale nic poza tym.

Jak korzystać z producenta idempotentnego Kafki?

To najłatwiejsza część. Wystarczy dodać do konfiguracji producenta ustawienie `enable.idempotence=true`. Jeśli producent jest już skonfigurowany z ustawieniem `acks=all`, nie będzie różnicy w wydajności. Włączenie producenta idempotentnego spowoduje kilka zmian:

- Aby pobrać PID, producent podczas uruchamiania będzie musiał wykonać jedno dodatkowe wywołanie interfejsu API.
- Każda wysłana partia rekordów będzie zawierała identyfikator producenta i numer sekwencyjny dla pierwszego komunikatu w partii (numery sekwencyjne dla poszczególnych komunikatów w partii są obliczane na podstawie numeru sekwencyjnego pierwszego komunikatu powiększonego o deltę). Te nowe pola dodają do każdej partii rekordów 96 bitów (identyfikator producenta jest długi, a numer sekwencyjny jest liczbą całkowitą), co dla większości obciążeń roboczych nie stanowi zauważalnego narzutu.
- Brokery będą weryfikować numery sekwencyjne z każdej instancji poszczególnych producentów i gwarantować brak zduplikowanych komunikatów.
- Kolejność komunikatów produkowanych do każdej partycji będzie gwarantowana we wszystkich scenariuszach awarii, nawet jeśli dla parametru `max.inflight.requests.per.connection` będzie ustawiona wartość większa niż 1 (5 jest wartością domyślną, a także najwyższą wartością obsługiwaną przez producent idempotentny).



W wyniku zaimplementowania propozycji KIP-360 w wersji 2.5 znacznie poprawiły się logika i sposób obsługi błędów producenta idempotentnego (zarówno po stronie producenta, jak i brokera). We wcześniejszych wersjach stan producenta nie zawsze był utrzymywany wystarczająco długo, co skutkowało występowaniem błędów krytycznych UNK → NOWN_PRODUCER_ID w różnych scenariuszach (jeśli chodzi o ponowne przypisywanie partycji, znany był przypadek brzegowy, w którym nowa replika stawała się liderem, zanim doszło do jakichkolwiek zapisów z konkretnego producenta, co oznaczało, że nowy lider nie miał żadnego stanu dla danej partycji). Ponadto w poprzednich wersjach dochodziło do prób przepisywania numerów sekwencyjnych w niektórych scenariuszach błędów, co mogło prowadzić do powstawania duplikatów. W nowszych wersjach, jeśli napotkamy błąd krytyczny dla partii rekordów, odrzucone zostaną ta partia i wszystkie partie będące w toku. Użytkownik, który pisze aplikację, może obsługiwać ten wyjątek i zdecydować, czy pominąć te rekordy, czy spróbować ponownie, ryzykując duplikaty i zmianę kolejności.

Transakcje

Jak wspomnieliśmy we wstępie do tego rozdziału, do Kafki dodano transakcje gwarantujące poprawność aplikacji opracowanych przy użyciu biblioteki `Kafka Streams`. Aby aplikacja przetwarzania strumieniowego generowała poprawne wyniki, każdy rekord wejściowy musi być przetwarzany dokładnie raz, a rezultat jego przetwarzania ma zostać odzwierciedlony dokładnie raz, nawet w przypadku awarii. Transakcje w Apache Kafce umożliwiają aplikacjom przetwarzania strumieniowego generowanie dokładnych wyników. To z kolei pozwala programistom korzystać z aplikacji przetwarzania strumieniowego w przypadkach, w których dokładność jest kluczowym wymogiem.

Należy pamiętać, że transakcje w Kafce zostały opracowane specjalnie dla aplikacji przetwarzania strumieniowego. Dlatego zostały zaprojektowane do współpracy z wzorcem Konsumuj-Przetwarzaj-Produkuj (ang. *Consume-Process-Produce*), który stanowi podstawę aplikacji przetwarzania strumieniowego. Wykorzystanie transakcji może zagwarantować w tym kontekście semantykę „dokładnie raz” — przetwarzanie każdego rekordu wejściowego zostanie uznane za zakończone po zaktualizowaniu stanu wewnętrznego aplikacji i pomyślnym wyprodukowaniu wyników do tematów wyjściowych. W dalszej części rozdziału, w punkcie „Jakich problemów nie rozwiązują transakcje?”, przeanalizujemy kilka scenariuszy, w których nie będą miały zastosowania gwarancje „dokładnie raz” Kafki.



Transakcje to nazwa mechanizmu bazowego. Semantyka lub inaczej gwarancje „dokładnie raz” to zachowanie aplikacji przetwarzania strumieniowego. Do zaimplementowania swoich gwarancji „dokładnie raz” Kafka Streams korzysta z transakcji. Inne frameworki przetwarzania strumieniowego, takie jak Spark Streaming lub Flink, zapewniają swoim użytkownikom semantykę „dokładnie raz” przy użyciu innych mechanizmów.

Przypadki użycia transakcji

Transakcje są przydatne w każdej aplikacji przetwarzania strumieniowego, w której ważna jest dokładność, zwłaszcza gdy przetwarzanie strumieniowe obejmuje agregację i (lub) łączenie. Jeżeli aplikacja przetwarzania strumieniowego wykonuje tylko pojedynczą transformację i filtrowanie rekordów, nie ma stanu wewnętrznego do aktualizacji, a nawet jeśli w procesie wprowadzane są duplikaty, dość łatwo jest odfiltrować je ze strumienia wyjściowego. Gdy aplikacja przetwarzania strumieniowego agreguje kilka rekordów w jeden, znacznie trudniej będzie sprawdzić, czy rekord wynikowy jest nieprawidłowy, ponieważ niektóre rekordy wejściowe zostały policzone więcej niż raz; skorygowanie wyniku nie jest możliwe bez ponownego przetworzenia danych wejściowych.

Typowymi przykładami złożonych aplikacji przetwarzania strumieniowego są aplikacje finansowe, w których funkcjonalność „dokładnie raz” jest wykorzystywana w celu zagwarantowania dokładnej agregacji. Ponieważ jednak konfiguracja dowolnej aplikacji biblioteki Kafka Streams w celu zapewnienia gwarancji „dokładnie raz” jest raczej trywialna, napotykalismy jej stosowanie w bardziej przyziemnych przypadkach użycia, takich jak np. czatboty.

Jakie problemy rozwiązują transakcje?

Rozważmy prostą aplikację przetwarzania strumieniowego, która odczytuje zdarzenia z tematu źródłowego, prawdopodobnie je przetwarza i zapisuje wyniki do innego tematu. Chcemy mieć pewność, że dla każdego przetwarzanego komunikatu wyniki będą zapisywane dokładnie raz. Cóż mogłoby pójść nie tak?

Okazuje się, że całkiem sporo. Przyjrzyjmy się dwóm scenariuszom.

Ponowne przetwarzanie spowodowane awariami aplikacji

Po skonsumowaniu komunikatu z klastra źródłowego i przetworzeniu go aplikacja musi wykonać dwie rzeczy: wyprodukować wynik do tematu wyjściowego i zatwierdzić przesunięcie skonsumowanego komunikatu. Przypuśćmy, że te dwa oddzielne działania odbywają się we wskazanej właśnie kolejności. Co się stanie, jeśli aplikacja ulegnie awarii po wyprodukowaniu danych wyjściowych, ale przed zatwierdzeniem przesunięcia danych wejściowych?

W rozdziale 4. wyjaśniliśmy, co się dzieje, gdy awarii ulega konsument. Po kilku sekundach brak sygnału heartbeat spowoduje uruchomienie równoważenia obciążenia, a partycje, które przetwarzał dany konsument, zostaną przypisane do innego konsumenta. Nowy konsument rozpocznie przetwarzanie tych partycji od ostatniego zatwierdzonego przesunięcia. Oznacza to, że wszystkie rekordy, które zostały przetworzone przez aplikację między ostatnim zatwierdzonym przesunięciem a awarią, zostaną przetworzone ponownie, a wyniki zostaną jeszcze raz zapisane do tematu wyjściowego, tworząc duplikaty.

Ponowne przetwarzanie spowodowane przez aplikacje zombie

Co się stanie, jeśli nasza aplikacja skonsumuje partię rekordów z Kafki, a następnie zawiesi się lub utraci łączność z Kafką przed wykonaniem jakiegokolwiek innej czynności z tą partią rekordów?

Podobnie jak w poprzednim scenariuszu, po kilku brakujących sygnałach heartbeat aplikacja zostanie uznana za nieaktywną, a jej partycje zostaną przypisane ponownie innemu konsumentowi z danej grupy konsumentów. Konsument ten ponownie odczyta partię rekordów, przetworzy ją, wyprodukuje wyniki do tematu wyjściowego i będzie kontynuował normalną pracę.

Tymczasem pierwsza instancja aplikacji — ta, która się zawiesiła — może wznowić swoją aktywność: przetworzyć partię rekordów, którą ostatnio skonsumowała, i wyprodukować wyniki do tematu wyjściowego. Może zrobić to wszystko, zanim odpyta Kafkę o rekordy lub wysle sygnał heartbeat i odkryje, że powinna być nieaktywna, a właścicielem tych partycji jest teraz inna instancja.

Konsument, który powinien być nieaktywny (martwy), ale o tym nie wie, nazywa się zombie. W tym scenariuszu widzimy, że bez dodatkowych gwarancji zombie mogą produkować dane do tematu wyjściowego i duplikować wyniki.

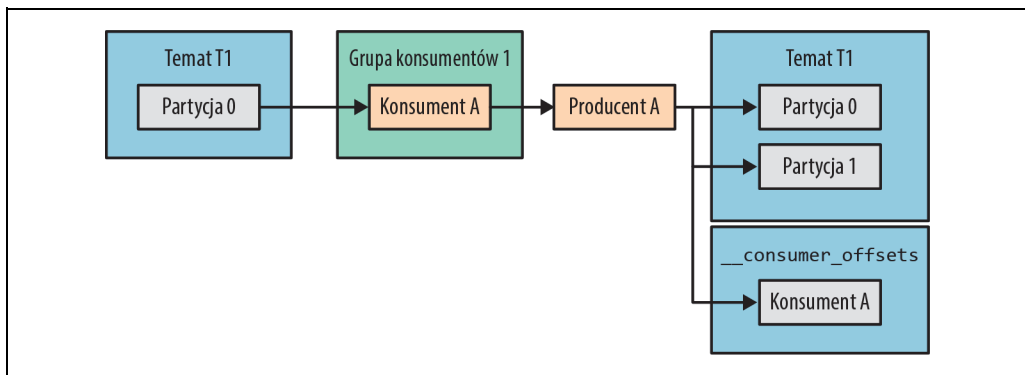
W jaki sposób transakcje gwarantują semantykę „dokładnie raz”?

Wróćmy do przykładu naszej prostej aplikacji przetwarzania strumieniowego. Odczytuje ona dane z jednego tematu, przetwarza je i zapisuje wynik do innego tematu. Przetwarzanie dokładnie raz oznacza, że konsumowanie, przetwarzanie i produkowanie odbywają się w sposób **niepodzielny** (ang. *atomically*). Albo przesunięcie oryginalnego komunikatu zostanie zatwierdzone, a wynik wyprodukowany z powodzeniem, albo nie wydarzy się żadna z tych rzeczy. Musimy się upewnić, że nie będą mogły zostać wyprodukowane częściowe wyniki, np. gdy przesunięcie zostanie zatwierdzone, ale wynik nie zostanie wyprodukowany, lub odwrotnie.

Aby obsłużyć to zachowanie, transakcje Kafki wprowadzają ideę **niepodzielnych zapisów wielopartycyjnych** (ang. *atomic multipartition writes*). Pomysł polega na tym, że zarówno zatwierdzanie

przesunięć, jak i produkowanie rezultatów wymaga zapisywania komunikatów w partycji. Wyniki są zapisywane do tematu wyjściowego, a przesunięcia do tematu `__consumer_offsets`. Jeżeli będziemy mogli otworzyć transakcję, zapisać oba komunikaty i zatwierdzić, jeśli oba zostały zapisane pomyślnie, lub w przeciwnym razie przerwać, aby spróbować ponownie, otrzymamy semantykę „dokładnie raz”, której szukamy.

Na rysunku 8.1 przedstawiliśmy prostą aplikację przetwarzania strumieniowego przeprowadzającą niepodzielny zapis wielopartycyjny na dwóch partycjach i zatwierdzającą jednocześnie przesunięcia dla skonsumowanych zdarzeń.



Rysunek 8.1. Producent transakcyjny z niepodzielnym zapisem wielopartycyjnym

Aby korzystać z transakcji i wykonywać niepodzielne zapisy wielopartycyjne, używamy **producenta transakcyjnego** (ang. *transactional producer*). Producent transakcyjny jest normalnym producentem Kafki skonfigurowanym z ustawieniem `transactional.id` (identyfikator transakcyjny) i zainicjowanym za pomocą metody `initTransactions()`. Inaczej niż ustawienie `producer.id`, które jest generowane automatycznie przez brokery Kafki, `transactional.id` jest częścią konfiguracji producenta i powinno być utrzymywane między restartami. W rzeczywistości główną rolę `transactional.id` jest identyfikacja tego samego producenta po restarcie. Brokery Kafki utrzymują mapowanie `transactional.id` na `producer.id`, dlatego jeśli metoda `initTransactions()` zostanie wywołana ponownie z istniejącym `transactional.id`, producentowi zostanie również przypisany ten sam identyfikator `producer.id` zamiast nowego losowego.

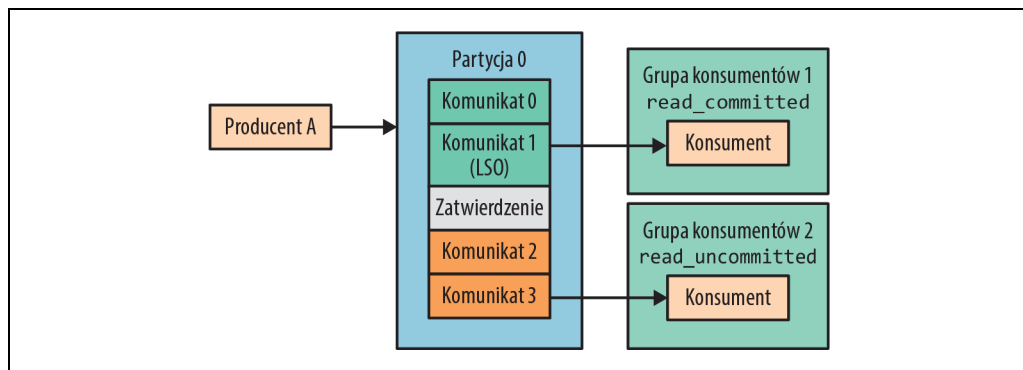
Zapobieganie tworzeniu duplikatów przez instancje zombie aplikacji wymaga mechanizmu **odgradzania zombie** (ang. *zombie fencing*), czyli uniemożliwiania zapisywania wyników do strumienia wyjściowego przez instancje zombie aplikacji. Tu korzystamy ze standardowego sposobu odgradzania zombie, używając epoki. Kafka inkrementuje numer epoki powiązany z `transactional.id`, gdy wywoływana jest metoda `initTransaction()` w celu zainicjowania producenta transakcyjnego. Wysyłanie, zatwierdzanie i przerywanie żądań przez producenty o tym samym identyfikatorze transakcyjnym, ale w mniejszych numerach epok, będzie odrzucane z błędem `FencedProducer`. Starszy producent nie będzie mógł zapisywać do strumienia wyjściowego i zostanie zmuszony do wywołania `close()`, uniemożliwiając zombie wprowadzanie duplikatów rekordów. W Apache Kafce w wersji co najmniej 2.5 istnieje możliwość dodawania do metadanych transakcji metadanych grupy konsumentów. Metadane te również będą wykorzystywane do odgradzania, co pozwoli producentom

o różnych identyfikatorach transakcyjnych na zapisywanie na tych samych partycjach przy jednoczesnym chronieniu się przed instancjami zombie.

W przeważającej części transakcje są funkcjonalnością producenta — tworzymy producent transakcyjny, rozpoczynamy transakcję, zapisujemy rekordy do wielu partycji, produkujemy przesunięcia, aby oznaczyć rekordy jako już przetworzone, oraz zatwierdzamy lub przerywamy transakcję. Robimy to przy użyciu producenta. To jednak nie wszystko — rekordy zapisywane transakcyjnie, nawet te, które są częścią ostatecznie przerwanych transakcji, są zapisywane na partycji, tak jak wszystkie inne rekordy. Konsumenty muszą być skonfigurowane z odpowiednimi gwarancjami izolacji, w przeciwnym razie nie będziemy mieli oczekiwanej gwarancji „dokładnie raz”.

Konsumpcję komunikatów zapisywanych transakcyjnie kontrolujemy przez ustawienie konfiguracji `isolation.level`. Jeśli ma ona wartość `read_committed`, to wywołanie `consumer.poll()` po zasubskrybowaniu zbioru tematów zwróci komunikaty, które były częścią pomyślnie zatwierdzonej transakcji lub zostały zapisane nietransakcyjnie; nie zwróci komunikatów stanowiących część przerwanej lub nadal otwartej transakcji. Domyślna wartość `isolation.level`, czyli `read_uncommitted`, zwróci wszystkie rekordy, w tym te należące do otwartych lub przerwanych transakcji. Skonfigurowanie trybu `read_committed` nie gwarantuje, że aplikacja otrzyma wszystkie komunikaty, które są częścią określonej transakcji. Możliwe jest zasubskrybowanie tylko podzbioru tematów należących do transakcji, a tym samym uzyskanie podzbioru komunikatów. Ponadto aplikacja nie ma informacji, kiedy się rozpoczynają lub kończą transakcje ani które komunikaty są częścią poszczególnych transakcji.

Na rysunku 8.2 pokazaliśmy, które rekordy są widoczne dla konsumenta w trybie `read_committed` i dla konsumenta z domyślnym trybem `read_uncommitted`.



Rysunek 8.2. Konsumenty w trybie `read_committed` będą pozostawać w tyle za konsumentami z konfiguracją domyślną

Aby zagwarantować odczytywanie komunikatów w kolejności, tryb `read_committed` nie będzie zwracał komunikatów, które zostały wyprodukowane po rozpoczęciu pierwszej transakcji, która wciąż pozostaje otwarta, czyli po przekroczeniu punktu ostatniego stabilnego przesunięcia (ang. *Last Stable Offset* — LSO). Komunikaty te zostaną wstrzymane do momentu, gdy transakcja zostanie zatwierdzona lub przerwana przez producent albo zostanie przerwana przez broker po upływie limitu

czasu `transaction.timeout.ms` (domyślnie 15 minut). Utrzymywanie transakcji otwartej przez dłuższy czas wprowadza większe opóźnienie całkowite, wstrzymując konsumenty.

Nasze proste zadanie przetwarzania strumieniowego będzie miało na wyjściu gwarancje „dokładnie raz”, nawet jeśli dane wejściowe zostały zapisane nietransakcyjnie. Niepodzielne produkowanie wielopartycyjne daje gwarancję, że jeśli rekordy wyjściowe zostały zatwierdzone do tematu wyjściowego, przesunięcie rekordów wejściowych dla danego konsumenta również zostało zatwierdzone i w rezultacie rekordy wejściowe nie będą ponownie przetwarzane.

Jakich problemów nie rozwiązują transakcje?

Jak wyjaśniliśmy wcześniej, transakcje dodano do Kafki w celu zapewnienia niepodzielnych zapisów (ale nie odczytów) wielopartycyjnych oraz w celu odgradzenia producentów zombie w aplikacjach przetwarzania strumieniowego. W efekcie oferują one gwarancje „dokładnie raz”, gdy są wykorzystywane w łańcuchach konsumuj-przetwarzaj-produkuj zadań przetwarzania strumieniowego. W innych kontekstach transakcje po prostu nie będą działać albo będą wymagały dodatkowego wysiłku, by osiągnąć pożądane gwarancje.

Dwa główne błędy są następujące: założenie, że gwarancje „dokładnie raz” mają zastosowanie do działań innych niż produkowanie do Kafki, oraz że konsumenty zawsze odczytują całe transakcje i mają informacje o granicach transakcji.

W kolejnych podpunktach przedstawiliśmy kilka scenariuszy, w których transakcje Kafki nie pomagają w uzyskaniu gwarancji „dokładnie raz”.

Efekty uboczne podczas przetwarzania strumieniowego

Powiedzmy, że w naszej aplikacji przetwarzania strumieniowego etap przetwarzania rekordów obejmuje wysyłanie użytkownikom wiadomości e-mailowych. Włączenie w tej aplikacji semantyki „dokładnie raz” nie zagwarantuje, że e-maile będą wysłane tylko raz. Te gwarancje dotyczą wyłączenia rekordów zapisywanych w Kafce. Używanie numerów sekwencyjnych do duplikowania rekordów lub znaczników do przerywania lub anulowania transakcji działa w Kafce, ale nie powoduje „cofnięcia” wysłania e-maila. To samo dotyczy wszelkich działań z efektami zewnętrznymi wykonywanych w aplikacji przetwarzania strumieniowego: wywoływanie REST API, zapisywanie w pliku itp.

Odczyt z tematu Kafki i zapis w bazie danych

W tym przypadku aplikacja zapisuje w zewnętrznej bazie danych, a nie w Kafce. Scenariusz nie uwzględnia producenta — rekordy są zapisywane w bazie danych przy użyciu sterownika bazy danych (np. JDBC), a przesunięcia są zatwierdzane w Kafce w ramach konsumenta. Nie istnieje mechanizm umożliwiający zapisywanie wyników w zewnętrznej bazie danych i zatwierdzanie przesunięć w Kafce w ramach pojedynczej transakcji. Zamiast tego moglibyśmy zarządzać przesunięciami w bazie danych (jak wyjaśniliśmy w rozdziale 4.) i zatwierdzać w niej zarówno dane, jak i przesunięcia w obrębie jednej transakcji — opierałoby się to raczej na gwarancjach transakcyjnych bazy danych niż Kafki.



Mikrouslugi często muszą aktualizować bazę danych *oraz* publikować komunikaty w Kafce w ramach pojedynczej transakcji niepodzielnej, aby wydarzyło się jedno i drugie albo do skutku nie doszło żadne z działań. Jak wyjaśniliśmy w dwóch ostatnich przykładach, transakcje Kafki tego nie robią.

Typowym rozwiązaniem tego powszechnego problemu jest tzw. **wzorzec Skrzynka Nadawcza** (ang. *outbox*). Mikrousluga publikuje jedynie komunikaty do tematu Kafki („skrzynka nadawcza”), a osobna usługa przekazywania komunikatów odczytuje zdarzenia z Kafki i aktualizuje bazę danych. Jak się właśnie przekonałeś, Kafka nie gwarantuje aktualizacji bazy danych „dokładnie raz”, dlatego należy mieć pewność, że aktualizacja będzie idempotentna.

Korzystanie z tego wzorca gwarantuje, że komunikat dotrze ostatecznie do Kafki, konsumentów tematu i bazy danych — lub do żadnego z tych komponentów.

Używany jest również wzorzec odwrotny, gdzie tablica bazy danych służy jako skrzynka nadawcza, a usługa przekazywania zapewnia, że aktualizacje tablicy dotrą także do Kafki jako komunikaty. Ten wzorzec jest preferowany, gdy użyteczne są wbudowane ograniczenia RDBMS, takie jak unikatowość i klucze obce. Lider projektu Debezium opublikował na blogu obszerny post na temat wzorca Skrzynka Nadawcza (<https://oreil.ly/PB3Vb>) ze szczegółowymi przykładami.

Odczyt danych z bazy danych, zapis w Kafce, a następnie zapis z Kafki do kolejnej bazy danych

Bardzo kuszące jest przekonanie, że możemy zbudować aplikację, która będzie odczytywała dane z bazy danych, identyfikowała transakcje bazodanowe, zapisywała rekordy w Kafce, a stamtąd zapisywała je do kolejnej bazy danych, zachowując jeszcze pierwotne transakcje ze źródłowej bazy danych.

Niestety transakcje Kafki nie mają funkcjonalności niezbędnych do obsługi tego rodzaju gwarancji kompleksowych typu „od początku do końca” (ang. *end-to-end*). Oprócz problemu z zatwierdzeniem zarówno rekordów, jak i przesunięć w ramach tej samej transakcji, istnieje jeszcze jedna trudność: gwarancje `read_committed` w konsumentach Kafki są zbyt słabe, aby zachować transakcje bazodanowe. Tak, konsument nie zobaczy rekordów, które nie zostały zatwierdzone. Nie ma jednak gwarancji, że zobaczy wszystkie rekordy, które zostały zatwierdzone w ramach danej transakcji, ponieważ w niektórych tematach może być opóźnienie; konsument nie ma informacji pozwalających zidentyfikować granice transakcji, stąd nie może wiedzieć, kiedy transakcja się rozpoczęła i zakończyła oraz czy zobaczył wszystkie swoje rekordy, ich część lub nie zobaczył żadnego.

Kopiowanie danych z jednego klastra Kafki do drugiego

Ten przypadek jest bardziej subtelny — podczas kopiowania danych z jednego klastra Kafki do drugiego można obsługiwać gwarancje „dokładnie raz”. Opis wykonania znajdziesz w dokumencie KIP-656 dotyczącym dodawania funkcjonalności „dokładnie raz” w narzędziu MirrorMaker 2.0 (<https://oreil.ly/EoM6w>). Gdy piszemy ten rozdział, propozycja jest nadal w wersji roboczej, ale sam algorytm został jasno opisany. Ta propozycja zawiera gwarancję, że każdy rekord w klastrze źródłowym zostanie skopiowany do klastra docelowego dokładnie raz.

Nie gwarantuje to jednak, że transakcje będą miały charakter niepodzielny. Jeżeli aplikacja wyprodukuje transakcyjnie kilka rekordów i przesunięć, a następnie MirrorMaker 2.0 skopiuje je do

innego klastra Kafki, wówczas właściwości i gwarancje transakcyjne zostaną utracone podczas procesu kopiowania. Z tego samego powodu są one tracone podczas kopiowania danych z Kafki do relacyjnej bazy danych: konsument odczytujący dane z Kafki nie może wiedzieć ani zagwarantować, że otrzyma wszystkie zdarzenia z transakcji. Może replikować np. część transakcji, jeśli subskrybuje tylko podzbiór tematów.

Wzorzec Publikuj-Subskrybuj

Oto nieco bardziej finezyjny scenariusz. Gwarancje „dokładnie raz” omówiliśmy w kontekście wzorca Konsumuj-Przetwarzaj-Produkuj, ale wzorzec Publikuj-Subskrybuj jest bardzo częstym przypadkiem użycia. Korzystanie z transakcji w przypadku użycia pub-sub daje pewne gwarancje: konsumenty skonfigurowane w trybie `read_committed` nie zobaczą rekordów opublikowanych w ramach transakcji, która została przerwana. Gwarancje te nie spełniają jednak warunku „dokładnie raz”. Konsumenty mogą przetwarzać komunikat więcej niż raz, w zależności od własnej logiki zatwierdzenia przesunięć.

Gwarancje udzielane przez Kafkę w tym przypadku są podobne do gwarancji zapewnianych przez transakcje JMS, ale zagwarantowanie, że niezatwierdzone transakcje pozostaną niewidoczne, zależy od konsumentów w trybie `read_committed`. Brokery JMS wstrzymują niezatwierdzone transakcje przed wszystkimi konsumentami.



Ważne jest unikanie wzorca, który polega na publikowaniu komunikatu, a następnie przed zatwierdzeniem transakcji oczekiwaniu na odpowiedź innej aplikacji. Ta druga aplikacja otrzyma komunikat dopiero po zatwierdzeniu transakcji, co skutkuje zakleszczeniem.

Jak korzystać z transakcji?

Transakcje są funkcjonalnością brokerską i częścią protokołu Kafki, istnieje więc wiele klientów, które obsługują transakcje.

Najczęstszym i najbardziej zalecanym sposobem korzystania z transakcji jest włączenie gwarancji „dokładnie raz” w Kafka Streams. W ten sposób nie będziemy używać transakcji bezpośrednio, ale za kulisami Kafka Streams zastosuje je dla nas, by zapewnić nam pożądane gwarancje. Transakcje zostały zaprojektowane właśnie z myślą o tym przypadku użycia, stąd korzystanie z nich za pośrednictwem biblioteki Kafka Streams jest najłatwiejsze i daje największe prawdopodobieństwo, że wszystko będzie działać zgodnie z oczekiwaniami.

Aby włączyć gwarancje „dokładnie raz” dla aplikacji Kafka Streams, wystarczy dla parametru `processing.guarantee` ustawić wartość `exactly_once` lub `exactly_once_beta`. To wszystko.

A jeśli chcielibyśmy gwarancji „dokładnie raz” bez korzystania z Kafka Streams? W takim przypadku trzeba skorzystać bezpośrednio z transakcyjnych interfejsów API. W poniższym listingu pokazaliśmy, jak to będzie działać. Pełny przykład znajdziesz w repozytorium GitHuba Apache Kafki, który zawiera sterownik demo (<https://oreil.ly/45dE4>) i prosty procesor „dokładnie raz” (<https://oreil.ly/CrXHU>) działające w osobnych wątkach:



Ustawienie `exactly_once_beta` to nieco inna metoda obsługi instancji aplikacji, które ulegają awarii lub zawieszają się przy transakcjach w toku. Zostało to wprowadzone dla brokerów Kafki w wersji 2.5, a w wersji 2.6 dla biblioteki Kafka Streams. Główną zaletą tej metody jest możliwość obsługi wielu partycji za pomocą jednego producenta transakcyjnego, a tym samym tworzenie bardziej skalowalnych aplikacji Kafka Streams. Więcej informacji na temat tych zmian znajdziesz w propozycji KIP-447 (<https://oreil.ly/O3dSA>), gdzie zostały omówione po raz pierwszy.

```
Properties producerProps = new Properties();
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
producerProps.put(ProducerConfig.CLIENT_ID_CONFIG, "DemoProducer");
producerProps.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionalId); ❶

producer = new KafkaProducer<>(producerProps);

Properties consumerProps = new Properties();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); ❷
consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed"); ❸

consumer = new KafkaConsumer<>(consumerProps);

producer.initTransactions(); ❹

consumer.subscribe(Collections.singleton(inputTopic)); ❺

while (true) {
    try {
        ConsumerRecords<Integer, String> records = consumer.poll(Duration.ofMillis(200));
        if (records.count() > 0) {
            producer.beginTransaction(); ❻
            for (ConsumerRecord<Integer, String> record : records) {
                ProducerRecord<Integer, String> customizedRecord = transform(record); ❼
                producer.send(customizedRecord);
            }
            Map<TopicPartition, OffsetAndMetadata> offsets = consumerOffsets();
            producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata()); ❽
            producer.commitTransaction(); ❾
        }
    } catch (ProducerFencedException|InvalidProducerEpochException e) { ❿
        throw new KafkaException(String.format(
            "To transactional.id %s jest używane przez inny proces", transactionalId));
    } catch (KafkaException e) {
        producer.abortTransaction(); ⓫
        resetToLastCommittedPositions(consumer);
    }
}
```

- ❶ Skonfigurowanie producenta z ustawieniem `transactional.id` sprawia, że jest on producentem transakcyjnym zdolnym do produkowania niepodzielnych zapisów wielopartycyjnych. Identyfikator transakcyjny musi być unikatowy i długotrwały. Zasadniczo definiuje on instancję aplikacji.
- ❷ Konsumenty, które są częścią transakcji, nie zatwierdzają własnych przesunięć — producent zapisuje przesunięcia w ramach transakcji. Dlatego zatwierdzanie przesunięć powinno być wyłączone.

- 3 W tym przykładzie konsument odczytuje z tematu wejściowego. Zakładamy, że rekordy w temacie wejściowym również zostały zapisane przez producenta transakcyjnego (tylko dla zabawy — nie ma takiego wymogu dla danych wejściowych). Aby czysto odczytywać transakcje (tj. ignorować transakcje w toku i przerwane), ustawiamy poziom izolacji konsumentów na `read_committed`. Zwróć uwagę, że konsument oprócz odczytywania zatwierdzonych transakcji nadal będzie odczytywał zapisy nietransakcyjne.
- 4 Producent transakcyjny musi najpierw zostać zainicjowany. Powoduje to zarejestrowanie identyfikatora transakcyjnego, zwiększenie wartości epoki, aby zagwarantować, że inne producenci o tym samym identyfikatorze będą uważane za zombie, oraz przerwanie starszych transakcji w toku z tym samym identyfikatorem transakcyjnym.
- 5 Korzystamy z konsumentckiego API `subscribe`, co oznacza, że partycje przypisane do tej instancji aplikacji mogą ulec zmianie w dowolnym momencie w wyniku równoważenia obciążenia. Przed wydaniem wersji 2.5, w której wprowadzono zmiany API z propozycji KIP-447, było to znacznie trudniejsze. Producenci transakcyjne musiały mieć statycznie przypisany zbiór partycji, ponieważ mechanizm odgradzania transakcji opierał się na tym samym identyfikatorze transakcyjnym, który był używany dla tych partycji (w przypadku zmiany identyfikatora transakcyjnego nie istniała ochrona w postaci odgradzania zombie). Propozycja KIP-447 dodała używane w tym przykładzie nowe interfejsy API, które dołączają do transakcji informacje o grupie konsumentów, a informacje te są używane do odgradzania. Gdy korzysta się z tej metody, sensowne jest również zatwierdzanie transakcji za każdym razem, gdy anulowane są przypisania powiązanych partycji.
- 6 Skonsumowaliśmy rekordy, a teraz chcemy je przetworzyć i wyprodukować wyniki. Ta metoda gwarantuje, że wszystko, co zostało wyprodukowane od momentu jej wywołania do zatwierdzenia lub przerwania transakcji, będzie częścią pojedynczej transakcji niepodzielnej.
- 7 Przetwarzamy rekordy — tu znajduje się cała nasza logika biznesowa.
- 8 Jak wyjaśniliśmy wcześniej w tym rozdziale, ważne jest zatwierdzanie przesunięć w ramach transakcji. Gwarantuje to, że jeśli nie uda nam się wyprodukować wyników, to nie zatwierdzimy przesunięć dla rekordów, które w rzeczywistości nie zostały przetworzone. Metoda ta powoduje zatwierdzanie przesunięć w ramach transakcji. Należy pamiętać, by nie zatwierdzać przesunięć w żaden inny sposób — wyłączamy automatyczne zatwierdzanie przesunięć i nie wywołujemy żadnych konsumentckich interfejsów API zatwierdzania. Zatwierdzanie przesunięć przez jakąkolwiek inną metodę nie daje gwarancji transakcyjnych.
- 9 Wyprodukowaliśmy wszystko, czego potrzebowaliśmy, zatwierdziliśmy przesunięcia w ramach transakcji i nadszedł czas, by zatwierdzić transakcję i dobić targu. Gdy ta metoda zostanie pomyślnie zakończona, cała transakcja zostanie zrealizowana i będziemy mogli kontynuować odczytywanie i przetwarzanie następnej partii zdarzeń.
- 10 Jeżeli otrzymamy ten wyjątek, oznacza, że jesteśmy zombie. W jakiś sposób nasza aplikacja się zawiesiła lub rozłączyła i uruchomiona jest nowsza instancja aplikacji z naszym identyfikatorem transakcyjnym. Najprawdopodobniej transakcja, którą rozpoczęliśmy, została przerwana i ktoś inny przetwarza te rekordy. Możemy już tylko z gracją zejść ze sceny.

- 11 Jeśli podczas zapisywania transakcji wystąpił błąd, możemy przerwać transakcję, przywrócić pozycję konsumenta i spróbować ponownie.

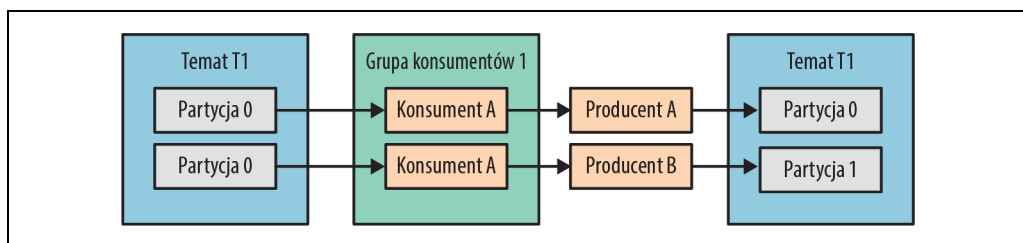
Identyfikatory transakcyjne i odgradzanie

Wybór identyfikatora transakcyjnego dla producentów jest ważny i nieco trudniejszy niż się wydaje. Niepoprawne przypisanie identyfikatora transakcyjnego może prowadzić do błędów aplikacji lub utraty gwarancji „dokładnie raz”. Kluczowe wymagania są takie, że identyfikator transakcyjny musi być spójny dla tej samej instancji aplikacji między restartami i różny dla poszczególnych instancji aplikacji — w przeciwnym razie brokery nie będą w stanie odgradzić instancji zombie.

Do wydania 2.5 jedynym sposobem zapewnienia odgradzenia było statyczne mapowanie identyfikatora transakcyjnego na partycje. Gwarantowało to konsumowanie poszczególnych partycji zawsze za pomocą tego samego identyfikatora transakcyjnego. Jeśli producent o identyfikatorze transakcyjnym A przetworzył komunikaty z tematu T i utracił łączność, a zastępujący go nowy producent otrzymał identyfikator transakcyjny B i później producent A wrócił jako zombie, wtedy zombie A nie był odgradzany, ponieważ jego identyfikator nie był zgodny z identyfikatorem nowego producenta B. Chcemy, aby producent A był zawsze zastępowany przez producent A, a nowy producent A miał wyższy numer epoki i zombie A był odpowiednio odgradzany. W tych wydaniach poprzedni przykład nie działałby prawidłowo — identyfikatory transakcyjne są przypisywane do wątków losowo bez upewniania się, że ten sam identyfikator transakcji jest zawsze używany do zapisywania na tej samej partycji.

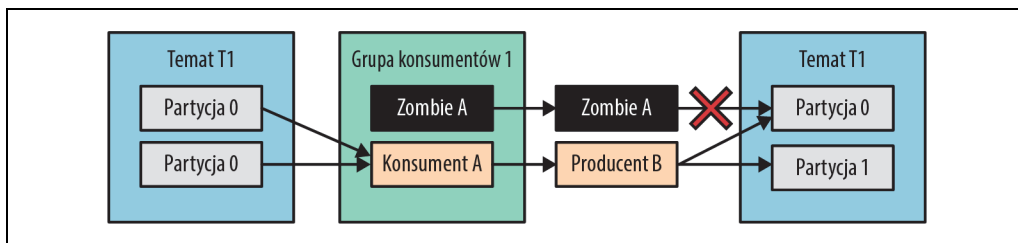
W wersji Apache Kafka 2.5 propozycja KIP-447 wprowadziła drugą metodę odgradzania, która oprócz identyfikatorów transakcyjnych wykorzystuje metadane grupy konsumentów. Używamy producenckiej metody zatwierdzania przesunięć i jako argument przekazujemy metadane grupy konsumentów, a nie tylko identyfikator tej grupy.

Załóżmy, że mamy temat T1 z dwiema partycjami: 0 i 1. Każda z nich jest przetwarzana przez osobny konsument z tej samej grupy; konsumenci przekazują rekordy odpowiadającym im producentowi transakcyjnemu — jednemu o identyfikatorze transakcyjnym A, drugiemu o identyfikatorze transakcyjnym B. Producenci zapisują dane wyjściowe odpowiednio do partycji 0 i 1 tematu T2. Ten scenariusz przedstawiliśmy na rysunku 8.3.



Rysunek 8.3. Transakcyjny procesor rekordów

Jak pokazaliśmy na rysunku 8.4, jeżeli instancja aplikacji z konsumentem A i producentem A stanie się zombie, konsument B rozpocznie przetwarzanie rekordów z obu partycji. Jeśli chcemy zagwarantować, żeby żaden zombie nie zapisywał na partycji 0, to konsument B nie może po prostu rozpocząć



Rysunek 8.4. Transakcyjny procesor rekordów po równoważeniu obciążenia

odczytu z partycji 0 i zapisu na partycji 0 z identyfikatorem transakcyjnym B. Aplikacja będzie musiała utworzyć instancję nowego producenta z identyfikatorem transakcyjnym A w celu bezpiecznego zapisywania na partycji 0 i odgrodzenia starego identyfikatora transakcyjnego A. Jest to marnotrawstwo. Zamiast tego w transakcjach uwzględniamy informacje o grupie konsumentów. Transakcje od producenta B pokażą, że pochodzą z nowszego pokolenia grupy konsumentów, więc zostaną przepuszczone, natomiast transakcje od producenta zombie A pokażą stare pokolenie grupy konsumentkiej i zostaną odgrodzone.

Jak działają transakcje?

Z transakcji możemy korzystać, wywołując interfejsy API i nie rozumiejąc nawet ich działania. Jednak posiadanie jakiegoś wyobrażenia o tym, co się dzieje za kulisami, pomoże nam rozwiązywać problemy z aplikacjami, które nie zachowują się zgodnie z oczekiwaniami.

Podstawowy algorytm transakcji w Kafce został zainspirowany migawkami Chandy-Lamport, w których komunikaty kontrolne znacznika są wysyłane do kanałów komunikacyjnych, a spójny stan jest określany na podstawie dotarcia znacznika. Transakcje Kafki wykorzystują komunikaty znaczników, aby wskazać, że dla wielu partycji transakcje zostały zatwierdzone lub przerwane — gdy producent zdecyduje się na zatwierdzenie transakcji, wysyła komunikat „zatwierdź” do koordynatora transakcji, który następnie zapisuje znaczniki zatwierdzenia na wszystkich partycjach zaangażowanych w transakcję. Co się jednak stanie, jeśli producent ulegnie awarii po zapisaniu komunikatów zatwierdzenia tylko w podzbiorze partycji? Transakcje Kafki rozwiązują ten problem za pomocą dwufazowego zatwierdzenia i dziennika transakcji. Generalnie algorytm działa w następujący sposób:

1. Zarejestrowanie istnienia trwającej transakcji oraz powiązanych z nią partycji.
2. Zarejestrowanie zamiaru zatwierdzenia lub przerwania — po tym zarejestrowaniu jesteśmy skazani na zatwierdzenie lub przerwanie.
3. Zapisanie wszystkich znaczników transakcji na wszystkich partycjach.
4. Zarejestrowanie zakończenia transakcji.

Aby zaimplementować ten podstawowy algorytm, Kafka potrzebuje dziennika transakcji. W tym celu używamy wewnętrznego tematu o nazwie `__transaction_state`.

Zobaczmy, jak ten algorytm działa w praktyce, i przeanalizujemy wewnętrzne mechanizmy działania wywołań transakcyjnych interfejsów API, których użyliśmy w poprzednim fragmencie kodu.

Zanim rozpoczniemy pierwszą transakcję, producenci muszą się zarejestrować jako transakcyjne, wywołując metodę `initTransaction()`. Żądanie to jest wysyłane do brokera, który będzie **koordynatorem transakcji** dla danego producenta transakcyjnego. Każdy broker jest koordynatorem transakcji dla podzbioru producentów, jak również koordynatorem grupy konsumentów dla podzbioru grup konsumentów. Koordynator transakcji dla każdego identyfikatora transakcyjnego jest liderem partycji dziennika transakcji, na który zmapowany jest dany identyfikator transakcyjny.

Interfejs API `initTransaction()` rejestruje nowy identyfikator transakcyjny u koordynatora lub zwiększa numer epoki istniejącego identyfikatora transakcyjnego, aby odgrodzić poprzednie producenci, które mogły stać się zombie. Gdy epoka jest inkrementowana, oczekujące transakcje zostają przerwane.

Kolejnym krokiem dla producenta jest wywołanie `beginTransaction()`. Wywołanie tego API nie jest częścią protokołu — producent zostaje jedynie poinformowany, że transakcja jest w toku. Koordynator transakcji po stronie brokera nadal nie wie, że transakcja się rozpoczęła. Jednak gdy producent zacznie wysyłać rekordy, przy każdym wykryciu, że wysyła rekordy do nowej partycji, wysyła również do brokera żądanie `AddPartitionsToTxnRequest`, informując go, że dla tego producenta trwa transakcja i jej częścią stały się dodatkowe partycje. Informacje te są zapisywane w dzienniku transakcji.

Kiedy skończymy produkować wyniki i jesteśmy gotowi do zatwierdzenia, zaczynamy od zatwierdzenia rekordów, które przetworzyliśmy w danej transakcji. Przesunięcia można zatwierdzić w dowolnym momencie, ale należy to zrobić przed zatwierdzeniem transakcji. Wywołanie `sendOffsetsToTransaction()` spowoduje wysłanie do koordynatora transakcji żądania, które zawiera przesunięcia oraz identyfikator grupy konsumentów. Koordynator transakcji użyje identyfikatora grupy konsumentów, aby znaleźć koordynator grupy i zatwierdzić przesunięcia, tak jak zwykle zrobiłaby to grupa konsumentów.

Teraz nadszedł czas, aby zatwierdzić lub przerwać. Wywołanie `commitTransaction()` lub `abortTransaction()` spowoduje wysłanie do koordynatora transakcji żądania `EndTransactionRequest`. Koordynator transakcji rejestruje w dzienniku transakcji zamiar zatwierdzenia lub przerwania. Po pomyślnym zakończeniu tego kroku za zakończenie procesu zatwierdzania (lub przerwania) odpowiedzialny jest koordynator transakcji. Zapisuje znacznik zatwierdzenia na wszystkich partycjach zaangażowanych w transakcję, a następnie zapisuje w dzienniku transakcji, że zatwierdzenie zakończyło się pomyślnie. Zwróć uwagę, że jeśli po zarejestrowaniu zamiaru zatwierdzenia i przed zakończeniem procesu koordynator transakcji zostanie zamknięty lub ulegnie awarii, wybrany zostanie nowy koordynator transakcji, który pobierze zamiar zatwierdzenia z dziennika transakcji i zakończy proces.

Jeśli transakcja nie zostanie zatwierdzona lub przerwana w czasie określonym przez `transaction.timeout.ms`, koordynator transakcji automatycznie ją przerwie.



Każdy broker, który otrzymuje rekordy od producentów transakcyjnych lub idempotentnych, przechowuje w pamięci identyfikatory producenta lub transakcji wraz ze stanem związanym z każdą z ostatnich pięciu partii wysłanych przez producent: numery sekwencyjne, przesunięcia itd. Ten stan jest przechowywany przez określony w milisekundach czas `transactional.id.expiration.ms` (domyślnie siedem dni) po tym, jak producent

przestanie być aktywny. Pozwala to producentowi na wznowienie działalności bez napotykania błędów UNKNOWN_PRODUCER_ID. Tworzenie nowych idempotentnych producentów lub nowych identyfikatorów transakcyjnych w bardzo szybkim tempie i nigdy niewykonywanie ich ponownie może wywołać w brokerze coś zbliżonego do wycieku pamięci. Trzy nowe producenci idempotentne na sekundę nagromadzone w ciągu tygodnia dają 1,8 miliona wpisów stanu producenta z przechowywanymi łącznie metadanymi 9 milionów partii, wykorzystując ok. 5 GB pamięci RAM. Może to spowodować w brokerze problemy z wyczerpywaniem się pamięci lub częstymi uruchomieniami mechanizmu odzyskiwania pamięci. Zalecamy projektowanie aplikacji w taki sposób, aby po uruchomieniu inicjowała kilka długotrwałych producentów, które będą potem wykorzystywane wielokrotnie przez cały okres życia aplikacji. Jeśli nie jest to możliwe (utrudnia to np. FaaS, czyli funkcja jako usługa), zalecamy skrócenie limitu czasu `transactional.id.expiration.ms`, dzięki czemu identyfikatory będą wygasły szybciej, a zatem stary stan, który nigdy nie zostanie ponownie użyty, nie zajmie znacznej części pamięci brokera.

Wydajność transakcji

Transakcje generują dla producenta umiarkowany narzut. Żądanie zarejestrowania identyfikatora transakcyjnego pojawia się raz w cyklu życia producenta. Dodatkowe wywołania w celu zarejestrowania partycji w ramach transakcji mają miejsce co najwyżej raz na partycję dla każdej transakcji, a następnie każda transakcja wysyła żądanie zatwierdzenia, co powoduje zapisanie dodatkowego znacznika zatwierdzenia na każdej partycji. Żądania inicjowania transakcyjnego i zatwierdzania transakcji są synchroniczne, więc żadne dane nie zostaną wysłane, dopóki żądania nie zostaną pomyślnie ukończone, nie zakończą się niepowodzeniem lub nie upłynie limit czasu, co jeszcze zwiększa narzut.

Należy zauważyć, że narzut generowany dla producenta przez transakcje jest niezależny od liczby komunikatów w transakcji. Stąd większa liczba komunikatów na transakcję zmniejsza zarówno względny narzut, jak i liczbę synchronicznych zatrzymań, co zwiększa ogólną przepustowość.

Po stronie konsumenta istnieje pewien narzut związany z odczytywaniem znaczników zatwierdzania. Kluczowy wpływ transakcji na wydajność konsumenta ma fakt, że konsumenci w trybie `read_committed` nie będą zwracać rekordów, które są częścią otwartej transakcji. Długie interwały między zatwierdzeniami transakcji oznaczają, że przed zwróceniem komunikatów konsument będzie musiał czekać dłużej, co w rezultacie zwiększy całkowite opóźnienie.

Należy jednak pamiętać, że konsument nie musi buforować komunikatów, które należą do otwartych transakcji. Broker nie zwróci ich w odpowiedzi na żądania pobrania wysyłane przez konsument. Ponieważ przy odczytywaniu transakcji konsument nie ma do wykonania żadnej dodatkowej pracy, nie ma również spadku przepływności.

Podsumowanie

Semantyka „dokładnie raz” w Kafce jest przeciwieństwem szachów: trudno ją zrozumieć, ale jest łatwa w użyciu.

W tym rozdziale omówiliśmy dwa kluczowe mechanizmy, które zapewniają w Kafce gwarancje „dokładnie raz”. Są to: producent idempotentny, który pozwala uniknąć duplikatów powstających w wyniku działania mechanizmu ponawiania prób, oraz transakcje stanowiące podstawę semantyki „dokładnie raz” w bibliotece Kafka Streams.

Oba te mechanizmy mogą być włączone w jednej konfiguracji i pozwalają nam używać Kafki do aplikacji, które wymagają mniejszej liczby duplikatów i silniejszych gwarancji poprawności.

Omówiliśmy szczegółowo konkretne scenariusze i przypadki użycia, aby pokazać oczekiwane zachowanie, a nawet przyjrzelśmy się niektórym szczegółom implementacji. Te szczegóły są ważne podczas rozwiązywania problemów z aplikacjami lub podczas bezpośredniego korzystania z transakcyjnych interfejsów API.

Rozumiejąc, co gwarantuje semantyka „dokładnie raz” Kafki w poszczególnych przypadkach użycia, możemy zaprojektować aplikacje, które będą używały tej semantyki, gdy będzie to konieczne. Zachowanie aplikacji nie powinno być zaskakujące, a zagadnienia opisane w tym rozdziale pomogą Ci uniknąć niespodzianek.

Budowanie potoków danych

Gdy mowa o budowaniu potoków danych za pomocą Apache Kafki, zwykle chodzi o parę przypadków użycia. Pierwszym jest budowa potoku danych, w którym Apache Kafka jest jednym z dwóch punktów końcowych, np. przesyłanie danych z Kafki do S3 lub przesyłanie danych z MongoDB do Kafki. Drugi przypadek użycia polega na budowie potoku między dwoma różnymi systemami, ale z wykorzystaniem Kafki jako pośrednika. Przykładem może być pobieranie danych z Twittera do Elasticsearcha przez wysyłanie danych najpierw z Twittera do Kafki, a następnie z Kafki do Elasticsearcha.

Po tym, jak zapoznaliśmy się ze sposobem stosowania Kafki w obu przypadkach użycia, zarówno w LinkedInie, jak i innych dużych organizacjach, do platformy Apache Kafka w wersji 0.9 dodaliśmy framework Kafka Connect. Zauważyliśmy, że z integracją Kafki z potokami danych związane są konkretne wyzwania, którym każda organizacja musiała stawić czoła. Postanowiliśmy więc dodać do Kafki interfejsy API rozwiązujące niektóre z tych problemów, aby organizacje nie musiały budować rozwiązań od zera.

Główną wartością, jaką Kafka zapewnia potokom danych, jest jej zdolność do służenia jako bardzo duży, niezawodny bufor między różnymi etapami potoku. Powoduje to skuteczne oddzielenie producentów i konsumentów danych w ramach potoku i umożliwia wykorzystanie tych samych danych ze źródła w wielu docelowych aplikacjach i systemach, przy czym wszystkie te dane mają różne wymagania związane z terminowością i dostępnością. To oddzielenie w połączeniu z niezawodnością, bezpieczeństwem i wydajnością sprawia, że Kafka dobrze się sprawdza w przypadku wielkości potoków danych.



Umieszczanie integracji danych w kontekście

Niektóre organizacje traktują Kafkę jako *punkt końcowy* potoku. Zastanawiają się nad takimi kwestiami jak przesyłanie danych z Kafki do silnika Elastic. Jest to ważne zagadnienie, szczególnie jeśli w Elasticu potrzebne są dane, które znajdują się obecnie w Kafce, dlatego przyjrzymy się dokładnie sposobom rozwiązania tego problemu. Zaczniemy jednak od omówienia użycia Kafki w szerszym kontekście, obejmującym co najmniej dwa punkty końcowe (a być może o wiele więcej), które same nie są Kafką. Jeżeli musisz zmierzyć się z problemem integracji danych, zachęcamy Cię, byś rozważył szerszy obraz, a nie skupiał się tylko na bezpośrednich punktach końcowych. Koncentrowanie się na krótkoterminowych integracjach może skończyć się skomplikowanym i kosztownym w utrzymaniu bałaganem związanym z integracją danych.

W tym rozdziale omówimy kilka typowych kwestii, które należy wziąć pod uwagę przy budowie potoków danych. Wyzwania te nie są charakterystyczne dla Kafki, ale stanowią ogólne problemy związane z integracją danych. Niemniej jednak pokażemy, dlaczego Kafka dobrze nadaje się do integracji danych i jak poradzić sobie z wieloma z tych wyzwań. Opiszemy, w jaki sposób Kafka Connect API różni się od zwykłych klientów produkcyjnych i konsumenckich oraz kiedy należy używać poszczególnych typów klienta. Potem przejdziemy do szczegółów dotyczących frameworku Kafka Connect. Chociaż pełne omówienie Kafka Connect API wykracza poza zakres tego rozdziału, zaprezentujemy przykłady podstawowego użycia i damy Ci wskazówki, gdzie będziesz mógł znaleźć więcej informacji. Na koniec zajmiemy się innymi systemami integracji danych i sposobami ich integracji z Kafką.

Zagadnienia związane z budowaniem potoków danych

Chociaż nie będziemy tutaj omawiać wszystkich szczegółów dotyczących budowania potoków danych, chcielibyśmy wskazać kilka najważniejszych rzeczy, które należy wziąć pod uwagę przy projektowaniu architektur oprogramowania z zamiarem integracji wielu systemów.

Terminowość

W jednych systemach wymagane jest dostarczanie danych w dużych ilościach raz dziennie, podczas gdy w drugich dane muszą być dostarczane w kilka milisekund po ich wygenerowaniu. Większość potoków danych plasuje się gdzieś pomiędzy tymi dwiema skrajnościami. Dobre systemy integracji danych mogą obsługiwać odmienne wymagania dotyczące terminowości dla różnych potoków danych oraz ułatwiać migrację między różnymi harmonogramami wraz ze zmianą wymagań biznesowych. Jako platforma danych strumieniowych ze skalowalną i niezawodną pamięcią masową Kafka może być używana do obsługi wszystkiego: od potoków czasu zbliżonego do rzeczywistego po przetwarzanie wsadowe raz dziennie. Producenci mogą zapisywać w Kafce z dowolną pożądaną częstotliwością, a konsumenci mogą odczytywać i dostarczać najnowsze zdarzenia w miarę ich przybywania albo pracować wsadowo: uruchamiać się np. co godzinę, łączyć się z Kafką i odczytywać zdarzenia, które nagromadziły się w ciągu ostatniej godziny.

W tym kontekście Kafkę można postrzegać jako gigantyczny bufor, który oddziela wymogi terminowości producentów od wymogów konsumentów. Producenci mogą zapisywać zdarzenia w czasie rzeczywistym, natomiast konsumenci mogą przetwarzać partie zdarzeń, lub odwrotnie. Dzięki temu trywialne jest również używanie mechanizmu ciśnienia zwrotnego — sama Kafka stosuje go do producentów (w razie potrzeby opóźniając potwierdzenia), ponieważ prędkość konsumpcji zależy wyłącznie od konsumentów.

Niezawodność

Dążymy do wyeliminowania pojedynczych punktów awarii oraz zapewnienia szybkiego i automatycznego odzyskiwania sprawności po wszelkiego rodzaju awariach. Potoki danych zapewniają często sposób dostarczania danych do systemów o kluczowym znaczeniu dla prowadzonej działalności. Awaria trwająca ponad kilka sekund może być bardzo uciążliwa, zwłaszcza gdy wymóg terminowości

jest bliski kilku milisekundom. Inną ważną kwestią z zakresu niezawodności jest gwarancja dostarczenia danych — niektóre systemy mogą sobie pozwolić na utratę danych, ale w większości przypadków mamy do czynienia z wymogiem dostarczania **co najmniej raz**, co oznacza, że każde zdarzenie z systemu źródłowego musi dotrzeć do miejsca docelowego, ale czasami ponawianie prób będzie powodowało powstawanie duplikatów. Często istnieje nawet wymóg dostarczania **dokładnie raz** — każde zdarzenie z systemu źródłowego musi dotrzeć do miejsca docelowego i niedopuszczalna jest utrata lub duplikacja danych.

Gwarancje dostępności i niezawodności Kafki omówiliśmy szerzej w rozdziale 7. Jak pisaliśmy, Kafka samodzielnie może dostarczać dane co najmniej raz, a dokładnie raz w połączeniu z zewnętrznym magazynem danych, który ma model transakcyjny lub unikatowe klucze. Ponieważ wiele punktów końcowych to magazyny danych zapewniające odpowiednią semantykę dla dostarczania dokładnie raz, potok oparty na Kafce można często zaimplementować właśnie z takimi gwarancjami. Warto podkreślić, że Kafka Connect API zapewnia interfejs do integracji z systemami zewnętrznymi podczas obsługi przesunięć, co ułatwia komponentom połączeniowym (konektorom) zbudowanie potoku typu „dokładnie raz od początku do końca”. Ponadto na licencji *open source* dostępnych jest wiele konektorów, które obsługują dostarczanie dokładnie raz.

Wysoka i zmienna przepustowość

Potoki danych, które budujemy, powinny być w stanie skalować się do bardzo dużej przepustowości, co jest często wymagane w nowoczesnych systemach danych. Co ważniejsze, powinny dostosowywać się, jeżeli przepływność nagle wzrośnie.

Skoro Kafka działa jako bufor między producentami a konsumentami, przepływność konsumenta nie musi być już powiązana z przepływnością producenta. Nie trzeba wdrażać złożonego mechanizmu ciśnienia zwrotnego, ponieważ jeśli przepływność producenta przekroczy przepływność konsumenta, dane będą gromadzone w Kafce, dopóki konsument będzie w stanie nadrobić zaległości. Funkcjonalność skalowania Kafki przez niezależne dodawanie konsumentów lub producentów umożliwia dynamiczne i niezależne skalowanie obu stron potoku, aby sprostać zmieniającym się wymaganiom.

Kafka to system rozproszony o wysokiej przepustowości — zdolny do przetwarzania setek megabajtów na sekundę na nawet skromnych klastrach — nie musimy się więc przejmować, że nasz potok nie będzie się skalował wraz ze wzrostem popytu. Ponadto Kafka Connect API koncentruje się na zrównoległaniu pracy i może to robić zarówno na pojedynczym węźle, jak i przez skalowanie poziome, w zależności od wymagań systemu. W kolejnych punktach opiszemy, w jaki sposób ta platforma pozwala źródłom i ujściom danych dzielić pracę na wiele wątków wykonywania i korzystać z dostępnych zasobów procesora nawet podczas pracy na pojedynczej maszynie.

Kafka obsługuje ponadto kilka rodzajów kompresji, dzięki czemu użytkownicy i administratorzy mogą kontrolować wykorzystanie zasobów sieciowych i pamięci masowej w miarę wzrostu zapotrzebowania na przepustowość.

Format danych

Jedną z najważniejszych kwestii w potoku danych jest uzgadnianie różnych formatów i typów danych. Obsługiwane typy danych różnią się w zależności od bazy danych i innych systemów pamięci masowej. Możesz np. ładować do Kafki pliki XML-a i dane relacyjne przy użyciu Avro w Kafce, a następnie konwertować dane na JSON podczas zapisywania w Elasticsearchu, na Parquet przy zapisywaniu w HDFS-ie i do CSV podczas zapisywania w S3.

Sama Kafka i interfejs Connect API zapewniają całkowitą dowolność, jeśli chodzi o formaty danych. Jak pisaliśmy w poprzednich rozdziałach, producenci i konsumenci mogą używać dowolnego serializatora do reprezentowania danych w dowolnym wygodnym dla Ciebie formacie. Kafka Connect przechowuje w pamięci własne obiekty, w tym typy danych i schematy, ale jak przekonasz się dalej w tym rozdziale, dzięki temu podpinane konwertery mogą przechowywać te rekordy w dowolnym formacie. Oznacza to, że format danych używany dla Kafki nie ogranicza wyboru konektorów.

Wiele źródeł i ujść danych ma schemat. Możemy odczytać ten schemat ze źródła z danymi, zapisać go i użyć do sprawdzenia zgodności lub nawet zaktualizować schemat w bazie danych ujścia. Klasycznym przykładem jest rurociąg danych z MySQL-a do Snowflake'a. Jeśli ktoś doda kolumnę w bazie danych MySQL, doskonały potok sprawi, że ta kolumna zostanie dodana również w bazie danych Snowflake, ponieważ będziemy ładować do niej nowe dane.

Ponadto podczas zapisywania danych z Kafki do systemów zewnętrznych za format, w jakim dane są zapisywane do systemu zewnętrznego, odpowiedzialne są konektory ujść. W niektórych konektorach ten format jest oparty na architekturze wtyczek. Konektor S3 umożliwia np. wybór między formatami Avro i Parquet.

Nie wystarczy jednak obsługiwać różne typy danych. Ogólne frameworki integracji danych powinny zajmować się także różnicami w zachowaniach w zależności od rodzaju źródeł i ujść danych. Syslog jest np. źródłem, które wysyła dane, podczas gdy relacyjne bazy danych wymagają frameworku pobierającego z nich dane. HDFS ma charakter „tylko do dopisywania” i możemy jedynie zapisywać w nim dane, natomiast większość systemów pozwala zarówno dopisywać dane, jak i aktualizować istniejące rekordy.

Transformacje

Transformacje (przekształcenia) są bardziej kontrowersyjne niż inne wymagania. Zasadniczo istnieją dwa podejścia do budowania potoków danych: ETL i ELT. Proces ETL (ang. *Extract-Transform-Load*), czyli wyodrębnianie, przekształcanie i ładowanie, oznacza, że potok danych jest odpowiedzialny za dokonywanie modyfikacji danych w miarę ich przepływu. Zaletą tego podejścia jest oszczędność czasu i pamięci masowej, ponieważ nie trzeba zapisywać danych, modyfikować ich i ponownie zapisywać. W zależności od rodzaju transformacji korzyść ta jest niekiedy rzeczywista, ale czasami przenosi ciężar obliczeń i przechowywania na sam potok danych, co może być pożądaną, ale nie musi. Główną wadą tego podejścia jest to, że transformacje danych w potoku mogą wiązać nam ręce, jeżeli chcemy przetwarzać dane dalej w potoku. Jeśli osoba, która zbudowała potok między bazami danych MongoDB i MySQL, postanowiła filtrować określone zdarzenia lub usuwać pola z rekordów, to wszyscy użytkownicy i aplikacje z dostępem do danych w MySQL-u będą mieć

dostęp tylko do częściowych danych. Jeżeli będą wymagać dostępu do brakujących pól, potok trzeba będzie przebudować, a dane historyczne ponownie przetworzyć (zakładając, że będą dostępne).

Proces ELT (ang. *Extract-Load-Transform*), czyli wyodrębnianie, ładowanie i przekształcanie, oznacza, że potok danych przeprowadza tylko minimalną transformację (głównie związaną z konwersją typów danych) w celu zapewnienia, by dane docierające do celu były jak najbardziej podobne do danych źródłowych. W tych systemach system docelowy gromadzi „surowe dane” i w nim odbywa się całe wymagane przetwarzanie. Zaletą tego podejścia polega na tym, że system zapewnia maksymalną elastyczność użytkownikom systemu docelowego, ponieważ mają oni dostęp do wszystkich danych. W tych systemach łatwiejsze wydaje się również rozwiązywanie problemów, ponieważ całe przetwarzanie danych jest ograniczone do jednego systemu, a nie podzielone między etapami potoku i dodatkowymi aplikacjami. Wadą jest to, że transformacje zużywają zasoby procesora i pamięci masowej w systemie docelowym. W niektórych przypadkach systemy te są stosunkowo drogie i istnieje silna potrzeba przesuwania obliczeń poza te systemy, jeśli to możliwe.

Kafka Connect zawiera funkcjonalność pojedynczej transformacji komunikatów (ang. *Single Message Transformation* — SMT), która przekształca rekordy podczas kopiowania ze źródła do Kafki lub z Kafki do miejsca docelowego. Obejmuje to przekierowywanie komunikatów do różnych tematów, filtrowanie komunikatów, zmianę typów danych, redagowanie określonych pól itd. Bardziej złożone transformacje, które obejmują łączenia i agregacje, są zwykle wykonywane przy użyciu frameworku Kafka Streams i omówimy je szczegółowo w osobnym rozdziale.



Budując system ETL z wykorzystaniem Kafki, należy pamiętać, że pozwala ona budować potoki typu „jeden do wielu”, gdzie dane źródłowe są zapisywane raz do Kafki, a następnie konsumowane przez wiele aplikacji i zapisywane w wielu systemach docelowych. Oczekiwane jest pewne wstępne przetwarzanie i czyszczenie, takie jak standaryzacja znaczników czasu i typów danych, dodawanie informacji o pochodzeniu czy usuwanie danych osobowych — transformacje, które przynoszą korzyści wszystkim konsumentom danych. Nie należy jednak przedwcześnie czyścić i optymalizować danych podczas ich konsumowania, ponieważ w innym miejscu mogą być potrzebne w postaci bardziej surowej.

Bezpieczeństwo

Zawsze należy dbać o bezpieczeństwo. Jeśli chodzi o potoki danych, główne kwestie związane z bezpieczeństwem są zazwyczaj następujące:

- Kto ma dostęp do danych pobieranych do Kafki?
- Czy można zapewnić szyfrowanie danych przechodzących przez potok? Dotyczy to głównie potoków danych, które przekraczają granice centrów danych.
- Kto może dokonywać modyfikacji potoków?
- Czy potok danych może się poprawnie uwierzytelniać, jeśli musi odczytywać dane z lokalizacji z kontrolowanym dostępem lub zapisywać w nich dane?
- Czy dane osobowe (ang. *Personally Identifiable Information* — PII) są przetwarzane zgodnie z przepisami i regulacjami dotyczącymi ich przechowywania, udostępniania i użytkowania?

Kafka umożliwia szyfrowanie danych przesyłanych przez potok ze źródeł danych do Kafki i z Kafki do ujęć danych. Obsługuje również uwierzytelnianie (za pośrednictwem SASL) i autoryzację, dzięki czemu możesz mieć pewność, że jeśli temat zawiera poufne informacje, to nie ma możliwości przesyłania ich potokiem do mniej zabezpieczonych systemów przez osobę nieupoważnioną. Kafka zapewnia również dziennik inspekcji do śledzenia dostępu — nieautoryzowanego i autoryzowanego. Przy odrobinie dodatkowego kodowania możliwe jest również śledzenie, skąd pochodzą zdarzenia w poszczególnych tematach i kto je zmodyfikował, dzięki czemu można podać cały rodowód dla każdego rekordu.

Bezpieczeństwo Kafki zostało omówione szczegółowo w rozdziale 11. Jednak framework Kafka Connect i jego konektory muszą być w stanie łączyć się z zewnętrznymi systemami danych i uwierzytelniać się w nich, a konfiguracja konektorów będzie obejmować poświadczenia do uwierzytelniania w zewnętrжных systemach danych.

Obecnie nie zaleca się przechowywania poświadczeń w plikach konfiguracyjnych, ponieważ oznacza to, że z tymi plikami konfiguracyjnymi trzeba obchodzić się z dodatkową ostrożnością i ograniczać do nich dostęp. Częstym rozwiązaniem jest użycie zewnętrznego systemu zarządzania sekretami, takiego jak Vault firmy HashiCorp (<https://www.vaultproject.io>). Kafka Connect obejmuje obsługę zewnętrznej konfiguracji sekretów (<https://oreil.ly/5eVRU>). Apache Kafka zawiera jedynie framework umożliwiający wprowadzanie wtyczek zewnętrznych dostawców konfiguracji oraz przykładowego dostawcę, który odczytuje konfigurację z pliku, a ponadto dostępni są opracowani przez społeczność Kafki zewnętrzni dostawcy konfiguracji (<https://oreil.ly/ovntG>), którzy integrują się z usługami Vault, AWS i Azure.

Postępowanie w razie awarii

Niebezpieczne jest zakładanie, że wszystkie dane będą przez cały czas w idealnej kondycji. Należy z wyprzedzeniem zaplanować postępowanie na wypadek awarii. Czy możemy zapobiec przedostawaniu się do potoku wadliwych rekordów? Czy możemy odzyskać dane z rekordów, których nie można sparsować? Czy uszkodzone rekordy mogą zostać naprawione (np. przez człowieka) i ponownie przetworzone? Co zrobić, jeśli uszkodzone zdarzenie będzie wyglądało tak samo jak normalne i odkryjesz problem dopiero kilka dni później?

Ponieważ Kafkę można skonfigurować do przechowywania wszystkich zdarzeń przez długi czas, możliwe jest „cofnięcie się w czasie” i naprawienie błędów, jeśli zajdzie taka potrzeba. Umożliwia to również systemowi docelowemu odtworzenie zapisanych w Kafce zdarzeń, gdy zostaną utracone.

Powiązania i zwinność

Pożądaną cechą charakterystyczną implementacji potoku danych jest oddzielenie źródeł danych od ujęć danych. Do powstania przypadkowych powiązań może dojść na wiele sposobów:

Potoki *ad hoc*

Niektóre firmy budują niestandardowy potok dla każdej pary aplikacji, które chcą połączyć. Używają np. narzędzia Logstash do zrzucania dzienników do Elasticsearcha, Flume’a do zrzucania dzienników do systemu plików HDFS, GoldenGate’a firmy Oracle do pobierania danych

z bazy danych Oracle do HDFS-a, narzędzia Informatica do pobierania danych z MySQL-a i XML-a do bazy danych Oracle itd. Tworzy to ściśle powiązanie potoku danych z konkretnymi punktami końcowymi i generuje bałagan punktów integracji, których wdrożenie, utrzymanie i monitorowanie wymaga znacznego wysiłku. Oznacza to również, że każdy nowy system wprowadzany przez firmę będzie wymagał budowy dodatkowych potoków, zwiększając koszty wdrażania nowych technologii i hamując innowacje.

Utrata metadanych

Jeśli potok danych nie zachowuje metadanych schematu i nie umożliwia jego ewolucji, tworzą się ściśle powiązania między oprogramowaniem produkującym dane u źródła a oprogramowaniem, które wykorzystuje dane w miejscu docelowym. Bez informacji o schemacie oba oprogramowania muszą zawierać informacje o tym, jak parsować i interpretować dane. Jeżeli dane przepływają z bazy Oracle do systemu HDFS, a administrator baz danych dodał w Oracle'u nowe pole bez zachowania informacji o schemacie i umożliwienia ewolucji schematu, każda aplikacja odczytująca dane z HDFS-a ulegnie awarii albo wszyscy programiści będą musieli uaktualnić swoje aplikacje w tym samym czasie. Żadna z opcji nie jest zwinna. Dzięki obsłudze ewolucji schematów w potoku każdy zespół może modyfikować swoje aplikacje we własnym tempie, nie przejmując się, że wszystko się zepsuje.

Ekstremalne przetwarzanie

Jak wspomnieliśmy podczas omawiania transformacji danych, z potokami danych nieodłącznie związane jest pewne przetwarzanie danych. W końcu przenosimy dane między różnymi systemami, w których stosowane muszą być odmienne formaty danych i obsługiwane są różne przypadki użycia. Jednak zbyt daleko posunięte przetwarzanie wiąże wszystkie systemy downstreamowe z podjętymi podczas budowy potoków decyzjami dotyczącymi tego, które pola należy zachowywać, jak agregować dane itp. Często prowadzi to do ciągłych zmian w potoku, w miarę jak zmieniają się wymagania aplikacji downstreamowych, co nie jest zwinne, wydajne ani bezpieczne. Bardziej zwinnym sposobem jest zachowywanie jak największej ilości surowych danych i umożliwienie twórcom aplikacji downstreamowych, w tym aplikacji Kafka Streams, podejmowania własnych decyzji dotyczących przetwarzania i agregacji danych.

Przypadki użycia dla frameworku Kafka Connect oraz dla producentów i konsumentów

Przy zapisywaniu w Kafce lub odczytywaniu z niej masz do wyboru skorzystanie z tradycyjnych klientów producenckich i konsumenckich, jak opisaliśmy w rozdziałach 3. i 4., albo użycie Kafka Connect API i konektorów, co omówimy w następnym podrozdziale. Zanim zaczniemy zagłębiać się w szczegóły frameworku Kafka Connect, winni jesteśmy wyjaśnienie, kiedy używamy jednego, a kiedy drugiego rozwiązania.

Jak pisaliśmy, klienci Kafki to klienci, które osadzasz w swojej aplikacji. Umożliwia to aplikacji zapisywanie danych w Kafce lub ich odczyt z tej platformy. Klienci Kafki mają zastosowanie, gdy możesz modyfikować kod aplikacji, do której chcesz podłączyć oprogramowanie, oraz gdy chcesz wysyłać dane do Kafki lub pobierać dane z Kafki.

Za pomocą interfejsu Connect połączysz Kafkę z magazynami danych, których nie napisałeś i których kodu lub interfejsów API nie możesz lub nie będziesz modyfikować. Connect będzie używany do pobierania danych z zewnętrznego magazynu danych do Kafki lub przysyłania danych z Kafki do zewnętrznego magazynu. Aby korzystać z frameworku Kafka Connect, potrzebujesz konektora dla magazynu danych, do którego chcesz się podłączyć, a obecnie dostępnych jest wiele konektorów. Oznacza to, że w praktyce użytkownicy Connect API muszą pisać jedynie pliki konfiguracyjne.

Jeśli musisz połączyć Kafkę z magazynem danych, a odpowiedni konektor jeszcze nie istnieje, możesz wybierać między napisaniem aplikacji za pomocą klientów Kafki a skorzystaniem z interfejsu Connect API. Zalecana jest ta druga opcja, ponieważ Connect zapewnia gotowe funkcjonalności, takie jak zarządzanie konfiguracją, magazyn przesunięć, zrównoleglenie, obsługa błędów, obsługa różnych typów danych i standardowe interfejsy RESTful API do zarządzania. Napisanie niewielkiej aplikacji łączącej Kafkę z magazynem danych wydaje się proste, ale trzeba zająć się wieloma szczegółami dotyczącymi typów danych i konfiguracji, przez co w ostatecznym rozrachunku jest to nietrywialne zadanie. Co więcej, będziesz musiał utrzymywać tę aplikację dla potoku i dokumentować ją, a Twoi koledzy z zespołu będą musieli nauczyć się z niej korzystać. Kafka Connect jest standardową częścią ekosystemu Kafki i obsługuje większość tych elementów, dzięki czemu możesz skupić się na transporcie danych do magazynów zewnętrznych i z powrotem.

Kafka Connect

Kafka Connect jest częścią platformy Apache Kafka i zapewnia skalowalny i niezawodny sposób kopiowania danych między Kafką a innymi magazynami danych. Oferuje interfejsy API i środowisko uruchomieniowe do tworzenia i uruchamiania **wtyczek konektorów** — wykonywanych przez framework Kafka Connect bibliotek, które są odpowiedzialne za przenoszenie danych. Kafka Connect działa jako klaster **procesów roboczych**. Wtyczki konektorów instaluje się na węzłach roboczych, a następnie używa się interfejsu RESTful API do konfigurowania **konektorów** i zarządzania tymi, które działają z określoną konfiguracją. **Konektory** uruchamiają dodatkowe **zadania**, aby równolegle przynosić duże ilości danych i efektywniej wykorzystywać zasoby dostępne na węzłach roboczych. Zadania konektora źródłowego muszą po prostu odczytywać dane z systemu źródłowego i dostarczać obiekty danych Connecta do procesów roboczych. Zadania konektora ujściowego pobierają obiekty danych z procesów roboczych i są odpowiedzialne za zapisywanie ich w docelowym systemie danych. Kafka Connect używa **konwerterów** do obsługi przechowywania tych obiektów danych w Kafce w różnych formatach — obsługa formatu JSON jest częścią platformy Apache Kafka, a Confluent Schema Registry zapewnia konwertery Avry, Protobufa i JSON Schema. Umożliwia to użytkownikom wybór formatu przechowywania danych w Kafce niezależnie od używanych przez nich konektorów, a także wybór sposobu obsługi schematu danych (jeśli w ogóle jest obsługiwany).

W tym rozdziale nie możemy zgłębić wszystkich szczegółów frameworku Kafka Connect i jego wielu konektorów. Jest to materiał na osobną książkę. Przedstawimy jednak przegląd tego narzędzia, omówimy sposób jego używania i wskażemy dodatkowe materiały referencyjne.

Uruchamianie frameworku Kafka Connect

Framework Kafka Connect jest dostarczany z Apache Kafką, nie trzeba więc instalować go osobno. Do użytku produkcyjnego, zwłaszcza jeśli planujesz wykorzystywać Connect do przenoszenia dużych ilości danych lub uruchamiania wielu konektorów, powinieneś uruchomić ten komponent na osobnych serwerach, na których nie ma brokerów Kafki. W takim przypadku zainstaluj platformę Apache Kafka na wszystkich komputerach i na jednych serwerach uruchom brokery, a na drugich komponent Connect.

Uruchamianie węzła roboczego komponentu Connect bardzo przypomina uruchamianie brokera — wywołujesz skrypt startowy z plikiem właściwości:

```
bin/connect-distributed.sh config/connect-distributed.properties
```

Węzły robocze Connecta mają kilka kluczowych konfiguracji:

`bootstrap.servers`

Lista brokerów Kafki, z którymi będzie współpracował Connect. Konektory będą za pomocą potoków przysyłać swoje dane do tych brokerów i z powrotem. Nie musisz podawać każdego brokera w klastrze, ale zaleca się wskazanie co najmniej trzech.

`group.id`

Wszystkie węzły robocze z tym samym identyfikatorem grupy są częścią tego samego klastra frameworku Connect. Konektor uruchomiony w klastrze będzie działał na każdym węźle roboczym, podobnie jak jego zadania.

`plugin.path`

Kafka Connect korzysta z architektury opartej na wtyczkach, która umożliwia pobieranie i dodawanie do platformy konektorów, konwerterów, transformacji i dostawców sekretów. Aby to obsługiwać, framework musi być w stanie znaleźć i załadować te wtyczki.

Jako lokalizacje dla konektorów i ich zależności możemy skonfigurować więcej niż jeden katalog, np. `plugin.path=/opt/connectors,/home/gwenshap/connectors`. Wewnątrz jednego z tych katalogów tworzymy zazwyczaj podfolder dla każdego konektora, więc dla podanego przykładu utworzymy `/opt/connectors/jdbc` i `/opt/connectors/elastic`. W każdym podfolderze umieścimy plik `jar` samego konektora i wszystkie jego zależności. Jeśli konektor jest dostarczany jako *uberJar* i nie ma żadnych zależności, można go umieścić bezpośrednio w lokalizacji `plugin`. ↪ `path` i nie wymaga podfolderu. Należy jednak pamiętać, że umieszczenie zależności na ścieżce najwyższego poziomu nie będzie działać.

Alternatywą jest dodanie konektorów i wszystkich ich zależności do zmiennej środowiskowej `CLASSPATH` frameworku Kafka Connect, ale nie jest to zalecane i może wprowadzać błędy, jeśli używasz konektora z zależnościami, która koliduje z jedną z zależności Kafki. Zalecanym podejściem jest użycie konfiguracji `plugin.path`.

`key.converter` i `value.converter`

Connect może obsługiwać wiele formatów danych przechowywanych w Kafce. Te dwie konfiguracje ustawiają konwerter odpowiednio dla klucza i wartości komunikatu, który będzie przechowywany w Kafce. Domyślnym formatem jest JSON korzystający z konwertera `JSONConverter` dostarczanego z Apache Kafką. Dla tych konfiguracji można ustawić również wartości ↪

AvroConverter, ProtobufConverter lub JscSchemaConverter, które są częścią rejestru schematów Confluenta.

Niektóre konwertery zawierają charakterystyczne dla siebie parametry konfiguracyjne. Należy użyć dla nich prefiksu `key.converter.` lub `value.converter.`, w zależności od tego, czy chcemy zastosować je do konwertera kluczy czy wartości. Komunikaty JSON mogą np. zawierać schemat lub być go pozbawione. Aby obsłużyć to ustawienie, można ustawić odpowiednio `key.converter.schemas.enable=true` lub `key.converter.schemas.enable=false`. Tę samą konfigurację można zastosować do konwertera wartości, ustawiając wartość `true` lub `false` dla `value.converter.schemas.enable`. Komunikaty Avry także zawierają schemat, ale należy skonfigurować lokalizację rejestru schematów za pomocą ustawień `key.converter.schema.registry.url` i `value.converter.schema.registry.url`.

`rest.host.name` i `rest.port`

Konektory są zwykle konfigurowane i monitorowane za pośrednictwem interfejsu RESTful API frameworku Kafka Connect. Dla RESTful API można skonfigurować określony port.

Gdy uruchomisz węzły robocze i klaster, sprawdź RESTful API, by upewnić się, że działają:

```
$ curl http://localhost:8083/
{"version":"3.0.0-SNAPSHOT", "commit":"fae0784ce32a448a", "kafka_cluster_id":"pfkYIGZQSxm8Ry1-vACQHdg"}%
```

Uzyskanie dostępu do bazowego URI RESTful powinno zwrócić aktualnie uruchomioną wersję. Uruchamiamy migawkę Kafki 3.0.0 (wersja wstępna). Możemy również sprawdzić, które wtyczki są dostępne:

```
$ curl http://localhost:8083/connector-plugins

[
  {
    "class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type": "sink",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type": "source",
    "version": "3.0.0-SNAPSHOT"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorCheckpointConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorHeartbeatConnector",
    "type": "source",
    "version": "1"
  },
  {
    "class": "org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "type": "source",
    "version": "1"
  }
]
```

Używamy czystej platformy Apache Kafka, więc jedynymi dostępnymi wtyczkami są źródło plików, ujście plików i konektory, które są częścią narzędzia MirrorMaker 2.0.

Zobaczmy, jak konfigurować i stosować te przykładowe konektory. Potem przejdziemy do bardziej zaawansowanych przykładów, które wymagają skonfigurowania zewnętrznych systemów danych, z którymi będziemy się łączyć.



Tryb samodzielny

Należy pamiętać, że Kafka Connect ma również tryb samodzielny. Jest on podobny do trybu rozproszonego — zamiast `bin/connect-distributed.sh` wystarczy uruchomić `bin/connect-standalone.sh`. Plik konfiguracyjny konektora można przekazać również w wierszu poleceń, a nie za pośrednictwem interfejsu RESTful API. W tym trybie wszystkie konektory i zadania są uruchamiane na pojedynczym samodzielnym węźle roboczym. Jest on używany w przypadkach, gdy konektory i zadania trzeba uruchamiać na konkretnej maszynie (np. konektor `syslog` nasłuchuje na określonym porcie, trzeba więc wiedzieć, na których maszynach jest uruchomiony).

Przykład konektora — źródło plików i ujście plików

W tym przykładzie użyjemy konektorów plików i konwertera JSON, które są częścią platformy Apache Kafka. Aby kontynuować, upewnij się, że masz uruchomione serwer ZooKeeper i Kafkę.

Na początek uruchommy węzeł roboczy rozproszonego Connecta. W rzeczywistym środowisku produkcyjnym wymagane są co najmniej dwa lub trzy węzły robocze, aby zapewnić wysoką dostępność. W tym przykładzie uruchomimy tylko jeden:

```
bin/connect-distributed.sh config/connect-distributed.properties &
```

Nadszedł czas na uruchomienie źródła plików. Jako przykład skonfigurujemy je do odczytu pliku konfiguracyjnego Kafki — zasadniczo przesyłając potokiem konfigurację Kafki do tematu Kafki:

```
echo '{"name":"load-kafka-config",
"config":{"connector.class":"FileStreamSource","file":"config/server.properties","topic":"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:8083/connectors -H "Content-Type: application/json"

{
  "name": "load-kafka-config",
  "config": {
    "connector.class": "FileStreamSource",
    "file": "config/server.properties",
    "topic": "kafka-config-topic",
    "name": "load-kafka-config"
  },
  "tasks": [
    {
      "connector": "load-kafka-config",
      "task": 0
    }
  ],
  "type": "source"
}
```

W celu utworzenia konektora napisaliśmy skrypt JSON zawierający nazwę konektora (load-kafka-
→config) i mapę konfiguracji konektora obejmującą klasę konektora, plik, który chcemy załadować, oraz temat, do którego chcemy załadować plik.

Użyjmy konsumenta Kafka Console, by sprawdzić, czy załadowaliśmy konfigurację do tematu:

```
gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic kafka-config-topic --from-beginning
```

Jeśli wszystko poszło dobrze, powinieneś zobaczyć mniej więcej coś takiego:

```
{ "schema": { "type": "string", "optional": false }, "payload": "# Licensed to the Apache Software Foundation (ASF) under one or more" }  
<fragment_usunięty>  
{ "schema": { "type": "string", "optional": false }, "payload": "#####  
Server Basics #####" }  
{ "schema": { "type": "string", "optional": false }, "payload": "" }  
{ "schema": { "type": "string", "optional": false }, "payload": "# The id of the broker. This must be set to a unique integer for each broker." }  
{ "schema": { "type": "string", "optional": false }, "payload": "broker.id=0" }  
{ "schema": { "type": "string", "optional": false }, "payload": "" }  
<fragment_usunięty>
```

Jest to dosłownie zawartość pliku *config/server.properties*, tak jak została przekonwertowana na format JSON linia po linii i umieszczona przez nasz konektor w temacie *kafka-config-topic*. Zwróć uwagę, że domyślnie konwerter JSON umieszcza schemat w każdym rekordzie. W tym konkretnym przypadku schemat jest bardzo prosty — istnieje tylko jedna kolumna o nazwie *payload* typu *string* i dla każdego rekordu zawiera pojedynczą linię z pliku.

Teraz użyjmy konwertera ujścia plików, aby zrzucić zawartość tego tematu do pliku. Wynikowy plik powinien być całkowicie identyczny z oryginalnym plikiem *server.properties*, ponieważ konwerter JSON konwertuje rekordy JSON z powrotem na zwykłe linie tekstu:

```
echo '{ "name": "dump-kafka-config",  
  "config": { "connector.class": "FileStreamSink", "file": "copy-of-server-properties", "topics": "kafka-config-topic" } }' | curl -X POST -d @-  
http://localhost:8083/connectors --header "content-Type:application/json"  
{ "name": "dump-kafka-config", "config": { "connector.class": "FileStreamSink", "file": "copy-of-server-properties", "topics": "kafka-config-topic", "name": "dump-kafka-config", "tasks":  
  [] }
```

Przyjrzyj się zmianom w konfiguracji źródła: używana klasa to teraz *FileStreamSink*, a nie *File* → *StreamSource*. Nadal mamy właściwość pliku, ale teraz nie odnosi się ona do źródła rekordów, tylko do pliku docelowego, a zamiast określania *tematu* (*topic*) określamy *tematy* (*topics*). Zwróć uwagę na tę liczbę mnogą — za pomocą ujścia można zapisywać wiele tematów w jednym pliku, podczas gdy źródło umożliwia zapisywanie tylko do jednego tematu.

Jeśli wszystko poszło dobrze, powinieneś mieć plik o nazwie *copy-of-server-properties*, który jest identyczny z plikiem *config/server.properties* użytym do wypełnienia tematu *kafka-config-topic*.

Aby usunąć konektor, możesz uruchomić następujące polecenie:

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```



W tym przykładzie zastosowaliśmy konektory FileStream, ponieważ są proste i wbudowane w Kafkę, co pozwala na utworzenie pierwszego potoku przy zainstalowaniu wyłącznie Kafki. Nie należy ich stosować w przypadku rzeczywistych potoków produkcyjnych, ponieważ mają wiele ograniczeń i nie oferują gwarancji niezawodności. Jeśli chcesz pobierać dane z plików, możesz użyć kilku innych konektorów, takich jak FilePulse Connector (<https://oreil.ly/VLCf2>), FileSystem Connector (<https://oreil.ly/Fcryw>) lub SpoolDir (<https://oreil.ly/qgsI4>).

Przykład konektora — z MySQL-a do Elasticsearcha

Skoro udało nam się uruchomić prosty przykład, zrobimy coś bardziej użytecznego. Weźmy tablicę bazy danych MySQL, prześlijmy ją do tematu Kafki, a stamtąd załadujmy ją do Elasticsearcha i zindeksujmy jej zawartość.

Testy przeprowadzamy na MacBooku. Aby zainstalować MySQL i Elasticsearch, wykonaj następujące polecenia:

```
brew install mysql
brew install elasticsearch
```

W następnym kroku powinieneś się upewnić, że masz konektory. Istnieje kilka opcji:

1. Pobranie i zainstalowanie przy użyciu klienta Confluent Huba (<https://oreil.ly/c7S5z>).
2. Pobranie ze strony internetowej Confluent Huba (<https://www.confluent.io/hub>) lub z dowolnej innej strony internetowej, na której znajduje się interesujący Cię konektor.
3. Skompilowanie z kodu źródłowego. W tym celu musisz wykonać kilka czynności:
 - a) sklonuj źródło konektora:
 - b) `git clone https://github.com/confluentinc/kafka-connect-elasticsearch`
 - c) uruchom polecenie `mvn install -DskipTests`, aby skompilować projekt;
 - d) powtórz te czynności dla konektora JDBC (<https://oreil.ly/yXg0S>).

Teraz musimy załadować te konektory. Utwórz katalog, np. `/opt/connectors`, i zaktualizuj plik `config/connect-distributed.properties`, aby zawierał konfigurację `plugin.path=/opt/connectors`.

Następnie do odpowiednich podkatalogów `plugin.path` skopiuj wraz z zależnościami poszczególne pliki `jar` utworzone w katalogu `target`, gdzie skompilowałeś każdy konektor:

```
gwen$ mkdir /opt/connectors/jdbc
gwen$ mkdir /opt/connectors/elastic
gwen$ cp ../kafka-connect-jdbc/target/kafka-connect-jdbc-10.3.x-SNAPSHOT.jar
/opt/connectors/jdbc
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-elasticsearch-11.1.0-
SNAPSHOT.jar /opt/connectors/elastic
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connect-elasticsearch-11.1.0-
SNAPSHOT-package/share/java/kafka-connect-elasticsearch/* /opt/connectors/elastic
```

Ponieważ musimy połączyć się nie z jakąś bazą danych, ale konkretnie z MySQL-em, musisz ponadto pobrać i zainstalować sterownik JDBC MySQL-a. Z powodów licencyjnych ten sterownik nie jest dostarczany z konektorem. Możesz go pobrać ze strony bazy danych MySQL (<https://oreil.ly/KZCPw>), a następnie umieścić plik `jar` w katalogu `/opt/connectors/jdbc`.

Zrestartuj węzły robocze komponentu Kafka Connect i sprawdź, czy na liście znajdują się nowe wtyczki konektorów:

```
gwen$ bin/connect-distributed.sh config/connect-distributed.properties &

gwen$ curl http://localhost:8083/connector-plugins
[
  {
    "class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "type": "sink",
    "version": "11.1.0-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "type": "sink",
    "version": "10.3.x-SNAPSHOT"
  },
  {
    "class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "type": "source",
    "version": "10.3.x-SNAPSHOT"
  }
]
```

Widzimy, że w naszym klastrze Connect mamy teraz dostępne dodatkowe wtyczki konektorów.

Następnym krokiem jest utworzenie w MySQL-u tablicy, którą będziemy mogli przesyłać strumieniowo do Kafki za pomocą konektora JDBC:

```
gwen$ mysql.server restart
gwen$ mysql --user=root

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)
```

Jak widać, utworzyliśmy bazę danych i tablicę, a jako przykład wstawiliśmy kilka wierszy.

Kolejnym krokiem jest skonfigurowanie konektora źródłowego JDBC. Dostępne opcje konfiguracyjne można znaleźć w dokumentacji, ale można również użyć w tym celu RESTful API:

```
gwen$ curl -X PUT -d '{"connector.class":"JdbcSource"}' localhost:8083/connector-
plugins/JdbcSourceConnector/config/validate/ --header "content-Type:application/json"

{
  "configs": [
    {
      "definition": {
        "default_value": "",
```

```

    "dependents": [],
    "display_name": "Timestamp Column Name",
    "documentation": "The name of the timestamp column to use to detect
        new or modified rows. This column may not be nullable.",
    "group": "Mode",
    "importance": "MEDIUM",
    "name": "timestamp.column.name",
    "order": 3,
    "required": false,
    "type": "STRING",
    "width": "MEDIUM"
  },
</fragment_usunięty>

```

Poprosiliśmy RESTful API o walidację konfiguracji dla konektora i wysłaliśmy mu konfigurację z samą nazwą klasy (jest to minimalna wymagana konfiguracja). W odpowiedzi otrzymaliśmy definicję JSON wszystkich dostępnych konfiguracji.

Mając te informacje na uwadze, możemy teraz utworzyć i skonfigurować nasz konektor JDBC:

```

echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSource-
Connector","connection.url":"jdbc:mysql://127.0.0.1:3306/test?user=root","mode":"timestamp
","table.whitelist":"login","validate.non.null":false,"timestamp.column.name":"login_time"
,"topic.prefix":"mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header
"content-Type:application/json"

{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector"
  },
  "tasks": []
}

```

Odczytując dane z tematu `mysql.login`, sprawdźmy, czy to zadziałało:

```

gwen$ bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic mysql.login
--from-beginning

```

Jeżeli otrzymasz komunikat o błędzie informujący, że temat nie istnieje, lub nie zobaczysz żadnych danych, sprawdź dzienniki węzła roboczego Connecta pod kątem błędów takich jak ten:

```

[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-login-connector
(org.apache.kafka.connect.runtime.WorkerConnector:108)
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: Access denied for
user 'root;'@'localhost' (using password: NO) at
io.confluent.connect.jdbc.JdbcSourceConnector.start(JdbcSourceConnector.java:78)

```

Inne potencjalne problemy mogą dotyczyć umieszczenia sterownika w zmiennej środowiskowej `CLASSPATH` lub uprawnień do odczytu tablicy.

Jeśli po uruchomieniu konektora wstawisz dodatkowe wiersze w tablicy login, powinieneś je natychmiast zobaczyć w temacie mysql.login.



Przechwytywanie zmian danych i Debezium Project

Używany przez nas konektor JDBC wykorzystuje JDBC i SQL do skanowania tablic baz danych w poszukiwaniu nowych rekordów. Wykrywa nowe rekordy za pomocą pól znacznika czasu lub zwiększającego się klucza podstawowego. Jest to stosunkowo nieefektywny i czasami niedokładny proces. Wszystkie relacyjne bazy danych posiadają jako część implementacji dziennik transakcji (zwany również dziennikiem powtórzeń, dziennikiem binarnym lub dziennikiem rejestrowania z wyprzedzeniem), a wiele z nich umożliwia zewnętrznym systemom odczytywanie danych bezpośrednio z dziennika transakcji — jest to znacznie dokładniejszy i wydajniejszy proces znany jako **przechwytywanie zmian danych** (ang. *change data capture*). Większość nowoczesnych systemów ETL traktuje przechwytywanie zmian danych jak źródło danych. Debezium Project (<https://debezium.io>) zapewnia zbiór dostępnych na licencji *open source* wysokiej jakości konektorów przechwytywania zmian dla różnych baz danych. Jeśli planujesz strumieniowe przesyłanie danych z relacyjnej bazy danych do Kafki, zalecamy użycie konektora przechwytywania zmian Debezium, jeżeli istnieje odpowiedni dla Twojej bazy danych. Ponadto dokumentacja Debezium jest jedną z najlepszych, z jakich korzystaliśmy — oprócz dokumentowania samych konektorów obejmuje przydatne wzorce projektowe i przypadki użycia związane z przechwytywaniem zmian danych, zwłaszcza w kontekście mikrousług.

Dostarczanie danych MySQL-a do Kafki jest użyteczne samo w sobie, ale uczynimy to bardziej zabawnym przez zapisywanie danych w Elasticsearchu.

Najpierw uruchamiamy Elasticsearch i uzyskując dostęp do jego lokalnego portu, sprawdzamy, czy działa:

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Chens-MBP",
  "cluster_name" : "elasticsearch_gwenshap",
  "cluster_uuid" : "X69zu3_sQNGb7zbMh7NDVw",
  "version" : {
    "number" : "7.5.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8bec50e1e0ad29dad5653712cf3bb580cd1afcdf",
    "build_date" : "2020-01-15T12:11:52.313576Z",
    "build_snapshot" : false,
    "lucene_version" : "8.3.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Teraz stworzymy i uruchamiamy konektor:

```
echo '{"name":"elastic-login-connector",
"config":{"connector.class":"ElasticsearchSinkConnector","connection.url":"http://localhos
```

```
t:9200","type.name":"mysql-data","topics":"mysql.login","key.ignore":true}}' | curl -X
POST -d @- http://localhost:8083/connectors --header "content-Type:application/json"
```

```
{
  "name": "elastic-login-connector",
  "config": {
    "connector.class": "ElasticsearchSinkConnector",
    "connection.url": "http://localhost:9200",
    "topics": "mysql.login",
    "key.ignore": "true",
    "name": "elastic-login-connector"
  },
  "tasks": [
    {
      "connector": "elastic-login-connector",
      "task": 0
    }
  ]
}
```

Jest tu kilka konfiguracji, które musimy wyjaśnić. `connection.url` jest po prostu adresem URL lokalnego serwera Elasticsearcha, który skonfigurowaliśmy wcześniej. Każdy temat z Kafki stanie się domyślnie oddzielnym indeksem Elasticsearcha o tej samej nazwie co temat. Jedynym tematem, który zapisujemy w Elasticsearchu, jest `mysql.login`. Konektor JDBC nie zapewnia klucza komunikatu. W rezultacie zdarzenia w Kafce mają klucze zerowe. Ponieważ zdarzeniom w Kafce brakuje kluczy, musimy poinstruować konektor Elasticsearcha, aby jako klucza dla każdego zdarzenia używał nazwy tematu, identyfikatora partycji i przesunięcia. Odbywa się to przez ustawienie wartości `true` dla konfiguracji `key.ignore`.

Sprawdźmy, czy utworzony został indeks z danymi `mysql.login`:

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index          uuid                pri rep docs.count
docs.deleted store.size pri.store.size
yellow open    mysql.login wkeyk9-bQea6NJmAFjv4hw  1   1           2
0          3.9kb      3.9kb
```

Jeżeli nie ma indeksu, poszukaj błędów w dzienniku węzła roboczego Connecta. Częstymi przyczynami błędów są brakujące konfiguracje lub biblioteki. Jeśli wszystko jest w porządku, możemy wyszukać nasze rekordy w indeksie:

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=true"
{
  "took" : 40,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    }
  }
}
```



```

"max_score" : 1.0,
"hits" : [
  {
    "_index" : "mysql.login",
    "_type" : "_doc",
    "_id" : "mysql.login+0+0",
    "_score" : 1.0,
    "_source" : {
      "username" : "gwenshap",
      "login_time" : 1621699811000
    }
  },
  {
    "_index" : "mysql.login",
    "_type" : "_doc",
    "_id" : "mysql.login+0+1",
    "_score" : 1.0,
    "_source" : {
      "username" : "tpalino",
      "login_time" : 1621699816000
    }
  }
]
}

```

Gdy dodasz nowe rekordy do tabeli w bazie danych MySQL, pojawią się one automatycznie w temacie `mysql.login` w Kafce oraz w odpowiednim indeksie Elasticsearcha.

Skoro dowiedziałeś się, jak skompilować i zainstalować źródło JDBC i ujęcie Elasticsearcha, możesz zbudować i wykorzystywać dowolną parę konektorów, które pasują do Twojego przypadku użycia. Firma Confluent utrzymuje w repozytorium Confluent Hub (<https://www.confluent.io/hub>) zestaw swoich wstępnie skompilowanych konektorów, a także kilka konektorów udostępnionych przez społeczność i innych dostawców. Możesz wybrać z listy dowolny konektor, który chcesz wypróbować, a następnie pobrać go, skonfigurować — na podstawie dokumentacji lub pobierając konfigurację z interfejsu RESTful API — i uruchomić w klastrze węzłów roboczych Connecta.



Buduj własne konektory

Interfejs Connector API jest publiczny i każdy może tworzyć nowe konektory. Dlatego jeśli dla magazynu danych, z którym chcesz dokonać integracji, nie ma dostępnego konektora, zachęcamy Cię do napisania własnego. Możesz potem udostępnić go w repozytorium Confluent Hub, aby inni mogli go znaleźć i wykorzystać. Omówienie wszystkich szczegółów związanych z budowaniem konektora wykracza poza zakres tego rozdziału, ale istnieje wiele postów na blogu, które wyjaśniają, jak to zrobić (<https://oreil.ly/WUqLZ>), a ponadto udostępnione są dobre prelekcje z konferencji Kafka Summit NY 2019 (<https://oreil.ly/rV9RH>), Kafka Summit London 2018 (<https://oreil.ly/Jz7XV>) i ApacheCon (<https://oreil.ly/8QsOL>). Zalecamy również przyjrzenie się istniejącym konektorom jako punktom wyjścia i być może zabranie się od razu do pracy przy użyciu archetypu Apache Maven (<http://bit.ly/2sc9E9q>). Zawsze zachęcamy do proszenia o pomoc lub popisywania się najnowszymi konektorami na liście mailingowej społeczności Apache Kafki (users@kafka.apache.org), lub umieszczania ich w repozytorium Confluent Hub, aby można je było łatwo znaleźć.

Pojedyncze transformacje komunikatów

Kopiowanie rekordów z bazy danych MySQL do Kafki i stamtąd do Elasticu jest przydatne samo w sobie, jednak potoki ETL obejmują zazwyczaj etap transformacji. W ekosystemie Kafki przekształcenia wyodrębniane są do postaci funkcjonalności pojedynczych transformacji komunikatów (ang. *Single Message Transformations* — SMT), które są bezstanowym (ale mogą być też stanowym) przetwarzaniem strumieniowym. SMT można wykonywać za pomocą frameworku Kafka Connect, przekształcając komunikaty podczas ich kopiowania, często bez pisania jakiegokolwiek kodu. Bardziej złożone transformacje, które obejmują zazwyczaj łączenia lub agregacje, będą wymagały stanowego frameworku Kafka Streams, który omówimy dalej w książce.

Apache Kafka obejmuje następujące SMT:

Cast

Zmiana typu danych pola.

MaskField

Zastąpienie zawartości pola wartością null. Jest to przydatne do usuwania danych poufnych lub osobowych.

Filter

Zrobienie rzutu lub załączenie wszystkich komunikatów, które spełniają określony warunek. Do wbudowanych warunków należą: dopasowanie nazwy tematu, dopasowanie określonego nagłówka oraz sprawdzenie, czy komunikat jest nagrobkiem (czyli ma wartość null).

Flatten

Przekształcenie zagnieżdżonej struktury danych w płaską. Odbywa się to przez konkatencję nazw wszystkich pól w danej ścieżce do określonej wartości.

HeaderFrom

Przeniesienie lub skopiowanie pól z komunikatu do nagłówka.

InsertHeader

Dodanie do nagłówka każdego komunikatu statycznego łańcucha znaków.

InsertField

Dodanie nowego pola do komunikatu przy użyciu wartości z jego metadanych, takiej jak przesunięcie, lub wartości statycznej.

RegexRouter

Zmiana tematu docelowego za pomocą wyrażenia regularnego i zastępczego łańcucha znaków.

ReplaceField

Usunięcie pola z komunikatu lub zmiana nazwy pola.

TimestampConverter

Modyfikowanie formatu czasu pola, np. z czasu uniksowego na String.

TimestampRouter

Modyfikowanie tematu na podstawie znacznika czasu komunikatu. Jest to przydatne głównie w konektorach ujęcia, gdy chcemy kopiować komunikaty do określonych partycji tablicy na podstawie ich znacznika czasu, a pole tematu służy do znalezienia równoważnego zbioru danych w systemie docelowym.

Dostępne są również transformacje od kontrybutorów spoza głównej bazy kodu platformy Apache Kafka. Przydatne kolekcje można znaleźć na GitHubie w repozytorium Lenses.io (<https://oreil.ly/fWAyh>), Aiven (<https://oreil.ly/oQRG5>) i Jeremy Custenborder (<https://oreil.ly/OdPHW>) lub na Confluent Hubie (<https://oreil.ly/Up8dM>).

Więcej informacji na temat funkcjonalności SMT frameworku Kafka Connect znajdziesz w serii blogów *Twelve Days of SMT* (<https://oreil.ly/QnpQV>), gdzie opisane są szczegółowe przykłady wielu transformacji. Ponadto z samouczka *How to Use Single Message Transforms in Kafka Connect* (<https://oreil.ly/rw4CU>) możesz się dowiedzieć, jak pisać własne przekształcenia.

Dla przykładu założmy, że chcemy dodać nagłówek rekordu (<https://oreil.ly/ISiWs>) do każdego rekordu wyprodukowanego przez utworzony wcześniej konektor MySQL-a. Nagłówek będzie wskazywał, że rekord został utworzony przez ten konektor MySQL-a, co jest przydatne w sytuacji, gdy audytorzy będą chcieli zbadać pochodzenie tych rekordów.

W tym celu zastąpimy poprzednią konfigurację konektora MySQL-a następującą konfiguracją:

```
echo '{
  "name": "mysql-login-connector",
  "config": {
    "connector.class": "JdbcSourceConnector",
    "connection.url": "jdbc:mysql://127.0.0.1:3306/test?user=root",
    "mode": "timestamp",
    "table.whitelist": "login",
    "validate.non.null": "false",
    "timestamp.column.name": "login_time",
    "topic.prefix": "mysql.",
    "name": "mysql-login-connector",
    "transforms": "InsertHeader",
    "transforms.InsertHeader.type": "org.apache.kafka.connect.transforms.InsertHeader",
    "transforms.InsertHeader.header": "MessageSource",
    "transforms.InsertHeader.value.literal": "mysql-login-connector"
  }}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
```

Jeśli do tablicy MySQL-a, którą utworzyliśmy w poprzednim przykładzie, wstawisz teraz jeszcze kilka rekordów, zobaczysz, że nowe komunikaty w temacie `mysql.login` mają nagłówki (pamiętaj, że do wypisywania nagłówków w konsoli konsumenckiej potrzebujesz platformy Apache Kafka w wersji co najmniej 2.7):

```
bin/kafka-console-consumer.sh --bootstrap-server=localhost:9092 --topic mysql.login --from-
beginning --property print.headers=true
NO_HEADERS
{"schema":{"type":"struct","fields":[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Timestamp","version":
:1,"field":"login_time"}],"optional":false,"name":"login"},"payload":{"username":"tpalino",
"login_time":1621699816000}}
MessageSource:mysql-login-connector
{"schema":{"type":"struct","fields":[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Timestamp","version":
:1,"field":"login_time"}],"optional":false,"name":"login"},"payload":{"username":"rajini",
"login_time":1621803287000}}
```

Jak widać, stare rekordy pokazują `NO_HEADERS`, ale nowe rekordy pokazują `MessageSource:mysql-1o-gin-connector`.



Obsługa błędów i kolejki niedostarczonych komunikatów

Transformacje to przykład konfiguracji, która nie jest charakterystyczna dla konkretnego konektora, ale może być używana w dowolnym z nich. Inną bardzo przydatną konfiguracją konektora, która może być używana w każdym konektorze ujęcia, jest `error.tolerance` — konektor można skonfigurować w taki sposób, aby po cichu porzucał uszkodzone komunikaty lub kierował je do specjalnego tematu zwanego „kolejką niedostarczonych komunikatów” (ang. *dead letter queue*). Więcej informacji na ten temat znajdziesz na blogu Confluenta w poście *Kafka Connect Deep Dive — Error Handling and Dead Letter Queues* (<https://oreil.ly/935hH>).

Jak działa framework Kafka Connect?

Aby zrozumieć, jak działa Kafka Connect, musisz poznać trzy podstawowe koncepcje i ich wzajemne relacje. Jak wyjaśniliśmy wcześniej i pokazaliśmy na przykładach, w celu skorzystania z frameworku Kafka Connect trzeba uruchomić klaster węzłów roboczych i utworzyć lub usunąć konektory. Dodatkową kwestią, o której dotąd nie pisaliśmy, jest obsługa danych przez konwertery, czyli komponenty konwertujące wiersze bazy danych MySQL na rekordy w formacie JSON, które konektor zapisał w Kafce.

Przyjrzyjmy się bliżej poszczególnym systemom i interakcjom między nimi.

Konektory i zadania

Wtyczki konektorów implementują interfejs Connector API, który składa się z dwóch części:

Konektory

Konektor jest odpowiedzialny za trzy ważne elementy:

- określanie liczby zadań uruchamianych dla konektora;
- decydowanie o przydzielaniu zadaniom pracy polegającej na kopiowaniu danych;
- pobieranie z węzłów roboczych konfiguracji dla zadań i przekazywanie ich dalej.

Konektor źródłowy JDBC połączy się np. z bazą danych, wykryje istniejące tablice do skopiowania i na ich podstawie określi liczbę potrzebnych zadań — wybierając niższą konfigurację `tasks.max` i liczbę tablic. Gdy zdecyduje, ile zadań zostanie uruchomionych, wygeneruje konfigurację dla każdego zadania, używając zarówno konfiguracji konektora (np. `connection.url`), jak i listy tablic przypisywanych do skopiowania poszczególnym zadaniom. Metoda `taskConfigs()` zwraca listę map (tzn. konfigurację dla każdego zadania, które chcemy uruchomić). Następnie węzły robocze uruchamiają zadania i przydzielają każdemu z nich unikatową konfigurację, by kopiowało określony podzbiór tablic z bazy danych. Należy pamiętać, że konektor zainicjowany za pośrednictwem interfejsu RESTful API może zostać uruchomiony na dowolnym węźle, a uruchamiane przez niego zadania mogą być również wykonywane na którymkolwiek z węzłów.

Zadania

Zadania są odpowiedzialne za przysyłanie danych do Kafki i pobieranie danych z Kafki. Wszystkie zadania są inicjowane przez otrzymanie kontekstu z węzła roboczego. Kontekst źródłowy zawiera obiekt, który umożliwia zadaniu źródłowemu przechowywanie przesunięć rekordów źródłowych (np. w konektorze plików przesunięcia są pozycjami w pliku, a w konektorze źródłowym JDBC przesunięcia mogą być kolumną znacznika czasu w tablicy). Kontekst dla konektora ujęcia obejmuje metody, które pozwalają mu kontrolować rekordy otrzymywane z Kafki — jest to wykorzystywane np. do stosowania mechanizmu ciśnienia zwrotnego, ponawiania prób i zewnętrznego przechowywania przesunięć w celu zapewnienia dostarczania dokładnie raz. Po zainicjowaniu zadania są uruchamiane za pomocą obiektu `Properties`, który zawiera konfigurację utworzoną przez Connector dla zadania. Zadania źródłowe odpytują zewnętrzny system i zwracają listy rekordów, które węzeł roboczy wysyła do brokerów Kafki. Zadania ujęcia otrzymują rekordy z Kafki za pośrednictwem węzła roboczego i są odpowiedzialne za zapisywanie tych rekordów w systemie zewnętrznym.

Węzły robocze

Węzły robocze frameworku Kafka Connect to procesy „kontenerowe”, które wykonują konektory i zadania. Są odpowiedzialne za obsługę żądań HTTP definiujących konektory i ich konfigurację, a także za przechowywanie konfiguracji konektorów w wewnętrznym temacie Kafki, inicjowanie konektorów i ich zadań oraz przekazywanie odpowiednich konfiguracji. Jeśli proces roboczy zostanie zatrzymany lub ulegnie awarii, dostrzegą to inne węzły robocze w klastrze Connecta (używając sygnału heartbeat z protokołu konsumentckiego Kafki) i ponownie przypiszą pozostawione konektory i zadania pozostałym procesom roboczym. Jeśli do klastra Connecta dołączy nowy proces roboczy, zauważą to pozostałe procesy robocze i przypiszą mu konektory lub zadania, aby zapewnić równomierne obciążenie wszystkich węzłów roboczych. Procesy robocze są ponadto odpowiedzialne za automatyczne zatwierdzanie w wewnętrznych tematach Kafki przesunięć zarówno dla konektorów źródłowych, jak i ujęciowych oraz za obsługę ponawiania prób, gdy zadania rzucają błędy.

Najlepszym sposobem na zrozumienie sposobu działania węzłów roboczych jest uświadomienie sobie, że konektory i zadania są odpowiedzialne za ten element integracji danych, który jest związany z „przenoszeniem danych”, podczas gdy węzły robocze są odpowiedzialne za RESTful API, zarządzanie konfiguracją, niezawodność, wysoką dostępność, skalowanie i równoważenie obciążenia.

Ta separacja zagadnień jest główną przewagą interfejsu Connect API nad klasycznymi interfejsami API konsument-producent. Doświadczeni programiści wiedzą, że pisanie kodu, który odczytuje dane z Kafki i wstawia je do bazy danych, zajmuje może dzień lub dwa, ale jeśli trzeba obsłużyć konfigurację, błędy, interfejsy RESTful API, monitorowanie, wdrażanie, skalowanie w górę i w dół oraz awarie, może to potrwać kilka miesięcy. Ponadto większość potoków integracji danych obejmuje więcej niż tylko jedno źródło lub system docelowy. Wyobraź sobie teraz wysiłek poświęcony na przygotowywanie na zamówienie kodu dla samej integracji bazy danych powtórzony wiele razy dla innych technologii. Jeżeli zaimplementujesz kopiowanie danych za pomocą konektora, podłączy się on do węzłów roboczych zajmujących się wieloma skomplikowanymi problemami operacyjnymi, którymi nie będziesz musiał się przejmować.

Konwertery i model danych Connecta

Ostatni element układanki Connect API to model danych konektora i konwertery. Connect API Kafki obejmuje interfejs programistyczny danych, który zawiera zarówno obiekty danych, jak i schemat opisujący te dane. Źródło JDBC odczytuje np. kolumnę z bazy danych i na podstawie typów danych kolumn zwracanych przez bazę danych konstruuje obiekt Connect Schema. Następnie używa schematu do zbudowania struktury (Struct) zawierającej wszystkie pola z rekordu bazy danych. Dla każdej kolumny przechowujemy jej nazwę i wartość. Każdy konektor źródłowy robi coś podobnego — odczytuje zdarzenie z systemu źródłowego i generuje parę Schema i Value. Konektory ujścia wykonują czynność odwrotną — pobierają parę Schema i Value, a następnie używają tego obiektu Schema do parsowania wartości i wstawiania ich do systemu docelowego.

Chociaż konektory źródłowe potrafią generować obiekty na podstawie interfejsu Data API, pozostaje pytanie, w jaki sposób węzły robocze Connecta zapisują te obiekty w Kafce. W tym miejscu na scenę wkraczają konwertery. Kiedy użytkownicy konfiguruje proces roboczy (lub konektor), wybierają konwerter do zapisywania danych w Kafce. Aktualnie dostępne opcje to typy proste, tablice bajtów, łańcuchy znaków, Avro, JSON, JSON Schema lub Protobufs. Konwerter JSON można skonfigurować w taki sposób, aby uwzględnił schemat w rekordzie wynikowym lub nie — dzięki czemu możemy obsługiwać zarówno dane strukturalne, jak i półstrukturalne. Gdy konektor zwraca rekord Data API węzłowi roboczemu, ten następnie używa skonfigurowanego konwertera do konwersji rekordu na obiekt Avry, obiekt JSON lub łańcuch znaków, a wynik jest zapisywany w Kafce.

Przeciwny proces zachodzi w przypadku konektorów ujścia. Gdy węzeł roboczy Connecta odczytuje rekord z Kafki, używa skonfigurowanego konwertera do konwersji rekordu z formatu Kafki (tj.: typów prostych, tablic bajtów, łańcuchów znaków, Avry, formatu JSON, JSON Schema lub Protobufs) na rekord interfejsu Data API Connect, a następnie przekazuje go do konektora ujścia, który wstawia go do systemu docelowego.

Umożliwia to interfejsowi Connect API obsługę różnych typów danych przechowywanych w Kafce, niezależnie od implementacji konektora (to znaczy, że z dowolnym typem rekordu może być używany dowolny konektor, o ile dostępny jest odpowiedni konwerter).

Zarządzanie przesunięciami

Zarządzanie przesunięciami jest jedną z usług pomocniczych wykonywanych przez węzły robocze dla konektorów (oprócz zarządzania wdrażaniem i konfiguracją za pośrednictwem interfejsu RESTful API). Chodzi o to, że konektory muszą wiedzieć, które dane już przetworzyły, a do przechowywania informacji o przetworzonych zdarzeniach mogą używać interfejsów API dostarczanych przez Kafkę.

W przypadku konektorów źródłowych oznacza to, że rekordy zwracane przez konektor procesom roboczym Connecta zawierają partycję logiczną i przesunięcie logiczne. Nie są to partycje i przesunięcia Kafki, ale raczej te partycje i przesunięcia, które są potrzebne w systemie źródłowym. W źródle plików partycją może być np. plik, a przesunięciem numer linii lub numer znaku w pliku. W źródle JDBC partycją może być tablica bazy danych, a przesunięciem identyfikator lub znacznik czasu rekordu w tablicy. Jedną z najważniejszych decyzji projektowych związanych z pisanem konektora źródłowego jest wybór dobrego sposobu partycjonowania danych w systemie źródłowym

i śledzenia przesunięć — wpływa to na poziom zrównoleglenia, jaki może osiągnąć konektor, i określa semantykę dostarczania, czyli „co najmniej raz” lub „dokładnie raz”.

Gdy konektor źródłowy zwraca listę rekordów, która zawiera źródłowe partycję i przesunięcie dla każdego rekordu, węzeł roboczy wysyła te rekordy do brokerów Kafki. Jeśli brokery potwierdzą pomyślne odebranie rekordów, węzeł roboczy zapisuje przesunięcia rekordów, które wysłał do Kafki. Umożliwia to konektorom po restarcie lub awarii rozpoczęcie przetwarzania zdarzeń od ostatnio zapisanego przesunięcia. Mechanizm przechowywania danych jest oparty na architekturze wtyczek i zazwyczaj jest tematem Kafki; nazwę tematu można kontrolować za pomocą konfiguracji `offset.storage.topic`. Connect wykorzystuje tematy Kafki również do przechowywania konfiguracji wszystkich utworzonych przez nas konektorów i statusu każdego konektora — używane są nazwy konfigurowane odpowiednio przez `config.storage.topic` i `status.storage.topic`.

Konektory ujęcia mają przeciwny, choć podobny przepływ pracy: odczytują rekordy Kafki, które mają już identyfikatory tematu, partycji i przesunięcia. Następnie wywołują metodę `put()`, która powinna zapisać te rekordy w systemie docelowym. Jeśli konektor zgłasza powodzenie, zatwierdza otrzymane przesunięcia z powrotem w Kafce, stosując standardowe konsumenckie metody zatwierdzania.

Śledzenie przesunięcia zapewniane przez sam framework powinno ułatwić programistom pisanie konektorów i zagwarantować pewien poziom spójnego zachowania podczas korzystania z różnych konektorów.

Alternatywy dla frameworku Kafka Connect

Omówiliśmy szczegółowo interfejs Connect API Kafki. Chociaż kochamy wygodę i niezawodność oferowane przez Connect API, nie jest to jedyna metoda wprowadzania danych do Kafki i pobierania ich z Kafki. Przyjrzyjmy się alternatywnym rozwiązaniom i przypadkom ich powszechnego stosowania.

Frameworki pobierania dla innych magazynów danych

Chociaż lubimy traktować Kafkę jako centrum wszechświata, nie każdy zgadza się z takim podejściem. Niektórzy budują większość swoich architektur danych wokół takich systemów jak Hadoop lub Elasticsearch. Te systemy mają własne narzędzia do konsumowania danych — Flume dla Hadoopa i Logstash lub Fluentd dla Elasticsearcha. Connect API Kafki zalecamy wtedy, gdy Kafka jest integralną częścią architektury, a celem jest podłączenie dużej liczby źródeł i ujęć danych. Jeśli tak naprawdę budujesz system skupiony wokół Hadoopa lub Elastica, a Kafka jest tylko jednym z wielu źródeł danych dla tego systemu, sensowne jest użycie Flume’a lub Logstasha.

Narzędzia ETL oparte na GUI

Staromodne systemy, takie jak Informatica, alternatywy udostępniane na licencji *open source*, jak Talend i Pentaho, a nawet nowsze alternatywy, np. Apache NiFi i StreamSets, obsługują Apache Kafkę zarówno jako źródło danych, jak i system docelowy. Jeżeli korzystasz już z tych systemów,

np. robisz wszystko za pomocą platformy Pentaho, niekoniecznie musi Cię interesować dodanie kolejnego systemu integracji danych tylko dla Kafki. Systemy te mają zastosowanie również wtedy, gdy do budowania potoków ETL używasz podejścia opartego na graficznym interfejsie użytkownika (GUI). Główną wadą tych systemów jest to, że są one zwykle budowane dla skomplikowanych przepływów pracy, więc jeśli chcesz jedynie wysyłać dane do Kafki i pobierać je z niej, będą nieco ciężkim i stosunkowo złożonym rozwiązaniem. Uważamy, że integracja danych powinna koncentrować się na wiernym dostarczaniu komunikatów w każdych warunkach, podczas gdy większość narzędzi ETL niepotrzebnie zwiększa złożoność.

Zachęcamy do potraktowania Kafki jako platformy, która może obsługiwać integrację danych (za pomocą Connecta), integrację aplikacji (przy użyciu producentów i konsumentów) oraz przetwarzanie strumieniowe. Kafka może być realnym zamiennikiem dla narzędzia ETL, które integruje tylko magazyny danych.

Frameworki przetwarzania strumieniowego

Prawie wszystkie frameworki przetwarzania strumieniowego umożliwiają odczytywanie zdarzeń z Kafki i zapisywanie ich w kilku innych systemach. Jeżeli Twój system docelowy jest obsługiwany i zamierzasz użyć określonego frameworku przetwarzania strumieniowego do przetwarzania zdarzeń z Kafki, rozsądne wydaje się użycie tego samego frameworku również do integracji danych. Często można w ten sposób zredukować liczbę etapów w przepływie pracy przetwarzania strumieniowego (nie ma potrzeby zapisywania przetworzonych zdarzeń w Kafce — wystarczy je odczytać i zapisać w innym systemie), chociaż trudniejsze może być wtedy rozwiązywanie takich problemów, jak utracone i uszkodzone komunikaty.

Podsumowanie

W tym rozdziale omówiliśmy wykorzystanie Kafki do integracji danych. Zaczęliśmy od powodów użycia Kafki do integracji danych i opisaliśmy ogólne zagadnienia związane z rozwiązaniami, które służą do integracji danych. Pokazaliśmy, dlaczego uważamy, że Kafka i jej interfejs Connect API są dobrym rozwiązaniem. Dalej podaliśmy kilka przykładów wykorzystania frameworku Kafka Connect w różnych scenariuszach, przyjrzelśmy się mechanizmowi działania Connecta, a następnie omówiliśmy kilka alternatyw dla frameworku Kafka Connect.

Niezależnie od tego, jakie rozwiązanie do integracji danych ostatecznie wybierzesz, najważniejszą funkcjonalnością będzie zawsze jego zdolność dostarczania wszystkich komunikatów we wszystkich przypadkach awaryjnych. Według nas Kafka Connect jest niezwykle niezawodny, gdyż opiera się na integracji ze sprawdzonymi funkcjonalnościami Kafki, ale ważne jest, żebyś testował wybierany system, tak jak robimy to my. Upewnij się, że wybrany system integracji danych może przetrwać zatrzymywanie procesów, awarie maszyn, opóźnienia sieci i wysokie obciążenia bez utraty komunikatów. W końcu systemy integracji danych mają tylko jedno zadanie — dostarczanie tych komunikatów.

Chociaż niezawodność jest zazwyczaj najważniejszym wymogiem przy integracji systemów danych, jest to oczywiście tylko jeden z wymogów. Przy wyborze systemu danych ważne jest, aby najpierw

zapoznać się z wymaganiami (przykłady znajdziesz w podrozdziale „Zagadnienia związane z budowaniem potoków danych”), a następnie upewnić się, że wybrany system je spełnia. Ponadto musisz jeszcze poznać to rozwiązanie do integracji danych na tyle dobrze, aby używać go w sposób spełniający Twoje wymagania. Nie wystarczy, że Kafka obsługuje semantykę „co najmniej raz”; musisz mieć pewność, że konfigurujesz ją nieprzypadkowo i w sposób, który w efekcie zagwarantuje całkowitą niezawodność.

Mirroring danych między klastrami

Do tej pory omawialiśmy przede wszystkim konfigurację, utrzymywanie i wykorzystanie pojedynczego klastra Kafki. W kilku scenariuszach architektura może jednak wymagać więcej niż jednego klastra.

W niektórych przypadkach klastry są całkowicie odseparowane. Należą do różnych działów lub do różnych przypadków użycia i nie ma powodu, by kopiować dane z jednego klastra do drugiego. Czasami różne umowy SLA lub obciążenia robocze utrudniają dostosowanie pojedynczego klastra w celu obsługi wielu przypadków użycia. Zdarzają się również sytuacje, w których obowiązują odmienne wymogi dotyczące bezpieczeństwa. Te przypadki użycia są dość proste — zarządzanie wieloma odrębnymi klastrami jest jak wielokrotne uruchamianie pojedynczego klastra.

W określonych scenariuszach klastry są współzależne, a administratorzy muszą stale kopiować dane między klastrami. W większości baz danych ciągle kopiowanie danych między serwerami bazodanowymi nazywane jest **replikacją**. Ponieważ użyliśmy tego określenia do opisanego ruchu danych między węzłami Kafki, które są częścią tego samego klastra, kopiowanie danych między klastrami Kafki nazwiemy **mirroringiem**. Wbudowany replikator międzyklastrowy platformy Apache Kafka to **MirrorMaker**.

W tym rozdziale omówimy międzyklastrowy mirroring całości lub części danych. Zaczniemy od opisanego kilku typowych przypadków użycia mirroringu między klastrami. Następnie przedstawimy kilka architektur, które są wykorzystywane do zaimplementowania tych przypadków użycia, i opiszemy zalety oraz wady poszczególnych wzorców architektonicznych. Potem przejdziemy do samego narzędzia MirrorMaker i pokażemy, jak go używać. Podzielimy się wskazówkami operacyjnymi dotyczącymi m.in. dostosowywania wdrożeń i wydajności. Na koniec wskażemy kilka alternatyw dla MirrorMakera.

Przypadki użycia dla mirroringu między klastrami

Poniżej zamieściliśmy listę przykładów zastosowania mirroringu międzyklastrowego.

Klastry regionalne i centralne

Czasami firma ma jedno centrum danych lub kilka w różnych regionach geograficznych, miastach lub na kontynentach. Każde centrum danych ma swój własny klaster Kafki. Niektóre

aplikacje mogą działać, komunikując się wyłącznie z klastrem lokalnym, ale inne wymagają danych z wielu centrów danych (w przeciwnym razie nie szukałbyś rozwiązań replikacji między centrami danych). W wielu przypadkach jest to wymogiem, ale klasycznym przykładem jest firma, która modyfikuje ceny na podstawie podaży i popytu. Może ona posiadać centrum danych w każdym mieście, w którym jest obecna, zbierając informacje o lokalnej podaży i popycie oraz odpowiednio dostosowując ceny. Wszystkie te informacje są wtedy mirrorowane w centralnym klastrze, w którym analitycy biznesowi mogą generować ogólnofirmowe raporty dotyczące przychodów.

Wysoka dostępność (HA) i odzyskiwanie sprawności po katastrofie (DR)

Aplikacje działają tylko na jednym klastrze Kafki i nie potrzebują danych z innych lokalizacji, ale obawiasz się, że z jakiegoś powodu cały klaster może stać się niedostępny. W celu zapewnienia redundancji chcesz mieć drugi klaster Kafki ze wszystkimi danymi przechowywanymi w pierwszym klastrze, by w nagłych przypadkach móc przekierować aplikacje do drugiego klastra i kontynuować pracę bez zakłóceń.

Zgodność z regulacjami

Firmy działające w wielu krajach mogą wymagać stosowania różnych konfiguracji i reguł, aby spełnić wymogi prawne i regulacyjne w każdym kraju. Niektóre zbiory danych mogą być np. przechowywane w oddzielnych klastrach ze ścisłą kontrolą dostępu, przy czym podzbiory danych mogą być replikowane do innych klastrów z szerszym dostępem. Aby zachować zgodność z przepisami, które regulują okres przechowywania danych w każdym regionie, zbiory danych mogą być przechowywane w klastrach w różnych regionach z odmiennymi konfiguracjami.

Migracje w chmurze

Obecnie wiele firm prowadzi działalność zarówno w lokalnym centrum danych, jak i u dostawcy chmury obliczeniowej. Często aplikacje są uruchamiane w kilku regionach dostawcy chmury w celu zapewnienia redundancji, a czasami korzysta się z wielu dostawców chmury. W takich przypadkach często w każdym lokalnym centrum danych i w każdym regionie chmury utrzymywany jest co najmniej jeden klaster Kafki. Klaster jest wykorzystywany przez aplikacje z poszczególnych centrów danych i regionów do efektywnego przesyłania danych między centrami danych. Jeśli nowa aplikacja zostanie wdrożona np. w chmurze, ale będzie wymagała pewnych danych, które są aktualizowane przez aplikacje działające w lokalnym centrum danych i przechowywane w lokalnej bazie danych, możesz użyć frameworku Kafka Connect do przechwytywania zmian bazy danych w lokalnym klastrze Kafki, a następnie przeprowadzić mirroring tych zmian do klastra Kafki w chmurze, gdzie może ich używać nowa aplikacja. Pomaga to kontrolować koszty ruchu między centrami danych oraz poprawić zarządzanie ruchem i poziom jego bezpieczeństwa.

Agregacja danych z klastrów brzegowych

Niektóre branże, takie jak handel detaliczny, telekomunikacja, transport czy opieka zdrowotna, generują dane z niewielkich urządzeń o ograniczonej łączności. Zagregowany klaster o wysokiej dostępności może być używany do obsługi analiz i innych przypadków użycia dla danych z dużej liczby klastrów brzegowych. Zmniejsza to wymagania dotyczące łączności, dostępności i trwałości w klastrach brzegowych o niewielkim śladzie cyfrowym, np. w razie

użycia IoT. Wysoce dostępny klastr zagregowany zapewnia ciągłość działania nawet wtedy, gdy klastry brzegowe są niedostępne, i upraszcza tworzenie aplikacji, które nie muszą mieć bezpośrednio do czynienia z dużą liczbą klastrów brzegowych z niestabilnymi sieciami.

Architektury wieloklastrowe

Skoro poznałeś już kilka przypadków użycia wymagających wielu klastrów Kafki, przyjrzyjmy się popularnym wzorcom architektonicznym, które z powodzeniem wykorzystujemy przy implementowaniu tych przypadków użycia. Zanim przejdziemy do architektury, omówimy pokrótce realia dotyczące komunikacji między centrami danych. Rozwiązania, które przedstawimy, mogą wydawać się zbyt skomplikowane, gdy nie uwzględnimy tego, że stanowią one kompromisy w obliczu określonych warunków sieciowych.

Realia komunikacji między centrami danych

Poniżej wymieniliśmy kilka kwestii, które należy wziąć pod uwagę, jeśli chodzi o komunikację między centrami danych.

Wysokie opóźnienia

Opóźnienia w komunikacji między dwoma klastrami Kafki zwiększają się wraz z odległością i ze wzrostem liczby przeskoków sieciowych między tymi dwoma klastrami.

Ograniczona przepustowość

Sieci rozległe (ang. *Wide Area Network* — WAN) mają zazwyczaj znacznie niższą dostępną przepustowość niż pojedyncze centra danych, a ta przepustowość może się zmieniać z minuty na minutę. Większe opóźnienia sprawiają ponadto, że wykorzystanie całej dostępnej przepustowości staje się trudniejsze.

Wyższe koszty

Niezależnie od tego, czy uruchamiasz Kafkę lokalnie czy w chmurze, komunikacja między klastrami wiąże się z wyższymi kosztami. Jest to częściowo spowodowane ograniczoną przepustowością, a jej zwiększenie może być zbyt kosztowne, a do tego dochodzą też opłaty, które dostawcy pobierają za przesyłanie danych między centrami danych, regionami i chmurami.

Brokery i klienci Apache Kafki zostały zaprojektowane, opracowane, przetestowane i wyregulowane w ramach pojedynczego centrum danych. Założyliśmy, że mamy małe opóźnienia i wysoką przepustowość między brokerami a klientami. Odzwierciedlają to domyślne limity czasu i rozmiary różnych buforów. Z tego powodu nie zaleca się (z wyjątkiem szczególnych przypadków, które omówimy później) instalowania części brokerów Kafki w jednym centrum danych, a pozostałych w drugim.

W większości przypadków najlepiej jest unikać produkowania danych do zdalnego centrum danych, a jeśli to robisz, musisz wziąć pod uwagę większe opóźnienia i możliwość wystąpienia większej liczby błędów sieciowych. Z błędami możesz sobie poradzić, zwiększając liczbę ponawiania prób przez producenty, a w celu obsługi większych opóźnień możesz zwiększyć rozmiar buforów, które przechowują rekordy między próbami ich wysłania.

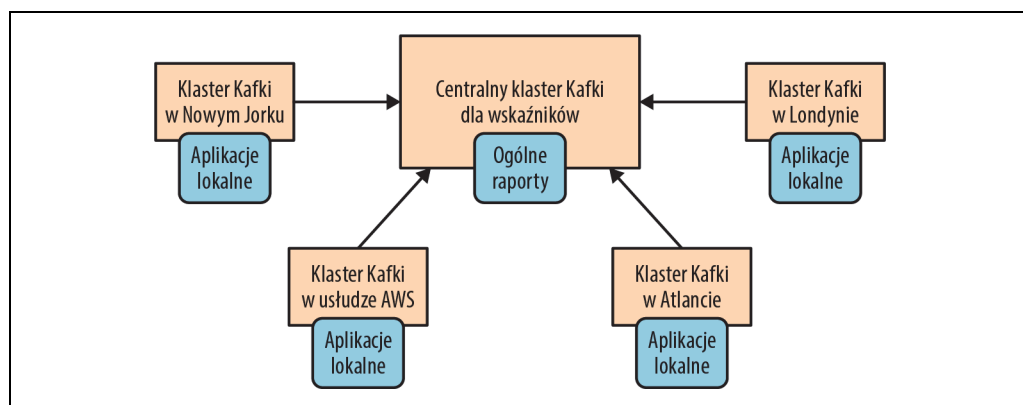
Jeżeli potrzebujemy jakiegokolwiek replikacji między klastrami i wykluczyliśmy komunikację między brokerami oraz między producentami a brokerami, musimy zapewnić komunikację między brokerami a konsumentami. Tak naprawdę jest to najbezpieczniejsza forma komunikacji między klastrami, bo w razie powstania partycji sieciowej, która uniemożliwia konsumentowi odczytywanie danych, rekordy pozostają bezpieczne w brokerach Kafki, dopóki komunikacja nie zostanie przywrócona i konsumenci nie będą mogli z powrotem ich odczytywać. Zjawisko powstawania partycji sieciowych nie powoduje ryzyka przypadkowej utraty danych. Ponieważ jednak przepustowość jest ograniczona, jeśli w jednym centrum danych znajduje się wiele aplikacji, które muszą odczytywać dane z brokerów Kafki w innym centrum danych, wolimy zainstalować klaster Kafki w każdym centrum danych i uruchomić jeden mirroring niezbędnych danych między nimi, niż mieć wiele aplikacji konsumujących te same dane za pośrednictwem sieci WAN.

Do tematu regulowania Kafki w celu komunikacji między centrami danych jeszcze wrócimy, natomiast większość architektur, które omówimy w kolejnych punktach, będzie kierować się następującymi zasadami:

- Nie mniej niż jeden klaster na centrum danych.
- Replikacja każdego zdarzenia dokładnie raz (z wyjątkiem powtarzania prób z powodu błędów) między każdą parą centrów danych.
- Jeśli to możliwe, konsumowanie ze zdalnego centrum danych zamiast produkowania do zdalnego centrum danych.

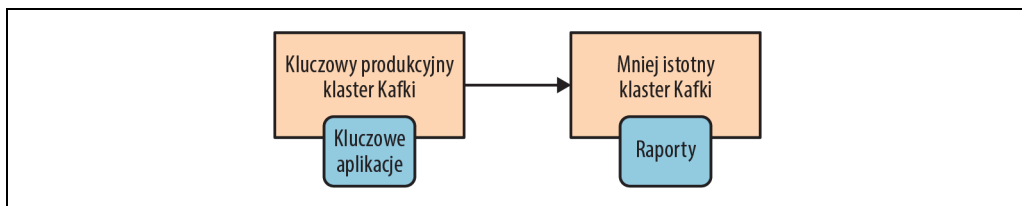
Architektura piasty i szprych

Architektura piasty i szprych (ang. *hub-and-spoke*) jest przeznaczona dla przypadku, gdy istnieje wiele lokalnych klastrów Kafki i jeden klaster centralny. Pokazaliśmy to na rysunku 10.1.



Rysunek 10.1. Architektura piasty i szprych

Istnieje również prostsza odmiana tej architektury z tylko dwoma klastrami: głównym i podrzędnym. Pokazaliśmy to na rysunku 10.2.



Rysunek 10.2. Prostsza wersja architektury piasty i szprych

Architektura ta jest używana, gdy dane są produkowane w wielu centrach danych, a niektóre konsumenci potrzebują dostępu do całego zbioru danych. Dzięki niej aplikacje w poszczególnych centrach danych mogą przetwarzać tylko dane lokalne dla konkretnego centrum danych. Nie zapewnia ona jednak dostępu do całego zbioru danych z każdego centrum danych.

Główna zaleta tej architektury polega na tym, że dane są zawsze produkowane do lokalnego centrum danych, a zdarzenia z każdego centrum danych są mirrorowane tylko raz — do centralnego centrum danych. Aplikacje przetwarzające dane z jednego centrum danych mogą być zlokalizowane w tym centrum. Aplikacje, które muszą przetwarzać dane z wielu centrów danych, będą znajdować się w centralnym centrum, gdzie mirrorowane są wszystkie zdarzenia. Ponieważ replikacja zawsze przebiega w jednym kierunku, a każdy konsument zawsze odczytuje z tego samego klastra, architektura ta jest łatwa do wdrożenia, skonfigurowania i monitorowania.

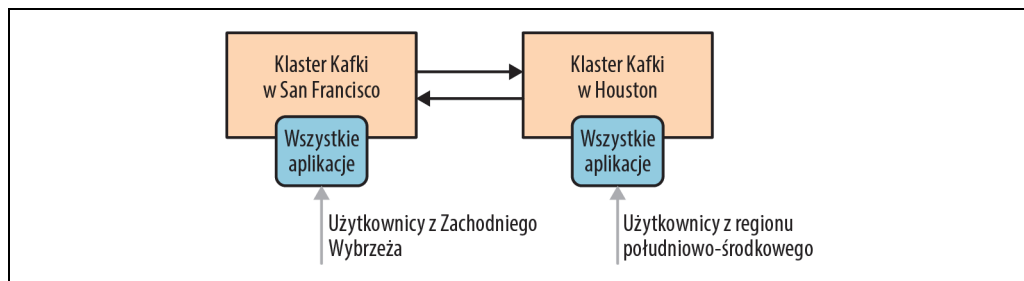
Podstawową wadą tej architektury jest bezpośredni efekt jej zalet i prostoty. Procesory z jednego regionalnego centrum danych nie mogą uzyskiwać dostępu do danych z drugiego centrum. Aby lepiej zrozumieć, dlaczego stanowi to ograniczenie, przyjrzyjmy się przykładowi tej architektury.

Żałóźmy, że jesteśmy dużym bankiem i mamy oddziały w wielu miastach. Powiedzmy, że postanawiamy przechowywać profile użytkowników i historię ich kont w klastrze Kafki w każdym mieście. Wszystkie te informacje replikujemy do centralnego klastra, który jest wykorzystywany do prowadzenia analityki biznesowej banku. Gdy użytkownicy łączą się ze stroną internetową banku lub stroną lokalnego oddziału, są kierowani, by wysłać zdarzenia do lokalnego klastra i z niego także odczytywać zdarzenia. Przypuśćmy jednak, że użytkownik odwiedza stronę oddziału z innego miasta. Ponieważ w tym mieście informacje o użytkowniku nie istnieją, oddział ten będzie zmuszony do interakcji ze zdalnym klastrzem (co jest niezalecane) lub nie będzie miał dostępu do informacji o użytkowniku (naprawdę żenujące). Dlatego stosowanie tego wzorca jest zwykle ograniczone tylko do tych części zbiorów danych, które są wyłączną własnością poszczególnych regionalnych centrów danych.

Przy implementacji tej architektury dla każdego regionalnego centrum danych potrzebny jest co najmniej jeden proces mirroringu w centralnym centrum. Proces ten będzie konsumował dane z poszczególnych klastrów regionalnych i produkował je do centralnego klastra. Jeżeli ten sam temat istnieje w wielu centrach danych, wszystkie zdarzenia z tego tematu można zapisywać do jednego tematu o tej samej nazwie w centralnym klastrze lub można zapisać zdarzenia z każdego centrum danych do osobnego tematu.

Architektura aktywny-aktywny

Architektura aktywny-aktywny (ang. *active-active*) jest używana, gdy co najmniej dwa centra danych współdzielą część lub całość danych, a każde centrum danych jest w stanie zarówno produkować, jak i konsumować zdarzenia. Pokazaliśmy to na rysunku 10.3.



Rysunek 10.3. Model architektury aktywny-aktywny

Podstawową zaletą tej architektury jest możliwość obsługi użytkowników z pobliskiego centrum danych, co zwykle zapewnia lepszą wydajność bez utraty funkcjonalności z powodu ograniczonej dostępności danych (jak widziałeś w architekturze piasty i szprych). Dodatkowe korzyści to redundancja i odporność na awarie. Ponieważ każde centrum danych ma wszystkie funkcjonalności, to gdy jedno centrum jest niedostępne, można skierować użytkowników do drugiego. Ten typ przełączenia awaryjnego wymaga jedynie sieciowego przekierowania użytkowników, co zazwyczaj jest najłatwiejszym i najbardziej przejrzystym sposobem.

Główna wada tej architektury to trudność w unikaniu konfliktów, gdy dane są asynchronicznie odczytywane i aktualizowane w wielu lokalizacjach. Obejmuje to wyzwania techniczne związane z mirroringiem zdarzeń — jak się upewnić, że to samo zdarzenie nie będzie np. mirrorowane w nieskończoność w tę i z powrotem? Ale co ważniejsze, wyzwaniem będzie utrzymanie spójności danych między dwoma centrami danych. Oto kilka przykładów trudności, jakie możesz napotkać:

- Jeżeli użytkownik wysłał zdarzenie do jednego centrum danych i odczytuje zdarzenia z drugiego, istnieje ryzyko, że zapisane przez niego zdarzenie nie dotarło jeszcze do drugiego centrum danych. Dla użytkownika będzie to wyglądało tak, jakby właśnie dodał do swojej listy życzeń jakąś książkę i kliknął listę życzeń, ale nie znalazł tam dodanej książki. Dlatego podczas korzystania z tej architektury programiści zwykle znajdują sposób, aby „przykleić” każdego użytkownika do określonego centrum danych i upewnić się, że przez większość czasu używa tego samego klastra (chyba że łączy się ze zdalnej lokalizacji lub centrum danych stało się niedostępne).
- Zdarzenie z jednego centrum danych informuje, że użytkownik zamówił książkę A, podczas gdy zdarzenie z mniej więcej tego samego czasu z drugiego centrum informuje, że ten sam użytkownik zamówił książkę B. Po mirroringu oba centra danych będą miały oba zdarzenia, możemy więc powiedzieć, że każde centrum danych będzie miało dwa sprzeczne zdarzenia. Aplikacje w obu centrach danych muszą wiedzieć, jak radzić sobie z tą sytuacją. Czy wybieramy jedno zdarzenie jako „poprawne”? Jeśli tak, potrzebujemy spójnych zasad wyboru

jednego zdarzenia, by aplikacje w obu centrach danych mogły dojść do tego samego wniosku. Czy może uznajemy, że oba zdarzenia są prawdziwe, wysyłamy użytkownikowi dwie książki, a inny dział zajmuje się zwrotami? W ten sposób konflikty rozwiązywał niegdyś Amazon, ale nie mogą tak robić np. organizacje zajmujące się obrotem akcjami. Metoda minimalizacji konfliktów i radzenia sobie z nimi, gdy występują, jest charakterystyczna dla każdego przypadku użycia. Zapamiętaj po prostu, że używając tej architektury, *będziesz mieć konflikty i będziesz musiał sobie z nimi radzić*.

Jeśli znajdziesz sposób radzenia sobie z wyzwaniami związanymi z asynchronicznymi operacjami odczytu i zapisu przeprowadzanymi na tym samym zbiorze danych z wielu lokalizacji, ta architektura jest szczególnie zalecana. To najbardziej skalowalna, odporna na awarie, elastyczna i opłacalna opcja, jaką znamy. Warto więc podjąć wysiłek, aby znaleźć rozwiązania pozwalające uniknąć cykli replikacji, zatrzymać użytkowników przez większość czasu w tym samym centrum danych i obsługiwać występujące konflikty.

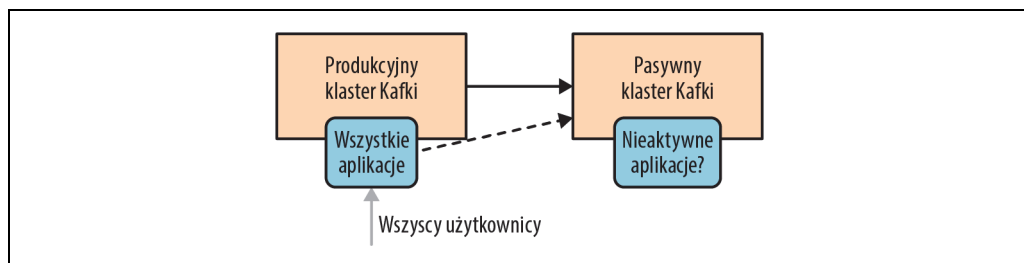
Jedno z wyzwań związanych z mirroringiem aktywny-aktywny, zwłaszcza w przypadku więcej niż dwóch centrów danych, polega na tym, że będziesz potrzebował zadań mirroringu dla każdej pary centrów i każdego kierunku. Obecnie wiele narzędzi mirroringowych może współdzielić procesy, używając np. tego samego procesu dla całego mirroringu do klastra docelowego.

Ponadto należy unikać pętli, w których to samo zdarzenie jest mirrorowane w nieskończoność w tę i z powrotem. Można to zrobić, nadając każdemu „tematowi logicznemu” osobne tematy dla poszczególnych centrów danych i unikając powielania tematów, które powstały w zdalnych centrach danych. Temat logiczny *users* będzie np. tematem *SF.users* w jednym centrum danych i tematem *NJ.users* w drugim centrum danych. Procesy mirroringu będą replikować temat *SF.users* z San Francisco do Nowego Jorku, a temat *NJ.users* z Nowego Jorku do San Francisco. W rezultacie poszczególne zdarzenia będą mirrorowane tylko raz, ale każde centrum danych będzie zawierało zarówno *SF.users*, jak i *NJ.users*, co oznacza, że będzie miało informacje dla wszystkich użytkowników. Jeśli konsumenci będą potrzebowały przetwarzać wszystkie zdarzenia użytkowników, będą musiały konsumować zdarzenia z **.users*. Innym sposobem jest potraktowanie tej konfiguracji jako osobnych przestrzeni nazw dla poszczególnych centrów — każda przestrzeń nazw będzie zawierać wszystkie tematy dla konkretnego centrum danych. W naszym przykładzie mielibyśmy przestrzenie nazw *NJ* i *SF*. Niektóre narzędzia do mirroringu, takie jak MirrorMaker, zapobiegają cykлом replikacji przy użyciu podobnej konwencji nazewnictwa.

Nagłówki rekordów wprowadzone w platformie Apache Kafka w wersji 0.11.0 umożliwiają znakovanie zdarzeń informacją o pochodzeniu z określonego centrum danych. Dane z nagłówków mogą być wykorzystywane również w celu uniknięcia niekończących się pętli mirroringu i umożliwienia osobnego przetwarzania zdarzeń z różnych centrów danych. Tę funkcjonalność można zaimplementować także przez użycie dla wartości rekordów ustrukturyzowanego formatu danych (naszym ulubionym przykładem jest Avro) i wykorzystanie tego do zawarcia znaczników i nagłówków w samym zdarzeniu. Wymaga to jednak dodatkowego wysiłku podczas mirroringu, ponieważ żadne z istniejących narzędzi do mirroringu nie będzie obsługiwać Twojego konkretnego formatu nagłówka.

Architektura aktywny-pasywny

W niektórych przypadkach jedynym wymogiem dla uruchamiania wielu klastrów jest obsługa pewnego rodzaju scenariusza katastrofy. Możesz mieć np. dwa klastry w tym samym centrum danych. Do wszystkich aplikacji stosujesz jeden klaster, ale potrzebujesz drugiego, zawierającego (prawie) wszystkie zdarzenia z oryginalnego klastra, którego będziesz mógł użyć, jeśli oryginalny klaster stanie się całkowicie niedostępny. Możesz też potrzebować np. geograficznej odporności na awarie. Załóżmy, że Twoja cała firma do działania wykorzystuje centrum danych znajdujące się w Kalifornii, ale potrzebujesz drugiego centrum w Teksasie, które zwykle będzie praktycznie bezczynne, ale będziesz mógł użyć go w przypadku trzęsienia ziemi. Centrum danych w Teksasie prawdopodobnie będzie miało nieaktywną („zimną”) kopię wszystkich aplikacji, które administratorzy będą mogli uruchomić w nagłych przypadkach, i będą one korzystać z drugiego klastra, jak pokazaliśmy na rysunku 10.4. Jest to często wymóg prawny, a nie coś, co firma sama planuje zrobić, ale i tak trzeba być przygotowanym.



Rysunek 10.4. Architektura aktywny-pasywny

Korzyści tej architektury to prostota konfiguracji i fakt, że można ją zastosować w prawie każdym przypadku użycia. Wystarczy zainstalować drugi klaster i skonfigurować proces mirroringu, który przesyła strumieniowo wszystkie zdarzenia z jednego klastra do drugiego. Nie trzeba przejmować się dostępem do danych, rozwiązywaniem konfliktów i innymi zawiłościami architektonicznymi.

Wady to marnotrawstwo dobrego klastra oraz to, że przełączanie awaryjne między klastrami Kafki jest w rzeczywistości znacznie trudniejsze, niż się wydaje. Najważniejsze jest to, że obecnie w Kafce nie jest możliwe awaryjne przełączenie klastrów bez utraty danych lub powielania zdarzeń, a często mamy do czynienia z jednym i drugim. Można to zminimalizować, ale nigdy nie da się tego całkowicie wyeliminować.

Powinno być oczywiste, że klaster, który nie robi nic poza oczekiwaniem na katastrofę, jest stratą zasobów. Ponieważ katastrofy są (lub powinny być) rzadkością, przez większość czasu obserwujemy całkowicie bezczynny klaster maszyn. Niektóre organizacje próbują walczyć z tym problemem, znacznie zmniejszając rozmiary klastra odzyskiwania sprawności po katastrofie (ang. *Disaster Recovery* — DR) w stosunku do rozmiarów klastra produkcyjnego. To jednak ryzykowna decyzja, ponieważ nie można być pewnym, że ten minimalnej wielkości klaster wytrzyma w nagłych wypadkach. Inne organizacje wolą, aby klaster DR był przydatny podczas zdarzeń niepowodujących katastrof, przenosząc na niego niektóre obciążenia robocze „tylko do odczytu”, co oznacza, że tak naprawdę uruchamiają niewielką wersję architektury piasty i szprych z pojedynczą szprychą.

Poważniejszym problemem jest w Kafce przełączenie awaryjne na klaster DR.

Przede wszystkim należy podkreślić, że niezależnie od wybranej metody przełączania awaryjnego Twój zespół inżynierów SRE musi to regularnie ćwiczyć. Plan, który działa dzisiaj, może przestać działać po uaktualnieniu albo istniejące narzędzia mogą stać się przestarzałe, gdy pojawiają się nowe przypadki użycia. Raz na kwartał to zazwyczaj absolutne minimum dla przećwiczenia procesu przełączania awaryjnego. Silne zespoły SRE ćwiczą znacznie częściej. Słynna usługa Chaos Monkey Netfliksa, która losowo powoduje katastrofy, działa w sposób ekstremalny — każdy dzień może okazać się dniem próby dla przełączania awaryjnego.

Zobaczmy, co wiąże się z przełączaniem awaryjnym.

Planowanie odzyskiwania sprawności po katastrofie

Podczas planowania usuwania skutków katastrofy ważne jest uwzględnienie dwóch kluczowych wskaźników. Zakładany czas odzyskiwania sprawności (ang. *Recovery Time Objective* — RTO) określa maksymalny czas na wznowienie działania wszystkich usług po wystąpieniu katastrofy. Zakładany punkt odzyskiwania sprawności (ang. *Recovery Point Objective* — RPO) określa maksymalny czas, przez jaki w wyniku katastrofy dane mogą pozostawać niedostępne. Im niższa wartość RTO, tym ważniejsze jest unikanie ręcznych procesów i restartów aplikacji, ponieważ bardzo krótkie RTO można osiągnąć tylko przy automatycznym przełączaniu awaryjnym. Niska wartość RPO wymaga mirroringu w czasie zbliżonym do rzeczywistego z małymi opóźnieniami, a RPO=0 wymaga replikacji synchronicznej.

Utrata danych i występowanie niespójności w przypadku nieplanowanego przełączania awaryjnego

Ponieważ wszystkie rozwiązania mirroringowe Kafki są asynchroniczne (rozwiązanie synchroniczne omówimy w następnym punkcie), klaster DR nie będzie miał najnowszych komunikatów z głównego klastra. Należy zawsze monitorować, jak daleko w tyle pozostaje klaster DR, i nigdy nie pozwalać na powstawanie zbyt dużego opóźnienia. Jednak w zajęтым systemie należy oczekiwać, że klaster DR będzie kilkaset, a nawet kilka tysięcy komunikatów za głównym. Jeśli klaster Kafki obsługuje milion komunikatów na sekundę, a opóźnienie między klastrem głównym a klastrem DR wynosi 5 milisekund, to w najlepszym przypadku klaster DR znajdzie się 5000 komunikatów za głównym. Przygotuj się więc, że nieplanowane przełączenie awaryjne będzie się wiązało z pewną utratą danych. W przypadku planowanego przełączania awaryjnego można zatrzymać główny klaster i przed przełączeniem aplikacji do klastra DR poczekać, aż proces mirroringu zreplikuje pozostałe komunikaty, dzięki czemu można uniknąć utraty tych danych. Gdy dojdzie do nieplanowanego przełączania awaryjnego i stracisz kilka tysięcy komunikatów, pamiętaj, że aktualnie rozwiązania mirroringowe nie obsługują transakcji, co oznacza, że jeśli niektóre zdarzenia w różnych tematach są ze sobą powiązane (np. sprzedaż i pozycje na fakturze), jedno z nich mogą zdążyć dotrzeć do lokalizacji DR przed przełączeniem awaryjnym, a inne nie. Po przełączeniu do klastra DR aplikacje będą musiały obsłużyć jakąś pozycję z faktury bez odpowiadającej jej sprzedaży.

Przesunięcie startu dla aplikacji po przełączeniu awaryjnym

Jednym z trudnych zadań związanych z przełączaniem awaryjnym do innego klastra jest upewnienie się, że aplikacje wiedzą, w którym miejscu rozpocząć konsumowanie danych. Istnieje kilka typowych rozwiązań tego problemu. Niektóre są proste, ale mogą powodować dodatkową utratę danych lub powielanie przetwarzania, inne są bardziej skomplikowane, ale minimalizują dodatkową utratę danych i ponowne przetwarzanie. Przyjrzymy się kilku z nich.

Automatyczne resetowanie przesunięcia

Można konfigurować zachowanie konsumentów Apache Kafki, które nie mają wcześniej zatwierdzonego przesunięcia — zaczynają wtedy odczytywanie od początku partycji lub od końca. Jeśli z jakiegoś powodu nie mirrorujesz tych przesunięć w ramach planu DR, musisz wybrać jedną z tych opcji. Zaczynij odczytywanie od początku dostępnych danych i obsługuj dużą liczbę duplikatów lub przeskocz do końca i pomij nieznaną (i miejmy nadzieję, że niewielką) liczbę zdarzeń. Jeżeli aplikacja bez większych problemów obsługuje duplikaty lub brak niektórych danych, wskazuje to, która opcja jest zdecydowanie najłatwiejsza. Ze względu na prostotę popularną metodą przełączania awaryjnego jest przeskoczenie do końca tematu.

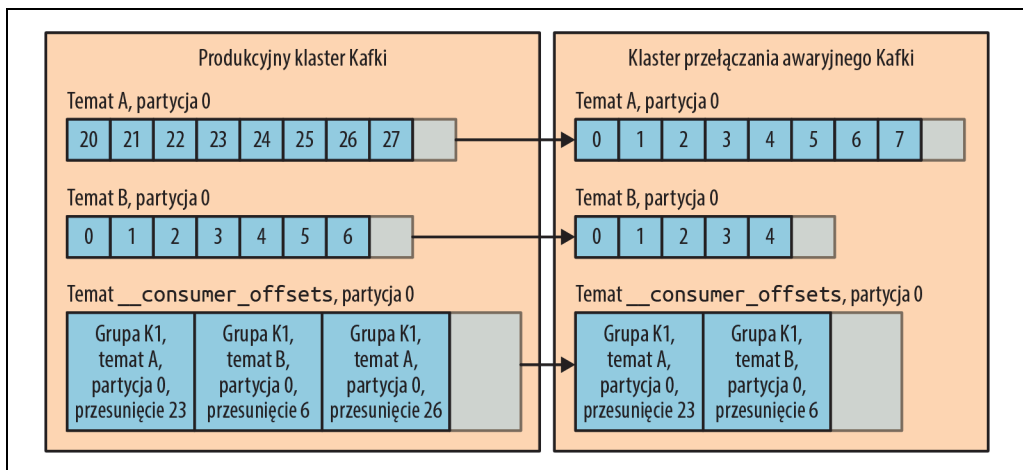
Replikacja tematu przesunięć

W Kafce od wersji 0.9.0 konsumenci zatwierdzają swoje przesunięcia w specjalnym temacie `__consumer_offsets`. Jeżeli mirrorujesz ten temat do klastra DR, gdy konsumenci zaczną konsumować z tego klastra DR, będą mogły pobrać swoje stare przesunięcia i kontynuować od miejsca, w którym przerwały. Jest to proste, ale istnieje długa lista związanych z tym zastrzeżeń.

Po pierwsze, nie ma gwarancji, że przesunięcia w klastrze głównym będą zgodne z przesunięciami w klastrze podrzędnym. Założmy, że w klastrze głównym przechowujesz dane tylko przez 3 dni, a mirroring tematu zaczynasz po tygodniu od jego utworzenia. W tym przypadku pierwszym przesunięciem dostępnym w klastrze głównym może być 57 000 000 (starsze zdarzenia pochodziły z pierwszych 4 dni i zostały już usunięte), ale pierwszym przesunięciem w klastrze DR będzie 0. Tak więc konsument, który spróbuje odczytać z klastra DR przesunięcie 57 000 003 (ponieważ będzie to jego następne zdarzenie do odczytania), nie zdoła tego zrobić.

Po drugie, nawet jeśli rozpoczynasz mirroring od razu po utworzeniu tematu, a zarówno temat klastra głównego, jak i temat klastra DR zaczynają się od 0, ponawianie prób przez producenty może powodować rozbieżności. Pod koniec tego rozdziału omówimy alternatywne rozwiązanie do mirroringu, które zachowuje przesunięcia między klastrami głównym i DR.

Po trzecie, nawet jeśli przesunięcia zostaną doskonale zachowane, ze względu na opóźnienie między klastrami głównym i DR oraz z uwagi na to, że aktualnie rozwiązania mirroringowe nie obsługują transakcji, przesunięcie zatwierdzone przez konsumenta Kafki może pojawić się przed lub za rekordem z tym przesunięciem. Konsument, który zostanie przełączony awaryjnie, może znaleźć zatwierdzone przesunięcia bez odpowiadających im rekordów. Może się również okazać, że ostatnie zatwierdzone przesunięcie w lokalizacji DR jest starsze niż ostatnie zatwierdzone przesunięcie w lokalizacji głównej, jak pokazaliśmy na rysunku 10.5.



Rysunek 10.5. Przełączanie awaryjne powoduje powstawanie przesunięć bez odpowiadających im rekordów

W takich przypadkach konieczne jest akceptowanie pewnej liczby duplikatów, jeśli ostatnie zatwierdzone przesunięcie w lokalizacji DR jest starsze niż to zatwierdzone w lokalizacji głównej lub jeśli przesunięcia w rekordach w lokalizacji DR wyprzedzają te z lokalizacji głównej z powodu ponawiania prób. Musisz również opracować sposób postępowania w sytuacjach, w których ostatnie zatwierdzone przesunięcie w lokalizacji DR nie ma odpowiadającego mu rekordu: zaczynasz przetwarzanie od początku tematu czy przeskakujesz do końca?

Jak widać, takie podejście ma swoje ograniczenia. Tak czy inaczej ta opcja pozwala na przełączanie awaryjne do klastra DR przy zmniejszonej liczbie zduplikowanych lub brakujących zdażeń w porównaniu z innymi podejściami, a jednocześnie jest łatwa do zaimplementowania.

Przełączanie awaryjne oparte na czasie

Od wersji 0.10.0 każdy komunikat zawiera znacznik czasu wskazujący godzinę wysłania komunikatu do Kafka. Od wersji 0.10.1.0 brokery zawierają indeks oraz API do wyszukiwania przesunięć według znacznika czasu. Dlatego jeśli w przypadku przełączenia awaryjnego do klastra DR będziesz wiedział, że Twój problem rozpoczął się np. o 4:05, będziesz mógł poinstruować konsumenty, by zaczęły przetwarzać dane od 4:03. Będzie trochę duplikatów z tych dwóch minut, ale prawdopodobnie jest to lepsze niż inne dostępne opcje, a zachowanie jest znacznie łatwiejsze do wyjaśnienia personelowi firmy — „cofnęliśmy się do 4:03” brzmi lepiej niż „przełączyliśmy się awaryjnie do tego, co być może jest ostatnim zatwierdzonym przesunięciem”. Stąd często bywa to dobry kompromis. Pozostaje tylko pytanie: jak poinstruować konsumenty, aby zaczęły przetwarzać dane od 4:03?

Jedną z możliwości jest wbudowanie tego bezpośrednio w aplikację w postaci konfigurowalnej przez użytkownika opcji określania czasu rozpoczynania dla aplikacji. Jeśli ta opcja zostanie skonfigurowana, aplikacja będzie mogła korzystać z nowych interfejsów API w celu pobierania przesunięcia na podstawie czasu, wyszukiwania tego czasu i rozpoczynania przetwarzania od właściwego punktu z zatwierdzaniem przesunięć jak zwykle.

To świetna opcja, jeżeli od razu napisałeś wszystkie swoje aplikacje w ten sposób. A co zrobić w przeciwnym razie? Apache Kafka udostępnia narzędzie `kafka-consumer-groups` do

resetowania przesunięć na podstawie różnych opcji, w tym resetowanie oparte na znaczniku czasu dodane w wersji 0.11.0. W momencie uruchamiania tego typu narzędzia grupa konsumentów powinna zostać zatrzymana, a po zakończeniu jego działania ponownie uruchomiona. Poniższe polecenie resetuje np. przesunięcia konsumentów do konkretnego czasu dla wszystkich tematów należących do określonej grupy:

```
bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-offsets --all-topics --group my-group --to-datetime 2021-03-31T04:03:00.000 --execute
```

Opcja ta jest zalecana we wdrożeniach, które muszą gwarantować określony poziom pewności podczas przełączania awaryjnego.

Translacja przesunięć

Jeśli chodzi o mirroring tematu przesunięć, jednym z największych wyzwań jest to, że przesunięcia w klastrach głównym i DR mogą się różnić. W przeszłości niektóre organizacje korzystały z zewnętrznego magazynu danych, takiego jak Apache Cassandra, do przechowywania mapowania przesunięć między poszczególnymi klastrami. Za każdym razem, gdy do klastra DR było produkowane zdarzenie, oba przesunięcia były wysyłane do zewnętrznego magazynu danych przez narzędzie do mirroringu, gdy przesunięcia się różniły. Obecnie rozwiązania mirroringowe, w tym MirrorMaker, do przechowywania metadanych translacji przesunięć używają tematu Kafki. Przesunięcia są zapisywane za każdym razem, gdy zmienia się różnica między tymi dwoma przesunięciami. Jeżeli przesunięcie 495 na klastrze głównym zostanie zmapowane na przesunięcie 500 na klastrze DR, to w zewnętrznym magazynie danych lub temacie translacji przesunięć zarejestrujemy (495,500). Jeśli później ta różnica zmieni się z powodu duplikatów i przesunięcie 596 zostanie zmapowane na 600, wówczas zarejestrujemy nowe mapowanie (596,600). Nie ma potrzeby zapisywania wszystkich mapowań przesunięć między 495 a 596. Zakładamy po prostu, że różnica pozostaje taka sama, i tak np. przesunięcie 550 w klastrze głównym będzie mapowane na 555 w klastrze DR. Gdy nastąpi przełączenie awaryjne, to zamiast mapować przesunięcia na znaczniki czasu (które są zawsze nieco niedokładne), mapujemy przesunięcia klastra głównego na przesunięcia klastra DR i z nich korzystamy. Aby zmusić konsumenty do rozpoczęcia korzystania z nowych przesunięć z mapowania, można wykorzystać jedną z dwóch wymienionych wcześniej technik. Nadal pozostaje problem z zatwierdzeniami przesunięć, które dotarły przed samymi rekordami, i zatwierdzeniami przesunięć, które nie zostały zreplikowane w lokalizacji DR na czas, ale mapowanie pozwala obsłużyć przynajmniej niektóre przypadki użycia.

Po przełączeniu awaryjnym

Powiedzmy, że przełączenie awaryjne się powiodło. Wszystko działa dobrze na klastrze DR. Teraz musimy zrobić coś z klastrzem głównym. Może przekształcić go np. w klaster DR?

Prosta modyfikacja procesów mirroringu, aby odwracały swój kierunek i rozpoczynały replikację z nowego klastra głównego do starego, jest kusząca. Rodzi to jednak dwa ważne pytania:

- Skąd będziemy wiedzieć, gdzie zacząć mirroring? Musimy rozwiązać ten sam problem, który mamy ze wszystkimi naszymi konsumentami w odniesieniu do aplikacji mirroringu. Pamiętaj także, że wszystkie nasze rozwiązania mają przypadki, w których tworzone są duplikaty lub pomijane dane — czasami jedno i drugie.

- Ponadto, z powodów omówionych wcześniej, pierwotny klaster główny prawdopodobnie będzie miał zdarzenia, których nie ma klaster DR. Jeśli zaczniesz mirrorować nowe dane w drugim kierunku, dodatkowa historia pozostanie i te dwa klastry będą niespójne.

Z tego powodu dla scenariuszy, w których gwarancje spójności i kolejności mają kluczowe znaczenie, najprostszym rozwiązaniem jest najpierw „zeskrobanie” pierwotnego klastra — usunięcie wszystkich danych i zatwierdzonych przesunięć — a następnie rozpoczęcie mirroringu z nowego klastra głównego do nowego klastra DR. Zapewnia to czyste konto, które jest identyczne z nowym klastrem głównym.

Kilka słów na temat wykrywania klastrów

Jedną z ważnych kwestii, które należy wziąć pod uwagę podczas planowania klastra rezerwowego, jest to, że w przypadku przełączania awaryjnego aplikacje będą musiały wiedzieć, jak rozpocząć komunikację z tym klastrem. Jeśli nazwy brokerów klastra głównego zakodowałeś na sztywno we właściwościach producentów i konsumentów, będzie to trudne. Większość organizacji zachowuje prostotę i tworzy nazwę DNS, która zwykle wskazuje główne brokery. W przypadku awarii nazwę DNS można przekierować na klaster rezerwowi. Usługa wykrywania (DNS lub inna) nie musi obejmować wszystkich brokerów — aby uzyskać metadane dotyczące klastra i wykryć pozostałe brokery, klienci Kafki muszą mieć dostęp tylko do jednego brokera. Dlatego zazwyczaj wystarczy uwzględnienie jedynie trzech brokerów. Niezależnie od metody wykrywania większość scenariuszy przełączania awaryjnego wymaga zrestartowania aplikacji konsumentckich po przełączeniu awaryjnym, by mogły znaleźć nowe przesunięcia, od których muszą zacząć konsumowanie zdarzeń. Aby zautomatyzowane przełączanie awaryjne bez restartowania aplikacji osiągnęło bardzo krótkie RTO, logika przełączania awaryjnego powinna być wbudowana w aplikacje klienckie.

Klastry rozciągnięte

Architektury aktywny-pasywny służą do ochrony biznesu przed awarią klastra Kafki przez przełączenie aplikacji na komunikowanie się z innym klastrem. Klastry rozciągnięte (ang. *stretched cluster*) mają na celu ochronę klastra Kafki przed awarią podczas postoju centrum danych. Osiąga się to przez zainstalowanie jednego klastra Kafki w wielu centrach danych.

Klastry rozciągnięte różnią się zasadniczo od innych scenariuszy z wieloma centrami danych. Przede wszystkim nie mamy do czynienia z kilkoma klastremi — to tylko jeden klaster. W rezultacie nie potrzebujemy procesu mirroringu, aby synchronizować dwa klastry. Normalny mechanizm replikacji Kafki jest używany jak zwykle do synchronizowania wszystkich brokerów w klastrze. Ta konfiguracja może obejmować replikację synchroniczną. Producenty zazwyczaj otrzymują potwierdzenie od brokera Kafki po tym, jak komunikat zostanie pomyślnie wysłany do Kafki. W przypadku klastra rozciągniętego możemy zastosować taką konfigurację, aby potwierdzenie było wysyłane po pomyślnym zapisaniu komunikatów w brokerach Kafki w dwóch centrach danych. Wymaga to użycia definicji szafy rackowej w celu upewnienia się, że każda partycja ma repliki w wielu centrach danych, a ponadto zastosowania ustawień `min.insync.replicas` i `acks=all`, które zagwarantują, że każdy zapis będzie potwierdzany z co najmniej dwóch centrów danych. Od wersji 2.4.0 brokery można skonfigurować również w taki sposób, aby umożliwić konsumentom pobieranie z najbliższej

repliki przy użyciu definicji szafy rackowej. Brokery dopasowują swój rack do racka konsumenta, by znaleźć najbardziej aktualną replikę lokalną, wracając do lidera, jeżeli odpowiednia replika lokalna nie jest dostępna. Konsumenty pobierające dane z replik podrzędnych w lokalnym centrum danych uzyskują wyższą przepustowość, mniejsze opóźnienia i niższe koszty dzięki ograniczeniu ruchu między centrami danych.

Jedną z zalet tej architektury jest replikacja synchroniczna — niektóre rodzaje działalności wymagają, aby ich lokalizacja DR była zawsze w 100% zsynchronizowana z lokalizacją główną. Jest to często wymóg prawny i jest stosowany do każdego magazynu danych w całej firmie, również do Kafki. Kolejną zaletą jest to, że wykorzystywane są oba centra danych i wszystkie brokery w klastrze. Nie ma marnotrawstwa, jakie widziałeś w architekturach aktywny-pasywny.

Architektura nie chroni przed wszystkimi rodzajami katastrof. Zabezpiecza tylko przed awariami centrów danych, ale nie przed awariami aplikacji lub Kafki. Ograniczona jest również złożoność operacyjna. Ta architektura wymaga fizycznej infrastruktury, którą nie wszystkie firmy mogą zapewnić.

Użycie tej architektury jest możliwe, jeśli można zainstalować Kafkę (i ZooKeeper) w co najmniej trzech centrach danych, pomiędzy którymi gwarantowane są wysoka przepustowość i niskie opóźnienia. Można to zrobić, jeśli firma posiada trzy budynki na tej samej ulicy lub korzysta z trzech stref dostępności w jednym regionie dostawcy chmury, co jest częściej spotykane.

Trzy centra danych są ważne dlatego, że ZooKeeper wymaga nieparzystej liczby węzłów w klastrze i pozostanie dostępny, jeżeli dostępna będzie większość węzłów. Przy dwóch centrach danych i nieparzystej liczbie węzłów jedno centrum danych zawsze będzie decydować o większości, co oznacza, że jeśli to centrum będzie niedostępne, także ZooKeeper będzie niedostępny, podobnie jak Kafka. Dzięki trzem centrům danych można łatwo alokować węzły, aby pojedyncze centrum danych nie decydowało o większości. Dlatego jeśli niedostępne będzie jedno centrum danych, większość węzłów będzie znajdowała się w pozostałych dwóch centrach, a klaster ZooKeepera pozostanie dostępny, a więc Kafka również.



Architektura 2,5 DC

Popularnym modelem dla klastrów rozciągniętych jest architektura 2,5 DC (centrum danych) z Kafką i ZooKeeperem działającymi w dwóch centrach danych i trzecim „połowicznym” centrum danych z jednym węzłem ZooKeepera, aby zapewnić kworum w przypadku awarii centrum danych.

Możliwe jest uruchomienie ZooKeepera i Kafki w dwóch centrach danych za pomocą konfiguracji grup ZooKeepera, która umożliwia ręczne przełączanie awaryjne między dwoma centrami danych. Jednak ta konfiguracja jest rzadkością.

MirrorMaker platformy Apache Kafki

Apache Kafka zawiera narzędzie o nazwie MirrorMaker, służące do mirroringu danych między dwoma centrami danych. Wczesne wersje MirrorMakera do odczytywania danych ze zbioru tematów źródłowych wykorzystywały kolekcję konsumentów, które były członkami grupy konsumentów,

a do wysyłania tych zdarzeń do klastra docelowego używały współdzielonego producenta Kafki w każdym procesie MirrorMakera. Chociaż w niektórych scenariuszach było to wystarczające do mirroringowania danych między klastrami, występowało kilka problemów, w szczególności skoki opóźnień związane ze zmianami w konfiguracji, a dodawanie nowych tematów skutkowało uruchamianiem procesu równoważenia obciążenia, który wszystko zatrzymywał. MirrorMaker 2.0 to rozwiązanie mirroringowe nowej generacji dla platformy Apache Kafka, które opiera się na frameworku Kafka Connect i jest pozbawione wielu wad swojego poprzednika. Złożone topologie można łatwo skonfigurować tak, aby obsługiwały szeroki zakres przypadków użycia, do których należą m.in.: przywracanie sprawności po katastrofie, tworzenie kopii zapasowych, migracja oraz agregacja danych.

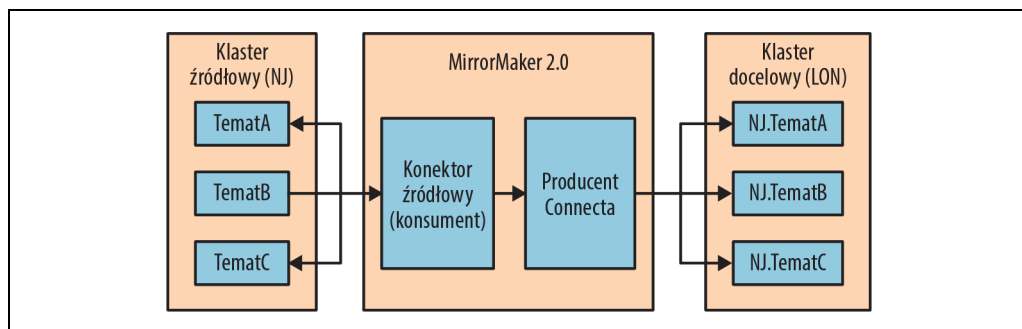


Kilka słów na temat MirrorMakera

MirrorMaker wydaje się bardzo prosty, ponieważ jednak staraliśmy się być bardzo wydajni i zbliżyć się do dostarczania dokładnie raz, okazało się, że poprawna implementacja była trudna. MirrorMaker był wielokrotnie przepisywany. Opis i szczegóły przedstawione w tym podrozdziale dotyczą MirrorMakera 2.0, który został wprowadzony w wersji 2.4.0.

MirrorMaker używa konektora źródłowego do konsumowania danych z innego klastra Kafki, a nie z bazy danych. Zastosowanie frameworku Kafka Connect minimalizuje koszty administracyjne dla pracownych działów IT w przedsiębiorstwach. Jak zapewne pamiętasz z opisu architektury frameworku Kafka Connect z rozdziału 9., każdy konektor rozdziela pracę między konfigurowalną liczbę zadań. W MirrorMakerze każde zadanie jest parą składającą się z konsumenta i producenta. Framework Connect przypisuje te zadania zgodnie z zapotrzebowaniem do różnych węzłów roboczych Connecta — stąd możesz mieć wiele zadań na jednym serwerze lub zadania rozłożone na wiele serwerów. Zastępuje to ręczną pracę polegającą na ustaleniu, ile strumieni MirrorMakera powinno zostać uruchomionych na instancję oraz ile instancji na maszynę. Connect posiada ponadto interfejs RESTful API do centralnego zarządzania konfiguracją konektorów i zadań. Jeśli założymy, że większość wdrożeń Kafki obejmuje framework Kafka Connect z innych powodów (bardzo popularnym przypadkiem użycia jest wysyłanie zdarzeń zmiany bazy danych do Kafki), uruchomienie MirrorMakera wewnątrz Connecta może zmniejszyć liczbę klastrów, którymi trzeba zarządzać.

MirrorMaker równomiernie przydziela partycje do zadań bez korzystania z protokołu zarządzania grupami konsumentów Kafki, aby uniknąć skoków opóźnień spowodowanych równoważeniem obciążenia po dodaniu nowych tematów lub partycji. Zdarzenia z każdej partycji w klastrze źródłowym są mirroringowane do tej samej partycji w klastrze docelowym, z zachowaniem partycjonowania semantycznego i utrzymaniem kolejności zdarzeń dla każdej partycji. Jeśli do tematów źródłowych dodawane są nowe partycje, są one automatycznie tworzone w temacie docelowym. Oprócz replikacji danych MirrorMaker obsługuje również migrację przesunąć konsumentów, konfigurację tematów i listy ACL tematów, dzięki czemu jest kompletnym rozwiązaniem mirroringowym dla wdrożeń wieloklastrowych. Przepływ replikacji definiuje konfigurację przepływu kierunkowego z klastra źródłowego do klastra docelowego. Dla MirrorMakera można zdefiniować wiele przepływów replikacji w celu zdefiniowania złożonych topologii, w tym wzorców architektonicznych, które omówiliśmy wcześniej, takich jak architektury piasty i szprych, aktywny-aktywny i aktywny-pasywny. Na rysunku 10.6 pokazaliśmy zastosowanie narzędzia MirrorMaker w architekturze aktywny-pasywny.



Rysunek 10.6. Proces MirrorMakera w Kafce

Konfigurowanie MirrorMakera

MirrorMaker jest w dużym zakresie konfigurowalny. Oprócz konfiguracji klastra definiujących topologię, frameworku Kafka Connect i ustawień konektora, dostosowywać można każdą właściwość konfiguracyjną producenta, konsumentów i klienta administracyjnego używanych przez MirrorMaker. Zaprezentujemy tutaj kilka przykładów i wskażemy niektóre z ważnych opcji konfiguracyjnych, ale wyczerpująca dokumentacja MirrorMakera wykracza poza zakres tego rozdziału.

Mając to na uwadze, przyjrzymy się przykładowi MirrorMakera. Poniższe polecenie uruchamia MirrorMaker z opcjami konfiguracji określonymi w pliku właściwości:

```
bin/connect-mirror-maker.sh etc/kafka/connect-mirror-maker.properties
```

Przyjrzymy się niektórym opcjom konfiguracji MirrorMakera:

Przepływ replikacji

Poniższy przykład pokazuje opcje konfiguracyjne do definiowania przepływu replikacji aktywny-pasywny między dwoma centrami danych w Nowym Jorku i Londynie:

```
clusters = NYC, LON                                ❶
NYC.bootstrap.servers = kafka.nyc.example.com:9092  ❷
LON.bootstrap.servers = kafka.lon.example.com:9092
NYC->LON.enabled = true                             ❸
NYC->LON.topics = .*                                 ❹
```

- ❶ Zdefiniowanie aliasów dla klastrów używanych w przepływach replikacji.
- ❷ Skonfigurowanie ładowania początkowego dla każdego klastra przy użyciu aliasu klastra jako prefiksu.
- ❸ Włączenie przepływu replikacji między parą klastrów za pomocą prefiksu *źródłowy->docelowy*. Wszystkie opcje konfiguracyjne dla tego przepływu wykorzystują ten sam prefiks.
- ❹ Skonfigurowanie tematów do mirrorowania dla tego przepływu replikacji.

Mirrorowane tematy

Jak pokazaliśmy w przykładzie, dla każdego przepływu replikacji można określić wyrażenie regularne dla nazw tematów, które będą mirrorowane. W tym przykładzie zdecydowaliśmy się na mirroring każdego tematu, ale często dobrą praktyką jest używanie np. wyrażenia *prod.**

i unikanie replikowania tematów testowych. Można również określić oddzielną listę wykluczeń tematów, zawierającą nazwy tematów lub wzorce, takie jak *test.**, aby wykluczyć tematy, które nie wymagają mirrorowania. Domyślnie nazwy tematów docelowych są automatycznie poprzedzane aliasem klastra źródłowego. W architekturze aktywny-aktywny MirrorMaker replikujący tematy z centrum danych NJ do centrum danych LON będzie np. mirrorował temat *orders* z NJ do tematu *NJ.orders* w LON. Ta domyślna strategia nazewnictwa zapobiega cykłom replikacji, które w nieskończoność mirrorują zdarzenia między dwoma klastrami w trybie aktywny-aktywny, jeśli tematy są mirrorowane zarówno z NJ do LON, jak i z LON do NJ. Rozróżnienie między tematami lokalnymi i zdalnymi obsługuje również przypadki użycia dla agregacji, ponieważ konsumenci mogą wybrać wzorce subskrypcji, aby konsumować dane produkowane tylko z regionu lokalnego lub subskrybować tematy ze wszystkich regionów w celu uzyskania kompletnego zbioru danych.

MirrorMaker okresowo sprawdza nowe tematy w klastrze źródłowym i automatycznie zaczyna mirrorować te tematy, jeżeli pasują do skonfigurowanych wzorców. Jeśli do tematu źródłowego zostanie dodanych więcej partycji, ta sama liczba partycji zostanie automatycznie dodana do tematu docelowego, zapewniając, że zdarzenia w temacie źródłowym pojawią się na tej samej partycji w tej samej kolejności w temacie docelowym.

Migracja przesunięć konsumentów

MirrorMaker zawiera klasę narzędziową `RemoteClusterUtils`, aby w przypadku przełączania awaryjnego z klastra głównego umożliwić konsumentom wyszukiwanie ostatniego zatwierdzonego przesunięcia w klastrze DR z translacją przesunięć. W wersji 2.7.0 dodano wsparcie dla okresowej migracji przesunięć konsumentów w celu automatycznego zatwierdzania przetłumaczonych przesunięć w docelowym temacie `__consumer_offsets`, by konsumenci przełączające się do klastra DR mogły ponownie rozpocząć od miejsca, w którym przerwały pracę w klastrze głównym, bez utraty danych i przy minimalnym duplikowaniu przetwarzania. Grupy konsumentów, dla których migrowane są przesunięcia, można dostosowywać, a w celu dodatkowej ochrony MirrorMaker nie nadpisuje przesunięć, jeżeli konsumenci w klastrze docelowym aktywnie korzystają z docelowej grupy konsumentów, dzięki czemu unika wszelkich przypadkowych konfliktów.

Konfiguracja tematów i migracja list ACL

Oprócz wykonywania mirroringu rekordów danych, MirrorMaker może zostać skonfigurowany do mirroringu konfiguracji tematów i list kontroli dostępu (ang. *Access Control List* — ACL) tematów, aby zachowanie mirrorowanego tematu pozostało niezmienione. Domyślna konfiguracja umożliwia tę migrację z rozsądnymi okresowymi interwałami odświeżania, które mogą być wystarczające w znacznej części przypadków. Większość ustawień konfiguracyjnych tematu ze źródła jest stosowana do tematu docelowego, ale niektóre, takie jak `min.insync.replicas`, nie są domyślnie stosowane. Listę wyłączonych konfiguracji można dostosowywać.

Migracja dotyczy tylko literalnych list ACL tematów, które pasują do mirrorowanych tematów, dlatego jeśli używasz prefiksów, list ACL ze znakami wieloznacznymi lub alternatywnych mechanizmów autoryzacji, musisz bezpośrednio skonfigurować je w klastrze docelowym. Listy ACL dla `TopicWrite` nie są migrowane, aby zagwarantować, że tylko MirrorMaker może zapisywać

w temacie docelowym. Odpowiedni dostęp musi zostać bezpośrednio przyznany w momencie przełączania awaryjnego w celu zapewnienia działania aplikacji z klastrem podrzędnym.

Zadania konektora

Opcja konfiguracyjna `tasks.max` ogranicza maksymalną liczbę zadań, z których może korzystać konektor powiązany z MirrorMakerem. Wartość domyślna to 1, ale zaleca się co najmniej 2. Podczas replikacji wielu partycji tematu należy w miarę możliwości stosować wyższe wartości w celu zwiększenia poziomu równoległości.

Prefiksy konfiguracji

MirrorMaker obsługuje dostosowywanie opcji konfiguracyjnych dla wszystkich swoich komponentów, w tym konektorów, producentów, konsumentów i klientów administracyjnych. Konfiguracje frameworku Kafka Connect i konektorów mogą być określane bez prefiksu. Ponieważ jednak konfiguracja MirrorMakera może zawierać ustawienia dla wielu klastrów, prefiksy mogą być stosowane do określenia konfiguracji charakterystycznych dla klastrów lub konfiguracji dla konkretnego przepływu replikacji. Jak widziałeś we wcześniejszym przykładzie, klastry są identyfikowane za pomocą aliasów, które są używane jako prefiks konfiguracji dla opcji związanych z tym klastrem. Prefiksy mogą być wykorzystywane do budowania konfiguracji hierarchicznej, gdzie im konkretniejszy prefiks konfiguracji, tym wyższy priorytet. MirrorMaker używa następujących prefiksów:

- `{klaster}.{konfiguracja_konektora}`;
- `{klaster}.admin.{konfiguracja_administratora}`;
- `{klaster_źródłowy}.consumer.{konfiguracja_konsumenta}`;
- `{klaster_docelowy}.producer.{konfiguracja_producenta}`;
- `{klaster_źródłowy}->{klaster_docelowy}.{konfiguracja_przepływu_replikacji}`.

Topologia replikacji wieloklastrowej

Widziałeś przykładową konfigurację MirrorMakera dla prostego przepływu replikacji aktywno-pasywny. Teraz przyjrzymy się rozszerzeniu konfiguracji, aby obsłużyć inne popularne wzorce architektoniczne.

Topologię aktywny-aktywny między Nowym Jorkiem a Londynem można skonfigurować przez włączenie przepływu replikacji w obu kierunkach. Chociaż wszystkie tematy z NJ są mirrorowane do LON i odwrotnie, w tym przypadku MirrorMaker zapewnia, że to samo zdarzenie nie będzie stale mirrorowane w tę i z powrotem między parą klastrów, ponieważ zdalne tematy stosują jako prefiks alias klastra. Dobrą praktyką jest używanie dla różnych procesów MirrorMakera tego samego pliku konfiguracyjnego, który zawiera pełną topologię replikacji, gdyż dzięki wykorzystaniu wewnętrznego tematu konfiguracji w docelowym centrum danych pozwala to uniknąć konfliktów, jeśli konfiguracje są współdzielone. Procesy MirrorMakera można uruchomić w docelowym centrum danych za pomocą współdzielonego pliku konfiguracyjnego, określając podczas inicjowania procesu MirrorMakera klastry docelowy przy użyciu opcji `--clusters`:

```
clusters = NYC, LON
NYC.bootstrap.servers = kafka.nyc.example.com:9092
LON.bootstrap.servers = kafka.lon.example.com:9092
```

```
NYC->LON.enabled = true
NYC->LON.topics = .*
LON->NYC.enabled = true
LON->NYC.topics = .*
```

❶
❷
❸
❹

- ❶ Włączenie replikacji z Nowego Jorku do Londynu.
- ❷ Określenie tematów replikowanych z Nowego Jorku do Londynu.
- ❸ Włączenie replikacji z Londynu do Nowego Jorku.
- ❹ Określenie tematów replikowanych z Londynu do Nowego Jorku.

Do tej topologii można również dodawać kolejne przepływy replikacji z dodatkowymi klastrami źródłowymi lub docelowymi. Dodając np. nowy przepływ replikacji dla SF, możemy rozszerzyć tę konfigurację, aby obsługiwała replikację z NJ do SF i LON:

```
clusters = NYC, LON, SF
SF.bootstrap.servers = kafka.sf.example.com:9092
NYC->SF.enabled = true
NYC->SF.topics = .*
```

Zabezpieczanie MirrorMakera

W przypadku klastrów produkcyjnych ważne jest zagwarantowanie bezpieczeństwa dla całego ruchu między centrami danych. Opcje zabezpieczenia klastrów Kafki opisaliśmy w rozdziale 11. MirrorMaker musi być skonfigurowany do korzystania z bezpiecznego nasłuchiwanca brokerów zarówno w klastrach źródłowych, jak i docelowych. Ponadto po stronie klienta dla każdego klastra muszą być skonfigurowane opcje zabezpieczeń, by umożliwić MirrorMakerowi nawiązywanie uwierzytelnionych połączeń. Do szyfrowania całego ruchu między centrami danych powinien być używany protokół SSL. Do zdefiniowania poświadczeń dla MirrorMakera można zastosować np. następującą konfigurację:

```
NYC.security.protocol=SASL_SSL
NYC.sasl.mechanism=PLAIN
NYC.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
    required username="MirrorMaker" password="MirrorMaker-password";
```

❶

❷

- ❶ Protokół zabezpieczeń powinien być zgodny z protokołem nasłuchiwanca brokerów odpowiadającym serwerom ładowania początkowego, które zostały zdefiniowane dla klastra. Zalecany jest SSL lub SASL_SSL.
- ❷ Ponieważ używany jest SASL, poświadczenia dla MirrorMakera są tutaj określone przy użyciu konfiguracji JAAS. W przypadku protokołu SSL należy określić magazyny kluczy, jeżeli połączone jest wzajemne uwierzytelnianie klienta.

Jeśli dla klastrów jest włączona autoryzacja, to głównemu użytkownikowi powiązanemu z MirrorMakerem trzeba również przyznać odpowiednie uprawnienia dla klastrów źródłowych i docelowych. Dla procesu MirrorMakera muszą zostać przyznane następujące uprawnienia ACL:

- Topic:Read w klastrze źródłowym do konsumowania z tematów źródłowych; Topic:Create i Topic:Write w klastrze docelowym w celu tworzenia tematów docelowych i produkowania do nich.

- `Topic:DescribeConfigs` w klastrze źródłowym do pozyskiwania konfiguracji tematów źródłowych; `Topic:AlterConfigs` w klastrze źródłowym w celu aktualizowania konfiguracji tematów źródłowych.
- `Topic:Alter` w klastrze docelowym, aby dodawać partycje w przypadku wykrycia nowych partycji źródłowych.
- `Group:Describe` w klastrze źródłowym w celu pozyskiwania metadanych źródłowej grupy konsumentów, w tym przesunięć; `Group:Read` w klastrze docelowym do zatwierdzania przesunięć dla tych grup.
- `Cluster:Describe` w klastrze źródłowym w celu pozyskiwania list ACL tematów źródłowych; `Cluster:Alter` w klastrze docelowym do aktualizowania list ACL tematów docelowych.
- `Topic:Create` i `Topic:Write` w klastrach źródłowych i docelowych dla wewnętrznych tematów MirrorMakera.

Wdrażanie MirrorMakera w środowisku produkcyjnym

W poprzednim przykładzie uruchomiliśmy narzędzie MirrorMaker w trybie dedykowanym w wierszu poleceń. Możesz zainicjować dowolną liczbę tych procesów, aby utworzyć dedykowany klastr MirrorMakera, który będzie skalowalny i odporny na błędy. Procesy mirrorujące ten sam klastr wykryją się wzajemnie i automatycznie rozłożą między siebie obciążenie robocze. Zwykle gdy stosuje się MirrorMaker w środowisku produkcyjnym, należy uruchomić go jako usługę działającą w tle z opcją `nohup` i przekierować rejestrowanie w konsoli do pliku dziennika. Narzędzie to posiada również opcję wiersza poleceń `-daemon`, która powinna zrobić to za Ciebie. Większość firm korzystających z MirrorMakera ma własne skrypty startowe zawierające również parametry konfiguracyjne. Do automatyzacji zarządzania wieloma opcjami konfiguracyjnymi często używane są produkcyjne systemy wdrażania, takie jak Ansible, Puppet, Chef i Salt. MirrorMaker można uruchomić również wewnątrz kontenera Dockera. MirrorMaker jest całkowicie bezstanowy i nie wymaga pamięci masowej na dysku (wszystkie dane i stan są przechowywane w samej Kafce).

Ponieważ MirrorMaker jest oparty na frameworku Kafka Connect, można używać z nim wszystkich trybów wdrażania Connecta. Tryb samodzielny może być stosowany do rozwoju i testowania oprogramowania, gdzie MirrorMaker działa jako samodzielny węzeł roboczy Connecta na pojedynczej maszynie. Przy bezpośrednim skonfigurowaniu konektorów MirrorMaker może być uruchomiony także jako konektor w istniejącym rozproszonym klastrze Connecta. Do użytku w środowisku produkcyjnym zalecamy uruchomienie MirrorMakera w trybie rozproszonym jako dedykowanego klastra MirrorMakera lub we współdzielonym rozproszonym klastrze Connecta.

Jeśli to możliwe, uruchamiaj MirrorMaker w docelowym centrum danych. Jeżeli wysyłasz dane z NJ do SF, MirrorMaker powinien działać w SF i konsumować dane NJ przez całe Stany Zjednoczone. Jest to spowodowane tym, że na dużych odległościach sieci mogą być nieco bardziej zawodne niż te wewnątrz centrum danych. Gdy powstanie partycja sieciowa i utracisz łączność między centrami danych, znacznie bezpieczniejsze jest posiadanie konsumenta, który nie jest w stanie połączyć się z klastrem, niż posiadanie producenta, który nie może się połączyć. Jeśli konsument nie może się połączyć, nie jest po prostu w stanie odczytywać zdarzeń, ale są one nadal przechowywane w źródłowym klastrze Kafki i mogą tam pozostać przez długi czas. Nie ma ryzyka utraty

zdarzeń. Natomiast jeśli zdarzenia zostały już skonsumowane i MirrorMaker nie może ich wyprodukować z powodu powstania partycji sieciowej, zawsze istnieje ryzyko, że te zdarzenia przypadkowo zostaną utracone przez MirrorMaker. Zdalne konsumowanie jest bezpieczniejsze niż zdalne produkowanie.

Kiedy trzeba konsumować lokalnie i produkować zdalnie? Robi się tak w sytuacji, gdy trzeba szyfrować dane podczas ich przesyłania między centrami danych, ale nie ma konieczności ich szyfrowania wewnątrz centrum danych. Konsumenty odczuwają znaczny spadek wydajności podczas łączenia się z Kafką za pomocą szyfrowania SSL — znacznie większy niż producenci. Dzieje się tak, ponieważ stosowanie protokołu SSL wymaga kopiowania danych dla szyfrowania, co oznacza, że konsumenci nie zyskują już na wydajności dzięki standardowej optymalizacji zerowego przepływu. Ten spadek wydajności dotyczy również samych brokerów Kafki. Jeżeli ruch między centrami danych wymaga szyfrowania, ale ruch lokalny niekoniecznie, być może lepiej umieścić MirrorMaker w źródłowym centrum danych, by konsumował niezaszyfrowane dane lokalnie, a następnie produkował je do zdalnego centrum danych za pośrednictwem szyfrowanego połączenia SSL. W ten sposób za pomocą SSL z Kafką łączy się producent, ale nie konsument, co nie wpływa tak bardzo na wydajność. Jeśli stosujesz to podejście lokalnego konsumowania i zdalnego produkowania, upewnij się, że connectowy producent MirrorMakera jest skonfigurowany w taki sposób, aby nigdy nie tracił zdarzeń — ustaw `acks=all` i wystarczającą liczbę ponawiania prób. Ponadto skonfiguruj MirrorMaker w trybie *fail fast* przy użyciu `errors.tolerance=none`, gdy nie może wysłać zdarzeń, co zazwyczaj jest bezpieczniejsze niż kontynuowanie i ryzykowanie utraty danych. Zwróć uwagę, że nowsze wersje Javy mają znacznie wyższą wydajność SSL, więc produkowanie lokalne i konsumowanie zdalne może być realną opcją nawet przy szyfrowaniu.

Innym przypadkiem, w którym potrzebne może być produkowanie zdalne i konsumowanie lokalne, jest scenariusz hybrydowy, gdy mirroringujemy z klastra lokalnego do klastra chmurowego. Bezpieczne klastry lokalne prawdopodobnie znajdują się za firewallem, który nie zezwala na połączenia przychodzące z chmury. Przy uruchomieniu MirrorMakera lokalnie wszystkie połączenia wychodzące do chmury są dozwolone.

Podczas wdrażania MirrorMakera w środowisku produkcyjnym należy pamiętać, aby monitorować go na różne sposoby opisane poniżej.

Monitorowanie za pomocą frameworku Kafka Connect

Kafka Connect oferuje szeroką gamę wskaźników do monitorowania różnych aspektów, np.: wskaźniki konektorów do monitorowania statusu konektorów, wskaźniki konektora źródłowego do kompleksowego monitorowania czy wskaźniki węzła roboczego do monitorowania opóźnień związanych z równoważeniem obciążenia. Connect zapewnia również interfejs RESTful API do przeglądania konektorów i zarządzania nimi.

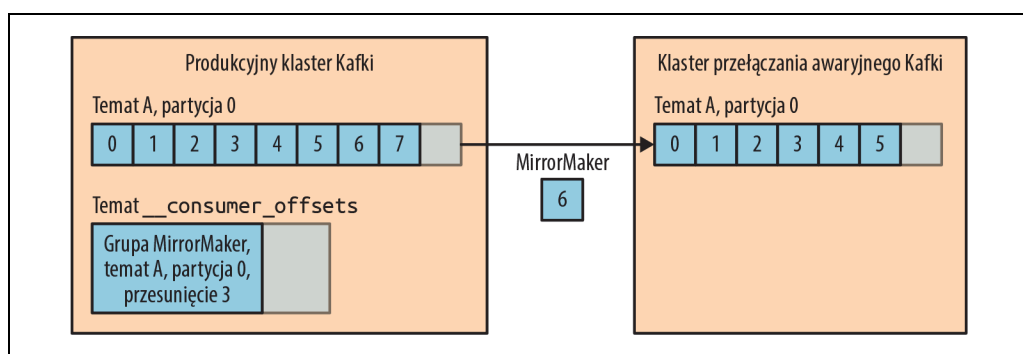
Monitorowanie wskaźników MirrorMakera

Oprócz wskaźników Connecta MirrorMaker oferuje wskaźniki do kompleksowego monitorowania mirroringu i opóźnień replikacji. Wskaźnik opóźnień replikacji `replication-laten-cy-ms` pokazuje interwał między znacznikiem czasu rekordu a momentem, w którym rekord został pomyślnie wyprodukowany dla klastra docelowego. Jest to przydatne do weryfikowania, czy klastr docelowy nadąża za klastrem źródłowym. Zwiększone opóźnienie w godzinach

szczytu może być do zaakceptowania, jeżeli istnieje wystarczająca pojemność, aby nadrobić zaległości w późniejszym czasie, ale stały wzrost opóźnień może wskazywać na niewystarczającą pojemność. Bardzo przydatne mogą być również inne wskaźniki, takie jak `record-lag-ms`, który pokazuje wiek rekordów w momencie replikacji, `byte-rate` pokazujący całkowitą replikację oraz `checkpoint-latency-ms`, który pokazuje opóźnienia migracji przesunięć. Domyślnie MirrorMaker emituje ponadto okresowo sygnał `heartbeat`, który może być używany do monitorowania kondycji tego narzędzia.

Monitorowanie opóźnień

Na pewno będziesz chciał wiedzieć, czy klaster docelowy nie pozostaje w tyle za klustrem źródłowym. To opóźnienie określa różnicę w przesunięciach między najnowszym komunikatem w klastrze źródłowym Kafki a najnowszym komunikatem w klastrze docelowym. Pokazaliśmy to na rysunku 10.7.



Rysunek 10.7. Monitorowanie opóźnienia wynikającego z różnicy w przesunięciach

Na rysunku 10.7 ostatnie przesunięcie w klastrze źródłowym wynosi 7, a ostatnie przesunięcie w klastrze docelowym to 5, co oznacza, że mamy opóźnienie o 2 komunikaty.

Istnieją dwa sposoby śledzenia tego opóźnienia i żaden z nich nie jest idealny:

- Sprawdzanie najnowszego przesunięcia zatwierdzonego przez MirrorMaker w klastrze źródłowym Kafki. Możesz użyć narzędzia `kafka-consumer-groups`, aby dla każdej partycji odczytywanej przez MirrorMaker sprawdzić przesunięcie ostatniego zdarzenia w partycji, ostatnie przesunięcie zatwierdzone przez MirrorMaker i opóźnienie między nimi. Wskaźnik ten nie jest w 100% dokładny, ponieważ MirrorMaker nie zatwierdza przesunięć przez cały czas. Domyślnie zatwierdza przesunięcia co minutę, więc zobaczysz, że opóźnienie będzie rosło przez minutę, a potem nagle spadnie. Na rysunku 10.7 rzeczywiste opóźnienie wynosi 2, ale narzędzie `kafka-consumer-groups` zgłosi opóźnienie wynoszące 5, ponieważ MirrorMaker nie zatwierdził jeszcze przesunięć dla nowszych komunikatów. Narzędzie Burrow LinkedIna monitoruje te same informacje, ale ma bardziej zaawansowaną metodę określania, czy opóźnienie stanowi prawdziwy problem, dzięki czemu nie będziesz otrzymywał fałszywych alertów.
- Sprawdzanie najnowszego przesunięcia odczytywanego przez MirrorMaker (nawet jeśli nie zostało zatwierdzone). Konsumenty osadzone w MirrorMakerze publikują kluczowe

wskaźniki w technologii JMX. Jednym z nich jest maksymalne opóźnienie konsumenta (na wszystkich przetwarzanych przez niego partycjach). To opóźnienie również nie jest w 100% dokładne, gdyż jest aktualizowane na podstawie tego, co odczytał konsument, ale nie uwzględnia tego, czy producentowi udało się wysłać te komunikaty do docelowego klastra Kafki i czy zostały one pomyślnie potwierdzone. W tym przykładzie konsument MirrorMakera zgłosi opóźnienie o 1 komunikat, a nie o 2, ponieważ odczytał już komunikat 6 — mimo że ten komunikat nie został jeszcze wyprodukowany do miejsca docelowego.

Zwróć uwagę, że jeśli MirrorMaker pomija lub porzuca komunikaty, żadna z metod nie wykryje problemu, ponieważ śledzi po prostu najnowsze przesunięcie. Confluent Control Center (<https://oreil.ly/KnvVV>) jest komercyjnym narzędziem, które monitoruje liczbę komunikatów i sumy kontrolne, a tym samym wypełnia tę lukę w monitorowaniu.

Monitorowanie wskaźników producentów i konsumentów

Framework Kafka Connect używany przez MirrorMaker zawiera producent i konsument. Oba mają wiele dostępnych wskaźników i zalecamy ich gromadzenie i śledzenie. Listę wszystkich dostępnych wskaźników znajdziesz w dokumentacji Kafki (<http://bit.ly/2sMfZWf>). Oto kilka wskaźników, które są przydatne w regulowaniu wydajności MirrorMakera:

Konsument

fetch-size-avg, fetch-size-max, fetch-rate, fetch-throttle-time-avg i fetch-throttle-time-max.

Producent

batch-size-avg, batch-size-max, requests-in-flight i record-retry-rate.

Konsument i producent

io-ratio oraz io-wait-ratio.

Kanarek

Jeśli monitorujesz wszystko inne, kanarek nie jest właściwie konieczny, ale lubimy dodawać go do wielu warstw monitorowania. Zapewnia proces, który co minutę wysyła zdarzenie do specjalnego tematu w klastrze źródłowym i próbuje odczytać zdarzenie z klastra docelowego. Powiadamia również, jeżeli dotarcie zdarzenia na miejsce trwa dłużej, niż wynosi akceptowalna ilość czasu. Może to oznaczać, że MirrorMaker jest opóźniony lub w ogóle nie jest dostępny.

Regulowanie MirrorMakera

MirrorMaker jest skalowalny poziomo. Wielkość klastra MirrorMakera zależy od wymaganej przepustowości i tolerowanego opóźnienia. Jeśli nie możesz tolerować żadnych opóźnień, musisz skonfigurować rozmiar MirrorMakera z pojemnością wystarczającą, by nadażyć za najwyższą przepływnością. Jeżeli możesz tolerować pewne opóźnienia, możesz ustalić taki rozmiar, aby przez 95 – 99% czasu MirrorMaker był wykorzystany w 75 – 80%. Należy wtedy oczekiwać, że pewne opóźnienie pojawi się przy szczytowej przepływności. Ponieważ MirrorMaker ma przez większość czasu wolną pojemność, nadrobi zaległości po ustaniu szczytowego ruchu.

Wydajność MirrorMakera należy mierzyć różną liczbą zadań konektora — skonfigurowaną za pomocą parametru `tasks.max`. W dużej mierze zależy to od Twojego sprzętu, centrum danych lub

dostawcy chmury, powinieneś więc uruchomić własne testy. Kafka jest dostarczana z narzędziem `kafka-performance-producer`. Użyj go do wygenerowania obciążenia w klastrze źródłowym, a następnie podłącz MirrorMaker i zacznij mirrorować to obciążenie. Przetestuj MirrorMaker po kolei z różną liczbą zadań: 1, 2, 4, 8, 16, 24 i 32. Zaobserwuj, kiedy wydajność zaczyna słabnąć, i ustaw `tasks.max` tuż poniżej tego punktu. Jeśli konsumujesz lub produkujesz skompresowane zdarzenia (co jest zalecane, ponieważ przepustowość jest głównym wąskim gardłem przy mirroringu między centrami danych), MirrorMaker będzie musiał dekompresować i ponownie kompresować zdarzenia. Powoduje to zużywanie dużej ilości zasobów procesora, przyglądaj się więc wykorzystaniu CPU wraz ze zwiększaniem liczby zadań. Stosując ten proces, znajdziesz maksymalną przepustowość, jaką możesz uzyskać za pomocą pojedynczego węzła roboczego MirrorMakera. Gdy to nie wystarczy, powinieneś poeksperymentować z dodatkowymi węzłami roboczymi. Jeśli używasz MirrorMakera w istniejącym klastrze Connecta z innymi konektorami, upewnij się, że podczas ustalania rozmiaru klastra wzięłeś pod uwagę także obciążenie tych konektorów.

Ponadto możesz do osobnego klastra MirrorMakera wydzielić wrażliwe tematy, czyli te, które bezwzględnie wymagają niskiego opóźnienia i dla których miejsce docelowe musi znajdować się jak najbliżej źródła. Zapobiegnie to spowolnieniu Twojego najbardziej wrażliwego potoku danych przez „rozdęty” temat lub producent znajdujący się poza kontrolą.

To właściwie całe dostrajanie, jakie można zastosować do MirrorMakera. Nadal można jednak zwiększyć przepustowość każdego zadania i każdego węzła roboczego MirrorMakera.

Jeśli uruchamiasz MirrorMaker w wielu centrach danych, wyregulowanie stosu TCP może pomóc zwiększyć efektywną przepustowość. W rozdziałach 3. i 4. dowiedziałeś się, że dla producentów i konsumentów można skonfigurować rozmiary buforów TCP za pomocą parametrów `send.buffer.bytes` i `receive.buffer.bytes`. Analogicznie rozmiary buforów po stronie brokera można skonfigurować za pomocą parametrów `socket.send.buffer.bytes` i `socket.receive.buffer.bytes`. Te opcje konfiguracyjne powinny być połączone z optymalizacją konfiguracji sieci w Linuksie, która wygląda następująco:

- Zwiększ rozmiar bufora TCP (`net.core.rmem_default`, `net.core.rmem_max`, `net.core.wmem_default`, `net.core.wmem_max` i `net.core.optmem_max`).
- Włącz automatyczne skalowanie okien (wykonaj `sysctl -w net.ipv4.tcp_window_scaling=1` lub dodaj `net.ipv4.tcp_window_scaling=1` do pliku `/etc/sysctl.conf`).
- Skróć czas powolnego startu TCP (ustaw w pliku `/proc/sys/net/ipv4/tcp_slow_start_after_idle` wartość 0).

Zwróć uwagę, że dostrajanie sieci Linuksa jest obszernym i złożonym tematem. Jeśli chcesz dowiedzieć się więcej o tych i innych parametrach, zalecamy przeczytanie przewodnika regulowania sieci takiego jak *Performance Tuning for Linux Servers* Sandry K. Johnson i in. (IBM Press).

Ponadto możesz chcieć dostroić bazowe producenty i konsumenty MirrorMakera. Po pierwsze, powinieneś określić, czy wąskim gardłem jest producent czy konsument — to producent czeka, aż konsument dostarczy więcej danych, czy odwrotnie? Jednym ze sposobów, by to ustalić, jest przyjrzenie się monitorowanym wskaźnikom producenta i konsumenta. Jeśli jeden proces jest bezczynny, podczas gdy drugi jest w pełni wykorzystywany, to wiesz, który wymaga dostrojenia. Inna metoda

polega na wykonaniu kilku rzutów wątków (przy użyciu narzędzia `jstack`) i sprawdzeniu, czy wątki `MirrorMakera` poświęcają większość czasu na odpytywanie, czy wysyłanie — więcej czasu poświęconego na odpytywanie oznacza zazwyczaj, że wąskim gardłem jest konsument, natomiast jeśli większa ilość czasu przeznaczona jest na wysyłanie, szala przechyla się na stronę producenta.

Gdy chcesz wyregulować producent, użyteczne mogą być następujące ustawienia konfiguracyjne:

`linger.ms` i `batch.size`

Jeśli monitorowanie wykaże, że producent konsekwentnie wysyła częściowo puste partie (tj. wskaźniki `batch-size-avg` i `batch-size-max` są niższe niż skonfigurowany rozmiar `batch.size`), można zwiększyć przepustowość przez wprowadzenie niewielkiego opóźnienia. Zwiększ wartość `linger.ms`, a producent przed wysłaniem partii odczeka kilka milisekund, aż się zapełni. Jeśli wysyłasz pełne partie i masz wolną pamięć, możesz zwiększyć `batch.size` i wysyłać większe partie.

`max.in.flight.requests.per.connection`

Ograniczenie liczby żądań w toku do 1 jest obecnie jedynym sposobem na zagwarantowanie przez `MirrorMaker` zachowania kolejności komunikatów, jeśli niektóre komunikaty wymagają wielokrotnego ponawiania, zanim zostaną pomyślnie potwierdzone. Oznacza to jednak, że każde żądanie wysłane przez producent musi zostać potwierdzone przez klastr docelowy, zanim wysłany zostanie następny komunikat. Może to ograniczyć przepustowość, zwłaszcza jeżeli występuje znaczne opóźnienie przed potwierdzaniem komunikatów przez brokery. Gdy kolejność komunikatów nie jest kluczowa dla Twojego przypadku użycia, zastosowanie dla `max.in.flight.requests.per.connection` wartości domyślnej 5 może znacznie zwiększyć przepustowość.

Przepustowość konsumenta mogą zwiększyć następujące konfiguracje:

`fetch.max.bytes`

Jeśli gromadzone wskaźniki pokazują, że `fetch-size-avg` i `fetch-size-max` są zbliżone do konfiguracji `fetch.max.bytes`, konsument odczytuje z brokera tyle danych, ile jest dozwolone. Jeżeli masz dostępną pamięć, spróbuj zwiększyć wartość `fetch.max.bytes`, aby umożliwić konsumentowi odczytywanie większej ilości danych w każdym żądaniu.

`fetch.min.bytes` i `fetch.max.wait.ms`

Jeśli we wskaźnikach konsumenta widzisz, że `fetch-rate` jest wysokie, konsument wysyła do brokerów zbyt wiele żądań i nie otrzymuje wystarczającej ilości danych w poszczególnych żądaniach. Spróbuj zwiększyć zarówno `fetch.min.bytes`, jak i `fetch.max.wait.ms`, aby konsument otrzymywał więcej danych w każdym żądaniu, a broker przed wysłaniem odpowiedzi na żądanie konsumenta czekał, aż dostępna będzie wystarczająca ilość danych.

Inne rozwiązania do mirroringu między klastrami

Przyjrzelśmy się dokładnie `MirrorMakerowi`, ponieważ to oprogramowanie mirroringowe jest częścią platformy Apache Kafka. Jednak `MirrorMaker` ma również pewne ograniczenia, gdy jest stosowany w praktyce. Warto przyjrzeć się kilku alternatywom dla `MirrorMakera` i sposobom, w jaki

rozwiązują one ograniczenia i złożoność tego narzędzia. Opiszemy kilka rozwiązań udostępnianych na licencji *open source* przez firmy Uber i LinkedIn oraz rozwiązania komercyjne Confluenta.

uReplicator Ubera

Uber uruchamiał starszą wersję MirrorMakera na bardzo dużą skalę, a ponieważ rosła liczba tematów i partycji oraz zwiększała się przepływność klastra, zaczął napotykać kilka problemów. Jak pisaliśmy wcześniej, starszy MirrorMaker do konsumowania z tematów źródłowych wykorzystywał konsumenty, które były członkami jednej grupy konsumentów. Dodawanie wątków MirrorMakera, dodawanie instancji MirrorMakera, restartowanie instancji MirrorMakera, a nawet dodawanie nowych tematów, które pasowały do wyrażenia regularnego używanego w filtrze załączania — wszystko to powodowało, że konsumenty uruchamiały równoważenie obciążenia. Jak dowiedzieć się w rozdziale 4., równoważenie obciążenia zatrzymuje wszystkie konsumenty, dopóki nie będzie można przypisać im nowych partycji. Przy bardzo dużej liczbie tematów i partycji może to chwilę potrwać. Dotyczy to szczególnie korzystania ze starych konsumentów, takich jak w przypadku Ubera. Czasami powodowało to od 5 do 10 minut bezczynności, przez co mirroring pozostawał w tyle, gromadził dużą liczbę zaległych zdarzeń do zmirrorowania i potrzeba było dużo czasu, by to nadrobić. Generowało to bardzo duże opóźnienie dla konsumentów odczytujących zdarzenia z klastra docelowego. Aby uniknąć uruchamiania procesów równoważenia obciążenia, gdy ktoś doda temat odpowiadający filtrowi załączania tematu, Uber postanowił prowadzić listę dokładnych nazw tematów do zmirrorowania, zamiast używać filtra opartego na wyrażeniu regularnym. Było to jednak trudne w utrzymaniu, gdyż w celu dodania nowego tematu wszystkie instancje MirrorMakera musiały być ponownie konfigurowane i restartowane. Jeśli nie zostało to wykonane prawidłowo, mogło doprowadzić do niekończących się równoważeń obciążenia, ponieważ konsumenty nie były w stanie uzgodnić subskrybowanych tematów.

Biorąc pod uwagę te problemy, Uber zdecydował się napisać własny klon MirrorMakera o nazwie **uReplicator**. Jako centralnego (ale wysoce dostępnego) kontrolera do zarządzania listą tematów i partycjami przypisanymi do każdej instancji uReplicatora postanowiono użyć platformy Apache Helix. W celu dodawania nowych tematów do listy w Heliksie administratorzy korzystają z RESTful API, a uReplicator jest odpowiedzialny za przypisywanie partycji do różnych konsumentów. Aby to osiągnąć, Uber zastąpił konsumenty Kafki stosowane w MirrorMakerze napisanym przez inżynierów Ubera konsumentem Kafki zwanym konsumentem Heliksa. Konsument ten pobiera swoje przypisania partycji z kontrolera Apache Heliksa, zamiast uzyskiwać je w wyniku uzgadniania między konsumentami (szczegółowe informacje na temat sposobu robienia tego w Kafce znajdziesz w rozdziale 4.). W rezultacie konsument Heliksa może unikać rebalansowania i zamiast tego nasłuchiwać przychodzących z Heliksa zmian w przypisanych partycjach.

Zespół Uber Engineering opublikował na blogu post (<https://oreil.ly/SGItx>) opisujący szczegółowo architekturę i przedstawiający ulepszenia, które uzyskano. Zależność uReplicatora od platformy Apache Helix wprowadza nowy komponent, którego trzeba się nauczyć i którym trzeba zarządzać, zwiększając złożoność każdego wdrożenia. Jak pokazaliśmy wcześniej, MirrorMaker 2.0 rozwiązuje wiele z tych problemów skalowalności i odporności na błędy starszej wersji MirrorMakera bez żadnych zewnętrznych zależności.

Brooklin LinkedIna

Podobnie jak Uber, LinkedIn również korzystał ze starszej wersji narzędzia MirrorMaker do przesyłania danych między klastrami Kafki. Wraz ze wzrostem skali danych napotykał podobne problemy ze skalowalnością i wyzwania operacyjne. I tak na bazie swojego systemu strumieniowego przesyłania danych LinkedIn zbudował rozwiązanie mirroringowe o nazwie Brooklin. Jest to usługa rozproszona, która może przesyłać strumieniowo dane między różnymi heterogenicznymi źródłami danych i systemami docelowymi z włączeniem Kafki. Jako ogólny framework pobierania danych, który można wykorzystać do budowania potoków danych, Brooklin obsługuje wiele przypadków użycia. Należą do nich m.in.:

- Mostek danych dostarczający dane do systemów przetwarzania strumieniowego z różnych źródeł danych.
- Strumieniowe przesyłanie zdarzeń przechwytywania zmian danych (ang. *Change Data Capture* — CDC) z różnych magazynów danych.
- Rozwiązanie mirroringu między klastrami dla Kafki.

Brooklin to skalowalny system rozproszony zaprojektowany z myślą o wysokiej niezawodności, który został przetestowany z Kafką na dużą skalę. Jest używany do mirroringu bilionów komunikatów dziennie i został zoptymalizowany pod kątem stabilności, wydajności i funkcjonalności. Brooklin jest wyposażony w RESTful API dla operacji zarządzania. Jest to usługa współdzielona, która może przetwarzać dużą liczbę potoków danych, umożliwiając mirrowanie danych między wieloma klastrami Kafki.

Rozwiązania do mirroringu między centrami danych firmy Confluent

Kiedy Uber opracowywał swój uReplicator, firma Confluent niezależnie opracowała Confluent Replicator. Oprócz podobieństw w nazwach projekty nie mają ze sobą wiele wspólnego — są różnymi rozwiązaniami dla dwóch różnych zestawów problemów MirrorMakera. Podobnie jak MirrorMaker 2.0, który pojawił się później, Replikator firmy Confluent jest oparty na frameworku Kafka Connect i został opracowany w celu rozwiązania problemów napotkanych przez jej klientów korporacyjnych podczas korzystania ze starszego narzędzia MirrorMaker do zarządzania wdrożeniami wieloklastrowymi.

Dla klientów, którzy wykorzystują klastry rozciągnięte ze względu na prostotę działania oraz niskie wartości RTO i RPO, firma Confluent dodała klaster wieloregionowy (ang. *Multi-Region Cluster* — MRC) jako wbudowaną funkcjonalność serwera Confluenta, który jest komercyjnym komponentem platformy Confluent. MRC rozszerza wsparcie Kafki dla klastrów rozciągniętych przy użyciu asynchronicznych replik, aby ograniczyć wpływ na opóźnienia i przepustowość. Podobnie jak klastry rozciągające, to rozwiązanie nadaje się do replikacji między strefami dostępności lub regionami o opóźnieniach poniżej 50 milisekund i korzysta na przejrzystym przełączaniu awaryjnym klientów. Dla odległych klastrów z bardziej zawodnymi sieciami niedawno do serwera Confluenta dodana została nowa wbudowana funkcjonalność o nazwie Cluster Linking. Rozszerza ona zachowujący przesunięcia wewnątrzklastrowy protokół Kafki w celu mirrowania danych między klastrami.

Przyjrzyjmy się funkcjonalnościom obsługiwanym przez poszczególne rozwiązania:

Confluent Replicator

Confluent Replicator jest narzędziem mirroringowym podobnym do MirrorMakera, które opiera się na frameworku Kafka Connect w zakresie zarządzania klastrami i może być uruchamiane w istniejących klastrach Connecta. Oba narzędzia obsługują replikację danych dla różnych topologii, a także migrację przesunięć konsumentów i konfiguracji tematów. Występują pewne różnice w funkcjonalnościach, np. MirrorMaker obsługuje migrację ACL i translację przesunięć dla każdego klienta, natomiast Replicator nie migruje list ACL, a translację przesunięć obsługuje (przy użyciu przechwytywacza znaczników czasu) tylko dla klientów Javy. Replicator nie ma koncepcji lokalnych i zdalnych tematów jak MirrorMaker, ale obsługuje łączne agregowanie. Podobnie jak MirrorMaker, Replicator również unika cykli replikacji, ale robi to przy użyciu nagłówków pochodzenia. Replikator zapewnia wiele wskaźników, takich jak opóźnienie replikacji, i może być monitorowany za pomocą swojego RESTful API lub Control Center UI. Obsługuje również migrację schematów między klastrami i może wykonywać translację schematów.

Klasy wieloregionowe (MRC)

Dowiedziałeś się wcześniej, że klasy rozciągnięte zapewniają proste transparentne przełączanie awaryjne i przywracanie sprawności po awarii dla klientów bez konieczności translacji przesunięć lub restartów klienta. Klasy rozciągnięte wymagają jednak, aby centra danych były blisko siebie i zapewniały stabilną sieć o niskim opóźnieniu w celu umożliwienia synchronicznej replikacji między centrami danych. Klasy MRC nadają się ponadto tylko dla centrów danych z opóźnieniem do 50 milisekund, ale wykorzystują kombinację replikacji synchronicznej i asynchronicznej, by ograniczyć wpływ na wydajność producenta i zapewnić wyższą tolerancję sieciową.

Jak pisaliśmy wcześniej, Apache Kafka obsługuje pobieranie z replik podrzędnych (obserwatorów), aby umożliwić klientom pobieranie z najbliższych brokerów na podstawie identyfikatora szafy rackowej, zmniejszając w ten sposób ruch między centrami danych. Confluent Server również dodaje koncepcję **obserwatorów**, będących asynchronicznymi replikami, które nie dołączają do replik zsynchronizowanych, a zatem nie mają wpływu na producenci używające konfiguracji `acks=all`, ale są w stanie dostarczać rekordy konsumentom. Administratorzy mogą konfigurować replikację synchroniczną w obrębie regionu i replikację asynchroniczną między regionami, aby jednocześnie korzystać z niskich opóźnień i wysokiej niezawodności. W serwerze Confluenta ograniczenia umieszczania replik pozwalają określić minimalną liczbę replik na region przy użyciu identyfikatorów szaf rackowych, aby zapewnić, że repliki zostaną rozmieszczone w różnych regionach w celu zagwarantowania niezawodności. Confluent Platform 6.1 dodaje również automatyczną promocję obserwatora z konfigurowalnymi kryteriami, umożliwiając szybkie automatyczne przełączanie awaryjne bez utraty danych. Gdy wartość `min.insync.replicas` spadnie poniżej skonfigurowanej minimalnej liczby replik zsynchronizowanych, obserwatory, które nadrobiły zaległości, są automatycznie promowane, by mogły dołączyć do replik zsynchronizowanych, co przywraca wymaganą minimalną liczbę tych replik. Promowane obserwatory korzystają z replikacji synchronicznej i mogą wpływać na przepustowość, ale klaster pozostaje operacyjny przez cały czas bez utraty danych, nawet jeśli jakiś region ulegnie awarii. Gdy region odzyska sprawność po awarii, obserwatory są automatycznie degradowane, dzięki czemu klaster wraca do normalnego poziomu wydajności.

Cluster Linking

Cluster Linking (łączenie klastrów), wprowadzone jako funkcjonalność przeglądowna w Confluent Platform 6.0, wbudowuje replikację wewnątrzklastrową bezpośrednio w Confluent Server. Korzystając z tego samego protokołu co replikacja między brokerami w klastrze, Cluster Linking wykonuje zachowującą przesunięcia replikację między klastrami, umożliwiając bezproblemową migrację klientów bez konieczności translacji przesunięć. Konfiguracja tematów, partycje, przesunięcia konsumentów i listy ACL są synchronizowane między dwoma klastrami, aby umożliwić przełączanie awaryjne z krótkim RTO w przypadku wystąpienia katastrofy. Łączy klastra definiuje konfigurację przepływu kierunkowego z klastra źródłowego do klastra docelowego. Brokery główne mirrorowanych partycji w klastrze docelowym pobierają dane partycji z odpowiadających im liderów źródłowych, podczas gdy obserwatory w klastrze docelowym replikują z lokalnego lidera przy użyciu standardowego mechanizmu replikacji z Kafki. W klastrze docelowym mirrorowane tematy są oznaczane jako „tylko do odczytu”, aby zapobiec lokalnemu produkowaniu do tych tematów, co zapewnia, że mirrorowane tematy są logicznie identyczne ze swoimi tematami źródłowymi.

Cluster Linking zapewnia prostotę działania bez potrzeby tworzenia osobnych klastrów, takich jak klastry Connecta, i jest bardziej wydajne niż narzędzia zewnętrzne, gdyż pozwala uniknąć dekompresji i ponownej kompresji podczas mirroringu. Inaczej niż w przypadku MRC, nie ma opcji replikacji synchronicznej, a przełączanie awaryjne klientów jest procesem ręcznym, który wymaga zrestartowania klientów. Cluster Linking może być jednak używane z odległymi centrami danych z zawodnymi sieciami o dużym opóźnieniu i zmniejszać ruch między centrami danych przez replikowanie tylko raz. Nadaje się do takich przypadków użycia, jak migracja klastrów i współdzielenie tematów.

Podsumowanie

Rozpoczęliśmy ten rozdział od opisanego powodów, dla których potrzebne może być zarządzanie więcej niż jednym klastrem Kafki, a następnie omówiliśmy kilka typowych architektur wieloklastrowych, od prostych do bardzo złożonych. Przedstawiliśmy szczegóły implementacji architektury przełączania awaryjnego dla Kafki i porównaliśmy różne dostępne aktualnie opcje. Potem przyjrzeliliśmy się dostępnym narzędziom. Zaczęliśmy od narzędzia MirrorMaker platformy Apache Kafka i opisaliśmy szczegółowo wykorzystanie go w środowisku produkcyjnym. Zakończyliśmy przeglądem alternatywnych opcji, które rozwiązują część problemów charakterystycznych dla MirrorMakera.

Niezależnie od tego, której architektury i jakich narzędzi będziesz używać, pamiętaj, że konfiguracja wieloklastrowa i mirroring potoków powinny być monitorowane i testowane, jak wszystko, co jest wdrażane w środowisku produkcyjnym. Ponieważ zarządzanie wieloma klastrami w Kafce może być łatwiejsze niż w relacyjnych bazach danych, niektóre organizacje odkładają to na później i zaniedbują stosowanie odpowiedniego projektowania, planowania, testowania, automatyzacji wdrażania, monitorowania i utrzymywania. Dzięki poważnemu potraktowaniu tego zagadnienia — najlepiej w ramach holistycznego planu na wypadek katastrofy lub planu różnicowania geograficznego dla całej organizacji, obejmującego wiele aplikacji i magazynów danych — znacznie zwiększysz swoje szanse na pomyślne zarządzanie wieloma klastrami Kafki.

Zabezpieczanie Kafki

Kafka jest wykorzystywana do wielorakich przypadków użycia, od śledzenia aktywności na stronie internetowej oraz potoków wskaźników po zarządzanie rekordami pacjentów i płatności online. Każdy przypadek użycia ma inne wymagania pod względem bezpieczeństwa, wydajności, niezawodności i dostępności. Chociaż zawsze lepiej jest korzystać z najsilniejszych i najnowszych dostępnych funkcjonalności bezpieczeństwa, często są konieczne kompromisy, gdyż zwiększony poziom zabezpieczeń wpływa na wydajność, koszty i doświadczenie użytkownika. Kafka obsługuje kilka standardowych technologii bezpieczeństwa z wieloma opcjami konfiguracyjnymi, aby umożliwić dostosowanie do każdego przypadku użycia.

Podobnie jak wydajność i niezawodność, bezpieczeństwo jest aspektem systemu, który należy rozważać w odniesieniu do całego systemu, a nie poszczególnych komponentów. Ogólny poziom bezpieczeństwa systemu określa najsłabsze ogniwo, a procesy i reguły bezpieczeństwa muszą być egzekwowane w całym systemie, również na platformie bazowej. Konfigurowalne funkcjonalności zabezpieczeń w Kafce umożliwiają integrację z istniejącą infrastrukturą bezpieczeństwa w celu zbudowania spójnego modelu bezpieczeństwa, który ma zastosowanie do całego systemu.

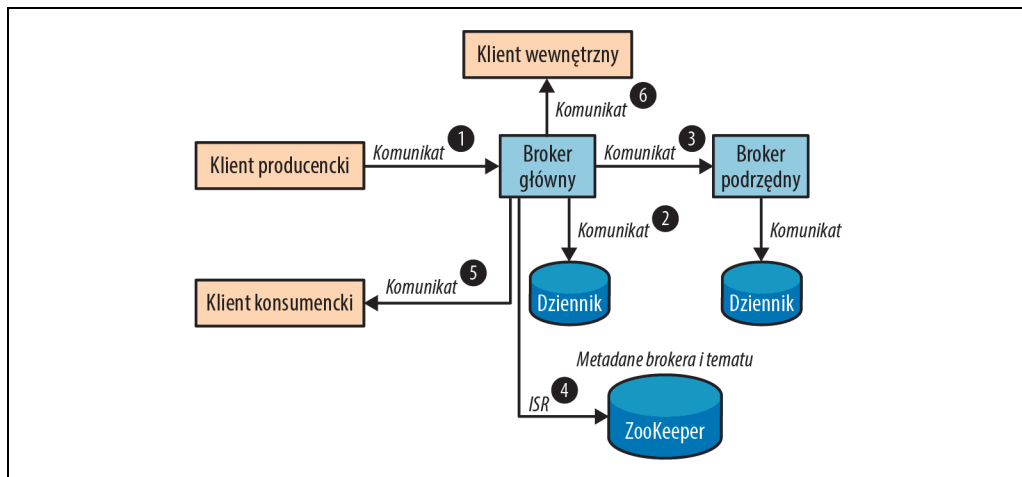
W tym rozdziale omówimy funkcjonalności bezpieczeństwa w Kafce i zobaczymy, w jaki sposób odnoszą się one do różnych aspektów zabezpieczeń i przyczyniają się do ogólnego bezpieczeństwa instalacji Kafki. Podzielimy się wiedzą w zakresie najlepszych praktyk, potencjalnych zagrożeń i technik ich ograniczania. Dokonamy również przeglądu dodatkowych środków, które można zastosować do zabezpieczania ZooKeepera i reszty platformy.

Zamykanie dostępu do Kafki

W Kafce używa się wielu procedur bezpieczeństwa w celu ustanowienia i zachowania poufności, integralności i dostępności danych:

- Uwierzytelnianie pozwala ustalić Twoją tożsamość i określić, **kim** jesteś.
- Autoryzacja określa, **co** możesz robić.
- Szyfrowanie chroni dane przed podsłuchiwaniami i fałszowaniem.
- Audyt (inspekcja) śledzi, co zrobiłeś lub próbowałeś zrobić.
- Kwoty kontrolują, ile zasobów możesz wykorzystać.

Aby zrozumieć, jak zablokować dostęp do wdrożenia Kafki, najpierw przyjrzyjmy się, w jaki sposób dane przepływają przez klaster Kafki. Na rysunku 11.1 pokazaliśmy główne etapy przykładowego przepływu danych. W tym rozdziale będziemy wykorzystywać ten przykładowy przepływ do badania różnych możliwości konfigurowania Kafki, aby chronić dane na każdym kroku i tym samym zagwarantować bezpieczeństwo całego wdrożenia.



Rysunek 11.1. Przepływ danych w klastrze Kafki

1. Alicja produkuje rekord zamówienia klienta do partycji tematu o nazwie customerOrders. Ten rekord jest wysyłany do lidera partycji.
2. Broker główny (lider) zapisuje rekord w pliku swojego lokalnego dziennika.
3. Broker podrzędny (obserwator) pobiera komunikat z lidera i zapisuje go w pliku dziennika swojej lokalnej repliki.
4. Broker główny aktualizuje stan partycji w ZooKeeperze, aby zaktualizować repliki zsynchronizowane, jeśli zachodzi potrzeba.
5. Bob konsumuje rekordy zamówień klientów z tematu customerOrders. Otrzymuje rekord wyprodukowany przez Alicję.
6. Wewnętrzna aplikacja przetwarza wszystkie komunikaty przychodzące do tematu customerOrders, aby produkować w czasie rzeczywistym wskaźniki dotyczące popularnych produktów.

Bezpieczne wdrożenie musi dawać kilka podstawowych gwarancji:

Autentyczność klienta

Gdy Alicja nawiązuje połączenie z brokerem, ten powinien uwierzytelnić klienta, aby się upewnić, że dany komunikat rzeczywiście pochodzi od Alicji.

Autentyczność serwera

Przed wysłaniem komunikatu do brokera głównego klient Alicji powinien zweryfikować, czy łączy się z prawdziwym brokerem.

Ochrona danych

Wszystkie połączenia, przez które przepływają komunikaty, jak również wszystkie dyski, na których komunikaty są przechowywane, powinny być zaszyfrowane lub fizycznie zabezpieczone, aby uniemożliwić podsłuchującym odczytanie danych i ich kradzież.

Integralność danych

W celu wykrywania manipulacji do danych przesyłanych za pośrednictwem niezabezpieczonych sieci należy dołączyć skróty komunikatów.

Kontrola dostępu

Przed zapisaniem komunikatu w dzienniku broker główny powinien sprawdzić, czy Alicja ma autoryzację do zapisywania w `customerOrders`. Zanim broker odeśle komunikat do konsumenta Boba, powinien sprawdzić, czy Bob ma autoryzację do odczytywania z danego tematu. Jeśli konsument Boba korzysta z zarządzania grupami, broker powinien także zweryfikować, czy Bob ma dostęp do tej grupy konsumentów.

Audyt

Rejestrowana powinna być ścieżka audytu, która pokazuje wszystkie operacje wykonane przez brokery, Alicję, Boba i inne klienty.

Dostępność

Brokery powinny stosować kwoty i limity, aby uniknąć sytuacji, w której jacyś użytkownicy blokują całą dostępną przepustowość lub przytłaczają broker atakami typu „odmowa usługi” (ang. *Denial of Service* — DoS). ZooKeeper powinien zostać zablokowany, aby zapewnić dostępność klastra Kafki, ponieważ dostępność brokera zależy od dostępności ZooKeepera i integralności przechowywanych na tym serwerze metadanych.

W kolejnych podrozdziałach omówimy funkcjonalności bezpieczeństwa Kafki, które mogą być stosowane do zapewniania tych gwarancji. Najpierw wprowadzimy model łączenia się z Kafką oraz protokoły bezpieczeństwa związane z połączeniami wychodzącymi od klientów do brokerów Kafki. Następnie szczegółowo przyjrzymy się poszczególnym protokołom bezpieczeństwa i zbadamy funkcjonalności uwierzytelniania każdego z nich, pozwalające uwierzytelniać klient i serwer. Przejrzymy opcje szyfrowania na różnych etapach, w tym wbudowane szyfrowanie przesyłanych danych w niektórych protokołach bezpieczeństwa, zapewniające prywatność i integralność danych. Potem zbadamy konfigurowalne uwierzytelnianie w Kafce służące do zarządzania kontrolą dostępu oraz główne dzienniki, które oferują funkcjonalność audytu. Na koniec omówimy kwestie bezpieczeństwa dla reszty systemu, w tym ZooKeepera i platformy, co jest niezbędne do utrzymania dostępności. Szczegółowe informacje na temat kwot, które przyczyniają się do zapewnienia dostępności usług przez sprawiedliwy podział zasobów między użytkowników, znajdziesz w rozdziale 3.

Protokoły bezpieczeństwa

Brokery Kafki są konfigurowane z nasłuchiwaniami na co najmniej jednym punkcie końcowym i na nich akceptują połączenia z klientami. Każdy nasłuchiwaniec może być skonfigurowany z własnymi ustawieniami bezpieczeństwa. Wymogi bezpieczeństwa dotyczące prywatnego nasłuchiwanca wewnętrznego, który jest fizycznie chroniony i dostępny tylko dla upoważnionego personelu,

mogą różnić się od wymogów bezpieczeństwa zewnętrznego nasłuchiacza dostępnego za pośrednictwem internetu. Wybór protokołu bezpieczeństwa determinuje poziom uwierzytelniania i szyfrowania przesyłanych danych.

Kafka obsługuje cztery protokoły bezpieczeństwa z wykorzystaniem dwu standardowych technologii: TLS i SASL. TLS (ang. *Transport Layer Security*), powszechnie nazywany SSL (ang. *Secure Sockets Layer*) po swoim poprzedniku, obsługuje szyfrowanie, a także uwierzytelnianie klienta i serwera. SASL (ang. *Simple Authentication and Security Layer*) to framework, który zapewnia uwierzytelnianie przy użyciu różnych mechanizmów protokołów opartych na połączeniu. Każdy protokół bezpieczeństwa Kafki łączy warstwę transportową (PLAINTEXT lub SSL) z opcjonalną warstwą uwierzytelniania (SSL lub SASL):

PLAINTEXT

PLAINTEXT to warstwa transportowa zwykłego tekstu bez uwierzytelniania. Nadaje się tylko do użytku w prywatnych sieciach do przetwarzania danych, które nie są poufne, ponieważ nie stosuje się uwierzytelniania ani szyfrowania.

SSL

Warstwa transportowa SSL z opcjonalnym uwierzytelnianiem SSL klienta. Ponieważ obsługuje uwierzytelnianie klienta i serwera, a także szyfrowanie, nadaje się do użytku w niezabezpieczonych sieciach.

SASL_PLAINTEXT

Warstwa transportowa PLAINTEXT z uwierzytelnianiem SASL klienta. Niektóre mechanizmy SASL obsługują również uwierzytelnianie serwera. Nie obsługuje szyfrowania i dlatego nadaje się do użytku tylko w sieciach prywatnych.

SASL_SSL

Warstwa transportowa SSL z uwierzytelnianiem SASL. Ponieważ obsługuje uwierzytelnianie klienta i serwera, a także szyfrowanie, nadaje się do użytku w niezabezpieczonych sieciach.



TLS/SSL

TLS jest jednym z najczęściej używanych protokołów kryptograficznych w internecie. Protokoły aplikacji, takie jak HTTP, SMTP i FTP, wykorzystują TLS do zapewnienia prywatności i integralności przesyłanych danych. Do tworzenia certyfikatów cyfrowych, które mogą być używane do szyfrowania asymetrycznego, zarządzania nimi i ich dystrybucji TLS stosuje infrastrukturę klucza publicznego (ang. *Public Key Infrastructure* — PKI), unikając konieczności dystrybucji współdzielonych sekretów między serwerami i klientami. Klucze sesji wygenerowane w procesie handshake TLS umożliwiają użycie dla następującej późniejszej transmisji danych szyfrowania symetrycznego o wyższej wydajności.

Nasłuchiawcy wykorzystywani do komunikacji między brokerami można wybrać przez konfigurację parametru `inter.broker.listener.name` lub `security.inter.broker.protocol`. Dla protokołu bezpieczeństwa używanego do komunikacji między brokerami w ustawieniach brokera muszą być zapewnione opcje konfiguracji zarówno po stronie serwera, jak i po stronie klienta. Dzieje się tak, ponieważ brokery muszą ustanawiać połączenia z klientami dla danego nasłuchiawcy. Poniższy

fragment kodu powoduje skonfigurowanie SSL dla komunikacji między brokerami i nasłuchiwczy wewnętrznych oraz SASL_SSL dla nasłuchiwarza zewnętrznego:

```
listeners=EXTERNAL://:9092,INTERNAL://10.0.0.2:9093,BROKER://10.0.0.2:9094
advertised.listeners=EXTERNAL://broker1.example.com:9092,INTERNAL://broker1.local:9093,BROKER://broker1.local:9094
listener.security.protocol.map=EXTERNAL:SASL_SSL,INTERNAL:SSL,BROKER:SSL
inter.broker.listener.name=BROKER
```

Klienci są konfigurowane z protokołem bezpieczeństwa i serwerami ładowania początkowego, które określają nasłuchiwarza brokera. Metadane zwracane klientom zawierają tylko punkty końcowe odpowiadające temu samemu nasłuchiwarzowi co serwery ładowania początkowego:

```
security.protocol=SASL_SSL
bootstrap.servers=broker1.example.com:9092,broker2.example.com:9092
```

W następnym podrozdziale, poświęconym uwierzytelnianiu, dokonamy przeglądu charakterystycznych dla protokołu opcji konfiguracyjnych brokerów i klientów dla poszczególnych protokołów bezpieczeństwa.

Uwierzytelnianie

Uwierzytelnianie to proces ustalania tożsamości klienta i serwera w celu weryfikacji autentyczności klienta i serwera. Gdy klient Alicji łączy się z brokerem głównym w celu wygenerowania rekordu zamówienia klienta, uwierzytelnianie serwera umożliwia klientowi ustalenie, czy serwerem, z którym się komunikuje, rzeczywiście jest broker. Uwierzytelnianie klienta weryfikuje tożsamość Alicji przez walidację jej poświadczeń, np. hasła lub certyfikatu cyfrowego, w celu ustalenia, czy połączenie pochodzi od Alicji, a nie od osoby podszywającej się pod nią. Po uwierzytelnieniu tożsamość Alicji zostaje powiązana z danym połączeniem przez cały okres jego trwania. Kafka używa instancji klasy `KafkaPrincipal` do reprezentowania tożsamości klienta i wykorzystuje tego pryncypała do przyznawania dostępu do zasobów i alokacji kwot dla połączeń z daną tożsamością klienta. `KafkaPrincipal` jest ustalany dla każdego połączenia podczas uwierzytelniania na podstawie protokołu uwierzytelniania. Na bazie nazwy użytkownika podanej do uwierzytelniania opartego na hasle dla Alicji może być używany np. pryncypał `User:Alicja`. `KafkaPrincipal` można dostosowywać przez konfigurowanie dla brokerów parametru `principal.builder.class`.



Połączenia anonimowe

Do nieuwierzytelnionych połączeń stosowany jest pryncypał `User:ANONYMOUS`. Dotyczy to klientów korzystających z nasłuchiwarza PLAINTEXT oraz nieuwierzytelnionych klientów korzystających z nasłuchiwarza SSL.

SSL

Gdy Kafka ma skonfigurowany dla nasłuchiwarza protokół bezpieczeństwa SSL lub SASL_SSL, jako bezpieczna warstwa transportowa dla połączeń na tym nasłuchiwarzu używany jest TLS. Po nawiązaniu połączenia przez TLS proces handshake TLS przeprowadza uwierzytelnianie, negocjuje parametry kryptograficzne i generuje współdzielone klucze dla szyfrowania. Klient w celu

ustalenia tożsamości serwera weryfikuje cyfrowy certyfikat serwera. Jeśli włączone jest uwierzytelnianie klienta za pomocą protokołu SSL, serwer weryfikuje również cyfrowy certyfikat klienta w celu ustalenia jego tożsamości. Cały ruch przez SSL jest szyfrowany, dlatego można używać tego protokołu w niezabezpieczonych sieciach.



Wydajność SSL

Kanały SSL są szyfrowane i tym samym wprowadzają zauważalny narzut w wykorzystaniu procesora. Dla protokołu SSL nie jest obecnie obsługiwany transfer zerowego przepływu. W zależności od wzorca ruchu narzut może wynosić nawet 20 – 30%.

Konfiguracja TLS

Gdy dla nasłuchiacza brokera używającego SSL lub SASL_SSL włączony jest TLS, brokery należy skonfigurować z magazynem kluczy (ang. *key store*) zawierającym klucz prywatny i certyfikat brokera, a klienty należy skonfigurować z magazynem zaufania (ang. *trust store*) zawierającym certyfikat brokera lub certyfikat urzędu certyfikacji (ang. *Certificate Authority* — CA), który podpisał certyfikat brokera. Certyfikaty brokera powinny zawierać nazwę hosta brokera jako rozszerzenie SAN (ang. *Subject Alternative Name*) lub jako nazwę zwyczajową (ang. *Common Name* — CN), aby umożliwić klientom weryfikację nazwy hosta serwera. W celu uproszczenia administrowania przy użyciu tego samego magazynu kluczy dla wszystkich brokerów w domenie można używać certyfikatów wieloznacznych (ang. *wildcard certificate*).



Weryfikacja nazwy hosta serwera

Domyślnie klienty Kafki sprawdzają, czy nazwa hosta serwera zapisana w certyfikacie serwera jest zgodna z hostem, z którym klient się łączy. Nazwą hosta dla połączenia może być serwer ładowania początkowego skonfigurowany przez klient lub rozgłoszona nazwa hosta nasłuchiacza, która została zwrócona przez broker w odpowiedzi na żądanie metadanych. Weryfikacja nazwy hosta jest kluczową częścią uwierzytelniania serwera, która chroni przed atakami typu „człowiek pośrodku” (ang. *man-in-the-middle*), i dlatego w systemach produkcyjnych nie powinna być wyłączana.

Ustawiając opcję konfiguracyjną `ssl.client.auth=required`, brokery można skonfigurować do uwierzytelniania klientów łączących się przez nasłuchiawce przy użyciu protokołu bezpieczeństwa SSL. Klienci powinny być skonfigurowane z magazynem kluczy, a brokery z magazynem zaufania zawierającym certyfikaty klienta lub certyfikat CA, który podpisał certyfikaty klienta. Jeżeli do komunikacji między brokerami stosowany jest protokół SSL, magazyny zaufania brokerów powinny zawierać CA certyfikatów brokerów oraz CA certyfikatów klientów. Domyślnie jako `KafkaPrincipal` dla autoryzacji i kwot używana jest nazwa wyróżniona (ang. *Distinguished Name* — DN) certyfikatu klienta. W celu dostarczenia listy reguł dostosowywania pryncypała można zastosować opcję konfiguracyjną `ssl.principal.mapping.rules`. Nasłuchiawce korzystający z SASL_SSL wyłączają uwierzytelnianie TLS klienta i polegają na uwierzytelnianiu SASL i utworzonym przez nie obiekcie `KafkaPrincipal`.

W poniższych przykładach przedstawiliśmy sposób tworzenia magazynów kluczy i magazynów zaufania do uwierzytelniania serwera i klienta za pomocą CA z podpisem własnym.



Uwierzytelnianie SSL klienta

Uwierzytelnianie SSL klienta może być opcjonalne, jeśli ustawi się `ssl.client.auth=requested`. W tym przypadku klienty, które nie zostały skonfigurowane z magazynami kluczy, ukończą proces handshake TLS, ale zostanie im przypisany pryncypał `User:ANONYMOUS`.

Generujemy dla brokerów parę kluczy z CA z podpisem własnym:

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -keystore server.ca.p12  
-storetype PKCS12 -storepass server-ca-password -keypass server-ca-password  
-alias ca -dname "CN=BrokerCA" -ext bc=ca:true -validity 365 ❶  
$ keytool -export -file server.ca.crt -keystore server.ca.p12  
-storetype PKCS12 -storepass server-ca-password -alias ca -rfc ❷
```

- ❶ Tworzymy parę kluczy dla CA i zapisujemy ją w pliku *server.ca.p12* w formacie PKCS12. Używamy tego do podpisywania certyfikatów.
- ❷ Eksportujemy publiczny certyfikat CA do pliku *server.ca.crt*. Będzie to uwzględniane w magazynach zaufania i łańcuchach znaków certyfikatów.

Tworzymy dla brokerów magazyn kluczy z certyfikatem podpisanym przez CA z podpisem własnym (jeśli używane są nazwy hostów ze znakiem wieloznacznym, ten sam magazyn kluczy może być używany dla wszystkich brokerów; w przeciwnym razie tworzymy magazyn kluczy dla każdego brokera za pomocą w pełni kwalifikowanej nazwy domenowej):

```
$ keytool -genkey -keyalg RSA -keysize 2048 -keystore server.ks.p12  
-storepass server-ks-password -keypass server-ks-password -alias server  
-storetype PKCS12 -dname "CN=Kafka,O=Confluent,C=GB" -validity 365 ❶  
$ keytool -certreq -file server.csr -keystore server.ks.p12 -storetype PKCS12  
-storepass server-ks-password -keypass server-ks-password -alias server ❷  
$ keytool -gencert -infile server.csr -outfile server.crt  
-keystore server.ca.p12 -storetype PKCS12 -storepass server-ca-password  
-alias ca -ext SAN=DNS:broker1.example.com -validity 365 ❸  
$ cat server.crt server.ca.crt > serverchain.crt  
$ keytool -importcert -file serverchain.crt -keystore server.ks.p12  
-storepass server-ks-password -keypass server-ks-password -alias server  
-storetype PKCS12 -noprompt ❹
```

- ❶ Generujemy klucz prywatny dla brokera i zapisujemy go w pliku *server.ks.p12* w formacie PKCS12.
- ❷ Generujemy żądanie podpisania certyfikatu.
- ❸ Używamy magazynu kluczy CA, aby podpisać certyfikat brokera. Podpisany certyfikat jest zapisywany w pliku *server.crt*.
- ❹ Do magazynu kluczy brokera importujemy łańcuch znaków certyfikatu brokera.

Jeśli do komunikacji między brokerami używany jest protokół TLS, tworzymy dla brokerów magazyn zaufania z certyfikatami CA poszczególnych brokerów, aby umożliwić im wzajemne uwierzytelnianie się:

```
$ keytool -import -file server.ca.crt -keystore server.ts.p12  
-storetype PKCS12 -storepass server-ts-password -alias server -noprompt
```

Generujemy dla klientów magazyn zaufania z certyfikatem CA brokera:

```
$ keytool -import -file server.ca.crt -keystore client.ts.p12  
-storetype PKCS12 -storepass client-ts-password -alias ca -noprompt
```

Jeżeli włączone jest uwierzytelnianie TLS klienta, klienty muszą być skonfigurowane z magazynem kluczy. Poniższy skrypt generuje dla klientów CA z podpisem własnym i tworzy magazyn kluczy z certyfikatem podpisanym przez CA klienta. CA klienta jest dodawany do magazynu zaufania brokera, aby brokery mogły weryfikować autentyczność klienta:

```
# Generujemy dla klientów parę kluczy CA z podpisem własnym  
keytool -genkeypair -keyalg RSA -keysize 2048 -keystore client.ca.p12  
-storetype PKCS12 -storepass client-ca-password -keypass client-ca-password  
-alias ca -dname CN=ClientCA -ext bc=ca:true -validity 365 ❶  
keytool -export -file client.ca.crt -keystore client.ca.p12 -storetype PKCS12  
-storepass client-ca-password -alias ca -rfc  
  
# Generujemy magazyn kluczy dla klientów  
keytool -genkey -keyalg RSA -keysize 2048 -keystore client.ks.p12  
-storepass client-ks-password -keypass client-ks-password -alias client  
-storetype PKCS12 -dname "CN=Metrics App,O=Confluent,C=GB" -validity 365 ❷  
keytool -certreq -file client.csr -keystore client.ks.p12 -storetype PKCS12  
-storepass client-ks-password -keypass client-ks-password -alias client  
keytool -gencert -infile client.csr -outfile client.crt  
-keystore client.ca.p12 -storetype PKCS12 -storepass client-ca-password  
-alias ca -validity 365  
cat client.crt client.ca.crt > clientchain.crt  
keytool -importcert -file clientchain.crt -keystore client.ks.p12  
-storepass client-ks-password -keypass client-ks-password -alias client  
-storetype PKCS12 -noprompt ❸  
  
# Dodajemy do magazynu zaufania brokera certyfikat CA klienta  
keytool -import -file client.ca.crt -keystore server.ts.p12 -alias client  
-storetype PKCS12 -storepass server-ts-password -noprompt ❹
```

- ❶ W tym przykładzie tworzymy nowy CA dla klientów.
- ❷ Klienci uwierzytelniają się za pomocą tego certyfikatu domyślnie używają pryncypała User:
↳CN=Metrics App,O=Confluent,C=GB.
- ❸ Dodajemy do magazynu kluczy klienta jego łańcuch znaków certyfikatu.
- ❹ Magazyn zaufania brokera powinien zawierać CA wszystkich klientów.

Skoro mamy już magazyny kluczy i zaufania, możemy skonfigurować TLS dla brokerów. Brokery wymagają magazynu zaufania tylko wtedy, gdy do komunikacji między brokerami używany jest TLS lub gdy włączone jest uwierzytelnianie klienta:

```
ssl.keystore.location=/path/to/server.ks.p12  
ssl.keystore.password=server-ks-password  
ssl.key.password=server-ks-password  
ssl.keystore.type=PKCS12  
ssl.truststore.location=/path/to/server.ts.p12  
ssl.truststore.password=server-ts-password  
ssl.truststore.type=PKCS12  
ssl.client.auth=required
```

Klienty są konfigurowane z wygenerowanym magazynem zaufania. Magazyn kluczy należy skonfigurować dla klientów, jeśli wymagane jest uwierzytelnienie klienta.

```
ssl.truststore.location=/path/to/client.ts.p12
ssl.truststore.password=client-ts-password
ssl.truststore.type=PKCS12
ssl.keystore.location=/path/to/client.ks.p12
ssl.keystore.password=client-ks-password
ssl.key.password=client-ks-password
ssl.keystore.type=PKCS12
```



Magazyny zaufania

Konfigurację magazynu zaufania można pominąć zarówno w brokerach, jak i klientach, gdy korzysta się z certyfikatów podpisanych przez zaufany urząd certyfikacji. W tym przypadku do ustanowienia zaufania wystarczą domyślne magazyny zaufania z instalacji Javy. Etapy instalacji opisaliśmy w rozdziale 2.

Aby uniknąć niepowodzenia procesów handshake TLS, należy okresowo aktualizować magazyny kluczy i zaufania przed wygaśnięciem certyfikatów. Magazyny SSL brokerów można aktualizować dynamicznie przez modyfikowanie tego samego pliku lub ustawienie w opcji konfiguracyjnej nowej wersji pliku. W obu przypadkach do uruchomienia aktualizacji można użyć interfejsu Admin API lub narzędzia configs Kafki. W poniższym przykładzie skorzystaliśmy z narzędzia configs do aktualizacji magazynu kluczy dla zewnętrznego nasłuchiacza brokera o identyfikatorze 0:

```
$ bin/kafka-configs.sh --bootstrap-server localhost:9092
--command-config admin.props
--entity-type brokers --entity-name 0 --alter --add-config
'listener.name.external.ssl.keystore.location=/path/to/server.ks.p12'
```

Kwestie bezpieczeństwa

TLS jest szeroko stosowany do zapewniania bezpieczeństwa warstwy transportowej dla kilku protokołów, w tym HTTPS. Podobnie jak w przypadku każdego protokołu bezpieczeństwa, przy przyjmowaniu określonego protokołu dla zastosowań o kluczowym znaczeniu należy zrozumieć potencjalne zagrożenia i poznać strategię ich ograniczania. Kafka domyślnie włącza tylko nowsze protokoły TLSv1.2 i TLSv1.3, ponieważ starsze protokoły, takie jak TLSv1.1, mają znane luki w zabezpieczeniach. Ze względu na problemy z niezabezpieczoną renegocjacją Kafka nie obsługuje renegocjacji dla połączeń TLS. Weryfikacja nazwy hosta jest domyślnie włączona, aby zapobiec atakom typu „człowiek pośrodku”. Poziom bezpieczeństwa można zwiększyć przez wprowadzenie ograniczeń odnośnie do stosowanych pakietów szyfrów. Silne szyfrowanie z co najmniej 256-bitowym kluczem chroni przed atakami kryptograficznymi i zapewnia integralność danych podczas ich przesyłania przez niezabezpieczoną sieć. Niektóre organizacje wymagają, aby protokół TLS i szyfry były zgodne ze standardami bezpieczeństwa takimi jak FIPS 140-2.

Ponieważ magazyny kluczy zawierające klucze prywatne są domyślnie przechowywane w systemie plików, ważne jest ograniczenie dostępu do plików magazynów kluczy przy użyciu uprawnień systemu plików. Jeśli klucz prywatny jest zagrożony, do unieważnienia certyfikatów można użyć standardowych funkcjonalności TLS Javy. W takim przypadku w celu ograniczenia ryzyka można użyć kluczy krótkotrwałych.

Procesy handshake TLS są kosztowne i znaczną ilość czasu pochłania działanie wątków sieciowych w brokerach. Nasłuchiwalce korzystające z TLS w niezabezpieczonych sieciach powinny być chronione przed atakami DoS za pomocą kwot i limitów połączeń, aby zapewnić dostępność brokerów. W celu opóźnienia odpowiedzi o niepowodzeniu w przypadku nieudanego uwierzytelniania można zastosować opcję konfiguracyjną `connection.failed.authentication.delay.ms`, aby zmniejszyć szybkość ponawiania prób przez klienty.

SASL

Protokół Kafki obsługuje uwierzytelnianie za pomocą SASL i ma wbudowaną obsługę kilku powszechnie stosowanych mechanizmów SASL. SASL można połączyć z TLS jako warstwą transportową, aby zapewnić bezpieczny kanał z uwierzytelnianiem i szyfrowaniem. Uwierzytelnianie SASL odbywa się za pośrednictwem sekwencji żądań serwera i odpowiedzi klienta, gdzie mechanizm SASL definiuje sekwencję i format przesyłania żądań i odpowiedzi. Brokery Kafki od ręki obsługują wymienione poniżej mechanizmy SASL za pomocą konfigurowalnych wywołań zwrotnych w celu zapewnienia integracji z istniejącą infrastrukturą bezpieczeństwa:

GSSAPI

Uwierzytelnianie protokołem Kerberos jest obsługiwane za pomocą SASL/GSSAPI i może być używane do integracji z serwerami Kerberos, takimi jak Active Directory lub OpenLDAP.

PLAIN

Uwierzytelnianie nazwą użytkownika i hasłem używane zwykle z niestandardowym wywołaniem zwrotnym po stronie serwera w celu weryfikacji haseł z zewnętrznego magazynu haseł.

SCRAM-SHA-256 i SCRAM-SHA-512

Uwierzytelnianie nazwą użytkownika i hasłem dostępne w Kafce od ręki bez konieczności stosowania dodatkowych magazynów haseł.

OAuthBEARER

Uwierzytelnianie tokenami okaziciela OAuth używane zwykle z niestandardowymi wywołaniami zwrotnymi do pozyskiwania i walidacji tokenów przyznawanych przez standardowe serwery OAuth.

W brokerze dla każdego nasłuchiwalcy z obsługą SASL można włączyć co najmniej jeden mechanizm SASL, konfigurując parametr `sasl.enabled.mechanisms`. Za pomocą konfiguracji `sasl.mechanism` klienty mogą wybrać dowolny z włączonych mechanizmów.

Do konfigurowania SASL Kafka używa usługi uwierzytelniania i autoryzacji Javy (ang. *Java Authentication and Authorization Service* — JAAS). Opcja `sasl.jaas.config` zawiera pojedynczy wpis konfiguracyjny JAAS, który określa moduł logowania i jego opcje. Podczas konfigurowania `sasl.jaas.config` brokerów korzysta się z prefiksów `listener` i `mechanism`. Parametr `listener.name.external.gssapi.sasl.jaas.config` definiuje np. wpis konfiguracyjny JAAS dla SASL/GSSAPI na nasłuchiwalcu o nazwie EXTERNAL. Proces logowania brokerów i klientów wykorzystuje konfigurację JAAS do określania publicznych i prywatnych poświadczeń używanych do uwierzytelniania.



Plik konfiguracyjny JAAS

Konfigurację JAAS można określić również w plikach konfiguracyjnych za pomocą właściwości systemowej Javy `java.security.auth.login.config`. Zalecane jest jednak stosowanie opcji `sasl.jaas.config` Kafki, ponieważ obsługuje ona ochronę hasłem i osobną konfigurację dla każdego mechanizmu SASL, gdy w nasłuchiwarzu włączonych jest wiele mechanizmów.

Wykorzystując procedury obsługi wywołań zwrotnych, mechanizmy SASL obsługiwane przez Kafkę można dostosowywać do zapewnienia integracji z zewnętrznymi serwerami uwierzytelniającymi. Brokerom lub klientom można zapewnić procedurę obsługi wywołań zwrotnych logowania, aby dostosować proces logowania, np. w celu uzyskania poświadczeń, które mają być używane do uwierzytelniania. Do uwierzytelniania poświadczeń klienta, np. w celu weryfikacji haseł przy użyciu zewnętrznego serwera haseł, można zapewnić procedurę obsługi wywołań zwrotnych serwera. Do wstrzykiwania poświadczeń klienta zamiast załączania ich w konfiguracji JAAS można zapewnić procedurę obsługi wywołań zwrotnych klienta.

W kolejnych podpunktach omówimy szerzej mechanizmy SASL obsługiwane przez Kafkę.

SASL/GSSAPI

Kerberos to szeroko stosowany protokół uwierzytelniania sieciowego, który wykorzystuje silną kryptografię do obsługi bezpiecznego wzajemnego uwierzytelniania w niezabezpieczonej sieci. GSSAPI (ang. *Generic Security Service Application Program Interface*) to framework do świadczenia usług bezpieczeństwa dla aplikacji wykorzystujących różne mechanizmy uwierzytelniania. Dokument RFC 4752 (<https://oreil.ly/wxTZt>) wprowadza mechanizm GSSAPI SASL do uwierzytelniania za pomocą mechanizmu Kerberos V5 frameworku GSSAPI. Dostępność oprogramowania dystrybuowanego na licencji otwartoźródłowej, a także komercyjne implementacje serwerów Kerberos klasy enterprise sprawiły, że Kerberos stał się popularnym wyborem dla uwierzytelniania w wielu sektorach o rygorystycznych wymogach bezpieczeństwa. Kafka obsługuje uwierzytelnianie Kerberos przy użyciu SASL/GSSAPI.

Konfiguracja SASL/GSSAPI

Do obsługi bezpiecznego uwierzytelniania z wykorzystaniem Kerberos Kafka używa dostawców zabezpieczeń GSSAPI zawartych w środowisku uruchomieniowym Javy. Konfiguracja JAAS dla GSSAPI obejmuje ścieżkę pliku tabeli kluczy (ang. *keytab*), który zawiera mapowanie pryncypałów na ich klucze długoterminowe w postaci zaszyfrowanej. Aby skonfigurować GSSAPI dla brokerów, należy dla każdego brokera utworzyć tabelę kluczy z pryncypałem zawierającym nazwę hosta brokera. Nazwy hostów brokera są weryfikowane przez klienty w celu zapewnienia autentyczności serwera i zapobiegania atakom typu „człowiek pośrodku”. Do wyszukiwania nazwy hosta podczas uwierzytelniania Kerberos wymaga bezpiecznej usługi DNS. We wdrożeniach, w których wyszukiwanie standardowe i wyszukiwanie odwrotne nie są zgodne, w pliku konfiguracyjnym Kerberos `krb5.conf` w klientach można skonfigurować wyłączenie wyszukiwania odwrotnego za pomocą ustawienia `rdns=false`. Konfiguracja JAAS dla każdego brokera powinna zawierać moduł logowania Kerberos V5 ze środowiska uruchomieniowego Javy, ścieżkę pliku tabeli kluczy oraz pełnego pryncypała brokera:

```

sasl.enabled.mechanisms=GSSAPI
listener.name.external.gssapi.sasl.jaas.config= ❶
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true
keyTab="/path/to/broker1.keytab" ❷
principal="kafka/broker1.example.com@EXAMPLE.COM"; ❸

```

- ❶ Używamy poprzedzonej prefiksem nasłuchiacza opcji `sasl.jaas.config`, która zawiera nazwę nasłuchiacza i mechanizm SASL zapisane małymi literami.
- ❷ Pliki tabel kluczy muszą być czytelne dla procesu brokera.
- ❸ Pryncypał usługi dla brokerów powinien zawierać nazwę hosta brokera.

Jeśli do komunikacji między brokerami używany jest SASL/GSSAPI, dla brokerów należy skonfigurować również mechanizm SASL i nazwę usługi Kerberos:

```

sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka

```

Klienty powinny być skonfigurowane za pomocą własnego pliku tabeli kluczy i pryncypała w konfiguracji JAAS oraz parametru `sasl.kerberos.service.name`, aby wskazać nazwę usługi, z którą się łączą:

```

sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka ❶
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true
keyTab="/ścieżka/do/alicja.keytab"
principal="Alicja@EXAMPLE.COM"; ❷

```

- ❶ Dla klientów powinna być określona nazwa usługi Kafki.
- ❷ Klienci mogą używać pryncypałów bez nazwy hosta.

Domyślnie jako tożsamość klienta używana jest skrócona nazwa pryncypała. W tym przykładzie pryncypałem klienta jest `User:Alicja`, a pryncypałem brokera `User:kafka`. Aby zastosować listę reguł w celu przekształcenia w pełni kwalifikowanego pryncypała w pryncypała niestandardowego, można wykorzystać konfigurację brokera `sasl.kerberos.principal.to.local.rules`.

Kwestie bezpieczeństwa

We wdrożeniach produkcyjnych wykorzystujących Kerberos do ochrony przepływu uwierzytelniania oraz transferu danych przez połączenie po uwierzytelnieniu zalecane jest użycie SASL_SSL. Jeżeli do zapewnienia bezpiecznej warstwy transportowej nie jest używany TLS, podsłuchiwanie w sieci mogą uzyskać wystarczającą ilość informacji, aby zastosować atak słownikowy (ang. *dictionary*) lub atak brutalnej siły (ang. *brute force*) i ukraść poświadczenia klientów. Bezpieczniej jest używać dla brokerów losowo generowanych kluczy zamiast kluczy generowanych na podstawie haseł, które są łatwiejsze do złamania. Należy unikać słabych algorytmów szyfrowania, takich jak DES-MD5, i stosować silniejsze. Dostęp do plików tabel kluczy musi być ograniczony przy użyciu uprawnień systemu plików, ponieważ każdy użytkownik będący w posiadaniu pliku może podszywać się pod danego użytkownika.

SASL/GSSAPI wymaga bezpiecznej usługi DNS do uwierzytelniania serwera. Ponieważ ataki DoS przeciwko usłudze KDC lub DNS mogą skutkować awariami uwierzytelniania w klientach, konieczne jest monitorowanie dostępności tych usług. Kerberos korzysta ponadto z luźno zsynchronizowanych zegarów z konfigurowalną zmiennością, aby wykrywać ataki powtórzenia (ang. *replay*). Należy się upewnić, że synchronizacja zegarów jest zabezpieczona.

SASL/PLAIN

Dokument RFC 4616 (<https://oreil.ly/wZrxB>) definiuje prosty mechanizm uwierzytelniania za pomocą nazwy użytkownika i hasła, który może być używany z TLS w celu zapewnienia bezpiecznego uwierzytelniania. Podczas uwierzytelniania klient wysyła do serwera nazwę użytkownika i hasło, a serwer weryfikuje hasło, korzystając z magazynu haseł. Kafka ma wbudowane wsparcie dla SASL/PLAIN, które można zintegrować z bezpieczną zewnętrzną bazą haseł za pomocą niestandardowej procedury obsługi wywołań zwrotnych.

Konfigurowanie SASL/PLAIN

Domyślna implementacja SASL/PLAIN wykorzystuje jako magazyn haseł konfigurację JAAS brokera. Wszystkie nazwy użytkowników i hasła klientów są zawarte w opcjach logowania, a broker sprawdza, czy hasło podane przez klienta podczas uwierzytelniania odpowiada jednemu z tych wpisów. Nazwa użytkownika i hasło brokera są wymagane tylko wtedy, gdy do komunikacji między brokerami używany jest SASL/PLAIN:

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.external.plain.sasl.jaas.config=
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="kafka" password="kafka-hasło" ❶
  user_kafka="kafka-hasło"
  user_Alicja="Alicja-hasło"; ❷
```

- ❶ Nazwa użytkownika i hasło używane do połączeń między brokerami inicjowanych przez broker.
- ❷ Gdy klient Alicji łączy się z brokerem, hasło dostarczone przez Alicję jest sprawdzane w konfiguracji brokera.

W celu uwierzytelniania klienty muszą być skonfigurowane z nazwą użytkownika i hasłem:

```
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
  required username="Alicja" password="Alicja-hasło";
```

Wbudowana implementacja, która przechowuje wszystkie hasła w konfiguracji JAAS poszczególnych brokerów, jest niezabezpieczona i mało elastyczna, gdyż w celu dodania lub usunięcia użytkownika trzeba zrestartować wszystkie brokery. Podczas korzystania z SASL/PLAIN w środowisku produkcyjnym można użyć niestandardowej procedury obsługi wywołań zwrotnych serwera, aby zintegrować brokery z bezpiecznym zewnętrznym serwerem haseł. Niestandardowe procedury obsługi wywołań zwrotnych mogą być stosowane również do obsługi rotacji haseł. Po stronie serwera procedura obsługi wywołań zwrotnych serwera powinna obsługiwać zarówno stare, jak i nowe hasła przez okres ich nakładania się, dopóki wszystkie klienty nie przełączą się na nowe hasła.

W poniższym przykładzie pokazaliśmy procedurę obsługi wywołań zwrotnych, która weryfikuje zaszyfrowane hasła z plików wygenerowanych za pomocą narzędzia `htpasswd` serwera Apache:

```
public class PasswordVerifier extends PlainServerCallbackHandler {

    private final List<String> passwdFiles = new ArrayList<>(); ❶

    @Override
    public void configure(Map<String, ?> configs, String mechanism,
        List<AppConfigurationEntry> jaasEntries) {
        Map<String, ?> loginOptions = jaasEntries.get(0).getOptions();
        String files = (String) loginOptions.get("password.files"); ❷
        Collections.addAll(passwdFiles, files.split(","));
    }

    @Override
    protected boolean authenticate(String user, char[] password) {
        return passwdFiles.stream().anyMatch(file -> authenticate(file, user, password)); ❸
    }

    private boolean authenticate(String file, String user, char[] password) {
        try {
            String cmd = String.format("htpasswd -vb %s %s %s", ❹
                file, user, new String(password));
            return Runtime.getRuntime().exec(cmd).waitFor() == 0;
        } catch (Exception e) {
            return false;
        }
    }
}
```

- ❶ Używamy wielu plików haseł, aby móc obsługiwać rotację haseł.
- ❷ W konfiguracji brokera przekazujemy ścieżki plików haseł jako opcję JAAS. Można również użyć niestandardowych opcji konfiguracyjnych brokera.
- ❸ Sprawdzamy, czy hasło pasuje do któregośkolwiek z plików, co umożliwia używanie przez pewien czas zarówno starych, jak i nowych haseł.
- ❹ Dla uproszczenia używamy narzędzia `htpasswd`. Dla wdrożeń produkcyjnych może być używana bezpieczna baza danych.

Brokery są konfigurowane z procedurą obsługi wywołań zwrotnych dla walidacji haseł i z jej opcjami:

```
listener.name.external.plain.sasl.jaas.config=
org.apache.kafka.common.security.plain.PlainLoginModule required
password.files="/ścieżka/do/htpassword.props,/ścieżka/do/oldhtpassword.props";
listener.name.external.plain.sasl.server.callback.handler.class=
com.example.PasswordVerifier
```

Po stronie klienta procedura obsługi wywołań zwrotnych klienta, która implementuje interfejs `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler`, może być używana do dynamicznego ładowania haseł podczas działania aplikacji, gdy nawiązywane jest połączenie, zamiast statycznego ładowania z konfiguracji JAAS podczas uruchamiania. Hasła mogą być ładowane z zaszyfrowanych plików lub przy użyciu zewnętrznego bezpiecznego serwera w celu poprawienia

poziomu bezpieczeństwa. Poniższy przykład definiuje dynamiczne ładowanie haseł z pliku za pomocą klas konfiguracyjnych z Kafki:

```
@Override
public void handle(Callback[] callbacks) throws IOException {
    Properties props = Utils.loadProps(passwdFile); ❶
    PasswordConfig config = new PasswordConfig(props);
    String user = config.getString("username");
    String password = config.getPassword("password").value(); ❷
    for (Callback callback: callbacks) {
        if (callback instanceof NameCallback)
            ((NameCallback) callback).setName(user);
        else if (callback instanceof PasswordCallback) {
            ((PasswordCallback) callback).setPassword(password.toCharArray());
        }
    }
}

private static class PasswordConfig extends AbstractConfig {
    static ConfigDef CONFIG = new ConfigDef()
        .define("username", STRING, HIGH, "Nazwa użytkownika")
        .define("password", PASSWORD, HIGH, "Hasło użytkownika"); ❸
    PasswordConfig(Properties props) {
        super(CONFIG, props, false);
    }
}
```

- ❶ W celu obsługi rotacji haseł wczytujemy plik konfiguracyjny w ramach wywołania zwrotnego, aby mieć pewność, że używamy najnowszego hasła.
- ❷ Bazowa biblioteka konfiguracji zwraca rzeczywistą wartość hasła, nawet jeśli hasło jest zewnętrzne.
- ❸ Definiujemy konfigurację haseł za pomocą typu PASSWORD, aby się upewnić, że hasła nie będą załączane we wpisach dziennika.

Klienci, tak samo jak brokery, które do komunikacji między sobą używają SASL/PLAIN, można skonfigurować z wywołaniem zwrotnym po stronie klienta:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
    required file="/ścieżka/do/credentials.props";
sasl.client.callback.handler.class=com.example.PasswordProvider
```

Kwestie bezpieczeństwa

Ponieważ SASL/PLAIN przesyła hasła w postaci zwykłego tekstu, mechanizm PLAIN powinno się włączać tylko z szyfrowaniem SASL_SSL w celu zapewnienia bezpiecznej warstwy transportowej. Hasła przechowywane w postaci zwykłego tekstu w konfiguracji JAAS brokerów i klientów nie są bezpieczne, należy więc rozważyć szyfrowanie lub zewnętrzne przechowywanie tych haseł w bezpiecznym magazynie. Zamiast używać wbudowanego magazynu haseł, który przechowuje wszystkie hasła klientów w konfiguracji brokera JAAS, lepiej skorzystać z bezpiecznego zewnętrznego serwera, który przechowuje zabezpieczone hasła i egzekwuje silne reguły dotyczące haseł.



Hasła w postaci zwykłego tekstu

Unikaj haseł tekstowych w plikach konfiguracyjnych, nawet jeśli te pliki mogą być chronione przy użyciu uprawnień systemu plików. Rozważ zewnętrzną obsługę lub szyfrowanie haseł, by mieć pewność, że hasła nie zostaną przypadkowo ujawnione. Funkcjonalność ochrony haseł Kafki została opisana dalej w tym rozdziale.

SASL/SCRAM

Dokument RFC 5802 (<https://oreil.ly/dXe3y>) wprowadza bezpieczny mechanizm uwierzytelniania za pomocą nazwy użytkownika i hasła, który rozwiązuje problemy związane z bezpieczeństwem mechanizmów uwierzytelniania haseł, takich jak SASL/PLAIN wysyłający hasła przez sieć. SCRAM (ang. *Soltded Challenge Response Authentication Mechanism*) pozwala uniknąć przesyłania haseł w postaci zwykłego tekstu i przechowuje hasła w formacie, który sprawia, że podszywanie się pod klienty staje się niepraktyczne. Solenie łączy hasła z losowymi danymi przed zastosowaniem jednokierunkowej funkcji skrótu kryptograficznego w celu bezpiecznego zapisania haseł. Kafka ma wbudowanego dostawcę SCRAM, który może być używany we wdrożeniach z bezpiecznym ZooKeeperem bez konieczności stosowania dodatkowych serwerów haseł. Dostawca Kafki obsługuje mechanizmy SCRAM-SHA-256 i SCRAM-SHA-512.

Konfigurowanie SASL/SCRAM

Początkowy zestaw użytkowników można utworzyć po uruchomieniu ZooKeepera i przed uruchomieniem brokerów. Brokery ładują metadane użytkowników SCRAM do pamięci podręcznej podczas uruchamiania, dzięki czemu wszyscy użytkownicy, w tym użytkownik brokera do komunikacji między brokerami, mogą się pomyślnie uwierzytelnić. Użytkowników można dodawać lub usuwać w dowolnym momencie. Brokery aktualizują pamięć podręczną za pomocą powiadomień opartych na obserwatorze ZooKeepera. W tym przykładzie tworzymy użytkownika z pryncypałem User:Alicja i hasłem Alicja-hasło dla mechanizmu SCRAM-SHA-512 SASL:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config  
'SCRAM-SHA-512=[iterations=8192,password=Alicja-hasło]'  
--entity-type users --entity-name Alicja
```

Na nasłuchiwarzu można włączyć co najmniej jeden mechanizm SCRAM, konfigurując mechanizmy w brokerze. Nazwa użytkownika i hasło są wymagane dla brokerów tylko wtedy, gdy dany nasłuchiwarz jest używany do komunikacji między brokerami:

```
sasl.enabled.mechanisms=SCRAM-SHA-512  
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512  
listener.name.external.scram-sha-512.sasl.jaas.config=  
org.apache.kafka.common.security.scram.ScramLoginModule required  
username="kafka" password="kafka-hasło"; ❶
```

❶ Nazwa użytkownika i hasło do połączeń między brokerami inicjowanych przez ten broker.

W klientach musi być skonfigurowane korzystanie z jednego z mechanizmów SASL włączonych w brokerze, a konfiguracja JAAS klienta musi zawierać nazwę użytkownika i hasło:

```
sasl.mechanism=SCRAM-SHA-512  
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule  
required username="Alicja" password="Alicja-hasło";
```

Nowych użytkowników SCRAM można dodawać za pomocą opcji `--add-config`, a usuwać za pomocą opcji `--delete-config` narzędzia `configs`. Gdy istniejący użytkownik zostanie usunięty, nie można ustawić dla niego nowych połączeń, ale istniejące połączenia użytkownika będą nadal działały. Dla brokera można skonfigurować interwał ponownego uwierzytelniania, aby ograniczyć czas, przez jaki istniejące połączenia będą mogły działać po usunięciu użytkownika. Poniższy przykład powoduje usunięcie konfiguracji SCRAM-SHA-512 dla użytkownika Alicja, aby usunąć poświadczenia Alicji dla tego mechanizmu:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config  
'SCRAM-SHA-512' --entity-type users --entity-name Alicja
```

Kwestie bezpieczeństwa

SCRAM stosuje do hasła jednokierunkową funkcję skrótu kryptograficznego w połączeniu z losowym soleniem, aby uniknąć przesyłania przez sieć lub zapisywania w bazie danych rzeczywistego hasła. Jednak każdy system oparty na hasłach jest tak bezpieczny jak same hasła. Aby chronić system przed atakami brutalnej siły lub atakami słownikowymi, należy wdrożyć rygorystyczne reguły dotyczące haseł. Kafka zapewnia zabezpieczenia przez to, że obsługuje tylko silne algorytmy mieszające SHA-256 i SHA-512, unikając słabszych algorytmów takich jak SHA-1. Jest to połączone z dużą domyślną liczbą iteracji 4096 i unikatowym losowym soleniem dla każdego zapisywanego klucza, aby ograniczyć negatywne konsekwencje naruszenia bezpieczeństwa ZooKeepera.

Aby chronić klucze przesyłane podczas procesu handshake i klucze przechowywane w ZooKeeperze przed atakami brutalnej siły, należy podjąć dodatkowe środki ostrożności. SCRAM musi być używany z protokołem bezpieczeństwa SASL_SSL, aby uniemożliwić podsłuchiwancom uzyskanie dostępu do skrótów kluczy podczas uwierzytelniania. ZooKeeper również musi mieć włączoną obsługę SSL, a dane ZooKeepera muszą być chronione za pomocą szyfrowania dyskowego, by udaremnić pobranie przechowywanych kluczy, nawet jeśli naruszone zostaną zabezpieczenia magazynu. W celu zapewnienia integracji z bezpiecznym zewnętrznym magazynem poświadczeń we wdrożeniach bez bezpiecznego ZooKeepera można używać wywołań zwrotnych SCRAM.

SASL/OAUTHBEARER

OAuth to framework autoryzacji, który umożliwia aplikacjom uzyskiwanie ograniczonego dostępu do usług HTTP. Dokument RFC 7628 (<https://oreil.ly/sPBfv>) definiuje mechanizm OAUTHBEARER SASL, dzięki któremu można uzyskać dostęp do chronionych zasobów w protokołach innych niż HTTP za pomocą poświadczeń uzyskanych przy użyciu OAuth 2.0. Stosując tokeny okaziciela OAuth 2.0 z krótszym okresem życia i ograniczonym dostępem do zasobów, OAUTHBEARER pozwala uniknąć luk w zabezpieczeniach w mechanizmach, które korzystają z haseł długoterminowych. Kafka obsługuje SASL/OAUTHBEARER dla uwierzytelniania klienta, co umożliwia integrację z zewnętrznymi serwerami OAuth. Wbudowana implementacja OAUTHBEARER wykorzystuje niezabezpieczone tokeny sieciowe JSON (ang. *JSON Web Token* — JWT) i nie nadaje się do użytku w środowisku produkcyjnym. W celu przeprowadzenia integracji ze standardowymi serwerami OAuth, aby zapewnić bezpieczne uwierzytelnianie za pomocą mechanizmu OAUTHBEARER we wdrożeniach produkcyjnych, można dodawać niestandardowe wywołania zwrotne.

Konfigurowanie SASL/OAUTHBEARER

Wbudowana implementacja SASL/OAUTHBEARER Kafki nie waliduje tokenów i dlatego wymaga określenia w konfiguracji JAAS jedynie modułu logowania. Jeżeli do komunikacji między brokerami używany jest nasłuchiwaniec, należy również podać szczegóły tokena używanego do połączeń klienckich inicjowanych przez brokery. Opcja `unsecuredLoginStringClaim_sub` to roszczenie podmiotu (ang. *subject claim*), które domyślnie określa instancję `KafkaPrincipal` dla połączenia:

```
sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
listener.name.external.oauthbearer.sasl.jaas.config=
  org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
  required unsecuredLoginStringClaim_sub="kafka"; ❶
```

- ❶ Roszczenie podmiotu dla tokena używanego do połączeń między brokerami.

Klienty muszą być skonfigurowane z opcją roszczenia podmiotu `unsecuredLoginStringClaim_sub`. Można również skonfigurować inne roszczenia i okres życia tokena:

```
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=
  org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
  required unsecuredLoginStringClaim_sub="Alicja"; ❶
```

- ❶ User: Alicja jest domyślną instancją `KafkaPrincipal` dla połączeń korzystających z tej konfiguracji.

Aby zintegrować Kafkę z zewnętrznymi serwerami OAuth w celu wykorzystania tokenów okaziciela w środowisku produkcyjnym, klienty Kafki muszą być skonfigurowane z opcją `sasl.login.callback.handler.class` umożliwiającą nabywanie tokenów z serwera OAuth przy użyciu hasła długoterminowego lub tokena odświeżania. Jeśli OAUTHBEARER jest używany do komunikacji między brokerami, brokery trzeba ponadto skonfigurować z procedurą obsługi wywołań zwrotnych logowania w celu pozyskiwania tokenów dla połączeń klienckich tworzonych przez broker:

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    OAuthBearerToken token = null;
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerTokenCallback) {
            token = acquireToken(); ❶
            ((OAuthBearerTokenCallback) callback).token(token);
        } else if (callback instanceof SaslExtensionsCallback) { ❷
            ((SaslExtensionsCallback) callback).extensions(processExtensions(token));
        } else
            throw new UnsupportedCallbackException(callback);
    }
}
```

- ❶ Klienty muszą nabyć token z serwera OAuth i ustawić prawidłowy token w wywołaniu zwrotnym.
- ❷ Klient może zawierać także opcjonalne rozszerzenia.

Brokery muszą być ponadto skonfigurowane z procedurą obsługi wywołań zwrotnych serwera przy użyciu klasy `listener.name.<nazwa_nasłuchiwarza>.oauthbearer.sasl.server.callback.handler`.
↳ class do walidacji tokenów dostarczanych przez klient:

```
@Override
public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    for (Callback callback : callbacks) {
        if (callback instanceof OAuthBearerValidatorCallback) {
            OAuthBearerValidatorCallback cb = (OAuthBearerValidatorCallback) callback;
            try {
                cb.token(validatedToken(cb.tokenValue())); ❶
            } catch (OAuthBearerIllegalTokenException e) {
                OAuthBearerValidationResult r = e.reason();
                cb.error(errorStatus(r), r.failureScope(), r.failureOpenIdConfig());
            }
        } else if (callback instanceof OAuthBearerExtensionsValidatorCallback) {
            OAuthBearerExtensionsValidatorCallback ecb =
                (OAuthBearerExtensionsValidatorCallback) callback;
            ecb.inputExtensions().map().forEach((k, v) ->
                ecb.valid(validateExtension(k, v))); ❷
        } else {
            throw new UnsupportedCallbackException(callback);
        }
    }
}
```

- ❶ Wywołanie zwrotne `OAuthBearerValidatorCallback` zawiera token od klienta. Ten token jest walidowany przez brokery.
- ❷ Brokery przeprowadzają walidację wszelkich opcjonalnych rozszerzeń od klienta.

Kwestie bezpieczeństwa

Ponieważ klienci SASL/OAUTHBEARER przesyłają przez sieć tokeny okaziciela OAuth 2.0, które mogą zostać użyte do podszywania się pod klienty, do szyfrowania ruchu uwierzytelniającego musi być włączony TLS. W celu ograniczenia skutków naruszenia bezpieczeństwa tokenów można stosować tokeny krótkotrwałe. Dla brokerów można włączyć ponowne uwierzytelnianie, aby uniemożliwić połączenia po wygaśnięciu tokenów używanych do uwierzytelniania. Interwał ponownego uwierzytelniania skonfigurowany w brokerach w połączeniu z obsługą unieważniania tokenów ogranicza czas, przez jaki istniejące połączenie może nadal korzystać z tokena po jego unieważnieniu.

Tokeny delegowania

Tokeny delegowania (ang. *delegation token*) to sekrety współdzielone przez brokery i klienty Kafki. Zapewniają lekki mechanizm konfiguracji bez konieczności dystrybucji magazynów kluczy SSL lub tablic kluczy Kerberos do aplikacji klienckich. Tokeny delegowania można wykorzystać do zmniejszenia obciążenia serwerów uwierzytelniających, takich jak KDC (ang. *Key Distribution Center*) Kerberos. Frameworki takie jak Kafka Connect mogą używać tokenów delegowania w celu uproszczenia konfiguracji zabezpieczeń dla węzłów roboczych. Klient, który uwierzytlnił się w brokerach Kafki, może tworzyć tokeny delegowania dla tego samego pryncypała użytkownika i dystrybuować je do węzłów roboczych, a te mogą wtedy uwierzytelniać się bezpośrednio w brokerach Kafki. Każdy token delegowania składa się z identyfikatora tokena i kodu HMAC (ang. *Hash-based*

Message Authentication Code) używanego jako współdzielony sekret. Uwierzytelnianie klienta tokenami delegowania odbywa się przy użyciu SASL/SCRAM z identyfikatorem tokena jako nazwą użytkownika i kodem HMAC jako hasłem.

Tokeny delegowania można tworzyć lub odnawiać za pomocą interfejsu Admin API Kafki lub polecenia `delegation-tokens`. Aby tworzyć tokeny delegacji dla pryncypała `User:Alicja`, klient musi być uwierzytelniony przy użyciu poświadczeń Alicji dla dowolnego protokołu uwierzytelniania innego niż tokeny delegowania. Klienci uwierzytelnione za pomocą tokenów delegowania nie mogą tworzyć innych tokenów delegowania:

```
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092
--command-config admin.props --create --max-life-time-period -1
--renewer-principal User:Bob ❶
$ bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 ❷
--command-config admin.props --renew --renew-time-period -1 --hmac c2VjcmV0
```

- ❶ Jeśli Alicja uruchomi to polecenie, wygenerowany token może zostać użyty do podszywania się pod Alicję. Właścicielem tego tokena jest `User:Alicja`. Konfigurujemy również pryncypała `User:Bob` jako odnawiacz tokena.
- ❷ Polecenie odnowienia może być uruchamiane przez właściciela tokena (Alicję) lub odnawiacza tokena (Boba).

Konfigurowanie tokenów delegowania

Aby tworzyć i weryfikować tokeny delegowania, wszystkie brokery muszą być skonfigurowane z tym samym kluczem głównym za pomocą opcji konfiguracyjnej `delegation.token.master.key`. Ten klucz można rotować tylko przez restartowanie wszystkich brokerów. Przed aktualizacją klucza głównego wszystkie istniejące tokeny trzeba usunąć, ponieważ nie można ich już używać, a nowe tokeny należy utworzyć po aktualizacji klucza we wszystkich brokerach.

W celu zapewnienia obsługi uwierzytelniania przy użyciu tokenów delegowania w brokerach musi być włączony co najmniej jeden z mechanizmów SASL/SCRAM. Klienci powinny być skonfigurowane do używania SCRAM z identyfikatorem tokena jako nazwą użytkownika i kodem HMAC jako hasłem. Instancją `KafkaPrincipal` dla połączeń wykorzystujących tę konfigurację będzie pierwotny pryncypał powiązany z tym tokenem, np. `User:Alicja`:

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required tokenauth="true" username="MTIz" password="c2VjcmV0"; ❶
```

- ❶ Do zdefiniowania tokenów delegowania stosowana jest konfiguracja SCRAM z `tokenauth`.

Kwestie bezpieczeństwa

Podobnie jak wbudowana implementacja SCRAM, tokeny delegowania nadają się do użytku w środowisku produkcyjnym tylko we wdrożeniach, w których ZooKeeper jest zabezpieczony. Wszystkie względy bezpieczeństwa, które opisaliśmy dla SCRAM, mają zastosowanie również do tokenów delegowania.

Klucz główny wykorzystywany przez brokery do generowania tokenów musi być chroniony za pomocą szyfrowania lub przez zewnętrzne przechowywanie klucza w bezpiecznym magazynie haseł.

W celu ograniczenia skutków naruszenia bezpieczeństwa tokenów można stosować tokeny krótkotrwałe. Dla brokerów można włączyć ponowne uwierzytelnianie, aby uniemożliwić połączenia po wygaśnięciu tokenów używanych oraz ograniczyć czas, przez jaki istniejące połączenia mogą działać po usunięciu tokena.

Ponowne uwierzytelnianie

Jak pisaliśmy wcześniej, brokery Kafki przeprowadzają uwierzytelnianie klienta po nawiązaniu przez niego połączenia. Poświadczenia klienta są weryfikowane przez brokery i jeśli są w danym momencie ważne, połączenie uwierzytelnia się pomyślnie. Niektóre mechanizmy bezpieczeństwa, takie jak Kerberos i OAuth, używają poświadczeń o ograniczonym czasie ważności. Aby uzyskać nowe poświadczenia przed wygaśnięciem starych, Kafka wykorzystuje działający w tle wątek logowania, ale nowe poświadczenia są domyślnie używane tylko do uwierzytelniania nowych połączeń. Istniejące połączenia, które zostały uwierzytelnione za pomocą starych poświadczeń, kontynuują przetwarzanie żądań, dopóki nie nastąpi rozłączenie z powodu przekroczenia limitu czasu żądania, przekroczenia limitu czasu bezczynności lub błędów sieci. Długotrwałe połączenia mogą kontynuować przetwarzanie żądania długo po wygaśnięciu poświadczeń użytych do ich uwierzytelniania. Za pośrednictwem opcji konfiguracyjnej `connections.max.reauth.ms` brokery Kafki obsługują ponowne uwierzytelnianie dla połączeń uwierzytelnianych przy użyciu SASL. Gdy dla tej opcji ustawiona jest dodatnia liczba całkowita, brokery Kafki określają czas życia sesji dla połączeń SASL i informują o nim klienty podczas procesu handshake SASL. Czas życia sesji to pozostały czas ważności poświadczeń lub `connections.max.reauth.ms`, w zależności od tego, która wartość jest mniejsza. Broker zakończy każde połączenie, które nie zostanie ponownie uwierzytelnione w tym przedziale czasu. Klienci przeprowadzają ponowne uwierzytelnianie przy użyciu najnowszych poświadczeń uzyskanych przez działający w tle wątek logowania lub wstrzykniętych za pomocą niestandardowych wywołań zwrotnych. Ponowne uwierzytelnianie można wykorzystać do zwiększenia poziomu bezpieczeństwa w kilku scenariuszach:

- W przypadku mechanizmów SASL, takich jak GSSAPI i OAUTHBEARER, które wykorzystują poświadczenia o ograniczonym okresie ważności, ponowne uwierzytelnianie gwarantuje, że wszystkie aktywne połączenia będą powiązane z ważnymi poświadczeniami. Krótkotrwałe poświadczenia ograniczają negatywne skutki na wypadek naruszenia bezpieczeństwa poświadczeń.
- Mechanizmy SASL oparte na hasłach, takie jak PLAIN i SCRAM, mogą obsługiwać rotację haseł przez dodanie okresowego logowania. Ponowne uwierzytelnienie ogranicza czas przetwarzania żądań na połączeniach uwierzytelnionych za pomocą starego hasła. Aby uniknąć przerw w działaniu, dopóki wszystkie klienty nie przejdą na nowe hasło, można użyć niestandardowego wywołania zwrotnego serwera, które dopuszcza stosowanie przez pewien czas zarówno starego, jak i nowego hasła.
- Opcja `connections.max.reauth.ms` wymusza ponowne uwierzytelnianie we wszystkich mechanizmach SASL, również w tych z niewygasającymi poświadczeniami. Ogranicza to czas, przez jaki unieważnione poświadczenia mogą być powiązane z aktywnym połączeniem.

- Wychodzące od klientów połączenia bez obsługi ponownego uwierzytelniania SASL są zamykane po wygaśnięciu sesji. Zmusza to klienty do ponownego nawiązania połączenia i uwierzytelnienia, zapewniając te same gwarancje bezpieczeństwa dla wygasłych lub unieważnionych poświadczeń.



Naruszenie bezpieczeństwa użytkownika

Jeśli naruszone zostaną zabezpieczenia jakiegoś użytkownika, należy podjąć działania w celu jak najszybszego usunięcia go z systemu. Po usunięciu użytkownika z serwera uwierzytelniania żadne nowe połączenia nie będą mogły uwierzytelnić się w brokerach Kafki. Istniejące połączenia będą kontynuować przetwarzanie żądań, dopóki nie upłynie limit czasu dla następnego uwierzytelnienia. Jeśli opcja `connections.max.reauth.ms` nie będzie skonfigurowana, limit czasu nie zostanie zastosowany, a istniejące połączenia będą mogły dalej korzystać z tożsamości użytkownika, którego bezpieczeństwo zostało naruszone. Kafka nie obsługuje renegocjacji SSL ze względu na znane luki w zabezpieczeniach w starszych protokołach SSL. Nowsze protokoły takie jak TLSv1.3 nie obsługują renegocjacji. Istniejące połączenia SSL mogą więc nadal korzystać z unieważnionych lub wygasłych certyfikatów. By uniemożliwić tym połączeniom wykonywanie jakichkolwiek operacji, można użyć list Deny ACL dla pryncypała użytkownika. Ponieważ zmiany ACL są stosowane z bardzo małymi opóźnieniami we wszystkich brokerach, jest to najszybszy sposób na wyłączenie dostępu dla użytkowników, których zabezpieczenia zostały naruszone.

Aktualizowanie zabezpieczeń bez przestojów

Wdrożenia Kafki wymagają regularnej konserwacji polegającej na rotowaniu sekretów, instalowaniu poprawek bezpieczeństwa i aktualizowaniu protokołów bezpieczeństwa do najnowszych wersji. Wiele z tych zadań konserwacyjnych jest wykonywanych za pomocą aktualizacji kroczących, gdzie brokery są po kolei wyłączane i ponownie uruchamiane ze zaktualizowaną konfiguracją. Niektóre zadania, takie jak aktualizacja magazynów kluczy SSL i magazynów zaufania, mogą być wykonywane przy użyciu dynamicznych aktualizacji konfiguracji bez restartowania brokerów.

Podczas dodawania nowego protokołu bezpieczeństwa do istniejącego wdrożenia można dodać do brokerów nowy nasłuchiwaniec z nowym protokołem, zachowując stary nasłuchiwaniec ze starym protokołem, aby mieć pewność, że w trakcie aktualizacji aplikacje klienckie będą mogły kontynuować działanie przy użyciu starego nasłuchiwanca. Poniższą sekwencję czynności możesz zastosować np. do przełączenia się w istniejącym wdrożeniu z PLAINTEXT na SASL_SSL:

1. Za pomocą narzędzia `configs` Kafki do każdego brokera dodaj nowego nasłuchiwanca na nowym porcie. Użyj pojedynczego polecenia aktualizacji konfiguracji w celu zaktualizowania nasłuchiwanca (`listeners`) i nasłuchiwanca rozgłaszanego (`advertised.listeners`), aby uwzględnić zarówno stary, jak i nowy nasłuchiwaniec. Przy użyciu prefiksu `listener` określ wszystkie opcje konfiguracyjne dla nowego nasłuchiwanca SASL_SSL.
2. Zmodyfikuj wszystkie aplikacje klienckie, aby korzystały z nowego nasłuchiwanca SASL_SSL.
3. Jeśli komunikacja między brokerami jest aktualizowana w celu używania nowego nasłuchiwanca SASL_SSL, wykonaj aktualizację kroczącą brokerów za pomocą nowej wartości `inter.broker.listener.name`.

4. Użyj narzędzia `configs`, aby usunąć stary nasłuchiwaniec z `listeners` i `advertised.listeners` oraz usunąć wszelkie niewykorzystywane opcje konfiguracyjne starego nasłuchiwanca.

Na tym samym porcie nasłuchiwanca można dodawać mechanizmy SASL do istniejących nasłuchiwaniec SASL lub usuwać je z nich bez przestojów za pomocą aktualizacji kroczących. Poniższą sekwencję czynności możesz zastosować do przełączenia się z mechanizmu PLAIN na SCRAM-SHA-256:

1. Przy użyciu narzędzia `configs` Kafki dodaj wszystkich istniejących użytkowników do magazynu SCRAM.
2. Ustaw `sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256`, skonfiguruj dla danego nasłuchiwanca `listener.name.<nazwa_nasłuchiwanca>.scram-sha-256.sasl.jaas.config` i wykonaj aktualizację kroczącą brokerów.
3. Zmodyfikuj wszystkie aplikacje klienckie, aby stosowały `sasl.mechanism=SCRAM-SHA-256`, i zaktualizuj `sasl.jaas.config` do korzystania ze SCRAM.
4. Jeśli dany nasłuchiwaniec jest używany do komunikacji między brokerami, użyj kroczącej aktualizacji brokerów, by ustawić `sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256`.
5. Wykonaj kolejną aktualizację kroczącą brokerów w celu usunięcia mechanizmu PLAIN. Ustaw `sasl.enabled.mechanisms=SCRAM-SHA-256` i usuń `listener.name.<nazwa_nasłuchiwanca>.plain.sasl.jaas.config` i wszelkie inne opcje konfiguracyjne dla PLAIN.

Szyfrowanie

Szyfrowanie jest stosowane w celu zachowania prywatności i integralności danych. Jak wspomnieliśmy wcześniej, nasłuchiwaniec Kafki korzystające z protokołów bezpieczeństwa SSL i SASL_SSL używają TLS jako warstwy transportowej, zapewniając bezpieczne, szyfrowane kanały, które chronią dane przesyłane przez niezabezpieczoną sieć. Stosowane pakiety szyfrów TLS można ograniczyć w celu zwiększenia poziomu bezpieczeństwa i spełnienia wymogów bezpieczeństwa, takich jak FIPS (ang. *Federal Information Processing Standard*).

W celu ochrony danych w stanie spoczynku, by uniemożliwić pozyskanie poufnych danych nawet użytkownikom z fizycznym dostępem do dysku, na którym przechowywane są dzienniki Kafki, należy zastosować dodatkowe środki bezpieczeństwa. Aby uniknąć naruszenia bezpieczeństwa, nawet jeśli dysk zostanie skradziony, fizyczna pamięć masowa może zostać zaszyfrowana za pomocą szyfrowania całego dysku lub szyfrowania woluminu.

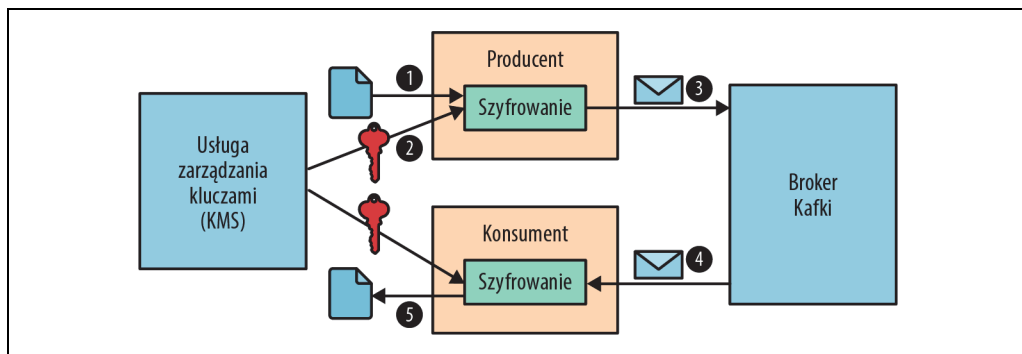
Chociaż szyfrowanie warstwy transportowej i magazynu danych może zapewnić odpowiednią ochronę w wielu wdrożeniach, wymagana może być dodatkowa ochrona, aby uniknąć automatycznego przyznawania dostępu do danych administratorom platformy. Niezaszyfrowane dane obecne w pamięci brokera mogą pojawiać się w zrzutach sterty, a administratorzy z bezpośrednim dostępem do dysku będą mogli uzyskać dostęp zarówno do nich, jak i do dzienników Kafki, które mogą zawierać poufne dane. We wdrożeniach ze szczególnie chronionymi danymi lub z danymi osobowymi (ang. *Personally Identifiable Information* — PII) wymagane są dodatkowe środki w celu zachowania prywatności danych. Aby zapewnić zgodność z regulacjami prawnymi, zwłaszcza w przypadku wdrożeń w chmurze, konieczne jest zagwarantowanie, że administratorzy platformy lub dostawcy usług

w chmurze nie będą mieli żadnego dostępu do poufnych danych. W celu zaimplementowania szyfrowania kompleksowego (ang. *end-to-end*), które gwarantuje szyfrowanie całego przepływu danych, do klientów Kafki można podłączyć niestandardowych dostawców szyfrowania.

Szyfrowanie kompleksowe

W rozdziale 3. poświęconym producentom Kafki dowiedziałeś się, że do konwersji komunikatów na tablice bajtów zapisywane w dziennikach Kafki używane są **serializatory**, a w rozdziale 4. dotyczącym konsumentów Kafki nauczyłeś się, że do konwertowania tablic bajtów z powrotem na komunikaty wykorzystuje się **deserializatory**. Serializatory i deserializatory można zintegrować z biblioteką szyfrowania w celu szyfrowania komunikatów podczas serializacji i deszyfrowania podczas deserializacji. Komunikaty szyfruje się zwykle przy użyciu algorytmów symetrycznych takich jak AES. Współdzielony klucz szyfrowania przechowywany w systemie zarządzania kluczami (ang. *key management system* — KMS) umożliwia producentom szyfrowanie komunikatów, a konsumentom odszyfrowywanie komunikatów. Brokery nie wymagają dostępu do klucza szyfrowania i nigdy nie widzą niezaszyfrowanej zawartości komunikatu, dzięki czemu takie podejście jest bezpieczne w środowiskach chmurowych. Parametry szyfrowania wymagane do odszyfrowywania komunikatów mogą być przechowywane w nagłówkach komunikatów albo w ładunku komunikatów, jeśli dostępu do komunikatów potrzebują starsze konsumenty bez obsługi nagłówka. W nagłówkach komunikatów zawarta może być również sygnatura cyfrowa w celu weryfikacji integralności komunikatów.

Na rysunku 11.2 przedstawiliśmy przepływ danych Kafki z szyfrowaniem kompleksowym.



Rysunek 11.2. Szyfrowanie kompleksowe

- ❶ Wysyłamy komunikat za pośrednictwem producenta Kafki.
- ❷ Producent używa klucza szyfrującego z usługi KMS, aby zaszyfrować komunikat.
- ❸ Zaszyfrowany komunikat jest wysyłany do brokera. Broker zapisuje zaszyfrowany komunikat w dziennikach partycji.
- ❹ Broker wysyła zaszyfrowany komunikat do konsumentów.
- ❺ Konsument używa klucza szyfrującego z usługi KMS, aby odszyfrować komunikat.

Dla producentów i konsumentów trzeba skonfigurować poświadczenia, by mogły pozyskiwać współdzielone klucze z KMS. Aby zwiększyć poziom bezpieczeństwa, zaleca się okresową rotację kluczy, ponieważ częsta rotacja ogranicza liczbę ujawnionych komunikatów w przypadku naruszenia zabezpieczeń, a ponadto chroni przed atakami brutalnej siły. Podczas okresu przechowywania komunikatów zaszyfrowanych starym kluczem musi być obsługiwane konsumowanie za pomocą zarówno starych, jak i nowych kluczy. Wiele systemów KMS od ręki obsługuje płynną rotację kluczy dla szyfrowania symetrycznego bez konieczności zapewniania specjalnej obsługi w klientach Kafki. W przypadku tematów skompaktowanych komunikaty zaszyfrowane starymi kluczami mogą być przechowywane przez długi czas i konieczne może być ponowne szyfrowanie starych komunikatów. Aby uniknąć ingerencji w nowsze komunikaty, podczas tego procesu producenci i konsumenci muszą pozostawać w trybie offline.



Kompresja zaszyfrowanych komunikatów

Jest mało prawdopodobne, aby kompresja komunikatów po szyfrowaniu przyniosła jakąkolwiek korzyść pod względem zmniejszenia zajmowanej przestrzeni w porównaniu z kompresją przed szyfrowaniem. Można skonfigurować serializatory do wykonywania kompresji przed szyfrowaniem komunikatów lub aplikacje do wykonywania kompresji przed wyprodukowaniem komunikatów. W obu przypadkach lepiej jest wyłączyć kompresję w Kafce, ponieważ generuje ona narzut, nie zapewniając żadnych dodatkowych korzyści. Dla komunikatów przesyłanych przez niezabezpieczoną warstwę transportową należy również wziąć pod uwagę znane eksploity skompresowanych, zaszyfrowanych komunikatów.

W wielu środowiskach, zwłaszcza gdy jako warstwa transportowa używany jest TLS, klucze komunikatów nie wymagają szyfrowania, gdyż zazwyczaj nie zawierają poufnych danych, takich jakie mogą zawierać ładunki komunikatów. Jednak w niektórych przypadkach klucze w postaci zwykłego tekstu mogą nie spełniać wymogów regulacyjnych. Ponieważ klucze komunikatów są używane do partycjonowania i kompaktowania, transformacja kluczy musi zachować wymaganą równowagę skrótu, aby zapewnić, że klucz zachowa tę samą wartość skrótu, nawet jeśli zmienią się parametry szyfrowania. Jednym z podejść byłoby przechowywanie bezpiecznego skrótu oryginalnego klucza jako klucza komunikatu, a zaszyfrowanego klucza komunikatu w ładunku komunikatu lub w nagłówku. Ponieważ Kafka serializuje klucze i wartości komunikatów niezależnie, do przeprowadzenia tej transformacji można użyć przechwytywacza produkcyjnego.

Autoryzacja

Autoryzacja to proces, który określa, jakie operacje można wykonywać na poszczególnych zasobach. Brokery Kafki zarządzają kontrolą dostępu za pomocą konfigurowalnego autoryzatora. Jak pisaliśmy wcześniej, za każdym razem, gdy klient ustanawia połączenie z brokerem, broker uwierzytelnia klienta i przypisuje do tego połączenia instancję `KafkaPrincipal`, która reprezentuje tożsamość danego klienta. Kiedy przetwarzane jest żądanie, broker sprawdza, czy pryncypał powiązany z połączeniem jest upoważniony do wykonywania tego żądania. Jeśli producent Alicji próbuje np. zapisać nowy rekord zamówienia klienta w temacie `customerOrders`, broker sprawdzi, czy `User:Alicja` ma autoryzację do zapisywania w tym temacie.

Kafka ma wbudowany autoryzator `AcLAuthorizer`, który można włączyć, konfigurując nazwę klasy autoryzatora w następujący sposób:

```
authorizer.class.name=kafka.security.authorizer.AcLAuthorizer
```



SimpleAcLAuthorizer

`AcLAuthorizer` został wprowadzony na platformie Apache Kafka 2.3. Starsze wersje od 0.9.0.0 miały wbudowany autoryzator `kafka.security.auth.SimpleAcLAuthorizer`, który został zarzucony, ale nadal jest obsługiwany.

AcLAuthorizer

`AcLAuthorizer` obsługuje precyzyjną kontrolę dostępu do zasobów Kafki za pomocą list kontroli dostępu (ang. *Access Control List* — ACL). Listy ACL są przechowywane w ZooKeeperze i buforowane w pamięci przez każdy broker, aby umożliwić bardzo wydajne wyszukiwanie dla autoryzacji żądań. Są ładowane do pamięci podręcznej podczas uruchamiania brokera, a pamięć podręczna jest aktualizowana za pomocą powiadomień opartych na obserwatorze ZooKeepera. Każde żądanie Kafki jest autoryzowane przez sprawdzenie, czy instancja `KafkaPrincipal` powiązana z danym połączeniem ma uprawnienia do wykonania żądanej operacji na żądanych zasobach.

Każde wiązanie ACL składa się z kilku elementów. Oto one:

- typ zasobu — `Cluster|Topic|Group|TransactionalId|DelegationToken`;
- typ wzorca — `Literal|Prefixed`;
- nazwa zasobu — nazwa zasobu, prefiks lub symbol wieloznaczny (*);
- operacja — `Describe|Create|Delete|Alter|Read|Write|DescribeConfigs|AlterConfigs`;
- typ uprawnień — `Allow|Deny`; `Deny` ma wyższy priorytet;
- pryncypał — pryncypał Kafki w postaci `<typ_pryncypała>:<nazwa_pryncypała>`, np. `User:Bob` lub `Group:Sales`; listy ACL mogą wykorzystywać pryncypała `User:*` w celu przyznania dostępu wszystkim użytkownikom;
- host — źródłowy adres IP połączenia klienta lub symbol wieloznaczny (*), jeśli autoryzowane są wszystkie hosty.

Lista ACL może określać np. następujące uprawnienia:

```
User:Alicja has Allow permission for Write to Prefixed Topic:customer from 192.168.0.1
```

`AcLAuthorizer` autoryzuje działanie, jeżeli nie zostało ono dopasowane do żadnej listy `Deny ACL`, a zostało dopasowane do co najmniej jednej listy `Allow ACL`. Uprawnienia `Describe` są przyznawane w sposób dorozumiany, jeśli przyznane zostały uprawnienia `Read`, `Write`, `Alter` lub `Delete`. Uprawnienia `DescribeConfigs` są przyznawane w sposób dorozumiany, jeśli przyznane zostały uprawnienia `AlterConfigs`.

Aby autoryzować żądania kontrolera i żądania pobrań repliki, brokery muszą mieć przyznane uprawnienia `Cluster:ClusterAction`. Producenty wymagają uprawnień `Topic:Write`, jeśli mają produkować do określonego tematu. Dla idempotentnego produkowania bez transakcji producenci muszą



Listy ACL z symbolem wieloznacznym

Listy ACL o typie wzorca `Literal` i nazwie zasobu `*` są używane jako wieloznaczne ACL, które pasują do wszystkich nazw zasobów danego typu zasobu.

otrzywać również uprawnienia `Cluster:IdempotentWrite`. W celu zatwierdzania przesunięć producenty transakcyjne wymagają dostępu `TransactionalId:Write` do ID transakcji i `Group:Read` dla grup konsumentów. Aby konsumować z tematu, konsumenci wymagają uprawnień `Topic:Read`, a także `Group:Read` dla grupy konsumentów, jeżeli stosowane jest zarządzanie grupami lub przesunięciami. Operacje administracyjne wymagają odpowiednio dostępu `Create`, `Delete`, `Describe`, `Alter`, `DescribeConfigs` lub `AlterConfigs`. W tabeli 11.1 przedstawiliśmy żądania Kafki, do których stosuje się poszczególne uprawnienia ACL.

Tabela 11.1 Dostęp przyznawany dla poszczególnych list ACL Kafki

ACL	Żądanie Kafki	Uwagi
<code>Cluster:ClusterAction</code>	Żądania między brokerami, w tym żądania kontrolera i żądania pobrań obserwatora dla replikacji	Powinno być przyznawane tylko brokerom
<code>Cluster:Create</code>	<code>CreateTopics</code> i automatyczne tworzenie tematów	Aby tworzyć określone tematy, należy użyć <code>Topic:Create</code> dla precyzyjnej kontroli dostępu
<code>Cluster:Alter</code>	<code>CreateAcls</code> , <code>DeleteAcls</code> , <code>AlterReplicaLogDirs</code> , <code>ElectReplicaLeader</code> , <code>AlterPartitionReassignments</code>	
<code>Cluster:AlterConfigs</code>	<code>AlterConfigs</code> i <code>IncrementalAlterConfigs</code> dla brokera i rejestratora brokera, <code>AlterClientQuotas</code>	
<code>Cluster:Describe</code>	<code>DescribeAcls</code> , <code>DescribeLogDirs</code> , <code>ListGroups</code> , <code>ListPartitionReassignments</code> , generowanie opisu autoryzowanych operacji dla klastra w żądaniu metadanych	Aby uzyskać precyzyjną kontrolę dostępu dla <code>ListGroups</code> , należy użyć <code>Group:Describe</code>
<code>Cluster:DescribeConfigs</code>	<code>DescribeConfigs</code> dla brokera i rejestratora brokera, <code>DescribeClientQuotas</code>	
<code>Cluster:IdempotentWrite</code>	<code>Idempotent</code> żądania <code>InitProducerId</code> i <code>Produce</code>	Wymagane tylko dla idempotentnych producentów nietransakcyjnych
<code>Topic:Create</code>	<code>CreateTopics</code> i automatyczne tworzenie tematów	
<code>Topic>Delete</code>	<code>DeleteTopics</code> , <code>DeleteRecords</code>	
<code>Topic:Alter</code>	<code>CreatePartitions</code>	
<code>Topic:AlterConfigs</code>	<code>AlterConfigs</code> i <code>IncrementalAlterConfigs</code> dla tematów	
<code>Topic:Describe</code>	Metadane żądania dla tematu, <code>OffsetForLeaderEpoch</code> , <code>ListOffset</code> , <code>OffsetFetch</code>	

Tabela 11.1 Dostęp przyznawany dla poszczególnych list ACL Kafki — ciąg dalszy

ACL	Żądanie Kafki	Uwagi
Topic:DescribeConfigs	DescribeConfigs dla tematów i dla zwracania konfiguracji w odpowiedzi CreateTopics	
Topic:Read	Consumer Fetch, OffsetCommit, TxnOffsetCommit, OffsetDelete	Powinno być przyznawane tylko konsumentom
Topic:Write	Produce, AddPartitionToTxn	Powinno być przyznawane tylko producentom
Group:Read	JoinGroup, SyncGroup, LeaveGroup, Heartbeat, OffsetCommit, AddOffsetsToTxn, TxnOffsetCommit	Wymagane dla konsumentów korzystających z zarządzania grupami konsumentów lub opartego na Kafce zarządzania przesunięciami. Wymagane również dla producentów transakcyjnych do zatwierdzania przesunięć w ramach transakcji
Group:Describe	FindCoordinator, DescribeGroup, ListGroups, OffsetFetch	
Group:Delete	DeleteGroups, OffsetDelete	
TransactionalId:Write	Produce i InitProducerId z transakcjami, AddPartitionToTxn, AddOffsetsToTxn, TxnOffsetCommit, EndTxn	Wymagane dla producentów transakcyjnych
TransactionalId:Describe	FindCoordinator dla koordynatora transakcji	
DelegationToken:Describe	DescribeTokens	

Kafka zapewnia narzędzie do zarządzania listami ACL za pomocą autoryzatora skonfigurowanego w brokerach. Listy ACL można tworzyć także bezpośrednio w ZooKeeperze. Poniższe polecenia są przydatne do tworzenia list ACL brokerów przed ich uruchomieniem:

```
$ bin/kafka-acls.sh --add --cluster --operation ClusterAction
--authorizer-properties zookeeper.connect=localhost:2181 ❶
--allow-principal User:kafka
$ bin/kafka-acls.sh --bootstrap-server localhost:9092
--command-config admin.props --add --topic customerOrders ❷
--producer --allow-principal User:Alicja
$ bin/kafka-acls.sh --bootstrap-server localhost:9092
--command-config admin.props --add --resource-pattern-type PREFIXED ❸
--topic customer --operation Read --allow-principal User:Bob
```

- ❶ Listy ACL dla użytkownika brokera są tworzone bezpośrednio w ZooKeeperze.
- ❷ Domyślnie polecenia list ACL przyznają uprawnienia literalnych ACL. User:Alicja przyznaje uprawnienia do zapisywania w temacie customerOrders.
- ❸ ACL z przedrostkiem pozwala Bobowi na odczytywanie ze wszystkich tematów zaczynających się od customer.

Ac1Authorizer ma dwie opcje konfiguracyjne do przyznawania szerokiego dostępu do zasobów lub pryncypałów, aby uprościć zarządzanie listami ACL, szczególnie podczas dodawania autoryzacji do istniejących klastrów po raz pierwszy:

```
super.users=User:Karolina;User:Admin  
allow.everyone.if.no.ac1.found=true
```

Użytkownicy o statusie super uzyskują dostęp do wszystkich operacji na wszystkich zasobach bez żadnych ograniczeń i nie można im odmówić dostępu za pomocą list Deny ACL. Jeśli naruszone zostanie bezpieczeństwo poświadczeń Karoliny, będzie ona musiała zostać usunięta z super.users, a w celu zastosowania zmian wymagane będzie zrestartowanie brokerów. Użytkownikom w systemach produkcyjnych bezpieczniej jest przyznawać konkretny dostęp za pomocą ACL, aby można było łatwo cofnąć uprawnienia, jeśli będzie to wymagane.



Separator superużytkowników

Inaczej niż w pozostałych konfiguracjach list w Kafce, które są rozdzielane przecinkami, użytkownicy super.users są rozdzielani średnikami, ponieważ pryncypały użytkowników, takie jak wyróżnione nazwy z certyfikatów SSL, często zawierają przecinki.

Jeśli włączona jest opcja allow.everyone.if.no.ac1.found, wszyscy użytkownicy mają dostęp do zasobów bez żadnych list ACL. Opcja ta może być przydatna przy włączaniu autoryzacji w klastrze po raz pierwszy lub podczas rozwoju oprogramowania, ale nie nadaje się do użytku w środowisku produkcyjnym, ponieważ dostęp może zostać przyznany w sposób niezamierzony do nowych zasobów. Dostęp może również zostać nieoczekiwanie usunięty, gdy zostaną dodane listy ACL dla pasującego prefiksu lub symbolu wieloznacznego, jeżeli warunek no.ac1.found nie będzie miał już zastosowania.

Dostosowywanie autoryzacji

W Kafce autoryzację można dostosowywać w celu implementowania dodatkowych ograniczeń lub dodawania nowych rodzajów kontroli dostępu, takich jak kontrola dostępu oparta na rolach.

Przedstawiony poniżej niestandardowy autoryzator ogranicza użycie niektórych żądań tylko do wewnętrznego nasłuchiwaacza. Dla uproszczenia żądania i nazwa nasłuchiwaacza zostały tu zakodowane na sztywno, ale w celu większej elastyczności można je skonfigurować przy użyciu niestandardowych właściwości autoryzatora:

```
public class CustomAuthorizer extends Ac1Authorizer {  
    private static final Set<Short> internalOps =  
        Utils.mkSet(CREATE_ACLS.id, DELETE_ACLS.id);  
    private static final String internalListener = "INTERNAL";  
  
    @Override  
    public List<AuthorizationResult> authorize(  
        AuthorizableRequestContext context, List<Action> actions) {  
        if (!context.listenerName().equals(internalListener) && ❶  
            internalOps.contains((short) context.requestType()))  
            return Collections.nCopies(actions.size(), DENIED);  
        else
```

```

        return super.authorize(context, actions); ❷
    }
}

```

- ❶ Autoryzatory otrzymują kontekst żądania z metadanymi, które zawierają nazwy nasłuchiwa-
czy, protokół bezpieczeństwa, typy żądań itp., co umożliwia niestandardowym autoryzatorom
dodawanie lub usuwanie ograniczeń na podstawie tego kontekstu.
- ❷ Przy użyciu publicznego API wykorzystujemy ponownie funkcjonalność z wbudowanego auto-
ryzatora Kafki.

Autoryzator Kafki można także zintegrować z zewnętrznymi systemami w celu zapewnienia obsługi kontroli dostępu opartej na grupach lub opartej na rolach. Do tworzenia list ACL dla pryncypałów grup lub ról można użyć różnych typów pryncypałów. Jak pokazaliśmy w poniższym listingu, można np. użyć ról i grup z serwera LDAP w celu okresowego uzupełniania obiektów `groups` i `roles` z klasy języka Scala, by obsługiwać listy Allow ACL na różnych poziomach:

```

class RbacAuthorizer extends AclAuthorizer {

    @volatile private var groups = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
        .withDefaultValue(Set.empty) ❶
    @volatile private var roles = Map.empty[KafkaPrincipal, Set[KafkaPrincipal]]
        .withDefaultValue(Set.empty) ❷

    override def authorize(context: AuthorizableRequestContext,
        actions: util.List[Action]): util.List[AuthorizationResult] = {
        val principals = groups(context.principal) + context.principal
        val allPrincipals = principals.flatMap(roles) ++ principals ❸
        val contexts = allPrincipals.map(authorizeContext(context, _))
        actions.asScala.map { action =>
            val authorized = contexts.exists(
                super.authorize(_, List(action).asJava).get(0) == ALLOWED
            )
            if (authorized) ALLOWED else DENIED ❹
        }.asJava
    }

    private def authorizeContext(context: AuthorizableRequestContext,
        contextPrincipal: KafkaPrincipal): AuthorizableRequestContext = {
        new AuthorizableRequestContext { ❺
            override def principal() = contextPrincipal
            override def clientId() = context.clientId
            override def requestType() = context.requestType
            override def requestVersion() = context.requestVersion
            override def correlationId() = context.correlationId
            override def securityProtocol() = context.securityProtocol
            override def listenerName() = context.listenerName
            override def clientAddress() = context.clientAddress
        }
    }
}

```

- ❶ Grupy, do których należy każdy użytkownik, uzupełniane z zewnętrznego źródła, takiego jak LDAP.
- ❷ Rola powiązane z każdym użytkownikiem, uzupełniane z zewnętrznego źródła, takiego jak LDAP.

- ❸ Wykonujemy autoryzację dla użytkownika, a także dla wszystkich grup i ról użytkownika.
- ❹ Jeśli któryś z kontekstów jest autoryzowany, zwracamy ALLOWED. Zwróć uwagę, że ten przykładowy kod nie obsługuje list Deny ACL dla grup lub ról.
- ❺ Dla każdego pryncypała tworzymy kontekst autoryzacji z tymi samymi metadanymi, jakie ma kontekst oryginalny.

Listy ACL można przypisać do grupy Sales lub roli Operator za pomocą standardowego narzędzia ACL Kafki:

```
$ bin/kafka-acls.sh --bootstrap-server localhost:9092
--command-config admin.props --add --topic customer --producer
--resource-pattern-type PREFIXED --allow-principal Group:Sales ❶
$ bin/kafka-acls.sh --bootstrap-server localhost:9092
--command-config admin.props --add --cluster --operation Alter
--allow-principal=Role:Operator ❷
```

- ❶ Aby utworzyć listę ACL, która ma zastosowanie do użytkowników należących do grupy Sales, używamy pryncypała Group:Sales z niestandardowym typem pryncypała Group.
- ❷ Aby utworzyć listę ACL, która ma zastosowanie do użytkowników z rolą Operator, używamy pryncypała Role:Operator z niestandardowym typem pryncypała Role.

Kwestie bezpieczeństwa

Ponieważ AclAuthorizer przechowuje listy ACL w ZooKeeperze, dostęp do ZooKeepera powinien zostać ograniczony. We wdrożeniach bez zabezpieczonego ZooKeepera można implementować niestandardowe autoryzatory, by przechowywać listy ACL w bezpiecznej zewnętrznej bazie danych.

W dużych organizacjach z dużą liczbą użytkowników zarządzanie listami ACL dla poszczególnych zasobów może stać się bardzo uciążliwe. Zarezerwowanie różnych prefiksów zasobów dla różnych działów umożliwia stosowanie list ACL z prefiksami, które minimalizują liczbę wymaganych list ACL. Żeby jeszcze bardziej uprościć kontrolę dostępu w dużych wdrożeniach, można to połączyć z listami ACL opartymi na grupach lub rolach, jak pokazaliśmy w poprzednim przykładzie.

Ograniczenie dostępu użytkowników przy użyciu zasady najmniejszych uprawnień może zmniejszyć negatywne skutki naruszenia bezpieczeństwa użytkownika. Oznacza to przyznanie dostępu tylko do zasobów niezbędnych każdemu pryncypałowi użytkownika do wykonywania żądanych operacji i usuwania list ACL, gdy przestają być potrzebne. Listy ACL powinny być usuwane natychmiast, gdy określony pryncypał użytkownika przestanie być używany, np. jeśli pracownik opuści organizację. Aby uniknąć zakłóceń podczas opuszczania organizacji przez pracowników, długo działające aplikacje można konfigurować za pomocą poświadczeń usługi, a nie poświadczeń powiązanych z konkretnym użytkownikiem. Ponieważ długotrwałe połączenia z pryncypałem użytkownika mogą kontynuować przetwarzanie żądań nawet po usunięciu danego użytkownika z systemu, można zastosować listy Deny ACL, by mieć pewność, że ten pryncypał nie uzyska niezamierzonego dostępu za pośrednictwem list ACL z symbolami wieloznacznymi. Jeśli to możliwe, należy unikać ponownego wykorzystywania pryncypałów, żeby zapobiec przyznawaniu dostępu połączeniom, które korzystają ze starszej wersji pryncypała.

Audyt

Brokery Kafki mogą być skonfigurowane do generowania kompleksowych dzienników **log4j** do celów audytu i debugowania. Poziom rejestrowania, a także dodatki używane do rejestrowania i ich opcje konfiguracyjne można określić w pliku *log4j.properties*. Instancje rejestratora `kafka.authorizer.logger` wykorzystywanego do rejestrowania autoryzacji i rejestratora `kafka.request.logger` wykorzystywanego do rejestrowania żądań można konfigurować niezależnie, aby dostosować poziom dziennika i retencję dla rejestrowania audytowego. Do analizy i wizualizacji tych dzienników systemy produkcyjne mogą stosować frameworki takie jak Elastic Stack.

Dla każdej próby wykonania operacji, której odmówiono dostępu, autoryzatory generują wpisy dziennika na poziomie INFO, a dla każdej operacji, której przyznano dostęp — wpisy dziennika na poziomie DEBUG. Oto przykład:

```
DEBUG Principal = User:Alicja is Allowed Operation = Write from host = 127.0.0.1 on
resource = Topic:LITERAL:customerOrders for request = Produce with resourceRefCount = 1
(kafka.authorizer.logger)
INFO Principal = User:Mallory is Denied Operation = Describe from host = 10.0.0.13 on
resource = Topic:LITERAL:customerOrders for request = Metadata with resourceRefCount = 1
(kafka.authorizer.logger)
```

Rejestrowanie żądań generowanych na poziomie DEBUG zawiera również szczegóły dotyczące pryncypała użytkownika i hosta klienta. Pełne informacje na temat żądań są uwzględniane, jeśli rejestrator żądań jest skonfigurowany do rejestrowania na poziomie TRACE. Oto przykład:

```
DEBUG Completed request:RequestHeader(apiKey=PRODUCE, apiVersion=8, clientId=producer-1,
correlationId=6) --{acks=-1,timeout=30000,partitionSizes=[customerOrders-
0:15514]},response:{responses=[{topic=customerOrders,partition_responses=[{partition=0,err
or_code=0,base_offset=13,log_append_time=-
1,log_start_offset=0,record_errors=[],error_message=null}]}],throttle_time_ms=0} from
connection 127.0.0.1:9094-127.0.0.1:61040-
0;totalTime:2.42,requestQueueTime:0.112,localTime:2.15,remoteTime:0.0,throttleTime:0,resp
onseQueueTime:0.04,sendTime:0.118,securityProtocol:SASL_SSL,principal=User:Alicja,listener:
SASL_SSL,clientInformation:ClientInformation(softwareName=apache-kafka-
java,softwareVersion=2.7.0-SNAPSHOT) (kafka.request.logger)
```

Dzienniki autoryzatora i żądań można analizować w celu wykrywania podejrzanych działań. Wskaźniki śledzące błędy uwierzytelniania oraz dzienniki błędów autoryzacji mogą być niezwykle przydatne podczas przeprowadzania audytu i dostarczać cennych informacji w przypadku ataku lub nieautoryzowanego dostępu. W celu zapewnienia kompleksowego audytu i śledzenia komunikatów metadane inspekcji mogą być załączane w nagłówkach komunikatów podczas ich tworzenia. Aby chronić integralność tych metadanych, można zastosować kompleksowe szyfrowanie.

Zabezpieczanie ZooKeepera

ZooKeeper przechowuje metadane Kafki, które mają kluczowe znaczenie dla utrzymywania dostępności jej klastrów, dlatego oprócz zabezpieczenia Kafki ważne jest zabezpieczenie ZooKeepera. ZooKeeper obsługuje SASL/GSSAPI dla uwierzytelniania przy użyciu Kerberos i SASL/DIGEST-MD5 dla uwierzytelniania za pomocą nazwy użytkownika i hasła. W wersji 3.5.0 w ZooKeeperze dodano także obsługę TLS, co umożliwiło wzajemne uwierzytelnianie oraz szyfrowanie przesyłanych

danych. Zwróć uwagę, że SASL/DIGEST-MD5 należy stosować tylko z szyfrowaniem TLS i nie nadaje się ono do użytku w środowisku produkcyjnym ze względu na znane luki w zabezpieczeniach.

SASL

Konfigurację SASL dla ZooKeepera zapewnia właściwość systemowa Javy `java.security.auth.login.config`. Należy w niej wskazać plik konfiguracyjny JAAS, który zawiera sekcję logowania z odpowiednim modulem logowania i jego opcjami dla serwera ZooKeeper. Aby klienci ZooKeepera mogli komunikować się z serwerami ZooKeeper z włączoną obsługą SASL, brokery Kafki muszą być skonfigurowane z sekcją logowania po stronie klienta. Pokazana poniżej sekcja `Server` zapewnia konfigurację JAAS dla serwera ZooKeeper, aby umożliwić uwierzytelnianie z wykorzystaniem Kerberosa:

```
Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true storeKey=true
    keyTab="/ścieżka/do/zk.keytab"
    principal="zookeeper/zk1.example.com@EXAMPLE.COM";
};
```

Aby włączyć uwierzytelnianie SASL na serwerach ZooKeeper, należy skonfigurować dostawców uwierzytelniania w pliku konfiguracyjnym ZooKeepera:

```
authProvider.sasl=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
kerberos.removeHostFromPrincipal=true
kerberos.removeRealmFromPrincipal=true
```



Principał brokera

Domyślnie ZooKeeper jako tożsamości klienta używa pełnego pryncypała Kerberosa, np. `kafka/broker1.example.com@EXAMPLE.COM`. Gdy dla autoryzacji ZooKeepera włączone są listy ACL, serwery ZooKeeper powinny być skonfigurowane z opcjami `kerberos.removeHostFromPrincipal=true` i `kerberos.removeRealmFromPrincipal=true`, aby zapewnić, że wszystkie brokery będą miały tego samego pryncypała.

Za pomocą pliku konfiguracyjnego JAAS, który zapewnia poświadczenia klienta dla brokera, należy skonfigurować brokery Kafki do uwierzytelniania w ZooKeeperze przy użyciu SASL:

```
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true storeKey=true
    keyTab="/ścieżka/do/broker1.keytab"
    principal="kafka/broker1.example.com@EXAMPLE.COM";
};
```

SSL

SSL może być włączone w dowolnym punkcie końcowym ZooKeepera, również w tym z uwierzytelnianiem SASL. Podobnie jak w Kafce, SSL może być skonfigurowane tak, aby umożliwiało uwierzytelnianie klientów. Jednak inaczej niż w Kafce, połączenia z włączonym uwierzytelnianiem klienta SASL i SSL są uwierzytelniane za pomocą obu protokołów jednocześnie i przypisywanych jest do nich wielu pryncypałów. Autoryzator ZooKeepera przyznaje dostęp do zasobu, jeśli uprawnienia ma którykolwiek z pryncypałów powiązanych z połączeniem.

Aby skonfigurować SSL na serwerze ZooKeeper, należy skonfigurować magazyn kluczy z nazwą hosta serwera lub z hostem z symbolem wieloznacznym. Jeżeli włączone jest uwierzytelnianie klienta, wymagany jest również magazyn zaufania do walidacji certyfikatów klienta:

```
secureClientPort=2181
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider
ssl.keyStore.location=/ścieżka/do/zk.ks.p12
ssl.keyStore.password=hasło_do_zk-ks
ssl.keyStore.type=PKCS12
ssl.trustStore.location=/ścieżka/do/zk.ts.p12
ssl.trustStore.password=hasło_do_zk-ts
ssl.trustStore.type=PKCS12
```

Aby skonfigurować SSL dla połączeń Kafki z ZooKeeperem, brokery powinny być skonfigurowane z magazynem zaufania do walidacji certyfikatów ZooKeepera. Jeśli włączone jest uwierzytelnianie klienta, wymagany jest również magazyn kluczy:

```
zookeeper.ssl.client.enable=true
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
zookeeper.ssl.keystore.location=/ścieżka/do/zkclient.ks.p12
zookeeper.ssl.keystore.password=hasło_do_zkclient-ts
zookeeper.ssl.keystore.type=PKCS12
zookeeper.ssl.truststore.location=/ścieżka/do/zkclient.ts.p12
zookeeper.ssl.truststore.password=hasło_do_zkclient-ts
zookeeper.ssl.truststore.type=PKCS12
```

Autoryzacja

Dla węzłów ZooKeepera autoryzację można włączyć przez ustawienie list ACL dla danej ścieżki. Gdy brokery są skonfigurowane z opcją `zookeeper.set.acl=true`, broker ustawia listy ACL dla węzłów ZooKeepera podczas tworzenia węzła. Domyślnie węzły metadanych mogą odczytywać wszyscy, ale modyfikować mogą je tylko brokery. Jeśli zajdzie potrzeba, można dodać dodatkowe listy ACL dla wewnętrznych użytkowników administracyjnych, którzy mogą mieć potrzebę aktualizowania metadanych bezpośrednio w ZooKeeperze. Ścieżki z poufnymi danymi, np. węzły zawierające poświadczenia SCRAM, domyślnie nie mogą być odczytywane przez wszystkich.

Zabezpieczanie platformy

W poprzednich podrozdziałach omówiliśmy opcje zamykania dostępu do Kafki i ZooKeepera w celu zabezpieczenia wdrożeń Kafki. Projekt zabezpieczeń dla systemu produkcyjnego powinien wykorzystywać model zagrożeń, który uwzględnia zagrożenia bezpieczeństwa nie tylko dla poszczególnych komponentów, ale także dla systemu jako całości. Modele zagrożeń budują abstrakcję systemu i identyfikują potencjalne zagrożenia i związane z nimi ryzyko. Po dokonaniu oceny, udokumentowaniu i ustaleniu priorytetów zagrożeń na podstawie ryzyka dla każdego potencjalnego zagrożenia należy wdrożyć strategię ograniczania możliwości jego wystąpienia, aby zapewnić ochronę całego systemu. Oceniając potencjalne zagrożenia, trzeba uwzględnić zarówno zagrożenia zewnętrzne, jak i wewnętrzne. W przypadku systemów przechowujących dane osobowe (PII) lub inne poufne dane należy zaimplementować także dodatkowe środki w celu zapewnienia zgodności z regulacjami. Szczegółowe omówienie standardowych technik modelowania zagrożeń wykracza poza zakres tego rozdziału.

Oprócz ochrony danych w Kafce i metadanych w ZooKeeperze za pomocą bezpiecznego uwierzytelniania, autoryzacji i szyfrowania należy podjąć dodatkowe kroki w celu zapewnienia bezpieczeństwa platformy. Obrona może obejmować zapórę sieciową dla ochrony sieci i szyfrowanie dla ochrony fizycznej pamięci masowej. Magazyny kluczy, magazyny zaufania i pliki *keytab* Kerberos, które zawierają poświadczenia wykorzystywane do uwierzytelniania, muszą być chronione przy użyciu uprawnień systemu plików. Dostęp do plików konfiguracyjnych zawierających informacje o kluczowym znaczeniu dla bezpieczeństwa, takie jak poświadczenia, musi być ograniczony. Ponieważ hasła przechowywane w postaci zwykłego tekstu w plikach konfiguracyjnych pozostają niezabezpieczone, nawet jeśli dostęp jest ograniczony, Kafka obsługuje zewnętrzne przechowywanie haseł w bezpiecznym magazynie.

Zabezpieczanie hasłem

Aby pobierać hasła z bezpiecznego zewnętrznego magazynu haseł, dla brokerów i klientów Kafki można zdefiniować niestandardowych dostawców konfiguracji. Hasła mogą być również przechowywane w postaci zaszyfrowanej w plikach konfiguracyjnych z niestandardowymi dostawcami konfiguracji wykonującymi odszyfrowywanie.

Przedstawiony poniżej niestandardowy dostawca konfiguracji używa narzędzia *gpg* do odszyfrowywania właściwości brokera lub klienta przechowywanych w pliku:

```
public class GpgProvider implements ConfigProvider {

    @Override
    public void configure(Map<String, ?> configs) {}

    @Override
    public ConfigData get(String path) {
        try {
            String passphrase = System.getenv("PASSPHRASE"); ❶
            String data = Shell.execCommand(                  ❷
                "gpg", "--decrypt", "--passphrase", passphrase, path);
            Properties props = new Properties();
            props.load(new StringReader(data));                ❸
            Map<String, String> map = new HashMap<>();
            for (String name : props.stringPropertyNames())
                map.put(name, props.getProperty(name));
            return new ConfigData(map);
        } catch (IOException e) {
            throw new RuntimeException(e);                        ❹
        }
    }

    @Override
    public ConfigData get(String path, Set<String> keys) { ❺
        ConfigData configData = get(path);
        Map<String, String> data = configData.data().entrySet()
            .stream().filter(e -> keys.contains(e.getKey()))
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
        return new ConfigData(data, configData.ttl());
    }

    @Override
    public void close() {}
}
```

- 1 Tekst szyfrujący do dekodowania haseł przekazujemy procesowi w zmiennej środowiskowej `PASSPHRASE`.
- 2 Odszyfrowujemy konfiguracje za pomocą `gpg`. Wartość zwracana zawiera pełny zestaw odszyfrowanych konfiguracji.
- 3 Konfiguracje parsujemy w obiekcie `data` jako właściwości Javy.
- 4 W przypadku wystąpienia błędu stosujemy zasadę *fail fast* i rzucamy wyjątek `RuntimeException`.
- 5 Kod wywołujący może zażądać podzbioru kluczy ze ścieżki; tutaj pobieramy wszystkie wartości i zwracamy żądany podzbiór.

Jak zapewne pamiętasz, podczas omawiania SASL/PLAIN użyliśmy standardowych klas konfiguracyjnych Kafki, aby załadować poświadczenia z pliku zewnętrznego. Możemy teraz zaszyfrować ten plik za pomocą `gpg`:

```
gpg --symmetric --output credentials.props.gpg  
--passphrase "$PASSPHRASE" credentials.props
```

Teraz do oryginalnego pliku właściwości dodajemy pośrednie konfiguracje i opcje dostawcy konfiguracji, aby klienci Kafki łączyli swoje poświadczenia z zaszyfrowanego pliku:

```
username=${gpg:/ścieżka/do/credentials.props.gpg:username}  
password=${gpg:/ścieżka/do/credentials.props.gpg:password}  
config.providers=gpg  
config.providers.gpg.class=com.example.GpgProvider
```

Poufne opcje konfiguracyjne brokera można również przechowywać w postaci zaszyfrowanej w ZooKeeperze przy użyciu narzędzia `configs` Kafki bez konieczności stosowania niestandardowych dostawców. Przed uruchomieniem brokerów można wykonać poniższe polecenie, by zaszyfrowane hasła magazynu kluczy SSL dla brokerów zapisywać w ZooKeeperze. Aby odszyfrować wartość, sekret koda haseł musi być skonfigurowany w pliku konfiguracyjnym każdego brokera:

```
$ bin/kafka-configs.sh --zookeeper localhost:2181 --alter  
--entity-type brokers --entity-name 0 --add-config  
'listener.name.external.ssl.keystore.password=server-ks-  
password,password.encoder.secret=encoder-secret'
```

Podsumowanie

W ciągu ostatniej dekady wzrosła częstotliwość i skala naruszeń bezpieczeństwa danych, ponieważ cyberataki stają się coraz bardziej wyrafinowane. Oprócz generowania znacznych kosztów związanych z izolowaniem i usuwaniem naruszeń oraz kosztów przestoju do czasu zastosowania środków bezpieczeństwa, złamanie zabezpieczeń danych może również skutkować sankcjami prawnymi i długotrwałym spadkiem reputacji marki. W tym rozdziale zbadaliśmy wiele opcji, które pozwalają zagwarantować poufność, integralność i dostępność danych przechowywanych w Kafce.

Przeanalizowaliśmy opcje dostępne dla różnych aspektów bezpieczeństwa w całym przykładowym przepływie danych przedstawionym na początku tego rozdziału:

Autentyczność klienta

Kiedy klient Alicji nawiązuje połączenie z brokerem Kafki, nasłuchiwaniec wykorzystujący SASL lub SSL z uwierzytelnianiem klienta może sprawdzić, czy połączenie naprawdę pochodzi od Alicji, a nie od oszusta. Aby ograniczyć negatywne skutki w przypadku naruszenia zabezpieczeń użytkownika, można skonfigurować ponowne uwierzytelnianie.

Autentyczność serwera

Za pomocą SSL z walidacją nazwy hosta lub mechanizmów SASL z wzajemnym uwierzytelnianiem, takich jak Kerberos lub SCRAM, klient Alicji może sprawdzić, czy połączył się z prawdziwym brokerem.

Ochrona danych

Użycie protokołu SSL do szyfrowania przesyłanych danych chroni dane przed podsłuchami. Szyfrowanie dysku lub woluminu chroni dane w spoczynku, nawet jeśli dysk zostanie skradziony. W przypadku poufnych danych szyfrowanie kompleksowe zapewnia precyzyjną kontrolę dostępu do danych i gwarantuje, że dostawcy usług w chmurze i administratorzy platform z fizycznym dostępem do sieci i dysków nie będą mogli uzyskać dostępu do danych.

Integralność danych

Do wykrywania fałszowania danych w niezabezpieczonej sieci można używać SSL. W celu weryfikacji integralności podczas korzystania z kompleksowego szyfrowania w komunikatach można umieszczać cyfrowe sygnatury.

Kontrola dostępu

Każda operacja wykonywana przez Alicję, Boba, a nawet brokery, jest autoryzowana za pomocą niestandardowego autoryzatora. Kafka ma wbudowany autoryzator, który umożliwia precyzyjną kontrolę dostępu za pośrednictwem list ACL.

Audyt

Dzienniki autoryzacji i dzienniki żądań mogą być używane do śledzenia operacji i prób wykonania operacji w celu przeprowadzania inspekcji i wykrywania anomalii.

Dostępność

Połączenie kwot i opcji konfiguracyjnych do zarządzania połączeniami może być wykorzystywane do ochrony brokerów przed atakami typu DoS. ZooKeeper można zabezpieczyć za pomocą SSL, SASL i list ACL, aby chronić metadane niezbędne do zapewnienia dostępności brokerów Kafki.

Przy szerokim wyborze opcji zabezpieczeń wybór odpowiednich opcji dla każdego przypadku użycia może być trudnym zadaniem. Omówiliśmy kwestie dotyczące bezpieczeństwa, aby rozważyć wszelkie mechanizmy bezpieczeństwa oraz środki kontroli i reguły, które można zastosować w celu ograniczenia potencjalnej powierzchni ataku. Dokonaliśmy również przeglądu dodatkowych środków wymaganych do zamknięcia dostępu do ZooKeepera i reszty platformy. Standardowe technologie bezpieczeństwa obsługiwane przez Kafkę i różne punkty rozbudowy do integracji z istniejącą w Twojej organizacji infrastrukturą bezpieczeństwa umożliwiają budowanie spójnych rozwiązań bezpieczeństwa w celu ochrony całej platformy.

Administrowanie Kafką

Zarządzanie klastrem Kafki wymaga dodatkowych narzędzi do wprowadzania zmian administracyjnych w tematach, konfiguracjach itd. Kafka oferuje kilka narzędzi interfejsu wiersza poleceń (ang. *Command-Line Interface* — CLI), które są przydatne przy wprowadzaniu zmian administracyjnych w klastrach. Narzędzia są zaimplementowane w klasach Javy i natywnie dostarczany jest zestaw skryptów do poprawnego wywoływania tych klas. Chociaż narzędzia te zapewniają podstawowe funkcje, możesz uznać, że brakuje im bardziej złożonych operacji lub są nieporęczne w użyciu na większą skalę. W tym rozdziale opisaliśmy tylko podstawowe narzędzia, które są dostępne w ramach projektu Apache Kafka dystrybuowanego na licencji *open source*. Więcej informacji na temat zaawansowanych narzędzi, które zostały opracowane przez społeczność poza głównym projektem, znajdziesz na stronie internetowej Apache Kafka (<https://kafka.apache.org>).



Autoryzacja operacji administratora

Chociaż Apache Kafka implementuje uwierzytelnianie i autoryzację w celu kontrolowania operacji w tematach, domyślne konfiguracje nie ograniczają korzystania z tych narzędzi. Oznacza to, że narzędzia CLI mogą być używane bez konieczności uwierzytelniania, co umożliwia wykonywanie operacji takich jak wprowadzanie zmian w tematach bez przeprowadzania kontroli bezpieczeństwa lub inspekcji. Aby zapobiec nieautoryzowanym zmianom, zawsze upewnij się, że we wdrożeniach dostęp do tych narzędzi jest ograniczony tylko do administratorów.

Operacje na tematach

Narzędzie `kafka-topics.sh` zapewnia łatwy dostęp do większości operacji na tematach. Umożliwia tworzenie, modyfikowanie i usuwanie tematów w klastrze oraz wyświetlanie dotyczących ich informacji. Chociaż za pomocą tego polecenia można wykonywać określone operacje konfigurowania tematów, jest to przestarzałe podejście i do wprowadzania zmian konfiguracyjnych zaleca się korzystanie z solidniejszego narzędzia `kafka-config.sh`. Aby użyć polecenia `kafka-topics.sh`, musisz za pomocą opcji `--bootstrap-server` podać łańcuch znaków dla połączenia z klastrem i port. W przedstawionych dalej przykładach łańcuch znaków dla połączenia z klastrem będziemy uruchamiać lokalnie na jednym z hostów z klastra Kafki i będziemy używać `localhost:9092`.

Wszystkie narzędzia omawiane w tym rozdziale znajdziesz w katalogu `/usr/local/kafka/bin/`. W przykładach uruchamiania poleceń przyjęliśmy założenie, że będziesz uruchamiał je z poziomu tego katalogu lub dodasz ścieżkę do zmiennej środowiskowej `$PATH`.



Sprawdź wersję

Poprawne działanie wielu narzędzi wiersza poleceń jest zależne od uruchamianej wersji Kafki. Dotyczy to również kilku poleceń, które zamiast łączenia się z samymi brokerami umożliwiają zapisywanie danych w ZooKeeperze. Dlatego powinieneś się upewnić, że używana wersja narzędzi jest zgodna z wersją brokerów w klastrze. Najbezpieczniejszym podejściem jest uruchamianie narzędzi na brokerach Kafki przy użyciu wdrożonej wersji.

Tworzenie nowego tematu

Do utworzenia w klastrze nowego tematu za pomocą polecenia `--create` wymaganych jest kilka argumentów. Argumenty te muszą być podawane podczas korzystania z tego polecenia, nawet jeśli niektóre z nich mogą mieć już skonfigurowane domyślne ustawienia na poziomie brokera. Możliwe jest stosowanie dodatkowych argumentów i nadpisań konfiguracji, a także opcji `--config`, ale omówimy je dalej w rozdziale. Oto trzy wymagane argumenty:

`--topic`

Nazwa tematu, który chcesz utworzyć.

`--replication-factor`

Liczba replik tematu, które mają być utrzymywane w klastrze.

`--partitions`

Liczba partycji, które mają być utworzone dla danego tematu.



Dobre praktyki nazywania tematów

Nazwy tematów mogą zawierać znaki alfanumeryczne, podkreślniki, myślniki i kropki. Nie zaleca się jednak używania kropek w nazwach tematów. Wewnętrzne wskaźniki Kafki konwertują kropki na podkreślniki (przy obliczaniu wskaźników przykładowy temat `temat.1` zostanie przekształcony w temat `temat_1`), co może powodować konflikty w nazwach tematów.

Kolejną rekomendacją jest unikanie rozpoczynania nazw tematów od podwójnego podkreślnika. Zgodnie z konwencją podwójnym podkreślnikiem zaczynają się nazwy wewnętrznych tematów operacji Kafki (takich jak temat `__consumer_offsets`, który śledzi zapisywanie przesunięć grupy konsumentów). Dlatego nie zaleca się stosowania tej konwencji nazewnictwa, aby uniknąć pomyłek.

Tworzenie nowego tematu jest proste. Wystarczy uruchomić polecenie `kafka-topics.sh` w następujący sposób:

```
# kafka-topics.sh --bootstrap-server <łańcuch_znaków_połączenia>:<port>
--create --topic <łańcuch_znaków> --replication-factor <liczba_całkowita>
--partitions <liczba_całkowita>
#
```

Wykonanie tego polecenia spowoduje, że klaster utworzy temat o określonej nazwie i liczbie partycji. Dla każdej partycji klaster utworzy wskazaną liczbę replik. Oznacza to, że jeśli klaster jest

skonfigurowany do przypisywania replik z uwzględnieniem szaf rackowych, repliki dla każdej partycji będą się znajdować w osobnych rackach. Jeżeli przypisywanie z uwzględnieniem szaf rackowych nie jest pożądane, podaj w wierszu poleceń argument `--disable-rack-aware`.

Utwórzmy np. temat o nazwie `my-topic` z 8 partycjami, które mają po 2 repliki:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-topic --
  replication-factor 2 --partitions 8
  Created topic "my-topic".
#
```



Poprawne używanie argumentów `if-exists` i `if-not-exists`

Wykorzystując narzędzie `kafka-topics.sh` w automatyzacji, możesz podczas tworzenia nowych tematów stosować argument `--if-not-exist`, który nie zwróci błędu, jeżeli temat już istnieje.

Chociaż dla polecenia `--alter` dostępny jest argument `--if-exists`, jego używanie nie jest zalecane. Gdy go zastosujesz, polecenie nie zwróci błędu, jeśli zmieniany temat nie istnieje. Może to maskować problemy dotyczące nieistniejących tematów, które powinny zostać utworzone.

Wyświetlanie listy wszystkich tematów w klastrze

Polecenie `--list` powoduje wyświetlenie wszystkich tematów w klastrze. Lista wypisuje po jednym temacie w linii bez określonej kolejności, co jest przydatne przy generowaniu pełnej listy tematów.

Oto przykład zastosowania opcji `--list` do wyświetlenia listy wszystkich tematów w klastrze:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --list
__consumer_offsets
my-topic
other-topic
```

Jak widać, wymieniony jest tutaj wewnętrzny temat `__consumer_offsets`. Uruchomienie polecenia z opcją `--exclude-internal` spowoduje usunięcie z listy wszystkich tematów, które zaczynają się od wspomnianego wcześniej podwójnego podkreślnika, co może być korzystne.

Generowanie opisu tematu

Możliwe jest również uzyskanie szczegółowych informacji o tematach w klastrze. Dane wyjściowe obejmują liczbę partycji, nadpisanie konfiguracji tematu oraz listę wszystkich partycji wraz z przypisanymi do nich replikami. Podając w poleceniu argument `--topic`, można to ograniczyć do pojedynczego tematu.

Tak wygląda wygenerowanie opisu naszego niedawno utworzonego w klastrze tematu `my-topic`:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 8      ReplicationFactor: 2      Configs:
segment.bytes=1073741824
    Topic: my-topic Partition: 0      Leader: 1      Replicas: 1,0      Isr: 1,0
    Topic: my-topic Partition: 1      Leader: 0      Replicas: 0,1      Isr: 0,1
```

Topic: my-topic	Partition: 2	Leader: 1	Replicas: 1,0	Isr: 1,0
Topic: my-topic	Partition: 3	Leader: 0	Replicas: 0,1	Isr: 0,1
Topic: my-topic	Partition: 4	Leader: 1	Replicas: 1,0	Isr: 1,0
Topic: my-topic	Partition: 5	Leader: 0	Replicas: 0,1	Isr: 0,1
Topic: my-topic	Partition: 6	Leader: 1	Replicas: 1,0	Isr: 1,0
Topic: my-topic	Partition: 7	Leader: 0	Replicas: 0,1	Isr: 0,1

#

Polecenie `--describe` ma również kilka przydatnych opcji filtrowania danych wyjściowych. Może to być pomocne przy diagnozowaniu problemów z klastrem. Dla tych poleceń zazwyczaj nie określamy argumentu `--topic`, ponieważ intencją jest znalezienie wszystkich tematów lub partycji w klastrze, które spełniają określone kryteria. Te opcje nie będą działać z poleceniem `list`. Oto kilka z nich:

`--topics-with-overrides`

Generuje opis tylko dla tych tematów, które mają konfiguracje inne niż wartości domyślne klastra.

`--exclude-internal`

To wspomniane wcześniej polecenie usuwa z listy wszystkie tematy, których nazwy zaczynają się od podwójnego podkreślnika.

Poniższe polecenia są używane do wyszukiwania tych partycji tematów, które mogą mieć problemy:

`--under-replicate-partitions`

Pokazuje wszystkie partycje, w których co najmniej jedna replika nie jest zsynchronizowana z liderem. Nie musi to wskazywać na jakiś poważny problem, ponieważ partycje z niewystarczającą liczbą replik (ang. *Under-Replicated Partition* — URP) mogą powstawać podczas konserwacji, wdrożeń i rebalansowania klastra, ale należy mieć tego świadomość.

`--at-min-isr-partitions`

Pokazuje wszystkie partycje, w których liczba replik włącznie z liderem jest równa minimalnej liczbie replik zsynchronizowanych (ang. *In-Sync Replica* — ISR). Tematy te są nadal dostępne dla klientów produkcyjnych lub konsumenckich, ale utracona została redundancja i istnieje ryzyko, że staną się niedostępne.

`--under-min-isr-partitions`

Pokazuje wszystkie partycje, w których liczba ISR jest mniejsza od minimum skonfigurowanego dla pomyślnego produkowania. W efekcie partycje te działają w trybie tylko do odczytu i nie można do nich produkować.

`--unavailable-partitions`

Pokazuje wszystkie partycje tematu, które nie mają lidera. Jest to poważna sytuacja i wskazuje, że partycja działa w trybie offline i jest niedostępna dla klientów produkcyjnych lub konsumenckich.

Oto przykład wyszukiwania tematów, które mają liczbę ISR na poziomie skonfigurowanego minimum. W tym przypadku temat jest skonfigurowany z liczbą ISR równą 1 i ma współczynnik replikacji (ang. *Replication Factor* — RF) równy 2. Host 0 jest online, a host 1 został wyłączony z powodu prac konserwacyjnych:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --describe --at-min-isr-partitions
Topic: my-topic Partition: 0 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 2 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 4 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 5 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 6 Leader: 0 Replicas: 0,1 Isr: 0
Topic: my-topic Partition: 7 Leader: 0 Replicas: 0,1 Isr: 0
#
```

Dodawanie partycji

Czasami konieczne jest zwiększenie liczby partycji dla określonego tematu. Tworzenie partycji umożliwia skalowanie i replikowanie tematów w klastrze. Najczęstszym powodem zwiększania liczby partycji jest poziome skalowanie tematu na większą liczbę brokerów przez zmniejszanie przepływności pojedynczej partycji. Liczbę partycji można również zwiększać, jeśli konsument musi uruchomić więcej swoich kopii w jednej grupie konsumentów, ponieważ partycja może być konsumowana tylko przez jednego członka grupy.

Poniżej przedstawiliśmy przykład zwiększenia liczby partycji dla tematu o nazwie `my-topic` do 16 za pomocą polecenia `--alter`, a następnie zweryfikowania, czy polecenie zadziałało:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --alter --topic my-topic --partitions 16

# kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-topic
Topic: my-topic PartitionCount: 16 ReplicationFactor: 2 Configs:
segment.bytes=1073741824
Topic: my-topic Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 2 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 4 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 5 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 6 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 7 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 8 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 9 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 10 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 11 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 12 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 13 Leader: 0 Replicas: 0,1 Isr: 0,1
Topic: my-topic Partition: 14 Leader: 1 Replicas: 1,0 Isr: 1,0
Topic: my-topic Partition: 15 Leader: 0 Replicas: 0,1 Isr: 0,1
#
```



Dostosowywanie tematów zawierających klucze

Z punktu widzenia konsumenta bardzo trudne może być dodawanie partycji w przypadku tematów, które są produkowane za pomocą komunikatów z kluczami. Jest to spowodowane tym, że mapowanie kluczy na partycje zmienia się po zmianie liczby partycji. Z tego powodu dla tematu, który będzie zawierał wiadomości z kluczami, zaleca się ustawianie liczby partycji tylko raz, gdy temat jest tworzony, i unikanie wprowadzania zmian w rozmiarze tematu.

Redukcja liczby partycji

Zmniejszenie liczby partycji dla tematu nie jest możliwe. Usunięcie partycji z tematu spowodowałoby również usunięcie części danych z tego tematu, co byłoby niespójne z punktu widzenia klienta. Ponadto redystrybucja danych do pozostałych partycji byłaby trudna i powodowałaby powstawanie komunikatów w niewłaściwej kolejności. W przypadku konieczności zmniejszenia liczby partycji zaleca się usunięcie danego tematu i odtworzenie go lub (jeśli usunięcie nie jest możliwe) utworzenie nowej wersji istniejącego tematu (np. „my-topic-v2”) i przeniesienie całego ruchu produkcyjnego do nowego tematu.

Usuwanie tematu

Nawet temat bez komunikatów wykorzystuje zasoby klastra, takie jak miejsce na dysku, otwarte uchwyty plików i pamięć. Kontroler posiada również metadane śmieci, o których musi zachować wiedzę, co przy dużej skali może mieć negatywny wpływ na wydajność. Jeżeli temat przestaje być potrzebny, można go usunąć, by zwolnić te zasoby. Jest to możliwe tylko wtedy, gdy brokery w klastrze mają ustawioną wartość `true` dla opcji `delete.topic.enable`. Jeśli mają skonfigurowaną wartość `false`, żądanie usunięcia tematu zostanie zignorowane i nie powiedzie się.

Usuwanie tematu jest operacją asynchroniczną. Oznacza to, że uruchomienie tego polecenia spowoduje wyznaczenie danego tematu do usunięcia, ale samo usunięcie może nie nastąpić natychmiast, w zależności od ilości danych i wymaganego czyszczenia. Kontroler powiadamia brokery o oczekującym usunięciu możliwie najszybciej (po zakończeniu wykonywania bieżących zadań), a brokery unieważniają metadane tematu i usuwają pliki z dysku. Ze względu na ograniczenia w sposobie wykonywania tych operacji przez kontroler zaleca się, by operatory nie usuwały więcej niż dwóch tematów naraz i przed przystąpieniem do usuwania kolejnych tematów odczekały wystarczająco dużo czasu, aby umożliwić ukończenie tego zadania. W małym klastrze pokazanym w przykładach w tej książce usuwanie tematu nastąpi niemal natychmiast, ale w większych klastrach może to potrwać dłużej.



Możliwa utrata danych

Usunięcie tematu spowoduje usunięcie również wszystkich jego komunikatów. Jest to operacja nieodwracalna, dlatego wykonuj ją ostrożnie.

Poniżej pokazaliśmy przykład usuwania tematu o nazwie `my-topic` za pomocą argumentu `--delete`. W zależności od wersji Kafki może pojawić się informacja, że argument nie będzie działał, jeśli nie zostanie ustawiona jakaś inna konfiguracja:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic my-topic
```

```
Note: This will have no impact if delete.topic.enable is not set to true.  
#
```

Jak widać, nie ma żadnych informacji zwrotnych o pomyślnym lub niepomyślnym zakończeniu usuwania tematu. Można to zweryfikować, uruchamiając opcję `--list` lub `--describe`, aby sprawdzić, czy temat znajduje się jeszcze w klastrze.

Grupy konsumentów

Grupy konsumentów to skoordynowane grupy konsumentów Kafki przetwarzających tematy lub kilka partycji pojedynczego tematu. Narzędzie `kafka-consumer-groups.sh` pomaga zarządzać grupami konsumentów przetwarzającymi tematy w klastrze i zbierać informacje na ich temat. Może być używane do wyświetlania listy grup konsumentów, generowania opisów określonych grup, usuwania grup konsumentów lub określonych informacji o grupie oraz resetowania informacji o przesunięciu grupy konsumentów.



Grupy konsumentów oparte na ZooKeeperze

W starszych wersjach Kafki grupami konsumentów można zarządzać w ZooKeeperze i tam można je utrzymywać. Zostało to zarzucone w wersji 0.11.0.* i stare grupy konsumentów nie są już używane. Niektóre wersje dostarczonych skryptów mogą nadal zawierać przestarzałe polecenia `--zookeeper` dla łańcuchów znaków połączenia, ale nie zaleca się ich stosowania, chyba że ma się stare środowisko z grupami konsumentów, które nie zostały zaktualizowane do nowszych wersji Kafki.

Wyświetlanie listy i generowanie opisu grup

Aby wyświetlić listę grup konsumentów, użyj parametrów `--bootstrap-server` i `--list`. Konsumenty *ad hoc* korzystające ze skryptu `kafka-consumer-groups.sh` pojawiają się na liście konsumentów jako `console-consumer-<wygenerowane_id>`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
console-consumer-95554
console-consumer-9581
my-consumer
#
```

Dla każdej z wymienionych grup można uzyskać więcej informacji, zmieniając parametr `--list` na `--describe` i dodając parametr `--group`. Spowoduje to wyświetlenie listy wszystkich tematów i partycji, z których konsumuje dana grupa, a także dodatkowych informacji, takich jak przesunięcia dla każdej partycji tematu. W tabeli 12.1 przedstawiliśmy pełny opis wszystkich pól występujących w danych wyjściowych.

Pobierzmy szczegółowe informacje np. dla grupy konsumentów *ad hoc* o nazwie `my-consumer`:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-consumer
GROUP          TOPIC          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET
LAG           CONSUMER-ID
HOST                  CLIENT-ID
my-consumer    my-topic      0           2                4
2              consumer-1-029af89c-873c-4751-a720-cefd41a669d6
127.0.0.1      consumer-1
my-consumer    my-topic      1           2                3
1              consumer-1-029af89c-873c-4751-a720-cefd41a669d6
127.0.0.1      consumer-1
my-consumer    my-topic      2           2                3
1              consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2
127.0.0.1      consumer-2
#
```

Tabela 12.1. Pola występujące w danych wyjściowych dla grupy o nazwie *my-consumer*

Pole	Opis
GROUP	Nazwa grupy konsumentów
TOPIC	Nazwa konsumowanego tematu
PARTITION	Identyfikator konsumowanej partycji
CURRENT-OFFSET	Następne przesunięcie do przetworzenia przez grupę konsumentów dla danej partycji tematu. Jest to pozycja konsumenta na partycji
LOG-END-OFFSET	Przesunięcie aktualnej wysokiej wody pobrane z brokera dla danej partycji tematu. Jest to przesunięcie następnego komunikatu, który ma zostać wyprodukowany do tej partycji
LAG	Różnica pomiędzy CURRENT-OFFSET konsumenta a LOG-END-OFFSET brokera dla tej partycji tematu
CONSUMER-ID	Unikatowy identyfikator konsumenta wygenerowany na podstawie dostarczonego identyfikatora klienta
HOST	Adres hosta, z którego odczytuje grupa konsumentów
CLIENT-ID	Dostarczony przez klient łańcuch znaków, który identyfikuje klient konsumujący z tej grupy

Usuwanie grup

Grupy konsumentów można usuwać za pomocą argumentu `--delete`. Wykonanie tego polecenia powoduje usunięcie całej grupy, w tym wszystkich zapisanych przesunięć dla wszystkich tematów przetwarzanych przez grupę. Aby wykonać tę czynność, należy zamknąć wszystkie konsumenty w grupie, ponieważ grupa konsumentów nie może mieć żadnych aktywnych członków. Jeśli spróbujesz usunąć grupę, która nie jest pusta, zgłoszony zostanie błąd *The group is not empty* (grupa nie jest pusta) i nic się nie stanie. Gdy doda się argument `--topic` i określi, które przesunięcia tematu mają zostać usunięte, można użyć tego samego polecenia również do usunięcia przesunięcia dla pojedynczego tematu przetwarzanego przez grupę bez usuwania całej grupy.

Oto przykład usunięcia całej grupy konsumentów o nazwie *my-consumer*:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete --group my-consumer
Deletion of requested consumer groups ('my-consumer') was successful.
#
```

Zarządzanie przesunięciami

Oprócz wyświetlania i usuwania przesunięć dla grupy konsumentów możliwe jest również pobieranie przesunięć i przechowywanie nowych przesunięć wykonanych w partii. Jest to przydatne w przypadku resetowania przesunięć dla konsumenta, gdy istnieje problem, który wymaga ponownego odczytania komunikatów, lub w przypadku pomijania komunikatu, z którym konsument ma problem (np. jeśli istnieje źle sformatowany komunikat, którego konsument nie może obsłużyć).

Eksportowanie przesunięć

Aby wyeksportować przesunięcia z grupy konsumentów do pliku CSV, użyj argumentu `--reset-of-fsets` z opcją `--dry-run`. Spowoduje to utworzenie eksportu aktualnych przesunięć w pliku o formacie, który może być ponownie wykorzystany do późniejszego importowania lub rollbacku przesunięć. Eksport w formacie CSV będzie miał następującą postać:

<nazwa_tematu>,<numer_partycji>,<przesunięcie>

Uruchomienie tego samego polecenia bez opcji `--dry-run` spowoduje całkowite zresetowanie przesunięć, należy więc zachować ostrożność.

Oto przykład eksportowania do pliku *offsets.csv* przesunięć dla tematu *my-topic*, który jest konsumowany przez grupę konsumentów o nazwie *my-consumer*:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--export --group my-consumer --topic my-topic
--reset-offsets --to-current --dry-run > offsets.csv

# cat offsets.csv
my-topic,0,8905
my-topic,1,8915
my-topic,2,9845
my-topic,3,8072
my-topic,4,8008
my-topic,5,8319
my-topic,6,8102
my-topic,7,12739
#
```

Importowanie przesunięć

Narzędzie importowania przesunięcia działa przeciwnie do eksportowania. Pobiera plik utworzony w poprzednim podpunkcie przez wyeksportowanie przesunięć i używa go do ustawienia aktualnych przesunięć dla grupy konsumentów. Powszechną praktyką jest eksportowanie aktualnych przesunięć dla grupy konsumentów, tworzenie kopii pliku (aby zabezpieczyć kopię zapasową) i edytowanie jej w celu zastąpienia przesunięć żądanymi wartościami.



Najpierw zatrzymaj konsumenty

Przed wykonaniem tego kroku ważne jest zatrzymanie wszystkich konsumentów w grupie. Nie będą odczytywać nowych przesunięć, jeśli będą one zapisywane, gdy grupa konsumentów jest aktywna. Konsumenty po prostu zastąpią wtedy importowane przesunięcia.

W poniższym przykładzie importujemy przesunięcia dla grupy konsumentów o nazwie *my-consumer* z utworzonego w ostatnim przykładzie pliku *offsets.csv*:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092
--reset-offsets --group my-consumer
--from-file offsets.csv --execute

TOPIC                PARTITION  NEW-OFFSET
my-topic              0          8905
my-topic              1          8915
my-topic              2          9845
my-topic              3          8072
my-topic              4          8008
my-topic              5          8319
my-topic              6          8102
my-topic              7          12739
#
```

Dynamiczne zmiany konfiguracji

Istnieje mnóstwo konfiguracji dla tematów, klientów, brokerów itd., które można dynamicznie aktualizować podczas działania bez konieczności zamykania lub ponownego wdrażania klastra. Głównym narzędziem do modyfikowania tych konfiguracji jest `kafka-configs.sh`. Obecnie istnieją cztery główne kategorie, czyli tzw. **typy encji** (ang. *entity type*), zmian w konfiguracji, które można wprowadzać dynamicznie: **tematy**, **brokery**, **użytkownicy** i **klienci**. Dla każdego typu encji można nadpisywać określone konfiguracje. W każdym kolejnym wydaniu Kafki wciąż dodawane są nowe dynamiczne konfiguracje, powinieneś więc upewnić się, że masz wersję narzędzia zgodną z wersją Kafki, którą uruchamiasz. Aby ułatwić modyfikowanie tych konfiguracji za pośrednictwem automatyzacji, argumentu `--add-config-file` można używać z wstępnie sformatowanym plikiem wszystkich konfiguracji wymagających zarządzania i aktualizowania.

Nadpisywanie wartości domyślnych konfiguracji tematu

Dla tematów domyślnie ustawionych jest wiele konfiguracji, które są zdefiniowane w statycznych plikach konfiguracyjnych brokera (np. reguła dla czasu retencji). Dzięki dynamicznym konfiguracjom możemy nadpisywać wartości domyślne poziomu klastra dla poszczególnych tematów, aby w ramach jednego klastra uwzględnić różne przypadki użycia. W tabeli 12.2 przedstawiono poprawne klucze konfiguracyjne dla tematów, które można zmieniać dynamicznie.

Tabela 12.2. Poprawne klucze dla tematów

Klucz konfiguracyjny	Opis
<code>cleanup.policy</code>	Jeśli ma ustawioną wartość <code>compact</code> (kompaktowanie dziennika), komunikaty z tego tematu będą porzucane i zachowywany będzie tylko najnowszy komunikat z danym kluczem
<code>compression.type</code>	Typ kompresji używany przez broker podczas zapisywania na dysk partii komunikatów dla tego tematu
<code>delete.retention.ms</code>	Określony w milisekundach czas przechowywania usuniętych nagrobków. Ma zastosowanie tylko do tematów z kompaktowaniem dziennika
<code>file.delete.delay.ms</code>	Określony w milisekundach czas oczekiwania przed usunięciem z dysku segmentów dziennika i indeksów dla tego tematu
<code>flush.messages</code>	Liczba komunikatów, po otrzymaniu których wymuszone zostanie spłukiwanie (przenoszenie) na dysk komunikatów tego tematu
<code>flush.ms</code>	Określony w milisekundach czas, po którym wymuszone zostanie spłukiwanie na dysk komunikatów tego tematu
<code>follower.replication.throttled.replicas</code>	Lista replik, dla których broker podrzędny (obserwator) powinien ograniczyć przepustowość replikacji dziennika
<code>index.interval.bytes</code>	Liczba bajtów komunikatów, które mogą zostać wyprodukowane pomiędzy wpisami w indeksie segmentu dziennika
<code>leader.replication.throttled.replicas</code>	Lista replik, dla których broker główny (lider) powinien ograniczyć przepustowość replikacji dziennika
<code>max.compaction.lag.ms</code>	Maksymalny limit czasu, przez który komunikat nie może być poddany kompaktowaniu w dzienniku

Tabela 12.2. Poprawne klucze dla tematów — ciąg dalszy

Klucz konfiguracyjny	Opis
<code>max.message.bytes</code>	Określony w bajtach maksymalny rozmiar pojedynczego komunikatu dla tego tematu
<code>message.downconversion.enable</code>	Jego włączenie umożliwia downgrade wersji formatu komunikatów, co generuje pewne obciążenie
<code>message.format.version</code>	Wersja formatu komunikatów, której broker będzie używał przy zapisywaniu komunikatów na dysku. Musi to być poprawny numer wersji API
<code>message.timestamp.difference.max.ms</code>	Określona w milisekundach maksymalna dopuszczalna różnica pomiędzy znacznikiem czasu komunikatu a znacznikiem czasu brokera podczas otrzymywania komunikatu. Ma to zastosowanie tylko wtedy, gdy klucz <code>message.timestamp.type</code> ma ustawioną wartość <code>CreateTime</code>
<code>message.timestamp.type</code>	Znacznik czasu, który ma być stosowany podczas zapisywania komunikatów na dysku. Aktualnie dostępne wartości to <code>CreateTime</code> dla znacznika czasu określanego przez klient i <code>LogAppendTime</code> dla znacznika czasu w momencie zapisywania komunikatu na partycji przez broker
<code>min.cleanable.dirty.ratio</code>	Częstotliwość prób kompaktowania partycji dla tego tematu podejmowanych przez mechanizm kompaktowania dziennika. Jest wyrażona jako stosunek liczby nieskompaktowanych segmentów dziennika do całkowitej liczby segmentów dziennika. Ma zastosowanie tylko do tematów z włączonym kompaktowaniem dziennika
<code>min.compaction.lag.ms</code>	Minimalny czas, przez który komunikat będzie pozostawał w dzienniku nieskompaktowany
<code>min.insync.replicas</code>	Minimalna liczba replik, które muszą być zsynchronizowane, aby partycja tematu była uznawana za dostępną
<code>preallocate</code>	Jeśli ma ustawioną wartość <code>true</code> , segmenty dziennika dla tego tematu powinny być wstępnie alokowane, gdy wprowadzany jest nowy segment
<code>retention.bytes</code>	Określona w bajtach liczba komunikatów, która ma być przechowywana dla tego tematu
<code>retention.ms</code>	Określony w milisekundach czas przechowywania komunikatów dla tego tematu
<code>segment.bytes</code>	Określona w bajtach liczba komunikatów, która ma być zapisywana w pojedynczym segmencie dziennika w partycji
<code>segment.index.bytes</code>	Określony w bajtach maksymalny rozmiar indeksu pojedynczego segmentu dziennika
<code>segment.jitter.ms</code>	Maksymalna liczba milisekund, która jest dodawana do <code>segment.ms</code> podczas wprowadzania nowych segmentów dziennika
<code>segment.ms</code>	Określona w milisekundach częstotliwość rotowania segmentu dziennika dla każdej partycji
<code>unclean.leader.election.enable</code>	Jeśli ma ustawioną wartość <code>false</code> , dla tego tematu nie jest dozwolony wybór nieczystego lidera

Format polecenia zmiany konfiguracji tematu jest następujący:

```
kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name <nazwa_tematu>
--add-config <klucz>=<wartość>[,<klucz>=<wartość>...]
```

Oto przykład ustawienia 1-godzinnej (3 600 000 ms) retencji dla tematu o nazwie my-topic:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--add-config retention.ms=3600000
Updated config for topic: "my-topic".
#
```

Nadpisywanie wartości domyślnych konfiguracji klienta i użytkownika

Dla klientów i użytkowników Kafki można nadpisywać tylko kilka konfiguracji, które zasadniczo są typami kwot. Dwie z najczęściej zmienianych konfiguracji to przepływności w bajtach na sekundę na broker dozwolone dla producentów i konsumentów o określonym identyfikatorze klienta. Pełną listę współdzielonych konfiguracji, które można modyfikować zarówno dla *użytkowników*, jak i *klientów*, przedstawiliśmy w tabeli 12.3.

Tabela 12.3. Konfiguracje (klucze) dla klientów

Klucz konfiguracyjny	Opis
consumer_bytes_rate	Określona w bajtach liczba komunikatów, którą pojedynczy identyfikator klienta może konsumować z pojedynczego brokera w sekundę
producer_bytes_rate	Określona w bajtach liczba komunikatów, którą pojedynczy identyfikator klienta może wyprodukować do pojedynczego brokera w sekundę
controller_mutations_rate	Akceptowalne tempo mutacji dla żądań tworzenia tematów, żądań tworzenia partycji i żądań usuwania tematów. Tempo jest akumulowane przez liczbę tworzonych lub usuwanych partycji
request_percentage	Procent na kwotę okna (z wartości całkowitej (num. i.o. threads+num.network.↪threads)·100%) dla żądań od użytkownika lub klienta



Nierówne dławienie w źle zrównoważonych klastrach

Ponieważ dławienie jest wykonywane osobno dla każdego brokera, zrównoważenie rozłożenia przywództwa partycji w klastrze staje się szczególnie ważne dla poprawnego egzekwowania dławienia. Jeśli będziesz miał w klastrze 5 brokerów i określisz dla klienta kwotę produkcyjną 10 MB/s, ten klient będzie mógł produkować 10 MB/s *na każdy* broker w tym samym czasie, czyli łącznie 50 MB/s przy założeniu zrównoważonego rozłożenia przywództwa na wszystkie 5 hostów. Jeśli jednak liderzy wszystkich partycji będą znajdować się na tym samym brokerze, jeden producent będzie w stanie wyprodukować maksymalnie 10 MB/s.



Identyfikator klienta a grupa konsumentów

Identyfikator klienta niekoniecznie musi być taki sam jak nazwa grupy konsumentów. Konsumenty mogą ustawiać swoje własne identyfikatory klienta i możesz mieć wiele konsumentów w różnych grupach, które określają ten sam identyfikator klienta. Najlepszą praktyką jest ustawienie dla każdej grupy konsumentów jakiegoś unikatowego identyfikatora klienta, który będzie ją identyfikował. Pozwala to pojedynczej grupie konsumentów na współdzielenie kwoty i ułatwia zidentyfikowanie w dziennikach, która grupa jest odpowiedzialna za żądania.

Pewne zmiany konfiguracji użytkownika i klienta można określić razem dla zgodnych konfiguracji, które mają zastosowanie do ich obu. Oto przykład polecenia zmiany tempa mutacji kontrolera zarówno dla użytkownika, jak i klienta w jednym kroku konfiguracyjnym:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --add-config "controller_mutations_rate=10"
--entity-type clients --entity-name <id_klienta>
--entity-type users --entity-name <id_uzytkownika>
#
```

Nadpisywanie wartości domyślnych konfiguracji brokera

Konfiguracje na poziomie brokera i klastra będą przede wszystkim ustawiane statycznie w plikach konfiguracyjnych klastra, ale wiele ustawień można nadpisać podczas działania bez konieczności ponownego wdrażania Kafki. Za pomocą narzędzia `kafka-configs.sh` dla brokerów można wprowadzać ponad 80 nadpisań. W związku z tym nie wymienimy ich wszystkich w tej książce, ale więcej informacji na ich temat znajdziesz przy użyciu polecenia `--help` lub w dokumentacji *open source* (<https://oreil.ly/R8hbb>). Oto kilka ważnych konfiguracji, na które warto zwrócić uwagę:

`min.insync.replicas`

Dostosowuje minimalną liczbę replik, które muszą potwierdzić zapis, aby żądanie produkcji zakończyło się powodzeniem, gdy producenci mają ustawione `acks=all` lub `acks=-1`.

`unclean.leader.election.enable`

Umożliwia wybór replik jako lidera, nawet jeśli powoduje to utratę danych. Jest to przydatne, gdy dozwolona jest pewna utrata danych. Można też włączać to na krótkie przedziały czasu w celu „odklejenia” klastra Kafki, jeżeli nie można uniknąć nieodwracalnej utraty danych.

`max.connections`

Maksymalna liczba jednoczesnych połączeń z brokerem. W celu bardziej precyzyjnego dławienia można użyć także konfiguracji `max.connections.per.ip` (maksymalnej liczby połączeń na adres IP) i `max.connections.per.ip.overrides` (nadpisanie maksymalnej liczby połączeń na adres IP).

Generowanie opisów nadpisanych konfiguracji

Za pomocą narzędzia `kafka-config.sh` można wyświetlić listę wszystkich nadpisań konfiguracji. Pozwala to sprawdzać konkretne konfiguracje tematu, brokera lub klienta. Podobnie jak w przypadku innych narzędzi odbywa się to za pomocą polecenia `--describe`.

W poniższym przykładzie pobieramy wszystkie nadpisanie konfiguracji dla tematu o nazwie `my-topic`. Jak widać, jest to tylko czas retencji:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--describe --entity-type topics --entity-name my-topic
Configs for topics:my-topic are retention.ms=3600000
#
```




Wyłącznie nadpisania tematu

Opis konfiguracji pokazuje tylko nadpisanie — nie zawiera domyślnych ustawień klastra. Nie ma żadnego sposobu na dynamiczne wykrywanie konfiguracji samych brokerów. Oznacza to, że podczas wykorzystywania tego narzędzia w automatyzacji do wykrywania ustawień tematu lub klienta użytkownik musi czerpać wiedzę na temat domyślnej konfiguracji klastra z innych źródeł.

Usuwanie nadpisanych konfiguracji

Konfiguracje dynamiczne można całkowicie usunąć, co spowoduje, że encja powróci do ustawień domyślnych klastra. Aby usunąć nadpisanie konfiguracji, użyj polecenia `--alter` z parametrem `--delete-config`.

Usuńmy np. nadpisanie konfiguracji dla `retention.ms` dla tematu `my-topic`:

```
# kafka-configs.sh --bootstrap-server localhost:9092
--alter --entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

Produkowanie i konsumowanie

Podczas pracy z Kafką często konieczne jest ręczne wyprodukowanie lub skonsumowanie kilku przykładowych komunikatów w celu sprawdzenia, co się dzieje z aplikacjami. Pomocne mogą być w tym dwa narzędzia, `kafka-console-consumer.sh` i `kafka-console-producer.sh`, które omówiliśmy pokrótce w rozdziale 2. przy weryfikacji naszej instalacji. Te narzędzia to skrypty opakowujące główne biblioteki klienckie Javy, pozwalające na interakcję z tematami Kafki bez konieczności pisania w tym celu całej aplikacji.



Potokowe przesyłanie danych wyjściowych do innej aplikacji

Chociaż możliwe jest pisanie aplikacji opakowujących narzędzia konsumenta lub producenta konsolowego (np. w celu konsumowania komunikatów i potokowego przesyłania ich do innej aplikacji do przetwarzania), tego typu aplikacje są dość kruche i należy ich unikać. Trudno jest wchodzić w interakcje z konsolowym konsumentem w sposób, który nie spowoduje utraty danych. Analogicznie konsolowy producent nie pozwala na korzystanie ze wszystkich funkcjonalności, a poprawne wysyłanie bajtów jest dość trudne. Najlepiej bezpośrednio używać bibliotek klienckich Javy lub zewnętrznych bibliotek klienckich dla innych języków, korzystających bezpośrednio z protokołu Kafki.

Producent konsolowy

Za pomocą narzędzia `kafka-console-producer.sh` można zapisywać komunikaty w temacie Kafki w klastrze. Domyślnie komunikaty są odczytywane po jednym na linię, przy czym znak tabulacji oddziela klucz i wartość (jeśli nie ma znaku tabulacji, klucz ma wartość `null`). Podobnie jak w przypadku konsolowego konsumenta, producent odczytuje i produkuje surowe bajty przy użyciu domyślnego serializatora (którym jest `DefaultEncoder`).

Producent konsolowy wymaga podania co najmniej dwóch argumentów, aby mieć informacje o tym, z jakim klastrem Kafki należy się połączyć i do jakiego tematu w tym klastrze należy produkować. Pierwszym argumentem jest zwyczajowy łańcuch znaków połączenia, `--bootstrap-server`, z którego już korzystaliśmy. Po zakończeniu produkowania wysyłamy znak końca pliku (ang. *end-of-file* — EOF), aby zamknąć klient. W większości popularnych terminali odbywa się to za pomocą przycisków `Ctrl+D`.

Oto przykład wygenerowania czterech komunikatów do tematu `my-topic`:

```
# kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic
>Komunikat 1
>Komunikat testowy 2
>Komunikat testowy 3
>Komunikat 4
>^D
#
```

Korzystanie z opcji konfiguracyjnych producenta

Producentowi konsolowemu można przekazywać również normalne opcje konfiguracyjne producenta. Można to zrobić na dwa sposoby, w zależności od tego, ile opcji musisz przekazać i jak wolisz to wykonać. Pierwszym z nich jest dostarczenie pliku konfiguracyjnego producenta poprzez podanie parametru `--producer.config <plik_konfiguracyjny>`, gdzie `<plik_konfiguracyjny>` jest pełną ścieżką do pliku zawierającego opcje konfiguracyjne. Drugim sposobem jest określenie tych opcji w wierszu poleceń z co najmniej jednym argumentem w postaci `--producer-property <klucz>=<wartość>`, gdzie `<klucz>` jest nazwą opcji konfiguracyjnej, a `<wartość>` jest ustawianą wartością. Może to być przydatne dla takich opcji producenta jak konfiguracje dotyczące partii komunikatów (np. `linger.ms` lub `batch.size`).



Mylące opcje wiersza poleceń

Opcja wiersza poleceń `--property` jest dostępna zarówno dla producenta, jak i konsumenta konsolowego, ale nie należy jej mylić z opcjami `--producer-property` i `--consumer-property`. Opcja `--property` jest stosowana tylko do przekazywania konfiguracji do narzędzia formatującego komunikaty, a nie do samego klienta.

Producent konsolowy ma wiele argumentów wiersza poleceń, których można używać z opcją `--producer-property` w celu dostosowania jego zachowania. Oto niektóre z bardziej przydatnych opcji:

`--batch-size`

Określa liczbę komunikatów wysłanych w jednej partii, jeśli nie są one wysyłane synchronicznie.

`--timeout`

Jeżeli producent pracuje w trybie asynchronicznym, ta opcja określa maksymalną ilość czasu oczekiwania na wypełnienie partii przed wyprodukowaniem, aby uniknąć długiego oczekiwania na tematy o powolnym produkowaniu.

`--compression-codec <łańcuch_znaków>`

Określa typ kompresji, która ma być stosowana podczas produkowania komunikatów. Poprawnymi typami są: `none`, `gzip`, `snappy`, `zstd` i `lz4`. Domyślną wartością jest `gzip`.

`--sync`

Produkowanie komunikatów synchronicznie i oczekiwanie na potwierdzenie każdego komunikatu przed wysłaniem następnego.

Opcje czytnika linii

Kilka przydatnych opcji, które można przekazać producentowi konsolowemu za pomocą opcji wiersza poleceń `--property`, ma również klasa `kafka.tools.ConsoleProducer$LineMessageReader` odpowiedzialna za odczytywanie standardowych danych wejściowych i tworzenie rekordów producenta. Oto one:

`ignore.error`

Ustawiana na `false`, aby rzucać wyjątek, gdy `parse.key` ma wartość `true`, a separator kluczy nie jest obecny. Wartość domyślna to `true`.

`parse.key`

Ustawiana na `false`, aby zawsze ustawiać dla klucza wartość `null`. Wartość domyślna to `true`.

`key.separator`

Określa znak separatora, który ma być używany między kluczem komunikatu a wartością komunikatu podczas odczytu. Domyślnym separatorem jest znak tabulatora.



Zmiana zachowania odczytu linii

Możesz dostarczyć Kafce własną klasę dla niestandardowych metod odczytu linii. Utworzona przez Ciebie klasa musi rozszerzać interfejs `kafka.common.MessageReader` i będzie odpowiedzialna za tworzenie obiektu `ProducerRecord`. Swoją klasę określ w wierszu poleceń za pomocą opcji `--line-reader` i upewnij się, że w ścieżce klas znajduje się zawierający ją plik JAR. Domyślny czytnik linii to `kafka.tools.ConsoleProducer$LineMessageReader`.

Podczas generowania komunikatów `LineMessageReader` rozdzieli dane wejściowe przy pierwszym wystąpieniu separatora kluczy (`key.separator`). Jeżeli dalej nie będzie już żadnych znaków, wartość komunikatu będzie pusta. Jeśli w wierszu nie ma znaku separatora kluczy lub `parse.key` ma ustawione `false`, klucz będzie miał wartość `null`.

Konsument konsolowy

Narzędzie `kafka-console-consumer.sh` pozwala konsumować komunikaty z co najmniej jednego tematu w klastrze Kafki. Komunikaty są wypisywane w standardowym strumieniu wyjściowym, rozdzielone znakami nowej linii. Domyślnie w komunikacie wysyłane są surowe bajty, bez klucza i bez formatowania (używany jest `DefaultFormatter`). Podobnie jak w przypadku producenta do uruchomienia potrzebnych jest kilka podstawowych opcji: łańcuch znaków połączenia z klastrem, temat do konsumowania oraz przedział czasowy konsumowania.



Sprawdzanie wersji narzędzia

Bardzo ważne jest, aby stosowany konsument miał tę samą wersję co klastr Kafka. Starsze konsumenty konsolowe mogą uszkodzić klastr przez niewłaściwy sposób interakcji z klastrzem lub ZooKeeperem.

Podobnie jak w innych poleceniach łańcuchem znaków dla połączenia z klastrzem będzie opcja `--bootstrap-server`. Natomiast w przypadku określania tematów do konsumowania można wybrać jedną z dwóch opcji:

`--topic`

Określa pojedynczy temat do konsumowania.

`--whitelist`

Wyrażenie regularne pasujące do wszystkich tematów, które mają być konsumowane (pamiętaj o odpowiednim zastosowaniu znaków ucieczki dla wyrażenia regularnego, aby zostało poprawnie przetworzone przez powłokę).

Należy wybrać i zastosować tylko jedną z powyższych opcji. Po uruchomieniu konsument konsolowy będzie cały czas próbował przetwarzać komunikaty, dopóki nie zostanie wydane polecenie ucieczki z powłoki (w tym przypadku `Ctrl+C`). Oto przykład konsumowania wszystkich tematów z naszego klastra, które pasują do prefiksu `my` (tutaj mamy akurat tylko jeden — `my-topic`):

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
Komunikat 1
Komunikat testowy 2
Komunikat testowy 3
Komunikat 4
^C
#
```

Korzystanie z opcji konfiguracyjnych konsumenta

Oprócz tych podstawowych opcji wiersza poleceń konsumentowi konsolowemu można przekazywać również standardowe opcje konfiguracyjne. Podobnie jak w przypadku narzędzia `kafka-console-producer.sh` można to robić na dwa sposoby, w zależności od tego, ile opcji musisz przekazać i jak wolisz to wykonać. Pierwszym z nich jest dostarczenie pliku konfiguracyjnego konsumenta przez określenie opcji `--consumer.config <plik_konfiguracyjny>`, gdzie `<plik_konfiguracyjny>` to pełna ścieżka do pliku zawierającego opcje konfiguracyjne. Drugim sposobem jest określenie tych opcji w wierszu poleceń z co najmniej jednym argumentem w postaci `--consumer-property <klucz>=<wartość>`, gdzie `<klucz>` jest nazwą opcji konfiguracyjnej, a `<wartość>` jest ustawioną dla niej wartością.

Warto poznać jeszcze kilka innych opcji powszechnie stosowanych dla konsumenta konsolowego:

`--formatter <nazwa_klasy>`

Określa klasę narzędzia formatowania komunikatów, która ma być używana do dekodowania komunikatów. Domyślnie jest to `kafka.tools.DefaultMessageFormatter`.

`--from-beginning`

Ustawienie konsumowania komunikatów z określonego tematu (lub tematów) od najstarszego przesunięcia. W przeciwnym razie konsumowanie rozpoczyna się od najnowszego przesunięcia.

`--max-messages <liczba_całkowita>`

Maksymalna liczba komunikatów do skonsumowania przed zamknięciem konsumenta.

`-- partition <liczba_całkowita>`

Konsumowanie tylko z partycji o podanym identyfikatorze.

`--offset`

Identyfikator przesunięcia, od którego należy zacząć konsumowanie, jeśli został on podany (jako `<liczba_całkowita>`). Inne poprawne opcje to `earliest`, czyli przetwarzanie od początku, i `latest`, czyli przetwarzanie od najnowszego przesunięcia.

`--skip-message-on-reror`

Pominięcie komunikatu zamiast zatrzymania konsumenta, jeżeli podczas przetwarzania wystąpił błąd. Przydatne przy debugowaniu.

Opcje narzędzia formatowania komunikatów

Oprócz domyślnego dostępne są jeszcze trzy narzędzia formatowania komunikatów:

`kafka.tools.LoggingMessageFormatter`

Wypisuje komunikaty za pomocą rejestratora, a nie standardowego strumienia wyjściowego. Komunikaty są wypisywane na poziomie rejestrowania `INFO` i zawierają znacznik czasu, klucz oraz wartość.

`kafka.tools.ChecksumMessageFormatter`

Wypisuje tylko sumy kontrolne komunikatów.

`kafka.tools.NoOpMessageFormatter`

Konsumuje komunikaty, ale w ogóle ich nie wypisuje.

Poniżej pokazaliśmy przykład konsumowania tych samych komunikatów co poprzednio, ale przy użyciu narzędzia formatowania `kafka.tools.ChecksumMessageFormatter` zamiast domyślnego:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--whitelist 'my.*' --from-beginning
--formatter kafka.tools.ChecksumMessageFormatter
checksum:0
checksum:0
checksum:0
checksum:0
#
```

Narzędzie formatowania `Kafka.tools.DefaultMessageFormatter` również ma kilka użytecznych właściwości, które można przekazać za pomocą opcji wiersza poleceń `--property`. Przedstawiliśmy je w tabeli 12.4.

Klasy deserializatora muszą zaimplementować interfejs `org.apache.kafka.common.serialization.Deserializer`, a konsument konsolowy będzie wywoływał w nich metodę `toString`, aby wyświetlić dane wyjściowe. Zazwyczaj te deserializatory implementuje się jako klasę Javy, którą wstawia się do ścieżki klas dla konsumenta konsolowego, ustawiając zmienną środowiskową `CLASSPATH` przed uruchomieniem `kafka-console-consumer.sh`.

Tabela 12.4. Właściwości narzędzia formatowania komunikatów

Właściwość	Opis
<code>print.timestamp</code>	Ustawienie <code>true</code> powoduje wyświetlanie znacznika czasu każdego komunikatu (jeśli jest dostępny)
<code>print.key</code>	Ustawienie <code>true</code> powoduje wyświetlanie oprócz wartości komunikatu także jego klucza
<code>print.offset</code>	Ustawienie <code>true</code> powoduje wyświetlanie oprócz wartości komunikatu także jego przesunięcia
<code>print.partition</code>	Ustawienie <code>true</code> powoduje wyświetlanie partycji tematu, z której komunikat został skonsumowany
<code>key.separator</code>	Określa znak separatora używanego pomiędzy kluczem a wartością komunikatu podczas wypisywania
<code>line.separator</code>	Określa znak separatora używanego pomiędzy komunikatami
<code>key.deserializer</code>	Określa nazwę klasy używanej do deserializacji klucza komunikatu przed wypisaniem
<code>value.deserializer</code>	Określa nazwę klasy używanej do deserializacji wartości komunikatu przed wypisaniem

Konsumowanie tematów przesunięć

Czasami przydatne jest sprawdzenie, jakie przesunięcia są zatwierdzane dla grup konsumentów klastra. Możesz np. sprawdzić, czy określona grupa w ogóle zatwierdza przesunięcia lub jak często są one zatwierdzane. W tym celu można wykorzystać konsolowy do konsumowania specjalnego wewnętrznego tematu o nazwie `__consumer_offsets`. Wszystkie przesunięcia konsumencie są zapisywane jako komunikaty w tym temacie. Aby zdekodować komunikaty z tego tematu, musisz użyć klasy formatowania `kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter`.

Poniżej przedstawiliśmy przykład konsumowania najstarszego komunikatu z tematu `__consumer_offsets`:

```
# kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic __consumer_offsets --from-beginning --max-messages 1
--formatter "kafka.coordinator.group.GroupMetadataManager$OffsetsMessageFormatter"
--consumer-property exclude.internal.topics=false
[my-group-name,my-topic,0]::[OffsetMetadata[1,NO_METADATA] CommitTime 1623034799990
ExpirationTime 1623639599990]
Processed a total of 1 messages
#
```

Zarządzanie partycjami

Domyślna instalacja Kafki zawiera również kilka skryptów do zarządzania partycjami. Jedno z tych narzędzi umożliwia ponowny wybór replik głównych (liderów). Inne jest niskopoziomowym narzędziem do przypisywania partycji brokerom. Wszystkie te narzędzia mogą być pomocne w sytuacjach, w których potrzebne jest ręczne równoważenie ruchu komunikatów w klastrze brokerów Kafki.

Wybór repliki preferowanej

Jak opisaliśmy w rozdziale 7., partycje mogą mieć wiele replik w celu zapewnienia niezawodności. Należy pamiętać, że w danym momencie tylko jedna z tych replik może być liderem dla partycji, a wszystkie operacje produkowania i konsumowania są wykonywane na tym brokerze. Utrzymanie

równowagi związanej z tym, które repliki partycji mają przywództwo na poszczególnych brokerach, jest niezbędne do równomiernego rozłożenia obciążenia na cały klastr Kafka.

W Kafce lidera definiuje się jako pierwszą replikę zsynchronizowaną na liście replik. Jednak gdy broker zostanie zatrzymany lub utraci łączność z resztą klastra, przywództwo jest przenoszone na inną replikę zsynchronizowaną, a pierwotny lider nie wznawia automatycznie przywództwa żadnej partycji. Może to powodować skrajnie nieefektywne rozłożenie przywództwa po wdrożeniu w całym klastrze, jeśli automatyczne równoważenie liderów nie jest włączone. Dlatego zaleca się włączenie tego ustawienia lub użycie innych narzędzi na licencji *open source*, takich jak Cruise Control, aby zapewnić utrzymywanie dobrej równowagi przez cały czas.

Jeśli okaże się, że klastr Kafka ma słabą równowagę, można przeprowadzić lekką, nieinwazyjną procedurę zwaną **wyborem preferowanego lidera**. Instruuje ona kontroler klastra, aby wybrał idealnego lidera dla partycji. Klienci mogą automatycznie śledzić zmiany w przywództwie, dzięki czemu mogą przenosić się do nowego brokera w klastrze, do którego przenoszone jest przywództwo. Operację tę można uruchomić ręcznie za pomocą narzędzia `kafka-leader-election.sh`. Dostępna jest również starsza wersja tego narzędzia o nazwie `kafka-preferred-replica-election.sh`, ale została ona zarzucona na rzecz nowego narzędzia, które oferuje więcej opcji, np. określanie, czy wybierana ma być replika „preferowana” czy „nieczysta”.

Wybór preferowanego lidera dla wszystkich tematów w klastrze można uruchomić np. za pomocą następującego polecenia:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --all-topic-partitions
#
```

Wybór można również przeprowadzić dla określonych partycji lub tematów. W tym celu należy przekazać bezpośrednio nazwę tematu za pomocą opcji `--topic` i partycję za pomocą opcji `--partition`. Możliwe jest też przekazanie listy kilku partycji. Wymaga to skonfigurowania pliku JSON, który nazwiemy *partitions.json*:

```
{
  "partitions": [
    {
      "partition": 1,
      "topic": "my-topic"
    },
    {
      "partition": 2,
      "topic": "foo"
    }
  ]
}
```

W tym przykładzie rozpoczniemy wybór preferowanej repliki z listą partycji określoną w pliku *partitions.json*:

```
# kafka-leader-election.sh --bootstrap-server localhost:9092
--election-type PREFERRED --path-to-json-file partitions.json
#
```

Zmianianie replik partycji

Czasami konieczna może być ręczna zmiana przypisań repliki dla partycji. Oto kilka sytuacji, w których może to być wymagane:

- Na brokerach występuje nierównomierne obciążenie, które nie jest prawidłowo obsługiwane przez automatyczną dystrybucję lidera.
- Broker został przełączony w tryb offline i partycja nie ma wystarczającej liczby replik zsynchronizowanych.
- Został dodany nowy broker i konieczne jest szybsze zrównoważenie na nim nowych partycji.
- Wymagane jest dostosowanie współczynnika replikacji tematu.

Do wykonania tej operacji można użyć narzędzia `kafka-reassign-partitions.sh`. Jest to wieloetapowy proces wygenerowania zestawu przesunięć, a następnie wykonania dostarczonej propozycji tego zestawu. Najpierw należy użyć listy brokerów i listy tematów do wygenerowania propozycji zestawu przesunięć. Będzie to wymagało wygenerowania pliku JSON z listą tematów, które mają zostać dostarczone. Następny krok to wykonanie przesunięć, które zostały wygenerowane przez poprzednią propozycję. Na koniec można użyć narzędzia z wygenerowaną listą, aby śledzić i weryfikować postęp lub ukończenie ponownego przypisywania partycji.

Wygenerujmy hipotetyczny scenariusz, w którym mamy klaster Kafki z 4 brokerami. Niedawno dodaliśmy 2 nowe brokery, zwiększając ich liczbę do 6, i chcemy przenieść 2 tematy na brokery 5. i 6.

Aby wygenerować zestaw przesunięć partycji, najpierw musimy utworzyć plik zawierający obiekt JSON z listą tematów. Obiekt JSON jest sformatowany w następujący sposób (aktualnie numer wersji to zawsze 1):

```
{
  "topics": [
    {
      "topic": "foo1"
    },
    {
      "topic": "foo2"
    }
  ],
  "version": 1
}
```

Po zdefiniowaniu pliku JSON możemy go użyć do wygenerowania zestawu przesunięć partycji, aby przenieść tematy wymienione w pliku `topics.json` na brokery o identyfikatorach 5 i 6:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--topics-to-move-json-file topics.json
--broker-list 5,6 --generate
{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
               {"topic":"foo1","partition":0,"replicas":[3,4]},
               {"topic":"foo2","partition":2,"replicas":[1,2]},
               {"topic":"foo2","partition":0,"replicas":[3,4]},
               {"topic":"foo1","partition":1,"replicas":[2,3]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]}
```



```

}

Proposed partition reassignment configuration

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
               {"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":2,"replicas":[5,6]},
               {"topic":"foo2","partition":0,"replicas":[5,6]},
               {"topic":"foo1","partition":1,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
#

```

Zaproponowane tutaj dane wyjściowe są sformatowane poprawnie i możemy je zapisać w 2 nowych plikach JSON, które nazwiemy *revert-reassignment.json* i *expand-cluster-reassignment.json*. Pierwszy plik może być stosowany do przeniesienia partycji z powrotem w pierwotne miejsce, jeśli z jakiegoś powodu trzeba wykonać rollback. Drugi plik może być wykorzystywany do następnego kroku, ponieważ jest to tylko propozycja i jeszcze niczego nie wykonaliśmy. Jak widać w tych danych wyjściowych, nie mamy tu dobrej równowagi przywództwa, ponieważ propozycja spowoduje przeniesienie całego przywództwa na broker 5. Na razie to zignorujemy i przyjmiemy, że w klastrze włączone jest automatyczne równoważenie przywództwa, co pomoże je później rozdystrybuować. Należy zwrócić uwagę, że jeśli wiemy, gdzie dokładnie chcemy przenieść partycje, pierwszy krok możemy pominąć i ręcznie utworzyć plik JSON do przeniesienia partycji.

Aby wykonać proponowane ponowne przypisywanie partycji z pliku *expand-cluster-reassignment.json*, uruchom następujące polecenie:

```

# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
               {"topic":"foo1","partition":0,"replicas":[3,4]},
               {"topic":"foo2","partition":2,"replicas":[1,2]},
               {"topic":"foo2","partition":0,"replicas":[3,4]},
               {"topic":"foo1","partition":1,"replicas":[2,3]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
               {"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":2,"replicas":[5,6]},
               {"topic":"foo2","partition":0,"replicas":[5,6]},
               {"topic":"foo1","partition":1,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
#

```

Spowoduje to rozpoczęcie ponownego przypisywania określonych replik partycji do nowych brokerów. Dane wyjściowe są takie same jak wygenerowana weryfikacja propozycji. Kontroler klastra wykonuje to ponowne przypisywanie, dodając nowe repliki do listy replik dla każdej partycji, co tymczasowo zwiększa współczynnik replikacji tych tematów. Nowe repliki kopiują następnie z aktualnego lidera wszystkie istniejące komunikaty dla poszczególnych partycji. W zależności od wielkości partycji na dysku może to zająć trochę czasu, ponieważ dane są kopiowane przez sieć do nowych replik. Po zakończeniu replikacji kontroler usuwa stare repliki z listy replik, co zmniejsza współczynnik replikacji do pierwotnej wartości.

Oto kilka innych przydatnych funkcjonalności tego polecenia, z których możesz skorzystać:

`--additional`

Pozwala dodawać przypisania do listy ponownych przypisań bez zakłócania procesu i konieczności oczekiwania, aż oryginalne czynności zostaną zakończone i rozpocznie się nowa sesja.

`--disable-rack-ware`

Czasami ze względu na włączenie informacji o szafach rackowych stan końcowy propozycji może być nieosiągalny. W razie potrzeby można to nadpisać tym poleceniem.

`--throttle`

Wartość wyrażona w bajtach na sekundę. Ponowne przypisywanie partycji ma duży wpływ na wydajność klastra, ponieważ powoduje zmiany w spójności pamięci podręcznej stron, wykorzystując przepustowość sieci oraz operacje we-wy dysku. W tej sytuacji przydatne może być ograniczanie (dławienie) ruchu partycji. Można je połączyć ze znacznikiem `--additional` w celu dławienia rozpoczętego już procesu ponownego przypisywania, który może powodować problemy.



Poprawa wykorzystania sieci podczas ponownego przypisywania replik

Podczas usuwania wielu partycji z jednego brokera, np. jeśli broker jest usuwany z klastra, warto najpierw usunąć z niego wszystkich liderów. Można to zrobić ręcznie, przenosząc liderów na inne brokery, jednak użycie do tego narzędzia `kafka-reassign-partitions.sh` jest uciążliwe. Prawdopodobnie najprostszym sposobem jest skorzystanie z jakiegoś narzędzia dostępnego na licencji *open source*, np. Cruise Control, oferującego takie funkcjonalności jak „degradacja” brokera, która bezpiecznie przenosi przywództwo z brokera.

Jeśli jednak nie masz dostępu do takich narzędzi, wystarczy zrestartować broker. Gdy broker przygotowuje się do zamknięcia, wszyscy liderzy dla partycji znajdujący się na tym konkretnym brokerze są przenoszeni na inne brokery w klastrze. Może to znacznie zwiększyć wydajność ponownego przypisywania i zmniejszyć wpływ na klaster, ponieważ ruch replikacji będzie dystrybuowany do wielu brokerów. Wymagane może być jednak tymczasowe wyłączenie funkcjonalności automatycznego ponownego przypisywania liderów, żeby liderzy nie powrócili na broker po restarcie.

Aby sprawdzić postępy przesunięć partycji, można użyć narzędzia `kafka-reassign-partitions.sh` w celu zweryfikowania statusu ponownego przypisywania. Spowoduje to wyświetlenie trwających procesów przypisywania partycji, procesów ukończonych oraz procesów, które zakończyły się niepowodzeniem (jeśli wystąpił błąd). Wymagany jest do tego plik z obiektem JSON, którego używaliśmy w kroku wykonywania.

Oto przykład wyników zastosowania opcji `--verify` podczas uruchamiania poprzedniego ponownego przypisywania partycji z pliku *expand-cluster-reassignment.json*:

```
# kafka-reassign-partitions.sh --bootstrap-server localhost:9092
--reassignment-json-file expand-cluster-reassignment.json
--verify
Status of partition reassignment:
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo1,1] is in progress
Reassignment of partition [foo1,2] is in progress
Reassignment of partition [foo2,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
Reassignment of partition [foo2,2] completed successfully
#
```

Zmiana współczynnika replikacji

Narzędzie `kafka-reassign-partitions.sh` można wykorzystać także do zwiększania lub zmniejszania współczynnika replikacji (ang. *replication factor* — RF) dla partycji. Może to być konieczne, gdy partycja została utworzona z niewłaściwym RF, chcemy zwiększyć redundancję w miarę rozszerzania klastra lub chcemy zmniejszyć redundancję w celu zmniejszenia kosztów. Ponadto jeśli zmienione zostanie domyślne ustawienie RF klastra, istniejące tematy nie zostaną automatycznie dostosowane. Można wtedy zastosować narzędzie `kafka-reassign-partitions.sh`, aby zwiększyć RF na istniejących partycjach.

Gdybyśmy chcieli np. zwiększyć do 3 wartość RF tematu `foo1` z poprzedniego przykładu, moglibyśmy utworzyć plik JSON podobny do używanej wcześniej propozycji wykonania, ale z dopisanym do zbioru replik dodatkowym identyfikatorem brokera. Byłby to plik JSON o nazwie *increase-foo1-RF.json*, w którym do istniejącego zbioru 5,6 dodalibyśmy broker 4:

```
{
  {"version":1,
   "partitions":[{"topic":"foo1","partition":1,"replicas":[5,6,4]},
                 {"topic":"foo1","partition":2,"replicas":[5,6,4]},
                 {"topic":"foo1","partition":3,"replicas":[5,6,4]},
  ]
}
```

Do wykonania tej propozycji wykorzystalibyśmy następnie pokazane wcześniej polecenie. Po zakończeniu wykonywania moglibyśmy użyć flagi `--verify` lub skryptu `kafka-topics.sh` do wygenerowania opisu tematu, aby sprawdzić, czy RF został zwiększony:

```
# kafka-topics.sh --bootstrap-server localhost:9092 --topic foo1 --describe
Topic:foo1 PartitionCount:3 ReplicationFactor:3 Configs:
Topic: foo1 Partition: 0 Leader: 5 Replicas: 5,6,4 Isr: 5,6,4
Topic: foo1 Partition: 1 Leader: 5 Replicas: 5,6,4 Isr: 5,6,4
Topic: foo1 Partition: 2 Leader: 5 Replicas: 5,6,4 Isr: 5,6,4
#
```

Anulowanie ponownego przypisywania replik

Anulowanie ponownego przypisywania replik było niegdyś niebezpiecznym procesem, który wymagał ręcznej manipulacji węzłami ZooKeepera (ang. *znode*) przez usunięcie węzła `/admin/reassign_partitions`. Na szczęście obecnie jest to łatwiejsze. Skrypt `kafka-reassign-partitions.sh` (oraz opakowywany przez niego interfejs `AdminClient`) obsługuje teraz opcję `--cancel`, która powoduje anulowanie aktywnych procesów ponownego przypisywania w klastrze. Podczas zatrzymywania trwającego przenoszenia partycji polecenie `--cancel` ma przywrócić zbiór repliki do stanu sprzed zainicjowania ponownego przypisywania. Dlatego jeśli repliki są usuwane z niedziałającego lub przeciążonego brokera, może to pozostawić klastr w niepożądanym stanie. Nie ma również gwarancji, że przywrócony zbiór replik będzie miał taką samą kolejność jak poprzednio.

Zrzucanie segmentów dziennika

Czasami może zająć potrzeba zapoznania się z treścią konkretnego komunikatu, np. gdy w temacie pojawi się uszkodzony komunikat „trującej pigułki” (ang. *poison pill*), a konsument nie będzie w stanie go obsłużyć. Narzędzie `kafka-dump-log.sh` służy do dekodowania segmentów dziennika dla partycji. Umożliwia to przeglądanie poszczególnych komunikatów bez konieczności ich konsumowania i dekodowania. Narzędzie przyjmuje jako argument rozdzielaną przecinkami listę plików segmentów dziennika i może wypisywać informacje podsumowujące komunikat albo szczegółowe dane komunikatu.

W tym przykładzie zrzucimy logi z przykładowego tematu `my-topic`, który jest nowym tematem zawierającym tylko cztery komunikaty. Najpierw zdekodujemy plik segmentu dziennika o nazwie `00000000000000000000.log` i pobierzemy podstawowe metadane o każdym komunikacie bez wypisywania samych zawartości komunikatów. W naszej przykładowej instalacji Kafki katalog danych tej platformy jest konfigurowany w katalogu `/tmp/kafka-logs`. W związku z tym naszym katalogiem do wyszukiwania segmentów dziennika będzie `/tmp/kafka-logs/<nazwa_tematu>-<partycja>`, czyli w tym przypadku `/tmp/kafka-logs/my-topic-0/`:

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 0
CreateTime: 1623034799990 size: 77 magic: 2
compresscodec: NONE crc: 1773642166 invalid: true
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 77
CreateTime: 1623034803631 size: 82 magic: 2
compresscodec: NONE crc: 1638234280 invalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
isTransactional: false isControl: false position: 159
CreateTime: 1623034808233 size: 82 magic: 2
compresscodec: NONE crc: 4143814684 invalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
```

```
isTransactional: false isControl: false position: 241
CreateTime: 1623034811837 size: 77 magic: 2
compresscodec: NONE crc: 3096928182 invalid: true
#
```

W następnym przykładzie dodamy opcję `--print-data-log`, która dostarczy nam m.in. rzeczywistych informacji o ładunku:

```
# kafka-dump-log.sh --files /tmp/kafka-logs/my-topic-0/00000000000000000000.log --print-
data-log
Dumping /tmp/kafka-logs/my-topic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 0
  CreateTime: 1623034799990 size: 77 magic: 2
  compresscodec: NONE crc: 1773642166 invalid: true
| offset: 0 CreateTime: 1623034799990 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 1
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 77
  CreateTime: 1623034803631 size: 82 magic: 2
  compresscodec: NONE crc: 1638234280 invalid: true
| offset: 1 CreateTime: 1623034803631 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 2
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 159
  CreateTime: 1623034808233 size: 82 magic: 2
  compresscodec: NONE crc: 4143814684 invalid: true
| offset: 2 CreateTime: 1623034808233 keysize: -1 valuesize: 14
  sequence: -1 headerKeys: [] payload: Test Message 3
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1
  producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0
  isTransactional: false isControl: false position: 241
  CreateTime: 1623034811837 size: 77 magic: 2
  compresscodec: NONE crc: 3096928182 invalid: true
| offset: 3 CreateTime: 1623034811837 keysize: -1 valuesize: 9
  sequence: -1 headerKeys: [] payload: Message 4
#
```

To narzędzie zawiera również kilka innych przydatnych opcji, takich jak walidacja pliku indeksu, który jest dołączany do segmentu dziennika. Indeks służy do wyszukiwania komunikatów w segmencie dziennika, a jeśli jest uszkodzony, będzie powodował błędy w konsumowaniu. Walidacja jest wykonywana za każdym razem, gdy broker jest uruchamiany w stanie nieczystym (tj. nie został poprawnie zatrzymany), ale może być również wykonywana ręcznie. Dostępne są dwie opcje sprawdzania indeksów, które oferują różny zakres walidacji. Opcja `--index-sanity-check` sprawdza jedynie, czy indeks jest w stanie używalności, natomiast `--verify-index-only` sprawdza niedopasowania w indeksie bez wypisywania wszystkich wpisów indeksu. Inna przydatna opcja, `--value-decoder-class`, pozwala na przekazanie do dekodera zserializowanych komunikatów w celu ich deserializacji.

Weryfikacja replik

Replikacja partycji działa podobnie do standardowego klienta konsumenckiego Kafki: broker podrzędny zaczyna replikację od najstarszego przesunięcia i regularnie tworzy na dysku punkty kontrolne bieżącego przesunięcia. Gdy replikacja zostanie zatrzymana i zrestartowana, wznowia działanie od ostatniego punktu kontrolnego. Może się zdarzyć, że zreplikowane wcześniej segmenty dziennika zostaną usunięte z brokera, a w takiej sytuacji broker podrzędny nie wypełni luk.

Aby sprawdzić, czy repliki dla partycji tematów są takie same w całym klastrze, można użyć narzędzia `kafka-replica-verification.sh`. Pobierze ono komunikaty ze wszystkich replik dla danego zestawu partycji tematów, sprawdzi, czy wszystkie komunikaty istnieją na wszystkich replikach, i wydrukuje maksymalne opóźnienie dla wskazanych partycji. Proces ten będzie działał w pętli do momentu jego przerwania. Aby go uruchomić, trzeba podać rozdzielaną przecinkami listę brokerów, z którymi chcemy się połączyć. Domyślnie weryfikowane są wszystkie tematy. Narzędziu można dostarczyć również wyrażenie regularne dopasowujące tematy, które mają zostać zweryfikowane.



Wpływ na wydajność klastra

Podobnie jak ponowne przypisywanie partycji, narzędzie weryfikacji replik będzie miało wpływ na wydajność klastra, gdyż w celu zweryfikowania repliki musi odczytać wszystkie komunikaty od najstarszego przesunięcia. Ponadto odczytuje komunikaty ze wszystkich replik partycji równolegle, dlatego należy go używać ostrożnie.

Zweryfikujmy np. repliki tematów zaczynających się od `my` na brokerach Kafki 1 i 2, które zawierają partycję 0 tematu `my-topic`:

```
# kafka-replica-verification.sh --broker-list
kafka.host1.domain.com:9092,kafka.host2.domain.com:9092 --topic-white-list 'my.*'

2021-06-07 03:28:21,829: verification process is started.
2021-06-07 03:28:51,949: max lag is 0 for partition my-topic-0 at offset 4 among 1
partitions
2021-06-07 03:29:22,039: max lag is 0 for partition my-topic-0 at offset 4 among 1
partitions
...
#
```

Pozostałe narzędzia

Dystrybucja Kafki zawiera jeszcze kilka innych narzędzi, których nie omówimy szczegółowo w tej książce, a które mogą być przydatne w administrowaniu klastrem Kafki w konkretnych przypadkach użycia. Więcej informacji na ich temat znajdziesz na stronie internetowej platformy Apache Kafka (<https://kafka.apache.org>):

Listy ACL klientów

Do interakcji z kontrolą dostępu dla klientów Kafki przeznaczone jest narzędzie wiersza poleceń `kafka-acls.sh`. Oferuje ono m.in. pełne funkcjonalności dla ustawiania właściwości

autoryzatora, konfigurowania reguł Deny lub Allow, wprowadzania ograniczeń na poziomie klastra lub tematu czy konfigurowania pliku TLS ZooKeepera.

Lekki MirrorMaker

Do mirroringu danych dostępny jest lekki skrypt `kafka-mirror-maker.sh`. Replikację opisaliśmy szerzej w rozdziale 10.

Narzędzia testowe

Dostępnych jest jeszcze kilka innych skryptów używanych do testowania Kafki lub uaktualniania funkcjonalności. Narzędzie `kafka-broker-api-versions.sh` pomaga identyfikować różne wersje użytecznych elementów API podczas aktualizacji z jednej wersji Kafki do drugiej i sprawdzać, czy nie występują problemy z kompatybilnością. Mamy skrypty do testowania wydajności producentów i konsumentów, kilka skryptów ułatwiających administrowanie ZooKeeperem, a także skrypt `trogdor.sh`, który jest frameworkiem testowym zaprojektowanym do uruchamiania benchmarków i innych obciążeń roboczych do przeprowadzania stress testów systemu.

Niebezpieczne operacje

Istnieją pewne zadania administracyjne, które są technicznie możliwe do wykonania, ale należy uciekać się do nich tylko w najbardziej ekstremalnych sytuacjach. Często diagnozujemy jakiś problem i kończą nam się opcje lub znajdujemy konkretny błąd, który musimy tymczasowo obejść. Zadania te zwykle nie są udokumentowane i nie mają wsparcia, a ponadto stwarzają pewne ryzyko dla aplikacji.

W tym podrozdziale omówimy kilka z bardziej powszechnych zadań, aby w sytuacji nadzwyczajnej była dostępna jakaś opcja przywrócenia sprawności. Ich stosowanie nie jest zalecane w normalnych operacjach klastra i należy je dokładnie rozważyć przed wykonaniem.



Niebezpieczeństwo: nisko latające smoki

Operacje opisane w tym podrozdziale często wymagają bezpośredniej ingerencji w metadane klastra przechowywane w ZooKeeperze. Mogą to być bardzo niebezpieczne działania, należy więc zachować szczególną ostrożność, aby z wyjątkiem wymienionych przypadków nie modyfikować informacji bezpośrednio w ZooKeeperze.

Przenoszenie kontrolera klastra

Każdy klaster Kafki ma jeden broker wyznaczony jako kontroler. Kontroler posiada specjalny wątek, który oprócz wykonywania normalnej pracy brokera jest odpowiedzialny za nadzorowanie operacji klastra. Zwykle wybór kontrolera odbywa się automatycznie przez monitorowanie efemerycznych węzłów ZooKeepera. Gdy kontroler zostanie wyłączony lub stanie się niedostępny, inne brokery nominują siebie możliwie najszybciej, gdyż po zamknięciu kontrolera węzeł ZooKeepera jest usuwany.

Czasami podczas rozwiązywania problemów z niewłaściwie działającym klastrem lub brokerem przydatne może być wymuszone przeniesienie kontrolera na inny broker bez wyłączania danego hosta. Przykładem jest sytuacja, gdy kontroler ucierpiał z powodu wyjątku lub innego problemu,

po którym pozostał działający, ale przestał wykonywać swoje funkcje. W takich przypadkach przenoszenie kontrolera zwykle nie wiąże się z wysokim ryzykiem, ponieważ jednak nie jest to normalne zadanie, nie powinno być wykonywane regularnie.

Aby wymusić przeniesienie kontrolera, należy ręcznie usunąć dany węzeł ZooKeepera z lokalizacji `/admin/controller`, co spowoduje rezygnację z bieżącego kontrolera, a klastr losowo wybierze nowy kontroler. Obecnie w Kafce nie ma możliwości wybrania na kontroler konkretnego brokera.

Rozwiązywanie problemów z usuwaniem tematów

Przy próbie usunięcia tematu z Kafki węzeł ZooKeepera żąda utworzenia zadania usuwania. Gdy wszystkie repliki zakończą usuwanie tematu i potwierdzą zakończenie usuwania, węzeł ZooKeepera zostanie usunięty. W normalnych okolicznościach klastr wykonuje to bardzo szybko. Czasami jednak proces ten może się nie powieść. Oto kilka scenariuszy, w których żądanie usunięcia może utknąć:

1. Operator wysyłający żądanie nie ma możliwości sprawdzenia, czy w klastrze włączone jest usuwanie tematu, i może zażądać usunięcia tematu z klastra, w którym możliwość usuwania jest wyłączona.
2. Operator wysłał żądanie usunięcia bardzo dużego tematu, ale zanim żądanie zostanie obsłużone, co najmniej jeden zbiór replik zostaje wyłączony z powodu awarii sprzętu, a usuwanie nie może zostać zakończone, ponieważ kontroler nie może potwierdzić, że usunięcie zostało zakończone pomyślnie.

Aby „odkleić” usuwanie tematów, najpierw usuń węzeł ZooKeepera z lokalizacji `/admin/delete_topic/<temat>`. Usunięcie węzłów ZooKeepera dla danego tematu (ale nie węzła nadrzędnego `/admin/delete_topic`) spowoduje usunięcie oczekujących żądań. Jeżeli usuwanie zostanie ponownie umieszczone w kolejce przez żądania buforowane w kontrolerze, konieczne może być również wymuszone przeniesienie kontrolera (o którym pisaliśmy wcześniej) natychmiast po usunięciu węzła ZooKeepera dla tematu, aby upewnić się, że żadne zbuforowane żądania nie są oczekujące w kontrolerze.

Ręczne usuwanie tematów

Jeśli masz klastr z wyłączoną opcją usuwania tematów lub potrzebujesz usunąć kilka tematów poza normalnym przepływem operacji, możliwe jest ręczne usunięcie ich z klastra. Wymaga to jednak całkowitego zamknięcia wszystkich brokerów w klastrze i nie można tego robić, gdy którykolwiek z brokerów w klastrze jest uruchomiony.



Najpierw wyłącz brokery

Modifikacja metadanych klastra w ZooKeeperze, gdy klastr jest online, jest bardzo niebezpieczną operacją i może wywołać niestabilność klastra. Nigdy nie próbuj usuwać ani modyfikować metadanych tematu w ZooKeeperze, jeśli klastr jest uruchomiony.

Aby usunąć temat z klastra, wykonaj następujące czynności:

1. Zamknij wszystkie brokery w klastrze.
2. Ze ścieżki klastra Kafki usuń ścieżkę ZooKeepera `/brokers/topics/<temat>`. Zwróć uwagę, że ten węzeł ma węzły potomne, które należy najpierw usunąć.

3. Z katalogów dziennika każdego brokera usuń katalogi partycji. Katalogi te mają nazwę `<temat>-<liczba_całkowita>`, gdzie `<liczba_całkowita>` określa identyfikator partycji.
4. Zrestartuj wszystkie brokery.

Podsumowanie

Uruchomienie klastra Kafki może być zniechęcającym przedsięwzięciem z licznymi konfiguracjami i zadaniami konserwacyjnymi, aby utrzymać szczytową wydajność systemów. W tym rozdziale omówiliśmy wiele rutynowych zadań, takich jak zarządzanie konfiguracjami tematów i klientów, które będziesz musiał często wykonywać. Opisaliśmy również bardziej ezoteryczne zadania, takie jak badanie segmentów dziennika, które będą przydatne przy rozwiązywaniu problemów. Na koniec omówiliśmy kilka operacji, które nie są bezpieczne ani rutynowe, ale mogą być wykorzystane w sytuacji podbramkowej. Wszystkie te narzędzia pomogą Ci zarządzać klastrem Kafki. Gdy zaczniesz powiększać swoje klastry Kafki, nawet korzystanie z tych narzędzi może stać się uciążliwe i trudne w zarządzaniu. Zdecydowanie zalecamy zaangażowanie się w społeczność Kafki i skorzystanie z wielu innych projektów *open source* z tego ekosystemu, aby zautomatyzować wiele zadań opisanych w tym rozdziale.

Po tym jak poznałeś narzędzia niezbędne do administrowania i zarządzania klastrem, musisz jeszcze zapewnić odpowiednie monitorowanie. W rozdziale 13. omówimy sposoby monitorowania kondycji i działania brokerów i klastrów, aby mieć pewność, że Kafka działa poprawnie (i wiedzieć, kiedy nie działa). Przedstawimy również najlepsze praktyki w zakresie monitorowania klientów, zarówno producenckich, jak i konsumenckich.

Monitorowanie Kafki

Aplikacje platformy Apache Kafka mają wiele wskaźników mierzących ich działanie — jest ich tak dużo, że łatwo się pogubić w tym, co warto obserwować, a co można odłożyć na bok. Obejmuje to proste wskaźniki dotyczące ogólnego natężenia ruchu, szczegółowe dane na temat czasu dla każdego typu żądania, a także wskaźniki dotyczące poszczególnych tematów i partycji. Wskaźniki te zapewniają szczegółowy wgląd w każdą operację w brokerze, ale mogą być również zmorą dla osoby odpowiedzialnej za zarządzanie systemem monitorowania.

W tym rozdziale omówimy najważniejsze wskaźniki, które należy monitorować przez cały czas, oraz sposób reagowania na nie. Opiszemy również kilka wskaźników, które warto mieć pod ręką podczas debugowania problemów. Nie będzie to jednak wyczerpująca lista dostępnych wskaźników, ponieważ często się ona zmienia, a wiele wskaźników będzie miało charakter informacyjny tylko dla zapalonego programisty Kafki.

Podstawy monitorowania

Zanim przejdziemy do konkretnych wskaźników dostarczanych przez brokery i klienty Kafki, omówmy podstawy monitorowania aplikacji Javy i kilka najlepszych praktyk dotyczących monitorowania i ostrzegania. Pozwoli Ci to lepiej zrozumieć, jak monitorować aplikacje i dlaczego konkretne wskaźniki opisane dalej w rozdziale zostały wybrane jako najważniejsze.

Gdzie są wskaźniki?

Dostęp do wszystkich danych udostępnianych przez Kafkę można uzyskać za pośrednictwem interfejsu JMX (ang. *Java Management Extensions*). Najprostszym sposobem wykorzystania ich w zewnętrznym systemie monitorowania jest użycie agenta gromadzenia danych dostarczonego przez wybrany system monitorowania i dołączenie go do procesu Kafki. Może to być osobny proces, który jest uruchamiany w systemie i łączy się z interfejsem JMX, jak w przypadku wtyczki `check_jmx` oprogramowania Nagios XI lub narzędzia `jmxtrans`. Możesz również uzyskiwać dostęp do danych za pośrednictwem połączenia HTTP, korzystając z agenta JMX, takiego jak Jolokia lub MX4J, który jest uruchamiany bezpośrednio w procesie Kafki.

Szczegółowe omówienie konfigurowania agentów monitorowania wykracza poza zakres tego rozdziału i dostępnych jest zbyt wiele możliwości, aby oddać sprawiedliwość im wszystkim. Jeśli Twoja

organizacja nie ma doświadczenia w monitorowaniu aplikacji Javy, warto rozważyć monitorowanie jako usługę. Wiele firm oferuje agenty monitorowania, punkty gromadzenia wskaźników, przechowywanie, generowanie wykresów i ostrzeganie w pakiecie usług. Świadczą one ponadto wsparcie dla konfigurowania wymaganych agentów monitorowania.



Wyszukiwanie portu JMX

Aby ułatwić konfigurację aplikacji takich jak systemy monitorowania, które łączą się bezpośrednio z interfejsem JMX w brokerze Kafki, broker ustawia skonfigurowany port JMX w informacjach o brokerze przechowywanych w ZooKeeperze. Węzeł ZooKeepera `/brokers/ids/<identyfikator>` zawiera dane w formacie JSON dla brokera, w tym klucze `hostname` i `jmx_port`. Należy jednak zauważyć, że zdalny JMX jest domyślnie wyłączony w Kafce ze względów bezpieczeństwa. Jeśli zamierzasz go włączyć, musisz odpowiednio skonfigurować zabezpieczenia portu. Dzieje się tak dlatego, że JMX nie tylko pozwala na wgląd w stan aplikacji, ale umożliwia także wykonywanie kodu. Zaleca się użycie agenta wskaźników JMX, który jest ładowany w aplikacji.

Wskaźniki nie pochodzące z aplikacji

Nie wszystkie wskaźniki będą pochodzić z samej Kafki. Można je pozyskiwać z pięciu ogólnych grup. W tabeli 13.1 opisaliśmy kategorie, na jakie możemy podzielić wskaźniki z monitorowania brokerów Kafki.

Tabela 13.1. Źródła wskaźników

Kategoria	Opis
Wskaźniki aplikacji	Są to wskaźniki pobierane z samej Kafki z interfejsu JMX
Dzienniki	Kolejny typ danych monitoringowych, który pochodzi z samej Kafki. Ponieważ jest to pewna forma tekstu lub danych strukturalnych, a nie sama liczba, wymaga to nieco przetwarzania
Wskaźniki infrastruktury	Te wskaźniki pochodzą z systemów, które znajdują się przed Kafką, ale nadal są w ścieżce żądania i masz nad nimi kontrolę. Przykładem takiego systemu jest moduł równoważenia obciążenia
Klienci syntetyczne	Są to dane z narzędzi zewnętrznych w stosunku do wdrożenia Kafki, takich jak klient, ale masz nad nimi bezpośrednią kontrolę i zwykle nie wykonują tej samej pracy co Twoje klienty. Do tej kategorii zalicza się zewnętrzny monitor taki jak Kafka Monitor
Wskaźniki klientów	Są to wskaźniki udostępniane przez klienty Kafki, które łączą się z Twoim klastrem

Dzienniki generowane przez Kafkę omówimy dalej w tym rozdziale, podobnie jak wskaźniki klientów. Pokróćce opiszemy także kwestię wskaźników syntetycznych. Wskaźniki infrastruktury są jednak zależne od konkretnego środowiska i wykraczają poza zakres tego rozdziału. Z każdym kolejnym etapem Twojej podróży po Kafce te źródła wskaźników będą stawały się coraz ważniejsze, byś mógł w pełni zrozumieć sposób działania swoich aplikacji, gdyż im niżej na liście znajdują się wskaźniki, tym bardziej obiektywny obraz Kafki zapewniają. Na początku wystarczy np. opieranie się na wskaźnikach z brokerów, ale później będziesz potrzebował bardziej obiektywnego obrazu ich działania. Dobrym przykładem wartości obiektywnych wskaźników jest monitorowanie kondycji strony internetowej. Załóżmy, że serwer internetowy działa poprawnie i potwierdzają to wszystkie generowane przez niego wskaźniki. Wystąpił jednak problem z siecią między serwerem WWW a zewnętrznymi użytkownikami, co oznacza, że żaden z Twoich użytkowników nie może skomunikować się

z tym serwerem. Może to wykryć (i zaalarmować Cię o tym) klient syntetyczny działający poza siecią i sprawdzający dostępność strony internetowej.

Jakich wskaźników potrzebujesz?

Jakie konkretne wskaźniki są dla Ciebie istotne? To pytanie jest prawie tak samo podchwytliwe, jak kwestia wyboru najlepszego edytora. Odpowiedź zależy w znacznym stopniu od tego, co zamierzasz z nimi robić, jakie masz narzędzia do gromadzenia danych, w jak bardzo zaawansowany sposób korzystasz z Kafki oraz ile masz czasu na budowanie infrastruktury wokół Kafki. Programista wewnętrznych mechanizmów działania brokera będzie miał zupełnie inne potrzeby niż inżynier SRE uruchamiający wdrożenie Kafki.

Ostrzeganie czy debugowanie?

Najpierw powinienś zadać sobie pytanie, czy Twoim głównym celem jest ostrzeganie o pojawiających się problemach z Kafką, czy debugowanie tych problemów. Zazwyczaj chodzi po trochu o jedno i drugie, ale podzielenie wskaźników na te dwie grupy pozwala traktować je inaczej podczas ich gromadzenia.

Wskaźnik przeznaczony do ostrzegania jest przydatny przez bardzo krótki czas — zazwyczaj niewiele dłużej niż zajmuje rozwiązanie problemu. Można go liczyć w godzinach lub dniach. Wskaźniki te będą konsumowane przez automatyzację reagującą na znane Ci problemy, a także przez operatorów ludzkich w przypadkach, gdy automatyzacja nie będzie jeszcze istniała. Zazwyczaj powinny to być wskaźniki bardziej obiektywne, ponieważ problem, który nie ma wpływu na klientów, jest znacznie mniej istotny niż ten, który taki wpływ ma.

Dane niezbędne do debugowania mają dłuższy horyzont czasowy, gdyż często diagnozujesz problemy, które istnieją już od jakiegoś czasu, lub przyglądasz się bliżej bardziej złożonemu problemowi. Dane te będą musiały pozostać dostępne przez dni lub tygodnie po ich zebraniu. Zwykle będą to ponadto pomiary bardziej subiektywne, czyli dane z samej aplikacji Kafki. Należy pamiętać, że nie zawsze konieczne jest gromadzenie tych danych w systemie monitorowania. Jeśli wskaźniki będą używane do debugowania istniejących problemów, wystarczy, że będą dostępne w razie potrzeby. Nie musisz przytłaczać systemu monitorowania gromadzeniem na bieżąco dziesiątek tysięcy wartości.



Dane historyczne

Istnieje jeszcze trzeci rodzaj danych, których będziesz w końcu potrzebował, a są to dane historyczne aplikacji. Najczęstszym zastosowaniem danych historycznych jest zarządzanie pojemnością, dlatego zawierają one informacje o wykorzystywanych zasobach, w tym zasobach obliczeniowych, pamięci masowej i sieci. Wskaźniki te będą musiały być przechowywane przez bardzo długi czas, mierzony w latach. Konieczne może być również zebranie dodatkowych metadanych, aby umieścić dane w odpowiednim kontekście, np. dodawania brokerów do klastra lub usuwania brokerów z klastra.

Automatyzacja czy ludzie?

Kolejna kwestia to określenie konsumenta wskaźników. Jeżeli wskaźniki są konsumowane przez automatyzację, powinny być bardzo konkretne. Nie ma nic złego w dużej liczbie wskaźników, z których każdy opisuje szczegółowo niewielki wycinek, ponieważ po to są komputery: by przetwarzać duże ilości danych. Im bardziej szczegółowe są dane, tym łatwiej tworzyć działającą na ich podstawie automatyzację, ponieważ takie dane nie pozostawiają tyle miejsca na interpretację, co na określanie ich znaczenia. Natomiast jeśli wskaźniki będą przetwarzane przez ludzi, przedstawienie dużej liczby wskaźników będzie przytłaczające. Staje się to jeszcze ważniejsze podczas definiowania alertów opartych na tych pomiarach. Bardzo łatwo jest ulec „zmęczeniu alarmowemu”, gdy pojawia się tak wiele alarmów, że ciężko stwierdzić, jak poważny jest problem. Trudno jest również właściwie zdefiniować progi dla poszczególnych wskaźników i aktualizować je na bieżąco. Gdy alerty są przytłaczające lub często niepoprawne, przestajemy ufać, że prawidłowo opisują stan aplikacji.

Analogią może być działanie samochodu. Aby prawidłowo dostosować stosunek powietrza do paliwa podczas jazdy, komputer potrzebuje licznych pomiarów gęstości powietrza, paliwa, spalin i innych drobiazgów dotyczących pracy silnika. Pomiary te byłyby jednak przytłaczające dla kierowcy pojazdu. Zamiast nich mamy lampkę „Sprawdź silnik”. Ten pojedynczy wskaźnik informuje, że istnieje jakiś problem i można uzyskać więcej szczegółowych informacji, aby dokładnie określić, na czym ten problem polega. W tym rozdziale podpowiemy, które wskaźniki zapewnią największą ilość informacji, aby ostrzeżenia mogły być proste.

Kontrolowanie kondycji aplikacji

Bez względu na to, w jaki sposób będziesz gromadzić wskaźniki z Kafki, powinieneś monitorować również ogólny stan procesu aplikacji za pomocą prostej kontroli kondycji. Dostępne są dwie metody:

- zewnętrzny proces, który informuje, czy broker jest uruchomiony (kontrola kondycji);
- wysyłanie alertów w przypadku braku wskaźników z brokera Kafki (tzw. **wskaźniki przestarzałe**; ang. *stale metrics*).

Chociaż ta druga metoda działa, może utrudniać rozróżnienie między awarią brokera Kafki a awarią samego systemu monitorowania.

W celu skontrolowania kondycji brokera Kafki wystarczy połączyć się z portem zewnętrznym (tym samym, którego klienci używają do łączenia się z brokerem), aby sprawdzić, czy odpowiada. Jeśli chodzi o aplikacje klienckie, może to być bardziej złożone: od prostego sprawdzenia, czy proces jest uruchomiony, po wewnętrzną metodę, która określa kondycję aplikacji.

SLO

Jednym z obszarów monitorowania, który ma szczególne znaczenie dla usług infrastrukturalnych, takich jak Kafka, są zakładane poziomy świadczenia usług, czyli SLO (ang. *Service-Level Objective*). W ten sposób komunikujemy naszym klientom, jakiego poziomu usług mogą oczekiwać od usługi infrastrukturalnej. Klienci chcą traktować usługi takie jak Kafka jako jednorodny system: nie chcą

i nie muszą rozumieć wewnętrznych mechanizmów jego działania. Wystarczy im znajomość interfejsu, z którego korzystają.

Definicje dotyczące poziomu usług

Zanim przejdziemy do omawiania SLO w Kafce, należy uzgodnić stosowaną terminologię. Często można spotkać się z nieprawidłowym używaniem terminów dotyczących przestrzeni „poziomu usług”, co prowadzi do nieporozumień w kwestii przedmiotu rozmowy.

Wskaźnik poziomu usług (ang. *Service-Level Indicator* — SLI) opisuje jeden z aspektów niezawodności usługi. Powinien iść w parze z doświadczeniem zbieranym przez klienta, więc im bardziej obiektywne będą pomiary, tym lepiej. W systemie przetwarzania żądań takim jak Kafka zwykle najlepiej jest wyrazić te pomiary jako stosunek liczby dobrych zdarzeń do całkowitej liczby zdarzeń, np. odsetek wysyłanych do serwera WWW żądań, które zwracają odpowiedź 2xx, 3xx lub 4xx.

Zakładany poziom świadczenia usług (SLO), który można nazwać również **progiem poziomu usług** (ang. *Service-Level Threshold* — SLT), łączy SLI z wartością docelową. Powszechnym sposobem wyrażenia wartości docelowej jest liczba dziesiętnych (np. 99,9% to „trzy dziesiątki”), choć w żadnym wypadku nie jest to wymagane. SLO powinien uwzględniać ponadto przedział czasu, do którego jest stosowany, często określany liczbą dni, np. 99% żądań wysyłanych do serwera w ciągu 7 dni musi zwracać odpowiedź 2xx, 3xx lub 4xx.

Umowa o gwarantowanym poziomie usług (ang. *Service-Level Agreement* — SLA) to umowa między usługodawcą a klientem. Zwykle obejmuje ona kilka SLO, a także szczegółowe informacje na temat sposobu ich pomiaru i raportowania, wsparcia zapewnianego klientowi przez dostawcę usług oraz kar nakładanych na dostawcę usług, jeśli nie dopełni warunków SLA. Umowa SLA dla zakładanego poziomu świadczenia usług może np. stanowić, że za okres, w którym usługa nie zapewniała SLO, dostawca usług zwróci wszystkie opłaty uiszczone przez klienta.



Umowa o gwarantowanym poziomie operacyjnym

Termin **umowa o gwarantowanym poziomie operacyjnym** (ang. *Operational-Level Agreement* — OLA) jest rzadziej stosowany. Opisuje on umowy między wieloma wewnętrznymi usługami lub dostawcami wsparcia w zakresie ogólnej realizacji umowy SLA. Ma to na celu prawidłowe opisanie i uwzględnienie w codziennych operacjach wielu działań, które są niezbędne do spełnienia wymagań SLA.

Bardzo często słyszy się, że ludzie mówią o umowach SLA, kiedy tak naprawdę chodzi o SLO. O ile dostawcy świadczący usługi na rzecz klientów uiszczających opłaty mogą mieć z tymi klientami umowy SLA, o tyle rzadko zdarza się, żeby inżynierowie uruchamiający aplikacje byli odpowiedzialni za coś więcej niż wydajność tej usługi w ramach SLO. Ponadto ci, którzy mają tylko klientów wewnętrznych (tj. uruchamiają Kafkę jako wewnętrzną infrastrukturę danych dla znacznie większej usługi), na ogół nie mają z nimi umów SLA. Nie powinno to jednak odwrócić Cię od ustalania i obwieszczania różnych SLO, ponieważ dzięki temu klienci będą przyjmowali mniej założeń co do tego, jak ich zdaniem powinna działać Kafka.

Jakie wskaźniki nadają się na SLI?

Generalnie wskaźniki dla SLI powinny być zbierane przy użyciu czegoś zewnętrznego w stosunku do brokerów Kafki, gdyż SLO powinny opisywać, czy typowy użytkownik usługi jest zadowolony, a nie można mierzyć tego subiektywnie. Dla klientów nie ma znaczenia, że według Ciebie usługa działa prawidłowo; liczy się ich (zagregowane) doświadczenie. Oznacza to, że prawdopodobnie dla większości SLI nadawać się będą wskaźniki infrastruktury, klienty syntetyczne, a prawdopodobnie najlepsze będą wskaźniki po stronie klienta.

Chociaż w żadnym wypadku nie jest to wyczerpująca lista, w tabeli 13.2 przedstawiliśmy najczęstsze wskaźniki SLI stosowane w systemach żądanie-odpowiedź i systemach przechowywania danych.

Tabela 13.2. Rodzaje wskaźników SLI

Dostępność	Czy klient jest w stanie wykonać żądanie i otrzymać odpowiedź?
Opóźnienie	Jak szybko zwracana jest odpowiedź?
Jakość	Czy odpowiedź zawiera poprawne dane?
Bezpieczeństwo	Czy żądanie i odpowiedź są właściwie chronione, np. za pomocą autoryzacji lub szyfrowania?
Przepustowość	Czy klient może uzyskać wystarczającą ilość danych z odpowiednią prędkością?



Klienci zawsze chcą więcej

Na niektóre SLO interesujące i ważne dla klientów nie masz wpływu. Klienci mogą być np. zaniepokojeni poprawnością lub świeżością danych produkowanych do Kafki. Nie zgadzaj się na gwarantowanie SLO, za które nie jesteś odpowiedzialny, ponieważ dołożysz sobie jedynie pracy niezwiązanej z podstawowymi zadaniami utrzymywania prawidłowego działania Kafki. Kieruj tych klientów do osób, które są odpowiedzialne za ustalanie, objaśnianie i zawieranie porozumień związanych z tymi dodatkowymi wymaganiami.

Pamiętaj, aby Twoje SLI były oparte na licznikach zdarzeń, które mieszczą się w progach SLO. Oznacza to, że najlepiej, gdyby każde zdarzenie było indywidualnie sprawdzane pod kątem tego, czy mieści się w progu SLO. Wyklucza to wskaźniki kwantylowe jako dobre wskaźniki SLI, ponieważ dowiesz się z nich jedynie, że 90% zdarzeń było poniżej danej wartości, i nie będziesz mógł kontrolować, jaka jest ta wartość. Podczas pracy z SLO przydatne może być jednak agregowanie wartości do koszyków („mniej niż 10 ms”, „10 – 50 ms”, „50 – 100 ms” itp.), zwłaszcza gdy nie jesteś jeszcze pewien, jaki próg jest dobry. Daje to wgląd w rozkład zdarzeń w zakresie SLO i pozwala skonfigurować koszyki w taki sposób, by granice były rozsądnymi wartościami dla określonego progu SLO.

Stosowanie SLO w ostrzeganiu

Krótko mówiąc: na SLO powinny opierać się główne alerty, gdyż SLO opisują problemy z punktu widzenia klientów, a o to powinienes zadbać w pierwszej kolejności. Ogólnie rzecz biorąc, jeśli problem nie dotyczy Twoich klientów, nie musisz z tego powodu zrywać się w nocy. SLO wskażą Ci także problemy, których nie potrafisz wykrywać, ponieważ nigdy wcześniej się z nimi nie spotkałeś. Nie pokażą Ci, co to za problemy, ale powiedzą Ci, że one istnieją.

Wyzwanie polega na tym, że bardzo trudno jest używać SLO bezpośrednio jako alertów. SLO najlepiej sprawdzają się w dłuższym okresie, takim jak tydzień, gdyż potrzebujemy możliwości zgłoszenia problemów kierownictwu i klientom w przyswajalny sposób. Ponadto kiedy alerty SLO zostaną uruchomione, będzie za późno — i tak będziesz już przekraczać progi SLO. Niektórzy używają wartości pochodnych, aby zapewnić wczesne ostrzeżenie, ale najlepszym sposobem podejścia wykorzystującego SLO do alarmowania jest obserwowanie tempa, z jakim przekraczany jest próg SLO w jego ramach czasowych.

Załóżmy, że Twój klaster Kafki otrzymuje milion żądań tygodniowo i masz zdefiniowane SLO, które stwierdza, że 99,9% żądań musi wysłać pierwszy bajt odpowiedzi w ciągu 10 milisekund. Oznacza to, że w ciągu tygodnia możesz mieć do tysiąca żądań, które odpowiedzą wolniej, a i tak spełnisz wymagania. Zwykle obserwujesz jedno takie żądanie na godzinę, czyli ok. 168 złych żądań tygodniowo, od niedzieli do soboty. Masz wskaźnik, który pokazuje to jako współczynnik przekraczania progu SLO, a jedno żądanie na godzinę na milion żądań tygodniowo daje tempo 0,1% na godzinę.

We wtorek o 10:00 Twój wskaźnik się zmienia i pokazuje, że tempo przekraczania progu wynosi 0,4% na godzinę. Nie jest to dobra wiadomość, ale nadal nie stanowi problemu, ponieważ na koniec tygodnia będziesz miał wynik mieszczący się w progu SLO. Otwierasz bilet, by przyrzeć się problemowi, ale wracasz do zadań o wyższym priorytecie. W środę o 14:00 współczynnik przekraczania progu wzrasta do 2% na godzinę i uruchomione zostają alerty. Wiesz, że w tym tempie do piątkowego popołudnia przekroczysz próg SLO. Rzucasz wszystko, diagnozujesz problem, a po ok. 4 godzinach współczynnik wraca do 0,4% na godzinę i pozostaje na tym poziomie przez resztę tygodnia. Dzięki zastosowaniu współczynnika przekraczania progu udało Ci się w tym tygodniu uniknąć niedotrzymania SLO.

Doskonałym źródłem informacji na temat wykorzystania SLO i współczynnika przekraczania progu do ustawiania alertów są pozycje *Site Reliability Engineering. Jak Google zarządza systemami produkcyjnymi* (Helion) i *The Site Reliability Workbook* (O'Reilly) autorstwa Betsy Beyer i in.

Wskaźniki brokerów Kafki

Brokery Kafki dostarczają różnych wskaźników. Wiele z nich to pomiary niskiego poziomu dodawane przez programistów podczas badania konkretnego problemu lub gromadzenia informacji do celów debugowania w późniejszym czasie. Dostępne są wskaźniki dostarczające informacji o prawie każdej funkcji w brokerze, ale te powszechnie stosowane dostarczają informacji potrzebnych do codziennego uruchamiania Kafki.



Kto obserwuje obserwatory?

Wiele organizacji używa Kafki do gromadzenia wskaźników aplikacji, wskaźników systemowych i dzienników, które są przetwarzane przez centralny system monitorowania. Jest to doskonały sposób na oddzielenie aplikacji od systemu monitorowania, ale stanowi szczególnie problem dla samej Kafki. Jeśli używasz tego samego systemu również do monitorowania Kafki, najprawdopodobniej nigdy się nie dowiesz, jeśli Kafka nie będzie działać prawidłowo, ponieważ przepływ danych do systemu monitorowania również zostanie uszkodzony.

Można temu zaradzić na wiele sposobów. Jednym z nich jest zastosowanie dla Kafki oddzielnego systemu monitorowania, który nie jest zależny od Kafki. Jeżeli masz wiele centrów danych, innym sposobem jest upewnienie się, że wskaźniki dla klastra Kafki w centrum danych A są produkowane do centrum danych B i odwrotnie. Niezależnie od tego, jak rozwiążesz ten problem, musisz zagwarantować, by monitorowanie i alarmowanie dla Kafki nie zależały od jej działania.

Ten podrozdział rozpoczniemy od omówienia ogólnego przepływu pracy dla diagnozowania problemów z klastrami Kafki, odwołując się do przydatnych wskaźników. Te i inne wskaźniki opisujemy szerzej dalej w tym rozdziale. W żadnym wypadku nie będzie to wyczerpująca lista wskaźników brokera, ale raczej kilka „obowiązkowych” wskaźników do sprawdzania kondycji brokera i klastra. Zanim przejdziemy do wskaźników klientów, na koniec zajmiemy się jeszcze rejestrowaniem.

Diagnozowanie problemów z klastrami

Jeśli chodzi o problemy z klastrami Kafki, mamy trzy główne kategorie:

- problemy z pojedynczym brokerem;
- przeciążone klastry;
- problemy z kontrolerem.

Problemy z poszczególnymi brokerami są zdecydowanie najłatwiejsze do zdiagnozowania i usunięcia. Wskazują na nie odstające wartości we wskaźnikach klastra i są często związane z powolnymi lub wadliwie działającymi urządzeniami pamięci masowej lub ograniczeniami obliczeniowymi innych aplikacji w systemie. Aby je wykrywać, należy monitorować dostępność poszczególnych serwerów oraz stan urządzeń pamięci masowej, korzystając ze wskaźników systemu operacyjnego (ang. *Operating System* — OS).

Jeśli jednak problem nie został zidentyfikowany na poziomie systemu operacyjnego lub sprzętu, przyczyną jest prawie zawsze brak równowagi w obciążeniu klastra Kafki. Chociaż Kafka próbuje utrzymać równomierne rozłożenie danych w klastrze na wszystkie brokery, nie oznacza to równomiernego rozłożenia dostępu klientów do tych danych. To monitorowanie nie zapewnia również wykrywania takich problemów jak gorące partycje. Aby utrzymywać równowagę klastra, zdecydowanie zaleca się korzystanie z zewnętrznego narzędzia. Jednym z takich narzędzi jest Cruise Control (<https://oreil.ly/rLybu>), który stale monitoruje klaster i równoważy obciążenie jego partycji. Zapewnia również wiele innych funkcji administracyjnych, takich jak dodawanie i usuwanie brokerów.



Wybór repliki preferowanej

Przed przystąpieniem do dalszego diagnozowania problemu należy się upewnić, że ostatnio przeprowadzony został wybór repliki preferowanej (zob. rozdział 12.). Brokery Kafki nie przywracają automatycznie przywództwa partycji (chyba że włączona jest funkcja automatycznego rebalansowania liderów) po uprzednim oddaniu przywództwa (np. gdy broker uległ awarii lub został zamknięty). Oznacza to, że równowaga replik głównych w klastrze może zostać bardzo łatwo zaburzona. Wybór repliki preferowanej jest bezpieczny i prosty do przeprowadzenia, dobrze jest więc najpierw to zrobić i sprawdzić, czy problem nie zniknie.

Przeciążone klastry to kolejny łatwy do wykrycia problem. Jeśli klastrer jest zrównoważony, a wiele brokerów wykazuje zwiększone opóźnienie dla żądań lub niski współczynnik bezczynności puli dla procedur obsługi żądań, oznacza to, że w celu obsłużenia ruchu dla klastra osiągnięte są limity brokerów. Przy dokładniejszej inspekcji może się okazać, że jakiś klient zmienił wzorzec żądań i obecnie powoduje problemy. Jednak nawet wtedy niewiele można zrobić w sprawie zmiany klienta. Dostępne rozwiązania to zmniejszenie obciążenia klastra albo zwiększenie liczby brokerów.

Problemy z kontrolerem w klastrze Kafki są znacznie trudniejsze do zdiagnozowania i często należą do kategorii błędów w samej Kafce. Problemy te objawiają się jako niezynchronizowane meta-dane brokerów, wyłączone repliki przy poprawnie działających brokerach i błędy podczas wykonywania zadań związanych z kontrolowaniem tematów, takich jak ich tworzenie. Jeśli zachodzisz w głowę, próbując zrozumieć, na czym polega problem w klastrze, i powtarzasz pod nosem: „To naprawdę dziwne”, istnieje duże prawdopodobieństwo, że kontroler zrobił coś nieprzewidywalnego i złego. Nie ma zbyt wielu sposobów na monitorowanie kontrolera, ale monitorowanie aktywnej liczby kontrolerów oraz rozmiarów ich kolejek zapewnia ogólny wskaźnik ewentualnych problemów.

Sztuka posługiwania się wskaźnikiem URP

Jednym z najpopularniejszych wskaźników podczas monitorowania Kafki są partycje z niewystarczającą liczbą replik zsynchronizowanych (ang. *Under-Replicated Partition* — URP). Pomiar ten jest zapewniany na każdym brokerze w klastrze i wskazuje liczbę jego partycji z replikami głównymi, które mają za mało replik zsynchronizowanych. Ten pojedynczy wskaźnik zapewnia wgląd w wiele problemów z klastrami Kafki, od niedziałającego brokera aż do wyczerpania zasobów. Z uwagi na szeroką gamę problemów, które może sygnalizować ten wskaźnik, warto zastanowić się, jak zareagować na wartość inną niż zero. Wiele wskaźników używanych w diagnozowaniu tego typu problemów opisujemy dalej w tym rozdziale. Dodatkowe informacje na temat wskaźnika partycji URP przedstawiliśmy w tabeli 13.3.

Tabela 13.3. Wskaźnik partycji URP

Nazwa wskaźnika	Partycje URP
JMX MBean	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
Zakres wartości	Nieujemna liczba całkowita



Pułapka alarmów URP

W poprzednim wydaniu tej książki i w wielu prelekcjach konferencyjnych autorzy szeroko rozwozdzili się nad tym, że wskaźnik partycji URP powinien być podstawowym wskaźnikiem ostrzegania ze względu na to, jak wiele problemów opisuje. Podejście to wiąże się ze znaczną liczbą problemów, z których nie mniej ważny jest fakt, że wskaźnik URP może często być niezerowy z niezbyt poważnych powodów. Oznacza to, że jako operator klastra Kafki będziesz otrzymywać fałszywe alerty, które będą prowadziły do ignorowania alertów. Ponadto zrozumienie informacji przekazywanych przez ten wskaźnik wymaga szerokiej wiedzy. Z tego powodu nie zalecamy już używania URP do celów ostrzegania. Zamiast tego do wykrywania nieznanymi problemów powinieneś wykorzystywać alerty oparte na SLO.

Stała (niezmieniająca się) liczba partycji URP zgłaszanych przez wiele brokerów w klastrze zwykle wskazuje, że jeden z brokerów w klastrze jest offline. Liczba partycji URP w całym klastrze będzie równa liczbie partycji, które są przypisane do tego brokera, a niedziałający broker nie będzie wysyłał wskaźników. W takim przypadku będziesz musiał zbadać, co się stało z tym brokerem, i rozwiązać tę sytuację. Często jest to awaria sprzętu, ale przyczyną może być również problem z systemem operacyjnym lub z Javą.

Jeśli liczba partycji URP jest zmienna albo jest stała, ale nie ma brokerów w trybie offline, zazwyczaj oznacza to problem z wydajnością w klastrze. Tego typu problemy są znacznie trudniejsze do zdiagnozowania ze względu na ich różnorodność, ale można wykonać kilka czynności, by zawęzić je do najbardziej prawdopodobnych przyczyn. Pierwszym krokiem jest próba ustalenia, czy problem dotyczy pojedynczego brokera, czy całego klastra. Czasami odpowiedź na to pytanie może być trudna. Jeżeli partycje URP znajdują się w pojedynczym brokerze, jak w poniższym przykładzie, wtedy problemem jest zazwyczaj ten broker. Błąd pokazuje, że inne brokery mają problem z replikacją komunikatów z tego brokera.

Jeżeli kilka brokerów ma partycje URP, może to być zarówno problem klastra, jak i pojedynczego brokera. W tym drugim przypadku byłoby to spowodowane tym, że pojedynczy broker ma problemy z replikacją jakichkolwiek komunikatów, i trzeba by się dowiedzieć, który to broker. Jednym ze sposobów jest uzyskanie listy partycji URP dla klastra i sprawdzenie, czy jakiś konkretny broker nie jest dla nich wspólny. Aby wyszukać wspólny wątek, można pobrać listę partycji URP za pomocą narzędzia `kafka-topics.sh` (które omówiliśmy szczegółowo w rozdziale 12.).

Przykładowa lista partycji URP w klastrze:

```
# kafka-topics.sh --bootstrap-server kafka1.example.com:9092/kafka-cluster --describe --
under-replicated
Topic: topicOne Partition: 5 Leader: 1 Replicas: 1,2 Isr: 1
Topic: topicOne Partition: 6 Leader: 3 Replicas: 2,3 Isr: 3
Topic: topicTwo Partition: 3 Leader: 4 Replicas: 2,4 Isr: 4
Topic: topicTwo Partition: 7 Leader: 5 Replicas: 5,2 Isr: 5
Topic: topicSix Partition: 1 Leader: 3 Replicas: 2,3 Isr: 3
Topic: topicSix Partition: 2 Leader: 1 Replicas: 1,2 Isr: 1
Topic: topicSix Partition: 5 Leader: 6 Replicas: 2,6 Isr: 6
Topic: topicSix Partition: 7 Leader: 7 Replicas: 7,2 Isr: 7
Topic: topicNine Partition: 1 Leader: 1 Replicas: 1,2 Isr: 1
Topic: topicNine Partition: 3 Leader: 3 Replicas: 2,3 Isr: 3
Topic: topicNine Partition: 4 Leader: 3 Replicas: 3,2 Isr: 3
Topic: topicNine Partition: 7 Leader: 3 Replicas: 2,3 Isr: 3
Topic: topicNine Partition: 0 Leader: 3 Replicas: 2,3 Isr: 3
Topic: topicNine Partition: 5 Leader: 6 Replicas: 6,2 Isr: 6
#
```

W tym przykładzie wspólnym brokerem jest numer 2. Oznacza to, że ten broker ma problem z replikacją komunikatów i w związku z tym nasze dochodzenie skoncentruje się na tym jednym brokerze. Jeżeli nie ma wspólnego brokera, prawdopodobnie występuje problem obejmujący cały klaster.

Problemy na poziomie klastra

Problemy klastrów zazwyczaj dzielą się na dwie kategorie:

- nierównoważone obciążenie;
- wyczerpanie zasobów.

Pierwszy problem, czyli nierówne rozłożenie partycji lub przywództwa, jest najłatwiejszy do wykrycia, ale jego usunięcie może być złożonym procesem. Aby zdiagnozować ten problem, potrzebnych jest kilka wskaźników od brokerów w klastrze. Oto one:

- liczba partycji;
- liczba partycji głównych;
- prędkość przysyłania komunikatów do wszystkich tematów;
- prędkość przysyłania bajtów do wszystkich tematów;
- prędkość wysyłania bajtów ze wszystkich tematów.

Zapoznaj się z tymi wskaźnikami. W idealnie zrównoważonym klastrze te liczby będą równe dla wszystkich brokerów w klastrze, jak pokazaliśmy w tabeli 13.4.

Tabela 13.4. Wskaźniki przepływności

Broker	Liczba partycji	Liczba liderów	Komunikaty przychodzące	Bajty przychodzące	Bajty wychodzące
1	100	50	13 130 kom/s	3,56 MB/s	9,45 MB/s
2	101	49	12 842 kom/s	3,66 MB/s	9,25 MB/s
3	100	50	13 086 kom/s	3,23 MB/s	9,82 MB/s

Oznacza to, że wszystkie brokery przyjmują mniej więcej taką samą ilość ruchu. Przy założeniu, że uruchomione zostały już wybory repliki preferowanej, duże odchylenia wskazywałyby, że ruch w obrębie klastra nie jest zrównoważony. Aby rozwiązać ten problem, trzeba przenieść partycje z mocno obciążonych brokerów na mniej obciążone brokery. Robi się to za pomocą narzędzia `kafka-reassign-partitions.sh`, które opisaliśmy w rozdziale 12.



Narzędzia równoważenia klastrów

Sam broker Kafki nie zapewnia automatycznego ponownego przypisywania partycji w klastrze. Oznacza to, że równoważenie ruchu w klastrze Kafki może być ogłupiającym procesem, polegającym na ręcznym przeglądaniu długich list wskaźników w celu znalezienia odpowiedniego przypisania repliki. Niektóre organizacje opracowały zautomatyzowane narzędzia, aby ułatwić wykonywanie tego zadania. Jednym z przykładów jest narzędzie `kafka-assigner`, które LinkedIn opublikował w repozytorium otwartoźródłowym `kafka-tools` na GitHubie (<https://oreil.ly/8ilPw>). Tę funkcjonalność zapewniają także niektóre oferty komercyjnego wsparcia dla Kafki.

Innym częstym problemem związanym z wydajnością klastra jest przekroczenie zdolności brokerów do obsługi żądań. Istnieje wiele możliwych wąskich gardeł, które mogą spowolnić ten proces, m.in.: CPU, operacje we-wy dysku i przepustowość sieci. Nie należy do nich wykorzystanie dysku, ponieważ brokery będą działać poprawnie aż do momentu zapełnienia dysku, a następnie ten dysk ulegnie nagłej awarii. Do diagnozowania problemów z wydajnością dostępnych jest wiele wskaźników, które można śledzić na poziomie systemu operacyjnego:

- wykorzystanie procesora;
- przepustowość sieci dla ruchu przychodzącego;
- przepustowość sieci dla ruchu wychodzącego;
- średni czas oczekiwania dysku;
- procentowe wykorzystanie dysku.

Wyczerpanie któregoś z tych zasobów zwykle objawia się jako ten sam problem: partycje URP. Należy pamiętać, że proces replikacji brokera działa dokładnie tak samo jak klienty Kafki. Jeżeli klastr ma problemy z replikacją, klienty również mają problemy z produkowaniem i konsumowaniem komunikatów. Rozsądne jest opracowanie punktu odniesienia dla tych wskaźników, gdy klastr działa prawidłowo, a następnie ustalenie progów, które wskazują na rozwijający się problem na długo przed wyczerpaniem się pojemności zasobów. Należy ponadto weryfikować trend dla tych wskaźników, ponieważ z upływem czasu zwiększa się ruch do klastra. Jeśli chodzi o wskaźniki brokera Kafki, dobrą wskazówką odnośnie do użycia klastra jest prędkość przysyłania bajtów do wszystkich tematów.

Problemy na poziomie hosta

Jeżeli problem z wydajnością Kafki nie występuje w całym klastrze i może być wyizolowany do jednego brokera lub dwóch, należy zbadać ten serwer i zobaczyć, co odróżnia go od reszty klastra. Tego typu problemy dzielą się na kilka ogólnych kategorii:

- awarie sprzętu;
- infrastruktura i konfiguracja sieciowa;
- konflikty z innymi procesami;
- lokalne różnice w konfiguracji.



Typowe serwery i problemy

Serwer i jego system operacyjny to złożona maszyna z tysiącami komponentów, z których każdy może mieć problemy i spowodować całkowitą awarię lub po prostu pogorszenie wydajności. Nie jesteśmy w stanie opisać w tej książce wszystkiego, co może zawieść — na ten temat zostało napisanych wiele pozycji i będą powstawać kolejne. Możemy jednak omówić niektóre z najczęściej spotykanych problemów. W tym podpunkcie skupimy się na problemach z typowym serwerem z systemem operacyjnym Linux.

Awarie sprzętu są czasami oczywiste, np. gdy serwer po prostu przestaje działać, ale problemy z wydajnością mogą być powodowane przez te mniej oczywiste kwestie. Są to zazwyczaj tzw. miękkie awarie, które pozwalają systemowi kontynuować pracę, ale pogarszają działanie. Może to być uszkodzony bit pamięci, w którym system wykrył problem i pominął ten segment (zmniejszając ilość całkowitej dostępnej pamięci). To samo może się zdarzyć w razie awarii procesora. W przypadku takich problemów należy korzystać z narzędzi oferowanych przez dostawców sprzętu, takich jak inteligentny interfejs zarządzania platformą (ang. *Intelligent Platform Management Interface* — IPMI), który służy do monitorowania kondycji sprzętu. Gdy występuje jakiś problem, można

skorzystać z polecenia `dmesg` i przyjrzeć się buforowi pierścieniowemu jądra, aby zobaczyć komunikaty dziennika, które są wrzucane do konsoli systemowej.

Najczęstszym rodzajem awarii sprzętu, która prowadzi do pogorszenia wydajności w Kafce, jest awaria dysku. Platforma Apache Kafka wykorzystuje dysk do utrwalania komunikatów, a wydajność producenta jest bezpośrednio związana z tym, jak szybko dyski wykonują te zapisy. Wszelkie odchylenia w tym zakresie będą się objawiać jako problemy z wydajnością producentów i wątków pobierania replik. To ostatnie prowadzi do powstawania partycji URP. Dlatego należy przez cały czas monitorować kondycję dysków i szybko rozwiązywać wszelkie problemy.



Jedno zepsute jajko

Awaria pojedynczego dysku na pojedynczym brokerze może zniszczyć wydajność całego klastra. Dzieje się tak dlatego, że klienci produkcyjne łączą się ze wszystkimi brokerami, które mają liderów (partycje główne) dla danego tematu, a jeśli postępujesz zgodnie z najlepszymi praktykami, partycje te będą równomiernie rozłożone po całym klastrze. Jeżeli jeden broker zmniejszy wydajność i będzie spowalniał żądania produkowania, spowoduje to uruchomienie mechanizmu ciśnienia zwrotnego w producentach, spowalniając żądania do wszystkich brokerów.

Na początek upewnij się, że monitorujesz informacje o stanie dysków z IPMI lub z interfejsu dostarczonego przez producenta Twojego sprzętu. Ponadto w ramach systemu operacyjnego powinien regularnie uruchamiać narzędzia SMART (ang. *Self-Monitoring, Analysis and Reporting Technology*) do monitorowania i testowania dysków. Dzięki temu będziesz ostrzegany o zbliżającej się awarii. Ważne jest również obserwowanie kontrolera dysku, zwłaszcza jeśli ma on funkcjonalność RAID, niezależnie od tego, czy używasz sprzętowego RAID-u. Wiele kontrolerów ma wbudowaną pamięć podręczną, która jest używana tylko wtedy, gdy kontroler jest sprawny i działa moduł podtrzymania baterijnego (ang. *Battery Backup Unit* — BBU). Awaria BBU może spowodować wyłączenie pamięci podręcznej, co pogorszy wydajność dysku.

Infrastruktura i konfiguracja sieciowa to kolejny obszar, w którym częściowe awarie będą powodowały problemy. Niektóre z tych problemów to problemy sprzętowe, takie jak uszkodzony kabel sieciowy lub niedziałające złącze. Inne to problemy z konfiguracją, zwykle ze zmianą prędkości lub ustawień duplexu połączenia po stronie serwera lub upstreamowego sprzętu sieciowego. Problemy z konfiguracją sieci mogą być również problemami z systemem operacyjnym, np. za małe bufor sieciowe lub zbyt wiele połączeń sieciowych zajmujących zbyt dużo całkowitej pamięci. Jednym z kluczowych wskaźników występowania problemów w tym obszarze będzie liczba błędów wykrywanych na interfejsach sieciowych. Jeśli liczba błędów rośnie, prawdopodobnie istnieje jakiś nierozwiązany problem.

Jeżeli nie ma problemów sprzętowych, kolejnym częstym problemem jest inna aplikacja działająca w systemie, która zużywa zasoby i wywiera presję na broker Kafki. Może to być coś zainstalowanego przez pomyłkę lub proces, który powinien być uruchomiony, np. agent monitorowania, ale nie działa poprawnie. Aby określić, czy istnieje proces, który wykorzystuje więcej zasobów procesora lub pamięci, niż zakładano, użyj narzędzi dostępnych w systemie, takich jak `top`.

Jeśli inne opcje zostały wyczerpane i nie znalazłeś jeszcze źródła rozbieżności na hoście, prawdopodobnie wkradła się różnica konfiguracji w brokerze albo w samym systemie. Biorąc pod uwagę liczbę aplikacji działających na dowolnym serwerze i liczbę opcji konfiguracyjnych dla każdego z nich, znalezienie rozbieżności może być trudnym zadaniem. Dlatego należy korzystać z systemu zarządzania konfiguracją, takiego jak Chef (<https://www.chef.io>) lub Puppet (<https://puppet.com>), w celu utrzymania spójnych konfiguracji w systemach operacyjnych i aplikacjach (włącznie z Kafką).

Wskaźniki brokera

Oprócz partycji URP należy monitorować także inne wskaźniki na ogólnym poziomie brokera. Chociaż możesz nie być skłonny do ustawiania progów alarmowych dla wszystkich z nich, dostarczają one cennych informacji na temat Twoich brokerów i klastra. Powinny być obecne w każdym tworzonym przez Ciebie dashboardzie monitorowania.

Liczba aktywnych kontrolerów

Liczba aktywnych kontrolerów wskazuje, czy dany broker jest aktualnie kontrolerem klastra. Wartość wskaźnika będzie wynosić 0 lub 1, przy czym 1 oznacza, że broker jest aktualnie kontrolerem. Zawsze tylko jeden broker powinien być kontrolerem w klastrze. Jeśli dwa brokery twierdzą, że są aktualnie kontrolerem, oznacza to problem polegający na tym, że utknął jakiś wątek kontrolera, który powinien już zostać zakończony. Może to powodować problemy z brakiem możliwości prawidłowego wykonywania zadań administracyjnych, takich jak przenoszenie partycji. Aby temu zaradzić, musisz zrestartować przynajmniej te dwa brokery. Jednak gdy w klastrze znajduje się dodatkowy kontroler, często występują problemy z bezpiecznym zamknięciem brokera, a zamiast tego trzeba będzie wymusić jego zatrzymanie. Więcej informacji na temat aktywnej liczby kontrolerów zebraliśmy w tabeli 13.5.

Tabela 13.5. Szczegóły wskaźnika liczby aktywnych kontrolerów

Nazwa wskaźnika	Liczba aktywnych kontrolerów
JMX MBean	kafka.controller:type=KafkaController,name=ActiveControllerCount
Zakres wartości	0 lub 1

Jeżeli żaden broker nie rości sobie prawa do bycia kontrolerem w klastrze, klaster przestaje reagować prawidłowo w obliczu zmian stanu, takich jak tworzenie tematów lub partycji, lub awarii brokerów. W takiej sytuacji należy przeprowadzić dalsze badania, aby dowiedzieć się, dlaczego wątki kontrolera nie działają właściwie. Taki problem może spowodować np. powstanie partycji sieciowej z klastrem ZooKeepera. Po rozwiązaniu tego bazowego problemu warto zrestartować wszystkie brokery w klastrze, aby zresetować stan dla wątków kontrolera.

Rozmiar kolejki kontrolera

Rozmiar kolejki kontrolera wskazuje, ile żądań brokerów oczekuje obecnie na przetworzenie przez kontroler. Wartość wskaźnika będzie nieujemną liczbą całkowitą, przy czym będzie się często zmieniać w miarę pojawiania się nowych żądań od brokerów i wykonywania działań administracyjnych,

takich jak tworzenie partycji, przenoszenie partycji i zmiana lidera przetwarzania. Należy spodziewać się nagłych wzrostów wartości wskaźnika, ale jeśli wartość będzie stale rosła lub utrzymywać się na stałym wysokim poziomie, będzie to oznaczało, że kontroler mógł utknąć. Może to uniemożliwiać prawidłowe wykonywanie zadań administracyjnych. Aby temu zaradzić, należy przenieść kontroler na inny broker, co wymaga wyłączenia brokera, który jest obecnie kontrolerem. Lecz gdy kontroler utknie, często będą występować problemy z kontrolowanym wyłączeniem dowolnego brokera. Więcej informacji na temat rozmiaru kolejki kontrolera zebraliśmy w tabeli 13.6.

Tabela 13.6. Szczegóły wskaźnika rozmiaru kolejki kontrolera

Nazwa wskaźnika	Rozmiar kolejki kontrolera
JMX MBean	kafka.controller:type=ControllerEventManager,name=EventQueueSize
Zakres wartości	Nieujemna liczba całkowita

Współczynnik beczynności procedur obsługi żądań

Do obsługi wszystkich żądań klientów Kafka wykorzystuje dwie pule wątków: **wątki sieciowe** i **wątki procedur obsługi żądań** (zwane również **wątkami we-wy**). Wątki sieciowe są odpowiedzialne za operacje odczytu i zapisu danych dla klientów w całej sieci. Nie wymaga to zbyt wiele przetwarzania, co oznacza, że wyczerpanie wątków sieciowych jest mniejszym zmartwieniem. Wątki procedur obsługi żądań są jednak odpowiedzialne za obsługę żądań klientów, która obejmuje odczyt komunikatów z dysku lub zapis komunikatów na dysku. W związku z tym, gdy brokery stają się coraz bardziej obciążone, ma to istotny wpływ na pulę wątków. Więcej informacji na temat współczynnika beczynności procedur obsługi żądania zebraliśmy w tabeli 13.7.

Tabela 13.7. Szczegóły współczynnika beczynności procedur obsługi żądań

Nazwa wskaźnika	Procentowa średnia beczynności wątków procedur obsługi żądań
JMX MBean	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent
Zakres wartości	Liczba zmiennoprzecinkowa z przedziału od 0 do 1 włącznie



Inteligentne wykorzystanie wątków

Choć może się wydawać, że będziesz potrzebował setek wątków procedur obsługi żądań, tak naprawdę nie musisz konfigurować więcej wątków, niż wynosi liczba procesorów w brokerze. Apache Kafka bardzo sprytnie korzysta z procedur obsługi żądań, przesuwając do czyścica te żądania, których przetworzenie zajmuje dużo czasu. Jest to stosowane np., gdy żądania są kwotowane lub wymagane jest więcej niż jedno potwierdzenie żądań produkowania.

Wskaźnik współczynnika beczynności procedur obsługi żądań określa, przez jaki procent czasu procedury obsługi żądań nie są wykorzystywane. Im mniejsza wartość, tym bardziej obciążony broker. Z naszego doświadczenia wynika, że współczynniki beczynności poniżej 20% wskazują na potencjalny problem, a poniżej 10% oznaczają istniejący problem z wydajnością. Oprócz zbyt małych rozmiarów klastra istnieją dwa powody wysokiego poziomu wykorzystania wątków z tej puli. Pierwszym jest niewystarczająca liczba wątków w puli. Generalnie liczbę wątków dla procedur obsługi żądań

należy ustawić równą liczbę procesorów w systemie (z uwzględnieniem procesorów hyper-threadingowych).

Innym częstym powodem wysokiego poziomu wykorzystania wątków dla procedur obsługi żądań jest to, że wątki wykonują niepotrzebną pracę dla każdego żądania. W wersjach Kafki wcześniejszych niż 0.10 wątek procedur obsługi żądań był odpowiedzialny za dekompresję każdej przychodzącej partii komunikatów, walidację komunikatów i przypisywanie przesunięć, a następnie ponowne kompresowanie partii komunikatów z przesunięciami przed zapisaniem jej na dysku. Co gorsza, wszystkie metody kompresji były wykonywane z synchroniczną blokadą. Od wersji 0.10 istnieje nowy format komunikatów, który dopuszcza względne przesunięcia w partii komunikatów. Oznacza to, że nowsze producenci przed wysłaniem partii komunikatów ustawiają względne przesunięcia, co pozwoli brokerowi pominąć ponowną kompresję tej partii komunikatów. Jeden z największych wzrostów wydajności można uzyskać, upewniając się, że wszystkie klienty producenckie i konsumenckie obsługują format komunikatów 0.10, oraz zmieniając wersję formatu komunikatów na 0.10 także w brokerach. Spowoduje to znaczne ograniczenie wykorzystania wątków procedur obsługi żądań.

Prędkość przysyłania bajtów do wszystkich tematów

Prędkość przysyłania bajtów do wszystkich tematów wyrażona w bajtach na sekundę jest przydatna do pomiaru natężenia ruchu komunikatów otrzymywanych przez brokery od klientów produkujących. Jest to dobry wskaźnik kształtowania się trendu w miarę upływu czasu, który pomaga określić, kiedy trzeba rozszerzyć klaster lub wykonać inną pracę związaną z rozwojem infrastruktury. Jest on również przydatny do oceny, czy jeden broker w klastrze odbiera większy ruch niż pozostałe, co wskazywałoby na konieczność wykonania rebalansowania partycji w klastrze. Więcej informacji zebraliśmy w tabeli 13.8.

Tabela 13.8. Szczegóły wskaźnika przysyłania bajtów do wszystkich tematów

Nazwa wskaźnika	Liczba bajtów przychodzących na sekundę
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
Zakres wartości	Liczba o podwójnej precyzji dla prędkości, liczba całkowita dla licznika

Ponieważ jest to pierwszy przytoczony wskaźnik prędkości, warto krótko omówić atrybuty dostarczane przez tego typu wskaźniki. Wszystkie wskaźniki prędkości mają siedem atrybutów, a wybór tych, które mają zostać użyte, zależy od typu pomiaru. Atrybuty te zapewniają dyskretną liczbę zdarzeń, a także średnią liczbę zdarzeń w różnych przedziałach czasu. Upewnij się, że korzystasz z tych wskaźników w odpowiedni sposób, gdyż w przeciwnym razie uzyskasz niewłaściwy obraz brokera.

Pierwsze dwa atrybuty nie są pomiarami, ale pomogą Ci zrozumieć otrzymywane dane:

EventType

Jest to jednostka miary dla wszystkich atrybutów. W tym przypadku są to bajty.

RateUnit

We wskaźnikach prędkości jest to przedział czasu, w którym mierzona jest prędkość. W tym przypadku jest to sekunda.

Te dwa atrybuty opisowe mówią nam, że niezależnie od przedziału czasu dla uśredniania wskaźnik jest prezentowany jako liczba bajtów na sekundę. Dostępne są cztery atrybuty prędkości o różnej precyzji:

- OneMinuteRate
Średnia z ostatniej minuty.
- FiveMinuteRate
Średnia z ostatnich 5 minut.
- FifteenMinuteRate
Średnia z ostatnich 15 minut.
- MeanRate
Średnia od momentu uruchomienia brokera.

Wartość OneMinuteRate szybko się zmienia i zapewnia obraz pomiaru dla określonego „punktu w czasie”. Jest to przydatne do obserwowania krótkich i nagłych wzrostów natężenia ruchu. MeanRate nie będzie się zbyt szybko zmieniać i zapewnia ogólną tendencję. Chociaż MeanRate ma swoje zastosowania, prawdopodobnie nie jest to wskaźnik, który miałby zastosowanie w ustawianiu alertów. FiveMinuteRate i FifteenMinuteRate zapewniają kompromis między OneMinuteRate i MeanRate.

Oprócz atrybutów prędkości istnieje również atrybut Count (licznik). Jest to stale rosnąca wartość wskaźnika od momentu uruchomienia brokera. Dla wskaźnika prędkości przysyłania bajtów do wszystkich tematów Count reprezentuje całkowitą liczbę bajtów wyprodukowanych do brokera od momentu uruchomienia procesu. Wykorzystywany z systemem wskaźników, który obsługuje przeciwniary, może zapewnić bezwzględny obraz pomiaru zamiast uśrednionej prędkości.

Prędkość wysyłania bajtów ze wszystkich tematów

Prędkość wysyłania bajtów ze wszystkich tematów, podobnie jak prędkość przysyłania bajtów, jest ogólnym wskaźnikiem wzrostu. W tym przypadku prędkość wysyłania bajtów pokazuje, jak szybko konsumenci odczytują komunikaty. Liczba bajtów wychodzących może kształtować się inaczej niż liczba bajtów przychodzących, gdyż Kafka może z łatwością obsługiwać wiele konsumentów. W licznych wdrożeniach Kafki wskaźnik ruchu wychodzącego może być sześciokrotnie wyższy od wskaźnika ruchu przychodzącego! Dlatego należy obserwować ten wskaźnik oddzielnie. Więcej informacji zebraliśmy w tabeli 13.9.

Tabela 13.9. Szczegóły wskaźnika wysyłania bajtów ze wszystkich tematów

Nazwa wskaźnika	Liczba bajtów wychodzących na sekundę
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
Zakres wartości	Liczba o podwójnej precyzji dla prędkości, liczba całkowita dla licznika



Uwzględnienie procesów replikacji

Prędkość wysyłania bajtów obejmuje *również* ruch repliki. Oznacza to, że jeśli wszystkie tematy są skonfigurowane ze współczynnikiem replikacji o wartości 2, wskaźnik bajtów wychodzących będzie równy wskaźnikowi bajtów przychodzących w przypadku braku klientów konsumenckich. Jeśli masz jednego klienta konsumenckiego odczytującego wszystkie komunikaty w klastrze, wartość wskaźnika ruchu wychodzącego będzie stanowiła dwukrotność wartości wskaźnika ruchu przychodzącego. Może to być mylące, gdy będziesz przyglądał się tym wskaźnikom, jeżeli nie będziesz wiedział, co jest liczone.

Prędkość przysyłania komunikatów do wszystkich tematów

O ile opisane w poprzednich punktach prędkości transmisji bajtów pokazują ruch brokera w bezwzględnych wartościach bajtowych, o tyle prędkość przysyłania komunikatów do wszystkich tematów pokazuje liczbę generowanych na sekundę pojedynczych komunikatów, niezależnie od ich rozmiarów. Jest to przydatne jako wskaźnik wzrostu, który stanowi inny pomiar ruchu producenckiego. Może być również używany w połączeniu ze wskaźnikiem bajtów przychodzących w celu określenia średniego rozmiaru komunikatu. Podobnie jak liczba bajtów przychodzących, pozwala ponadto zauważyć brak równowagi w brokerach, który ostrzega o konieczności przeprowadzenia prac konserwacyjnych. Więcej informacji zebraliśmy w tabeli 13.10.

Tabela 13.10. Szczegóły wskaźnika przysyłania komunikatów do wszystkich tematów

Nazwa wskaźnika	Liczba komunikatów przychodzących na sekundę
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
Zakres wartości	Liczba o podwójnej precyzji dla prędkości, liczba całkowita dla licznika



Dlaczego nie ma wskaźnika komunikatów wychodzących?

Ludzie często pytają, dlaczego nie ma wskaźnika komunikatów wychodzących dla brokera Kafki. Jest to spowodowane tym, że gdy komunikaty zostają skonsumowane, broker wysyła po prostu kolejną partię do konsumenta i nie rozpakowuje jej, aby sprawdzić, ile komunikatów jest w środku. W związku z tym broker tak naprawdę nie wie, ile komunikatów zostało wysłanych. Jedynym możliwym do podania wskaźnikiem jest liczba pobrań na sekundę, ale jest to prędkość żądań, a nie liczba komunikatów.

Liczba partycji

Liczba partycji dla brokera zasadniczo nie zmienia się tak bardzo, ponieważ jest to całkowita liczba partycji przypisanych do tego brokera. Obejmuje to każdą replikę, którą ma broker, niezależnie od tego, czy jest ona liderem czy obserwatorem danej partycji. Monitorowanie tego wskaźnika jest często bardziej interesujące w klastrze, który ma włączone automatyczne tworzenie tematu, ponieważ może to pozbawić osobę zarządzającą klastrem kontroli nad tworzeniem tematów. Więcej informacji zebraliśmy w tabeli 13.11.

Tabela 13.11. Szczegóły wskaźnika liczby partycji

Nazwa wskaźnika	Liczba partycji
JMX MBean	kafka.server:type=ReplicaManager,name=PartitionCount
Zakres wartości	Nieujemna liczba całkowita

Liczba liderów

Wskaźnik **liczby liderów** pokazuje liczbę partycji, dla których broker jest obecnie liderem. Podobnie jak w przypadku większości innych pomiarów w brokerach, ten wskaźnik powinien być na ogół równomierny w obrębie brokerów w klastrze. Znacznie ważniejsze jest regularne sprawdzanie liczby liderów i ewentualnie ostrzeganie na jej podstawie, ponieważ będzie ona wskazywać, kiedy klastr jest niezrównoważony, nawet jeśli repliki są idealnie zrównoważone pod względem liczby i wielkości w całym klastrze. Dzieje się tak, ponieważ broker może zrezygnować z przywództwa dla partycji z wielu powodów, takich jak wygaśnięcie sesji ZooKeepera, i automatycznie nie przejmuje z powrotem przywództwa po odzyskaniu sprawności (chyba że włączysz automatyczne równoważenie przywództwa). W takich przypadkach wskaźnik ten pokaże mniejszą liczbę liderów, a nie rzadko ich brak, co będzie oznaczało, że musisz przeprowadzić wybory repliki preferowanej, aby zrównoważyć przywództwo w klastrze. Więcej informacji zebraliśmy w tabeli 13.12.

Tabela 13.12. Szczegóły wskaźników liczby liderów

Nazwa wskaźnika	Liczba liderów
JMX MBean	kafka.server:type=ReplicaManager,name=LeaderCount
Zakres wartości	Nieujemna liczba całkowita

Przydatnym sposobem wykorzystania tego wskaźnika jest zastosowanie go wraz z liczbą partycji, aby pokazać procent partycji, dla których broker jest liderem. W dobrze zrównoważonym klastrze, który używa współczynnika replikacji 2, każdy broker powinien być liderem dla ok. 50% swoich partycji. Jeśli współczynnik replikacji wynosi 3, odsetek ten spada do 33%.

Liczba partycji offline

Wraz z liczbą partycji URP **liczba partycji offline** jest kluczowym wskaźnikiem monitorowania (tabela 13.13). Pomiar ten jest dostarczany tylko przez broker będący kontrolerem klastra (wszystkie pozostałe brokery zgłoszą 0) i pokazuje liczbę partycji w klastrze, które obecnie nie mają lidera. Partycje bez liderów mogą pojawiać się z dwóch głównych powodów:

- wszystkie brokery hostujące repliki dla tej partycji zostały wyłączone;
- żadna z replik zsynchronizowanych nie może przejąć przywództwa z powodu niezgodnej liczby komunikatów (przy wyłączonych wyborach nieczystego lidera).

W produkcyjnym klastrze Kafki partycja offline może wpływać na klienty produkcyjne i utratę komunikatów lub powodować uruchomienie mechanizmu ciśnienia zwrotnego w aplikacji. Jest to najczęściej problem powodujący przestój danej lokalizacji i należy go rozwiązać natychmiast.

Tabela 13.13. Szczegóły wskaźnika liczby partycji offline

Nazwa wskaźnika	Liczba partycji offline
JMX MBean	kafka.controller:type=KafkaController,name=OfflinePartitionsCount
Zakres wartości	Nieujemna liczba całkowita

Wskaźniki żądań

Protokół Kafki, który opisaliśmy w rozdziale 6., ma wiele różnych żądań. Zapewniane są wskaźniki dla każdego z tych żądań. Od wersji 2.5.0 wskaźniki mają żądania, które pokazyaliśmy w tabeli 13.14.

Tabela 13.14. Nazwy wskaźników żądań

AddOffsetsToTxn	AddPartitionsToTxn	AlterConfigs
AlterPartitionReassignments	AlterReplicaLogDirs	ApiVersions
ControlledShutdown	CreateAcls	CreateDelegationToken
CreatePartitions	CreateTopics	DeleteAcls
DeleteGroups	DeleteRecords	DeleteTopics
DescribeAcls	DescribeConfigs	DescribeDelegationToken
DescribeGroups	DescribeLogDirs	ElectLeaders
EndTxn	ExpireDelegationToken	Fetch
FetchConsumer	FetchFollower	FindCoordinator
Heartbeat	IncrementalAlterConfigs	InitProducerId
JoinGroup	LeaderAndIsr	LeaveGroup
ListGroups	ListOffsets	ListPartitionReassignments
Metadata	OffsetCommit	OffsetDelete
OffsetFetch	OffsetsForLeaderEpoch	Produce
RenewDelegationToken	SaslAuthenticate	SaslHandshake
StopReplica	SyncGroup	TxnOffsetCommit
UpdateMetadata	WriteTxnMarkers	

Dla każdego z tych żądań dostępnych jest osiem wskaźników, które zapewniają wgląd w każdą fazę przetwarzania żądania, np. dla żądania Fetch dostępne są wskaźniki przedstawione w tabeli 13.15.

Wskaźnik liczby żądań na sekundę zalicza się do wskaźników prędkości, o których pisaliśmy wcześniej, i pokazuje całkowitą liczbę żądań danego typu, które zostały odebrane i przetworzone w jednostce czasu. Zapewnia to wgląd w częstotliwość czasu każdego żądania, choć należy zauważyć, że wiele żądań, takich jak StopReplica i UpdateMetadata, jest dość rzadkich.

Każdy z siedmiu wskaźników *czasowych* zawiera zestaw percentyli dla żądań, a także dyskretny atrybut Count, podobnie jak w przypadku wskaźników prędkości. Wszystkie wskaźniki są obliczane od momentu uruchomienia brokera, więc pamiętaj o tym, gdy będziesz się przyglądał wskaźnikom, które nie zmieniają się przez dłuższy czas. Im dłużej broker jest uruchomiony, tym bardziej stabilne będą liczby. Wskaźniki te reprezentują następujące elementy przetwarzania żądań:

Tabela 13.15. Wskaźniki żądania Fetch

Nazwa	JMX MBean
Czas całkowity	kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Fetch
Czas kolejki żądań	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=Fetch
Czas lokalny	kafka.network:type=RequestMetrics,name=LocalTimeMs,request=Fetch
Czas zdalny	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=Fetch
Czas dławienia	kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=Fetch
Czas kolejki odpowiedzi	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=Fetch
Czas wysyłania odpowiedzi	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=Fetch
Liczba żądań na sekundę	kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Fetch

Całkowity czas

Całkowity czas, jaki broker poświęca na przetwarzanie żądania, od jego otrzymania do odeśłania odpowiedzi żądającemu.

Czas kolejki żądań

Czas spędzany przez żądanie w kolejce od momentu, w którym zostało odebrane, do rozpoczęcia jego przetwarzania.

Czas lokalny

Czas poświęcany przez lidera partycji na przetwarzanie żądania i wysłanie (ale niekoniecznie spłukanie) go na dysk.

Czas zdalny

Czas oczekiwania na obserwatory przed zakończeniem przetwarzania żądania.

Czas dławienia

Czas przetrzymywania odpowiedzi w celu spowolnienia żądającego, aby zapewnić zgodność z ustawieniami kwot klienta.

Czas kolejki odpowiedzi

Czas spędzany przez odpowiedź na żądanie w kolejce, zanim zostanie wysłana do żądającego.

Czas wysłania odpowiedzi

Ilość czasu faktycznie poświęconego na wysłanie odpowiedzi.

Oto atrybuty zapewniane dla każdego wskaźnika:

Count

Bezwzględna liczba żądań od rozpoczęcia procesu.

Min Minimalna wartość dla wszystkich żądań.

Max Maksymalna wartość dla wszystkich żądań.

Mean

Średnia wartość dla wszystkich żądań.

StdDev

Odchylenie standardowe dla pomiarów czasów żądania jako całości.

Percentiles

50thPercentile, 75thPercentile, 95thPercentile, 98thPercentile, 99thPercentile, 999th
↳Percentile.



Co to jest percentyl?

Percentyle są częstym sposobem postrzegania pomiarów czasu. 99. percentyl mówi nam, że 99% wszystkich wartości w grupie próbek (w tym przypadku czasów żądania) jest mniejsze niż wartość bazowa. Oznacza to, że 1% wartości jest większe niż określona wartość bazowa. Powszechnym wzorem jest przeglądanie wartości średniej oraz wartości percentyli 99% lub 99,9%. W ten sposób możesz zrozumieć, jaka jest średnia wydajność żądania i jakie są wartości odstające.

Które spośród wszystkich tych wskaźników i atrybutów dla żądań są ważne dla monitorowania? Należy gromadzić co najmniej średnią i jeden z wyższych percentyli (99% lub 99,9%) dla wskaźnika czasu całkowitego, a także liczbę żądań na sekundę dla każdego typu żądania. Daje to wgląd w ogólną wydajność żądań wysyłanych do brokera Kafki. Jeśli to możliwe, powinieneś również gromadzić te pomiary dla pozostałych sześciu wskaźników czasowych dla każdego typu żądania, gdyż pozwoli to zawęzić wszelkie problemy z wydajnością do określonej fazy przetwarzania żądania.

Ustawianie progów alertów na podstawie wskaźników czasowych może być trudne. Czas żądania Fetch może się np. bardzo różnić w zależności od wielu czynników, w tym ustawień klienta dotyczących czasu oczekiwania na komunikaty, obciążenia pobieranego tematu i szybkości połączenia sieciowego między klientem a brokerem. Bardzo przydatne może być jednak opracowanie wartości bazowej dla pomiaru 99,9. percentyla przynajmniej dla czasu całkowitego, zwłaszcza dla żądań Produce, i ustawianie na tej podstawie alertów. Podobnie jak w przypadku wskaźnika partycji URP, gwałtowny wzrost w 99,9. percentylu dla żądań Produce może ostrzegać o wielu problemach z wydajnością.

Wskaźniki tematów i partycji

Oprócz wielu wskaźników opisujących w ogólny sposób działanie brokera Kafki dostępne są wskaźniki charakterystyczne dla tematów i partycji. W większych klastrach tych wskaźników może być dość dużo i gromadzenie ich wszystkich w systemie wskaźników w ramach normalnych operacji może okazać się niemożliwe. Są one jednak dość przydatne do debugowania konkretnych problemów z klientem. Wskaźniki tematów można wykorzystać np. do zidentyfikowania konkretnego tematu, który powoduje znaczne zwiększenie natężenia ruchu do klastra. Ważne może być również dostarczenie tych wskaźników, aby mieli do nich dostęp użytkownicy Kafki (klienty produkcyjne i konsumenckie). Niezależnie od tego, czy jesteś w stanie regularnie gromadzić te wskaźniki, powinieneś być świadomy tego, co jest przydatne.

Dla wszystkich przykładów zamieszczonych w tabeli 13.16 będziemy używać przykładowego tematu *NAZWATEMATU* oraz partycji 0. Gdy będziesz uzyskiwał dostęp do opisanych wskaźników, musisz zastąpić nazwę tematu i numer partycji wartościami odpowiednimi dla Twojego klastra.

Wskaźniki tematów

Dla wszystkich wskaźników tematów pomiary są bardzo podobne do opisanych wcześniej wskaźników brokera. Tak naprawdę jedyną różnicą jest podawanie nazwy tematu i to, że wskaźniki będą charakterystyczne dla określonego tematu. Biorąc pod uwagę samą liczbę dostępnych wskaźników w zależności od liczby tematów obecnych w klastrze, prawie na pewno będą to wskaźniki, dla których nie będziesz chciał konfigurować monitorowania i alertów. Przydatne jest jednak dostarczanie ich klientom, aby mogli oni ocenić i debugować własne użycie Kafki. Więcej informacji na temat tych wskaźników zebraliśmy w tabeli 13.16.

Tabela 13.16. Wskaźniki dla każdego tematu

Nazwa	JMX MBean
Prędkość bajtów wchodzących	<code>kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=NAZWATEMATU</code>
Prędkość bajtów wychodzących	<code>kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=NAZWATEMATU</code>
Prędkość nieudanych pobrań	<code>kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=NAZWATEMATU</code>
Prędkość nieudanego produkowania	<code>kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=NAZWATEMATU</code>
Prędkość komunikatów wchodzących	<code>kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=NAZWATEMATU</code>
Prędkość żądań pobierania	<code>kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=NAZWATEMATU</code>
Prędkość żądań produkowania	<code>kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=NAZWATEMATU</code>

Wskaźniki partycji

Wskaźniki dotyczące poszczególnych partycji są zwykle mniej przydatne na bieżąco niż wskaźniki dotyczące poszczególnych tematów. Ponadto są one dość liczne, ponieważ setki tematów mogą składać się na tysiące partycji. Niemniej jednak mogą być użyteczne w niektórych sytuacjach. Wskaźnik wielkości partycji wskazuje np. ilość danych (w bajtach), które są obecnie przechowywane na dysku dla danej partycji. Łącznie będą one wskazywać ilość danych przechowywanych dla pojedynczego tematu, co może być przydatne w przydzielaniu poszczególnym klientom kosztów Kafki. Rozbieżność między wielkością dwóch partycji dla tego samego tematu może wskazywać na problem, w którym komunikaty nie są równomiernie rozmieszczone w kluczu używanym podczas produkowania. Wskaźnik liczby segmentów dziennika pokazuje liczbę plików segmentów dziennika na dysku dla danej partycji. Wraz z rozmiarem partycji może to być przydatne do śledzenia zasobów. Więcej informacji na temat tych wskaźników zebraliśmy w tabeli 13.17.

Wskaźniki przesunięcia końca dziennika i przesunięcia początku dziennika określają odpowiednio najnowsze i najstarsze przesunięcie dla komunikatów w tej partycji. Należy jednak zauważyć, że różnica między tymi dwiema liczbami niekoniecznie wskazuje na liczbę komunikatów w partycji, ponieważ kompaktowanie dziennika może skutkować „brakującymi” przesunięciami, które zostały

Tabela 13.17. Wskaźniki dla każdej partycji

Nazwa	JMX MBean
Rozmiar partycji	<code>kafka.log:type=Log,name=Size,topic=NAZWATEMATU,partition=0</code>
Liczba segmentów dziennika	<code>kafka.log:type=Log,name=NumLogSegments,topic=NAZWATEMATU,partition=0</code>
Przesunięcie końca dziennika	<code>kafka.log:type=Log,name=LogEndOffset,topic=NAZWATEMATU,partition=0</code>
Przesunięcie początku dziennika	<code>kafka.log:type=Log,name=LogStartOffset,topic=NAZWATEMATU,partition=0</code>

usunięte z partycji z powodu nowszych komunikatów z tym samym kluczem. W niektórych środowiskach przydatne może być śledzenie tych przesunięć dla partycji. Jednym z takich przypadków użycia jest zapewnienie bardziej precyzyjnego mapowania znacznika czasu na przesunięcie, co pozwala klientom konsumenckim łatwo wykonać rollback przesunięcia do określonego punktu w czasie (choć jest to mniej istotne, gdy ma się do dyspozycji wyszukiwanie indeksów czasowych wprowadzone w Kafce w wersji 0.10.1).



Wskaźniki partycji URP

Dla każdej partycji istnieje wskaźnik, który określa, czy ma ona wystarczającą liczbę replik zsynchronizowanych. Generalnie nie jest to zbyt przydatne w codziennych operacjach, ponieważ istnieje zbyt wiele wskaźników do gromadzenia i obserwowania. Znacznie łatwiej jest monitorować liczbę partycji URP w całym brokerze, a następnie użyć narzędzi wiersza poleceń (opisanych w rozdziale 12.) w celu określenia konkretnych partycji, które mają niewystarczającą liczbę replik zsynchronizowanych.

Monitorowanie JVM

Oprócz danych dostarczonych przez broker Kafki powinieneś monitorować standardowy zestaw wskaźników dla wszystkich swoich serwerów, a także samą maszynę wirtualną Javy (ang. *Java Virtual Machine* — JVM). Będą one przydatne do ostrzegania o takich sytuacjach jak zwiększenie aktywności mechanizmu odzyskiwania pamięci, powodujące obniżanie wydajności brokera. Wskaźniki te zapewnią również wgląd w powody obserwowania zmiany wskaźników w brokerze.

Mechanizm odzyskiwania pamięci

W przypadku JVM kluczowym wskaźnikiem do monitorowania jest status mechanizmu odzyskiwania pamięci (ang. *Garbage Collection* — GC). Konkretnie ziarna, które należy monitorować pod kątem tych informacji, będą się różnić w zależności od używanego środowiska uruchomieniowego Javy (ang. *Java Runtime Environment* — JRE) oraz stosowanych ustawień GC. Ziarna dla środowiska Oracle Java 1.8 JRE działającego z mechanizmem odzyskiwania pamięci G1 pokazaliśmy w tabeli 13.18.

Zwróć uwagę, że w semantyce GC „pełny cykl” odnosi się do „starego pokolenia”. Dla każdego z tych wskaźników należy obserwować dwa atrybuty: `CollectionCount` i `CollectionTime`. `CollectionCount` to liczba cykli GC danego typu (pełnych lub dla młodego pokolenia) od momentu uruchomienia JVM. `CollectionTime` to określony w milisekundach czas poświęcony na wykonywanie cykli GC

Tabela 13.18. Wskaźniki dla mechanizmu odzyskiwania pamięci G1

Nazwa	JMX MBean
Pełne cykle GC (dla starego pokolenia)	<code>java.lang:type=GarbageCollector,name=G1 Old Generation</code>
Cykle GC dla młodego pokolenia	<code>java.lang:type=GarbageCollector,name=G1 Young Generation</code>

danego typu od momentu uruchomienia JVM. Ponieważ te pomiary są licznikami, mogą być wykorzystane przez system wskaźników do podawania bezwzględnej liczby cykli GC i długości trwania GC na jednostkę czasu. Można ich również użyć do określania średniej ilości czasu na cykl GC, chociaż jest to mniej przydatne w normalnych operacjach.

Każdy z tych wskaźników ma również atrybut `LastGcInfo`. Jest to wartość złożona z pięciu pól, która dostarcza informacji o ostatnim cyklu mechanizmu odzyskiwania pamięci dla typu GC opisanego przez ziarno. Warto zwrócić uwagę na wartość `duration`, gdyż określa ona w milisekundach czas trwania ostatniego cyklu GC. Pozostałe wartości `LastGcInfo` (`GcThreadCount`, `id`, `startTime` i `endTime`) mają charakter informacyjny i nie są zbyt użyteczne. Zwróć uwagę, że za pomocą tego atrybutu nie będziesz mógł sprawdzić czasu każdego cyklu GC, ponieważ cykle GC mogą być wykonywane z dużą częstotliwością, w szczególności dla młodego pokolenia.

Monitorowanie systemu operacyjnego za pomocą JVM

JVM może dostarczyć pewnych informacji na temat systemu operacyjnego za pośrednictwem ziarna `java.lang:type=OperatingSystem`. Informacje te są jednak ograniczone i nie reprezentują wszystkiego, co powinieneś wiedzieć o systemie uruchomionym na brokerze. Dwa przydatne atrybuty, które można tutaj gromadzić, a które są trudne do zbierania w systemie operacyjnym, to `MaxFileDescriptorCount` i `OpenFileDescriptorCount`. `MaxFileDescriptorCount` informuje o maksymalnej liczbie deskryptorów plików (ang. *File Descriptor* — FD), które może mieć otwarta maszyna wirtualna Javy. Atrybut `OpenFileDescriptorCount` pokazuje liczbę otwartych FD. Dla każdego segmentu dziennika i połączenia sieciowego będą otwarte FD, których liczba może się szybko zwiększać. Problem z prawidłowym zamknięciem połączeń sieciowych może spowodować, że broker szybko wyczerpie dozwoloną liczbę deskryptorów plików.

Monitorowanie systemu operacyjnego

JVM nie może dostarczyć nam wszystkich niezbędnych informacji o systemie, w którym działa. Z tego powodu musimy gromadzić wskaźniki nie tylko z brokera, ale także z samego systemu operacyjnego. Większość systemów monitorowania zapewnia agenty, które będą gromadzić znacznie więcej danych dotyczących systemu operacyjnego, niż mógłbyś potrzebować. Główne obszary, które należy obserwować, to wykorzystanie procesora, pamięci, dysku, we-wy dysku i sieci.

W przypadku wykorzystania procesora należy przyjrzeć się przynajmniej średniej obciążenia systemu. Będzie to pojedyncza liczba, która wskaże względne wykorzystanie procesorów. Potrzebne może być również przechwytywanie procentowego wykorzystania procesora z podziałem na typy. W zależności od metody gromadzenia danych i konkretnego systemu operacyjnego mogą występować niektóre lub wszystkie z wymienionych niżej podziałów procentowego użycia procesora (wraz z zastosowanym skrótem):

- us Czas poświęcony na wykonywanie pracy w przestrzeni użytkownika.
- sy Czas poświęcony na wykonywanie pracy w przestrzeni jądra.
- ni Czas poświęcony na wykonywanie procesów o niskim priorytecie.
- id Czas bezczynności.
- wa Czas oczekiwania (na dysku).
- hi Czas poświęcony na obsługę przerw sprzętowych.
- si Czas poświęcony na obsługę przerw programowych.
- st Czas oczekiwania na hipernadzorcę.



Co to jest obciążenie systemu?

Chociaż powszechnie wiadomo, że obciążenie systemu jest miarą wykorzystania procesora w systemie, sposób dokonywania tego pomiaru nie jest do końca jasny. Średnia obciążenia to liczba uruchamialnych procesów, które czekają na wykonanie przez procesor. W Linuksie obejmuje to również wątki, które są w nieprzerywalnym stanie uśpienia, takim jak oczekiwanie na dysk. Obciążenie jest reprezentowane przez trzy liczby: średnią z ostatniej minuty, średnią z ostatnich 5 minut i średnią z 15 minut. W systemie z pojedynczym CPU wartość 1 wskazywałaby, że system jest obciążony w 100%, a wątek zawsze czeka na wykonanie. Oznacza to, że w systemie z wieloma procesorami liczba średniej obciążenia wskazująca 100% jest równa liczbie procesorów w systemie. Jeśli w systemie znajdują się np. 24 procesory, 100% będzie oznaczane średnią obciążenia wynoszącą 24.

Broker Kafki znaczną ilość przetwarzania przeznaczają na obsługę żądań. Dlatego podczas monitorowania Kafki istotne jest śledzenie wykorzystania procesora. Śledzenie pamięci jest mniej ważne dla samego brokera, ponieważ Kafka będzie zwykle uruchamiana ze stosunkowo niewielką stertą JVM. Będzie używać niewielkiej ilości pamięci spoza sterty dla funkcji kompresji, ale większość pamięci systemowej pozostanie do wykorzystania jako pamięć podręczna. Niemniej jednak powinno się śledzić wykorzystanie pamięci, by mieć pewność, że inne aplikacje nie ingerują w broker. Należy również monitorować ilości całkowitej i wolnej przestrzeni wymiany, aby upewnić się, że ta przestrzeń nie jest używana.

Jeśli chodzi o Kafkę, dysk jest zdecydowanie najważniejszym podsystemem. Na dysku utrwalane są wszystkie komunikaty, stąd wydajność Kafki zależy w dużym stopniu od wydajności dysków. Istotne jest monitorowanie wykorzystania miejsca na dysku oraz *i-węzłów* (*i-węzły* są obiektami metadanych plików i katalogów w uniksowych systemach plików), ponieważ trzeba pilnować, by nie zabrakło miejsca. Dotyczy to szczególnie partycji, w których przechowywane są dane Kafki. Konieczne jest ponadto monitorowanie statystyk operacji we-wy dysku, które wskazują, czy dysk jest używany efektywnie. Przynajmniej w przypadku dysków, na których przechowywane są dane Kafki, należy monitorować odczyty i zapisy na sekundę, średnie rozmiary kolejek odczytu i zapisu, średni czas oczekiwania i procentowe wykorzystanie dysku.

Trzeba także monitorować wykorzystanie sieci przez brokery. Będzie to po prostu ilość przychodzącego i wychodzącego ruchu sieciowego, zwykle podawana w bitach na sekundę. Należy pamiętać, że każdy bit przychodzący do brokera Kafki będzie generował liczbę bitów wychodzących równą

współczynniki replikacji tematów, nie wliczając konsumentów. W zależności od liczby konsumentów wychodzący ruch sieciowy może być nawet o rząd wielkości większy niż ruch przychodzący. Miej to na uwadze podczas ustawiania progów dla alertów.

Rejestrowanie

Omawianie tematu monitorowania nie będzie kompletne bez kilku słów o rejestrowaniu. Podobnie jak w przypadku wielu aplikacji, broker Kafki wypełni dyski komunikatami dziennika w ciągu kilku minut, jeśli na to pozwolisz. Aby uzyskać przydatne informacje z rejestrowania, istotne jest włączenie odpowiednich rejestratorów na właściwych poziomach rejestrowania. Wystarczy rejestrować wszystkie komunikaty na poziomie INFO, by przechwycić znaczną ilość ważnych danych na temat stanu brokera. Przydatne jest jednak wydzielenie z tego kilku rejestratorów w celu zapewnienia czystszej wersji plików dziennika.

Mamy dwa rejestratory zapisujące w osobnych plikach na dysku. Pierwszy to `kafka.controller`, nadal na poziomie INFO. Rejestrator ten służy do zapewniania komunikatów dotyczących konkretnie kontrolera klastra. Kontrolerem zawsze będzie tylko jeden broker, a zatem tylko jeden broker będzie zapisywał w tym rejestratorze. Informacje te obejmują tworzenie i modyfikację tematów, zmiany statusu brokera oraz działania klastra, takie jak wybory repliki preferowanej i przenoszenie partycji. Drugim rejestratorem, który należy wydzielić, jest `kafka.server.ClientQuotaManager`, również na poziomie INFO. Ten rejestrator służy do wyświetlania komunikatów związanych z zarządzaniem kwotami produkowania i konsumowania. Chociaż są to przydatne informacje, nie powinny znajdować się raczej w głównym pliku dziennika brokera.

Pomocne jest także rejestrowanie informacji dotyczących statusu wątków kompaktowania dziennika. Nie ma pojedynczego wskaźnika, który pokazywałby kondycję tych wątków, stąd istnieje ryzyko, że awaria kompaktowania pojedynczej partycji wstrzyma całkowicie i po cichu wątki kompaktowania dziennika. Włączenie rejestratorów `kafka.log.LogCleaner`, `kafka.log.Cleaner` i `kafka.log.LogCleanerManager` na poziomie DEBUG spowoduje rejestrowanie informacji o statusie tych wątków. Obejmuje to informacje o każdej kompaktowanej partycji, w tym rozmiar i liczbę komunikatów w każdej z nich. Przy normalnym działaniu nie jest to dużo rejestrowania, co oznacza, że możesz to włączyć domyślnie i nie zostaniesz przytłoczony danymi.

Istnieje również logowanie, które może być przydatne podczas debugowania problemów z Kafką. Jednym z takich rejestratorów jest `kafka.request.logger` włączony na poziomie DEBUG lub TRACE. Rejestruje on informacje o każdym żądaniu wysłanym do brokera. Na poziomie DEBUG dziennik zawiera punkty końcowe połączenia, czasy żądań oraz informacje podsumowujące. Na poziomie TRACE będzie zawierał także informacje dotyczące tematów i partycji — prawie wszystkie informacje o żądaniach z wyjątkiem samych ładunków komunikatów. Na każdym z tych poziomów rejestrator ten generuje znaczną ilość danych i nie zaleca się włączania go, chyba że jest to konieczne do debugowania.

Monitorowanie klienta

Wszystkie aplikacje wymagają monitorowania. Aplikacje tworzące instancję klienta Kafki, producencą lub konsumentką, mają charakterystyczne dla klienta wskaźniki, które należy przechwytywać. W tym podrozdziale omówimy oficjalne biblioteki klienckie Javy, chociaż inne implementacje powinny mieć dostępne własne wskaźniki.

Wskaźniki producenta

Klient producencki Kafki w dużym stopniu skompresował dostępne wskaźniki, oferując je jako atrybuty niewielkiej liczby ziaren MBeans technologii JMX. Natomiast poprzednia wersja klienta producenckiego (która nie jest już obsługiwana) używała większej liczby MBeans, ale miała więcej szczegółów w wielu wskaźnikach (zapewniając większą liczbę pomiarów percentylowych i różne średnie kroczące). W rezultacie ogólna liczba dostarczanych wskaźników obejmuje większy obszar, ale trudniejsze może być śledzenie wartości odstających.

Wszystkie wskaźniki producenta mają w nazwach ziarna identyfikator klienta producenckiego. W podanych przykładach w miejsce identyfikatora klienta podstawiliśmy *IDKLIENTA*. W przypadku gdy nazwa ziarna zawiera identyfikator brokera, został on zastąpiony przez *IDBROKERA*. Nazwy tematów zostały zastąpione przez *NAZWATEMATU*. Przykład znajdziesz w tabeli 13.19.

Tabela 13.19. Ziarna MBeans wskaźników producenta Kafki

Nazwa	JMX MBean
Ogólne wskaźniki producenta	kafka.producer:type=producer-metrics,client-id=IDKLIENTA
Wskaźniki konkretnego brokera	kafka.producer:type=producer-node-metrics,client-id=IDKLIENTA, ↳ node-id=node-IDBROKERA
Wskaźniki konkretnego tematu	kafka.producer:type=producer-topic-metrics,client-id=IDKLIENTA, ↳ topic=NAZWATEMATU

Każde ziarno wskaźników z tabeli 13.19 ma dostępnych wiele atrybutów opisujących stan producenta. Najbardziej użyteczne atrybuty omówimy w następnym podpunkcie. Zanim przejdziemy dalej, powinienesz upewnić się, że dobrze rozumiesz semantykę działania producenta, co opisaliśmy w rozdziale 3.

Ogólne wskaźniki producenta

Ziarno ogólnych wskaźników producenta zapewnia atrybuty opisujące prawie wszystko, od rozmiarów partii komunikatów po wykorzystanie bufora pamięci. Chociaż wszystkie te pomiary mają swoje zastosowanie w debugowaniu, tylko garstka z nich jest potrzebna na bieżąco, a jedynie kilka powinno być monitorowanych i mieć alarmy. Zwróć uwagę, że chociaż omówimy kilka wskaźników będących średnimi (kończących się na -avg), dostępne są również maksymalne wartości dla każdego wskaźnika (kończące się na -max), które mają ograniczoną użyteczność.

record-error-rate to jeden z atrybutów, dla którego z pewnością należy ustawić alert. Ten wskaźnik zawsze powinien mieć wartość 0, a jeśli ma jakąkolwiek większą wartość, oznacza to, że

producent porzuca komunikaty, które próbuje wysłać do brokerów Kafki. Producent ma skonfigurowaną liczbę ponawiania z określonym interwałem, a po wyczerpaniu prób komunikaty (zwane tutaj **rekordami**) są porzucane. Można śledzić również atrybut `record-retry-rate`, ale jest on mniej istotny niż wskaźnik błędów, ponieważ ponawianie prób jest czymś normalnym.

Kolejnym wskaźnikiem do ustawiania alertów jest `request-latency-avg`. Jest to średni czas, jaki zajmuje wysłanie do brokerów żądania produkcji. Powinienesz być w stanie ustalić wartość bazową dla tego wskaźnika przy normalnym działaniu i ustawić powyżej niej próg alarmowy. Zwiększenie opóźnienia żądania oznacza, że żądania produkcji stają się coraz wolniejsze. Może to być spowodowane problemami z siecią lub wskazywać na problemy brokerów. Tak czy inaczej jest to problem z wydajnością, który spowoduje uruchomienie mechanizmu ciśnienia zwrotnego i wywoła inne problemy w aplikacji produkującej.

Oprócz monitorowania tych kluczowych wskaźników zawsze dobrze jest wiedzieć, jak duży ruch wychodzący komunikatów generuje producent. Trzy dostępne atrybuty przedstawiają trzy różne aspekty tego obrazu. Atrybut `outgoing-byte-rate` opisuje komunikaty w rozmiarach bezwzględnych w bajtach na sekundę. Atrybut `record-send-rate` opisuje ruch w kategoriach liczby komunikatów produkowanych na sekundę. Wreszcie atrybut `request-rate` zapewnia liczbę żądań produkcji wysyłanych do brokerów na sekundę. Pojedyncze żądanie zawiera co najmniej jedną partię, pojedyncza partia zawiera co najmniej jeden komunikat, a każdy komunikat składa się oczywiście z pewnej liczby bajtów. Wszystkie te wskaźniki warto uwzględnić na dashboardzie aplikacji.

Istnieją również wskaźniki opisujące rozmiary rekordów, żądań i partii. Wskaźnik `request-size-avg` podaje w bajtach średni rozmiar żądań produkcji wysyłanych do brokerów. Wskaźnik `batch-size-avg` określa w bajtach średnią wielkość pojedynczej partii komunikatów (która z definicji składa się z komunikatów dla pojedynczej partycji tematu). Natomiast `record-size-avg` pokazuje w bajtach średni rozmiar pojedynczego rekordu. Dla producenta pojedynczego tematu dostarcza to użytecznych informacji na temat produkowanych komunikatów. Dla producentów wielu tematów, takich jak MirrorMaker, jest to mniej informacyjne. Oprócz tych trzech wskaźników istnieje jeszcze wskaźnik `records-per-request-avg`, który opisuje średnią liczbę komunikatów znajdujących się w pojedynczym żądaniu produkcji.

Ostatnim zalecanym ogólnym atrybutem wskaźników producenta jest `record-queue-time-avg`. Jest to pomiar średniej ilości czasu w milisekundach, przez jaki pojedynczy komunikat oczekuje w producencie po wysłaniu go przez aplikację, zanim zostanie faktycznie wyprodukowany do Kafki. Po tym, jak aplikacja wywoła klienta producenckiego w celu wysłania komunikatu (wywołując metodę `send`), producent czeka, aż spełniony zostanie jeden z dwóch wymienionych poniżej warunków:

- producent będzie miał wystarczającą liczbę komunikatów, by zapełnić partię zgodnie z konfiguracją `batch.size`;
- upłynie wystarczająco dużo czasu od wysłania ostatniej partii zgodnie z konfiguracją `linger.ms`.

Spełnienie jednego z tych warunków spowoduje, że klient producencki zamknie bieżącą partię i wyśle ją do brokerów. W celu uproszczenia można przyjąć, że dla zajętych tematów zastosowanie będzie miał pierwszy warunek, podczas gdy dla tematów powolnych — drugi warunek. Wskaźnik `record-queue-time-avg` będzie określał, ile czasu zajmuje średnio wyprodukowanie komunikatu,

a więc będzie pomocny podczas regulowania tych dwóch konfiguracji w celu spełnienia wymagań dotyczących opóźnienia dla danej aplikacji.

Wskaźniki poszczególnych brokerów i tematów

Oprócz ogólnych wskaźników producenta istnieją ziarna wskaźników, które zapewniają ograniczony zestaw atrybutów dla połączenia z każdym brokerem Kafki, a także dla każdego produkowanego tematu. Wskaźniki te są przydatne w niektórych przypadkach do debugowania problemów, ale nie będziesz raczej przeglądał ich na bieżąco. Wszystkie atrybuty tych ziaren są takie same jak opisane wcześniej atrybuty dla ogólnych ziaren producenta i mają również takie samo znaczenie (z tym wyjątkiem, że mają zastosowanie do konkretnego brokera lub tematu).

Najbardziej użytecznym atrybutem dostarczonym przez wskaźniki producenta dla poszczególnych brokerów jest `request-latency-avg`. Wynika to z faktu, że ten wskaźnik będzie przeważnie stabilny (biorąc pod uwagę stabilne grupowanie komunikatów w partie) i nadal może wskazywać na problem z połączeniami z konkretnym brokerem. Pozostałe atrybuty, takie jak `outgoing-byte-rate` i `request-latency-avg`, mogą mieć różne wartości w zależności od tego, dla których partycji broker jest liderem. Oznacza to, że „zakładana” wartość tych wskaźników może szybko ulec zmianie na podstawie stanu klastra Kafki.

Wskaźniki tematów są nieco bardziej interesujące niż wskaźniki poszczególnych brokerów, ale będą przydatne tylko dla producentów, które pracują z więcej niż jednym tematem. Ponadto można ich używać regularnie tylko wtedy, gdy producent nie pracuje z dużą liczbą tematów. MirrorMaker może np. produkować setki lub tysiące tematów. Trudno jest przejrzeć wszystkie te wskaźniki i prawie niemożliwe jest ustalenie dla nich rozsądnych progów alarmowych. Podobnie jak wskaźniki poszczególnych brokerów, również wskaźniki poszczególnych tematów najlepiej wykorzystywać podczas badania konkretnego problemu. Atrybuty `record-send-rate` i `record-error-rate` mogą być używane np. do wyodrębniania porzuconych komunikatów do określonego tematu (lub sprawdzania poprawności we wszystkich tematach). Ponadto istnieje wskaźnik `byte-rate`, który zapewnia dla tematu ogólną prędkość komunikatów w bajtach na sekundę.

Wskaźniki konsumenta

Podobnie jak klient producencki, konsument w Kafce konsoliduje wiele wskaźników w atrybuty zaledwie kilku ziaren wskaźników. Wskaźniki te wyeliminowały ponadto percentyle dla opóźnień i średnie kroczące dla prędkości, które były prezentowane w przestarzałym konsumencie Scali, podobnie jak w przypadku klienta producenckiego. Ponieważ logika konsumowania komunikatów jest nieco bardziej złożona niż zwykle wysyłanie komunikatów do brokerów Kafki, konsument ma jeszcze kilka dodatkowych wskaźników, aby to obsłużyć. Szczegóły dotyczące tych wskaźników zebraliśmy w tabeli 13.20.

Wskaźniki menedżera pobierania

W przypadku klienta konsumenckiego ziarno ogólnych wskaźników konsumenta jest mniej przydatne, ponieważ interesujące nas wskaźniki znajdują się zamiast tego w ziarnach **menedżera pobierania** (ang. *fetch manager*). Ogólne ziarno konsumenta ma wskaźniki dotyczące niskopoziomowych

Tabela 13.20. Ziarna MBeans wskaźników konsumenta Kafki

Nazwa	JMX MBean
Ogólne wskaźniki producenta	<code>kafka.consumer:type=consumer-metrics,client-id=IDKLIENTA</code>
Wskaźniki menedżera pobierania	<code>kafka.consumer:type=consumer-fetch-manager-metrics,</code> <code>↳client-id=IDKLIENTA</code>
Wskaźniki konkretnego tematu	<code>kafka.consumer:type=consumer-fetch-manager-metrics,</code> <code>↳client-id= IDKLIENTA,topic=NAZWATEMATU</code>
Wskaźniki konkretnego brokera	<code>kafka.consumer:type=consumer-node-metrics,client-id= IDKLIENTA,</code> <code>↳nodeid=node-IDBROKERA</code>
Wskaźniki koordynatora	<code>kafka.consumer:type=consumer-coordinator-metrics,</code> <code>↳client-id= IDKLIENTA</code>

operacji sieciowych, ale ziarno menedżera pobierania ma wskaźniki dotyczące prędkości bajtów, żądań i rekordów. Inaczej niż w przypadku klienta producenckiego, wskaźniki dostarczane przez konsument są przydatne do przeglądania, ale nie nadają się do tworzenia alertów.

Jeśli chodzi o menedżer pobierania, jedynym atrybutem, dla którego można skonfigurować monitorowanie i alerty, jest `fetch-latency-avg`. Podobnie jak odpowiednik `request-latency-avg` w kliencie producenckim, ten wskaźnik mówi nam, jak długo trwają żądania pobierania wysyłane do brokerów. Problem z ustawieniem alertów dla tego wskaźnika polega na tym, że opóźnienie jest regulowane przez konfigurację konsumentów `fetch.min.bytes` i `fetch.max.wait.ms`. Powolny temat będzie miał nieregularne opóźnienia, ponieważ czasami broker zareaguje szybko (gdy są dostępne komunikaty), a innym razem nie będzie odpowiadał przez `fetch.max.wait.ms` (gdy nie ma dostępnych komunikatów). Podczas konsumowania tematów, które mają bardziej regularny i obfity ruch komunikatów, ten wskaźnik może być bardziej przydatny.



Chwileczkę, nie ma zaległości?

Najlepsza rada dotycząca wszystkich konsumentów jest taka, że należy monitorować zaległości konsumentów. Dlaczego więc nie zalecamy monitorowania atrybutu `records-lag-max` w ziarnie menedżera pobierania? Wskaźnik ten pokazuje bieżącą zaległość (różnicę między przesunięciem konsumenta a przesunięciem końca dziennika brokera) dla najbardziej opóźnionej partycji.

Problem jest dwojaki: wskaźnik pokazuje zaległość tylko dla jednej partycji i zakłada prawidłowe funkcjonowanie konsumenta. Jeśli nie masz innej opcji, użyj tego atrybutu dla zaległości i skonfiguruj dla niego alertowanie. Najlepszą praktyką jest jednak stosowanie zewnętrznego monitorowania opóźnień, o czym napiszemy w podrozdziale „Monitorowanie opóźnień”.

Aby dowiedzieć się, jak duży ruch komunikatów obsługuje klient konsumencki, należy przechwytywać wskaźnik `bytes-consumed-rate` lub `records-consumed-rate`, a najlepiej oba. Wskaźniki te opisują ruch komunikatów konsumowanych przez tę instancję tego klienta odpowiednio w bajtach na sekundę i komunikatach na sekundę. Niektórzy użytkownicy ustalają minimalne progi alertowania dla tych wskaźników, by otrzymywać powiadomienia, jeśli konsument nie wykonuje wystarczającej pracy. Należy tu jednak zachować ostrożność. Kafka ma na celu rozdzielanie klientów konsumenckich i producenckich, umożliwiając im niezależne działanie. Szybkość, z jaką konsument jest w stanie konsumować komunikaty, często zależy od tego, czy producent działa prawidłowo,

stąd monitorowanie tych wskaźników u konsumenta opiera się na założeniach dotyczących stanu producenta. Może to prowadzić do generowania fałszywych alertów w klientach konsumenckich.

Dobrze jest również zrozumieć relacje między bajtami, komunikatami i żądaniem, a menedżer pobierania zapewnia pomocne wskaźniki. Wskaźnik `fetch-rate` pokazuje liczbę wykonywanych przez konsumenta żądań pobierania na sekundę. Wskaźnik `fetch-size-avg` podaje średni rozmiar tych żądań pobierania w bajtach. Wreszcie wskaźnik `records-per-request-avg` określa średnią liczbę komunikatów w każdym żądaniu pobierania. Zwróć uwagę, że konsument nie dostarcza odpowiednika wskaźnika `record-size-avg` producenta, aby poinformować nas, jaki jest średni rozmiar komunikatu. Jeśli to ważne, konieczne będzie wywnioskowanie tego rozmiaru z innych dostępnych wskaźników lub przechwycenie go w aplikacji po otrzymaniu komunikatów z biblioteki klienta konsumenckiego.

Wskaźniki poszczególnych brokerów i tematów

Podobnie jak w przypadku klienta producenckiego, wskaźniki dostarczane przez klient konsumencki dla każdego z połączeń brokera i każdego konsumowanego tematu są przydatne do debugowania problemów z konsumowaniem, ale prawdopodobnie nie będziesz przeglądał tych pomiarów codziennie. Jak u menedżera pobierania, atrybut `request-latency-avg` dostarczany przez wskaźniki poszczególnych brokerów ma ograniczoną użyteczność zależnie od ruchu komunikatów w konsumowanych tematach. Wskaźniki `incoming-byte-rate` i `request-rate` dzielą wskaźniki konsumowanych komunikatów dostarczane przez menedżer pobierania odpowiednio na pomiar bajtów na sekundę dla każdego brokera i pomiar żądań na sekundę. Można je wykorzystać do odizolowania problemów, które konsument ma z połączeniem z konkretnym brokerem.

Wskaźniki dla poszczególnych tematów dostarczane przez klient konsumencki są przydatne, jeśli konsumowany jest więcej niż jeden temat. W przeciwnym razie wskaźniki te są takie same jak wskaźniki menedżera pobierania i ich gromadzenie jest zbędne. Z drugiej strony, jeśli klient konsumuje wiele tematów (np. MirrorMaker Kafki), przeglądanie tych wskaźników będzie trudne. Jeżeli planujesz ich zbieranie, najważniejszymi wskaźnikami są `bytes-consumed-rate`, `records-consumed-rate` i `fetch-size-avg`. Wskaźnik `bytes-consumed-rate` pokazuje bezwzględny rozmiar bajtów konsumowanych na sekundę dla konkretnego tematu, podczas gdy `records-consumed-rate` pokazuje te same informacje w kategoriach liczby komunikatów. Wskaźnik `fetch-size-avg` zapewnia średni rozmiar w bajtach każdego żądania pobierania dla tematu.

Wskaźniki koordynatora konsumentów

Jak wspomnieliśmy w rozdziale 4., klienci konsumenckie generalnie współpracują ze sobą jako część grupy konsumentów. Grupa prowadzi działania koordynacyjne, takie jak dołączanie członków grupy i wysyłanie brokerom komunikatów heartbeat w celu utrzymania członkostwa w grupie. Koordynator konsumentów jest częścią klienta konsumenckiego odpowiedzialną za obsługę tych zadań i utrzymuje własny zestaw wskaźników. Podobnie jak w przypadku wszystkich wskaźników, dostarczanych jest wiele liczb, ale tylko kilka kluczowych należy regularnie monitorować.

Największym problemem, który mogą napotkać konsumenci w związku z działaniami koordynatora, jest przerwa w konsumowaniu podczas przeprowadzania synchronizacji grupy konsumentów.

Instancje konsumentów w grupie negocjują wtedy, które partycje będą konsumować. W zależności od liczby przetwarzanych partycji może to zająć trochę czasu. Koordynator zapewnia atrybut wskaźnika `sync-time-avg`, który jest określaną w milisekundach średnią ilością czasu, jaki zajmuje aktywność synchronizacji. Przydatne jest także przechwytywanie atrybutu `sync-rate`, który jest liczbą synchronizacji grupy na sekundę. W przypadku stabilnej grupy konsumentów przez większość czasu ta liczba powinna wynosić zero.

Konsument musi zatwierdzać przesunięcia, aby tworzyć punkty kontrolne swoich postępów w konsumowaniu komunikatów w sposób automatyczny w regularnych odstępach czasu lub za pomocą ręcznych punktów kontrolnych uruchamianych w kodzie aplikacji. Te zatwierdzenia są zasadniczo żadaniami produkcji (choć mają swój własny typ żądania), gdyż zatwierdzenie przesunięcia jest komunikatem produkowanym do specjalnego tematu. Koordynator konsumentów zapewnia atrybut `commit-latency-avg`, który mierzy średnią ilość czasu, jaki zajmują zatwierdzenia przesunięć. Należy monitorować tę wartość, podobnie jak w przypadku opóźnień żądań w producencie. Możliwe powinno być ustalenie oczekiwanej wartości odniesienia dla tego wskaźnika i ustalenie rozsądnych progów alarmowania powyżej tej wartości.

Ostatnim przydatnym do gromadzenia wskaźnikiem koordynatora jest `assigned-partitions`. Jest to liczba partycji przypisanych klientowi konsumenckiemu (jako pojedynczej instancji w grupie konsumentów) do przetwarzania. Bywa to pomocne, gdyż dzięki porównaniu wartości tego wskaźnika ze wszystkich klientów konsumenckich w grupie można określić rozłożenie obciążenia w całej grupie konsumentów. Można to wykorzystać do identyfikacji nierównowagi, która może być spowodowana problemami w algorytmie używanym przez koordynatora konsumentów do dystrybucji partycji do poszczególnych członków grupy.

Kwoty

Apache Kafka ma możliwość ograniczania (dławienia) żądań klientów w celu uniemożliwienia jednemu klientowi przytłoczenia całego klastra. Jest to konfigurowalne zarówno dla klientów produkcyjnych, jak i konsumenckich i wyrażane w kategoriach dozwolonej ilości ruchu w bajtach na sekundę wysyłanego z konkretnego identyfikatora klienta do konkretnego brokera. Dostępna jest konfiguracja brokera, która ustawia wartość domyślną dla wszystkich klientów, ale możliwe są także nadpisanie dla poszczególnych klientów, które można ustawiać dynamicznie. Gdy broker obliczy, że klient przekroczył swoją kwotę, spowalnia go przez wstrzymanie wysłania odpowiedzi do klienta tak długo, aby utrzymać ten klient poniżej kwoty.

Broker Kafki nie używa w odpowiedzi kodów błędów, aby wskazać, że klient jest dławiony. Oznacza to, że bez monitorowania wskaźników dostarczanych w celu pokazania ilości czasu, przez który klient jest dławiony, dla aplikacji nie jest oczywiste, iż ma miejsce dławienie. Wskaźniki, które należy monitorować, przedstawiliśmy w tabeli 13.21.

Kwoty nie są domyślnie włączone w brokerach Kafki, ale bezpiecznie jest monitorować te wskaźniki niezależnie od tego, czy korzysta się z kwot. Monitorowanie ich jest dobrą praktyką, ponieważ mogą zostać w pewnym momencie włączone i łatwiej jest monitorować je od początku, a nie dodawać wskaźniki później.

Tabela 13.21. Wskaźniki do monitorowania

Klient	Nazwa ziarna
Konsument	ziarno kafka.consumer:type=consumer-fetch-manager-metrics,client-id=IDKLIENTA, atrybut fetch-throttle-time-avg
Producent	ziarno kafka.producer:type=producer-metrics,client-id=IDKLIENTA, atrybut producethrottle-time-avg

Monitorowanie opóźnień

W przypadku konsumentów Kafki najważniejszą rzeczą do monitorowania są opóźnienia (zaległości) konsumentów. Jest to mierzona liczbą komunikatów różnica między ostatnim komunikatem wyprodukowanym do określonej partycji a ostatnim komunikatem przetworzonym przez konsument. Chociaż teoretycznie powinniśmy omówić ten temat w poprzednim podrozdziale, poświęconym m.in. monitorowaniu klientów konsumentów, jest to jeden z takich przypadków, w których zewnętrzny monitoring znacznie przekracza to, co jest udostępniane przez sam klient. Jak wspomniano wcześniej, w kliencie konsumentów dostępny jest wskaźnik opóźnień, ale korzystanie z niego jest problematyczne. Reprezentuje on tylko jedną partycję, tę z największym opóźnieniem, stąd nie pokazuje dokładnie, jak duże zaległości ma konsument. Ponadto wymagane jest poprawne działanie konsumenta, ponieważ wskaźnik jest obliczany przez konsumenta przy każdym żądaniu pobrania. Jeśli konsument uległ awarii lub przeszedł w tryb offline, wskaźnik jest niedokładny lub niedostępny.

Preferowaną metodą monitorowania opóźnień konsumentów jest posiadanie zewnętrznego procesu, który może obserwować zarówno stan partycji w brokerze, śledząc przesunięcie ostatnio wyprodukowanego komunikatu, jak i stan konsumenta, śledząc ostatnie przesunięcie zatwierdzone przez grupę konsumentów dla tej partycji. Zapewnia to obiektywny obraz, który może być aktualizowany niezależnie od statusu samego konsumenta. Kontrola ta musi być przeprowadzana dla każdej partycji przetwarzanej przez daną grupę konsumentów. W przypadku dużego konsumenta, takiego jak MirrorMaker, może to oznaczać dziesiątki tysięcy partycji.

W rozdziale 12. opisaliśmy korzystanie z narzędzi wiersza poleceń w celu pozyskiwania informacji o grupie konsumentów, w tym o zatwierdzonych przesunięciach oraz opóźnieniach. Monitorowanie tego rodzaju opóźnień ma jednak własne problemy. Przede wszystkim trzeba określić, jakie są rozsądne opóźnienia dla poszczególnych partycji. Temat, który otrzymuje 100 komunikatów na godzinę, będzie potrzebował innego progu niż temat otrzymujący 100 000 komunikatów na sekundę. Następnie trzeba zapewnić przetwarzanie tych wszystkich wskaźników opóźnień przez system monitorowania i ustawić na nich alerty. Jeżeli masz grupę konsumentów, która konsumuje 100 000 partycji w ponad 1500 tematach, może to być dość trudne zadanie.

Jednym ze sposobów zmniejszenia złożoności monitorowania grup konsumentów jest użycie narzędzia Burrow (<https://oreil.ly/supY1>). Jest to aplikacja opracowana przez LinkedIn i udostępniana na licencji *open source*. Zapewnia monitorowanie statusu konsumentów przez gromadzenie informacji o opóźnieniach dla wszystkich grup konsumentów w klastrze i obliczanie pojedynczego statusu dla każdej grupy, informującego, czy grupa konsumentów działa prawidłowo, pozostaje w tyle

lub jest zablokowana albo całkowicie zatrzymana. Nie wymaga w tym celu progów, monitorując postępy grupy konsumentów w przetwarzaniu komunikatów, chociaż można również uzyskać opóźnienie komunikatów jako liczbę bezwzględną. Na blogu LinkedIn Engineering (<http://bit.ly/2sanKZb>) znajdziesz szersze omówienie metodologii działania Burrowa. Wdrożenie tego narzędzia może być prostym sposobem na zapewnienie monitorowania dla wszystkich konsumentów w klastrze, a także w wielu klastrach, i można łatwo zintegrować Burrow z istniejącym systemem monitorowania i ostrzegania.

Jeśli nie ma innej opcji, wskaźnik `records-lag-max` z klienta konsumenckiego zapewni przynajmniej częściowy obraz statusu konsumenta. Stanowczo zaleca się jednak korzystanie z zewnętrznego systemu monitorowania, takiego jak Burrow.

Monitorowanie kompleksowe

Innym rodzajem monitorowania zewnętrznego, które jest zalecane do określania, czy klastry Kafka działają prawidłowo, jest system monitorowania kompleksowego, zapewniający kliencki punkt widzenia na temat kondycji klastra Kafka. Klienci konsumenckie i producenckie mają wskaźniki, które mogą określać, że może występować jakiś problem z klastrem Kafka, ale może to być gra w zgadywanie, czy zwiększone opóźnienie jest spowodowane problemem z klientem, siecią lub samą Kafką. Ponadto oznacza to, że jeśli jesteś odpowiedzialny za prowadzenie klastra Kafka, a nie klientów, będziesz musiał monitorować także wszystkie klienty. Tak naprawdę musisz wiedzieć dwie rzeczy:

- Czy można produkować komunikaty do klastra Kafka?
- Czy można konsumować komunikaty z klastra Kafka?

Najlepiej, gdybyś był w stanie monitorować te kwestie indywidualnie dla poszczególnych tematów. Jednak w większości sytuacji nierozsądne jest wstrzykiwanie w tym celu syntetycznego ruchu do każdego tematu. Można natomiast zapewnić odpowiedzi na te pytania przynajmniej dla każdego brokera w klastrze, a właśnie to oferuje Xinfra Monitor (<https://oreil.ly/QqXD9>), dawniej znany jako Kafka Monitor. Narzędzie to, udostępniane na licencji *open source* przez zespół Kafka w LinkedInie, nieustannie produkuje i konsumuje dane z tematu, który jest rozłożony na wszystkie brokery w klastrze. Mierzy dostępność żądań produkcji i konsumowania dla każdego brokera oraz całkowite opóźnienie od wyprodukowania do skonsumowania. Ten rodzaj monitorowania jest nieoceniony pod tym względem, że umożliwia zewnętrzne weryfikowanie, czy klastr Kafka działa zgodnie z założeniem, gdyż podobnie jak w przypadku monitorowania opóźnień konsumentów, broker Kafka nie może zgłaszać, czy klienty są w stanie prawidłowo korzystać z klastra.

Podsumowanie

Monitorowanie jest kluczowym aspektem prawidłowego uruchamiania platformy Apache Kafka, co wyjaśnia, dlaczego tak wiele zespołów poświęca znaczną ilość czasu na doskonalenie tych elementów. Wiele organizacji używa Kafka do obsługi przepływów danych w skali petabajtowej. Zapewnienie, że dane się nie zatrzymają, a komunikaty nie zostaną utracone, jest kluczowym wymogiem

biznesowym. Naszym obowiązkiem jest również pomaganie użytkownikom w monitorowaniu sposobu wykorzystywania Kafki przez ich aplikacje i dostarczanie w tym celu niezbędnych wskaźników.

W tym rozdziale omówiliśmy podstawy monitorowania aplikacji Javy, a konkretnie aplikacji Kafki. Przyjrzelśmy się sporemu podzbirowi wskaźników dostępnych w brokerze Kafki, zahaczając również o monitorowanie Javy i systemu operacyjnego oraz rejestrowanie. Następnie opisaliśmy szczegółowo monitorowanie dostępne w bibliotekach klienckich Kafki, w tym monitorowanie kwot. Na koniec przyjrzelśmy się wykorzystaniu zewnętrznych systemów do monitorowania opóźnień konsumentów oraz kompleksowej dostępności klastrów. Choć nie jest to wyczerpująca lista dostępnych wskaźników, w tym rozdziale omówiono te najbardziej istotne, na które należy zwracać uwagę.

Przetwarzanie strumieniowe

Kafka była tradycyjnie postrzegana jako potężna magistrala komunikatów, zdolna do dostarczania strumieni zdarzeń, ale bez możliwości przetwarzania lub transformacji. Niezawodne funkcjonalności dostarczania strumieni sprawiają, że Kafka jest idealnym źródłem danych dla systemów przetwarzania strumieniowego. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza i wiele innych systemów przetwarzania strumieniowego zostało zbudowanych z Kafką jako niezrządkiem jedynym niezawodnym źródłem danych.

Podczas gdy rosła popularność platformy Apache Kafka, najpierw jako prostej magistrali komunikatów, a później jako systemu integracji danych, wiele firm miało już systemy zawierające liczne strumienie interesujących danych przechowywane przez długi czas i doskonale uporządkowane, które czekały tylko na pojawienie się jakiegoś frameworku przetwarzania strumieniowego do ich przetworzenia. Innymi słowy: podobnie jak przetwarzanie danych było znacznie utrudnione przed wynalezieniem baz danych, przetwarzanie strumieniowe było wstrzymywane przez brak platformy przetwarzania strumieniowego.

Od wersji 0.10.0 Kafka nie tylko zapewnia niezawodne źródło strumieni danych dla każdego popularnego frameworku przetwarzania strumieniowego, ale zawiera również w ramach swojej kolekcji bibliotek klienckich potężną bibliotekę przetwarzania strumieniowego, zwaną Kafka Streams (lub czasami Streams API). Dzięki temu programiści mogą konsumować, przetwarzać i produkować zdarzenia we własnych aplikacjach bez korzystania z zewnętrznych frameworków przetwarzania.

Ten rozdział zaczniemy od wyjaśnienia, co rozumiemy przez przetwarzanie strumieniowe (gdyż termin ten jest często źle pojmowany), a następnie omówimy kilka podstawowych koncepcji przetwarzania strumieniowego oraz wzorce projektowe, które są wspólne dla wszystkich systemów przetwarzania strumieniowego. Potem przyjrzymy się bliżej bibliotece przetwarzania strumieniowego platformy Apache Kafka — jej założeniom i architekturze. Przedstawimy przykład wykorzystania biblioteki Kafka Streams do obliczania średniej kroczącej cen akcji. Dalej opiszemy kolejne przykłady dobrych przypadków użycia dla przetwarzania strumieniowego i zakończymy rozdział, podając kilka kryteriów, które można wykorzystać przy wyborze frameworku przetwarzania strumieniowego do zastosowania z Apache Kafką.

Rozdział ma być krótkim wprowadzeniem do rozległego i fascynującego świata przetwarzania strumieniowego i biblioteki Kafka Streams. Powstało na ten temat wiele książek.

Oto niektóre pozycje opisujące podstawowe pojęcia przetwarzania strumieniowego z perspektywy architektury danych:

- Martin Kleppmann, *Making Sense of Stream Processing* (wyd. O'Reilly). Omawia korzyści płynące z przeprojektowywania aplikacji na aplikacje przetwarzania strumieniowego oraz sposoby przebudowania architektury danych wokół idei strumieni zdarzeń.
- Tyler Akidau, Slava Chernyak i Reuven Lax, *Streaming Systems* (O'Reilly). Doskonałe ogólne wprowadzenie do tematu przetwarzania strumieniowego i niektórych podstawowych pomysłów z tej dziedziny.
- James Urquhart, *Flow Architectures* (O'Reilly). Ta książka jest skierowana do dyrektorów ds. technologii i omawia implikacje przetwarzania strumieniowego dla firmy.

Pozycje poświęcone konkretnym frameworkom:

- Mitch Seymour, *Mastering Kafka Streams and ksqlDB* (O'Reilly).
- William P. Bejeck Jr., *Kafka Streams in Action* (Manning).
- Fabian Hueske i Vasiliki Kalavri, *Stream Processing with Apache Flink* (O'Reilly).
- Gerard Maas i Francois Garillot, *Stream Processing with Apache Spark* (O'Reilly).

Kafka Streams to stale ewoluujący framework. Każde główne wydanie modernizuje interfejsy API i modyfikuje semantykę. W tym rozdziale opiszemy API i semantykę dla wersji 2.8 platformy Apache Kafka. Będziemy unikać korzystania z jakiegokolwiek interfejsu API, który ma zostać wycofany w wersji 3.0, a nasza dyskusja na temat semantyki łączenia oraz obsługi znaczników czasu nie będzie obejmowała żadnych zmian planowanych w wersji 3.0.

Czym jest przetwarzanie strumieniowe?

Istnieje mnóstwo nieporozumień co do znaczenia pojęcia przetwarzania strumieniowego. Wiele definicji łączy w sobie szczegóły implementacji, wymagania dotyczące wydajności, modele danych i dużo innych aspektów inżynierii oprogramowania. Podobna sytuacja miała miejsce w świecie relacyjnych baz danych — abstrakcyjne definicje modelu relacyjnego zostały na zawsze uwikłane w szczegóły implementacji i specyficzne ograniczenia popularnych silników bazodanowych.

Świat przetwarzania strumieniowego wciąż ewoluuje, a sam fakt, że jakaś popularna implementacja działa w określony sposób lub ma konkretne ograniczenia, nie oznacza, iż szczegóły te są nieodłączną częścią przetwarzania strumieni danych.

Zacznijmy od początku. Czym jest strumień danych (nazywany również **strumieniem zdarzeń** lub **strumieniowaniem danych**)? Przede wszystkim **strumień danych** jest abstrakcją reprezentującą nieograniczony zbiór danych. **Nieograniczony** oznacza: nieskończony i stale rosnący. Zbiór danych jest nieograniczony, ponieważ z biegiem czasu pojawiają się nowe rekordy. Definicja ta jest używana przez firmy Google (<http://oreil.ly/1p1AKux>), Amazon (<http://amzn.to/2sfc334>) i prawie wszystkich innych.

Zwróć uwagę, że ten prosty model (strumień zdarzeń) może być używany do reprezentowania prawie każdej działalności biznesowej, którą chcemy analizować. Możemy mieć strumień transakcji

kartami kredytowymi, obrotu akcjami, dostarczania paczek, zdarzeń sieciowych przechodzących przez przełącznik, zdarzeń zgłaszanych przez czujniki w sprzęcie produkcyjnym, wysyłanych e-maili, ruchów w grze itd. Lista przykładów jest nieskończona, ponieważ prawie wszystko można postrzegać jako sekwencję zdarzeń.

Oprócz nieograniczonego charakteru model strumieni zdarzeń ma jeszcze kilka innych atrybutów:

Strumienie zdarzeń są uporządkowane

Inherentną cechą strumieni zdarzeń jest to, że kolejne zdarzenia mają miejsce przed innymi zdarzeniami lub po nich. Stanie się to oczywiste, gdy przyjrzymy się zdarzeniom finansowym. Sekwencja, w której najpierw wpłacasz pieniądze na swoje konto, a następnie je wydajesz, różni się od sekwencji, w której najpierw wydajesz pieniądze, a następnie pokrywasz swój dług, dokonując wpłaty. Ta druga sekwencja będzie obciążać rachunek opłatami za debet, a ta pierwsza nie. Zwróć uwagę, że jest to jedna z różnic między strumieniem zdarzeń a tablicą bazy danych — rekordy w tablicy są zawsze traktowane jako nieuporządkowane, a klauzula `order by` z SQL-a nie jest częścią modelu relacyjnego; została dodana, aby pomóc w generowaniu raportów.

Niemutowalne rekordy danych

Zdarzenia, które już wystąpiły, nigdy nie mogą być modyfikowane. Transakcja finansowa, która została anulowana, nie znika. Zamiast tego w strumieniu zapisywane jest dodatkowe zdarzenie rejestrujące anulowanie poprzedniej transakcji. Kiedy klient zwraca towar do sklepu, nie usuwamy faktu, że towar został mu wcześniej sprzedany, a zamiast tego rejestrujemy zwrot jako dodatkowe zdarzenie. To kolejna różnica między strumieniem danych a tablicą bazy danych — możemy usuwać rekordy z tablicy lub aktualizować je, ale są to wszystkie dodatkowe transakcje, które występują w bazie danych, i jako takie mogą być rejestrowane w strumieniu zdarzeń, który rejestruje wszystkie transakcje. Jeżeli jesteś zaznajomiony z dziennikami binarnymi, rejestrowaniem z wyprzedzeniem lub dziennikami ponawiania z baz danych, możesz zobaczyć, że jeśli wstawimy do tablicy jakiś rekord, a później go usuniemy, tablica nie będzie już zawierała rekordu, ale dziennik ponawiania będzie zawierał dwie transakcje — wstawienie i usunięcie.

Strumienie zdarzeń można odtwarzać ponownie

Jest to pożądana właściwość. Chociaż łatwo można sobie wyobrazić strumienie nieodtwarzalne (pakiety TCP przesyłane strumieniowo przez gniazdo są na ogół nieodtwarzalne), dla większości aplikacji biznesowych kluczowa jest możliwość ponownego odtwarzania nieprzetworzonego strumienia zdarzeń, które wystąpiły miesiące (a czasami lata) temu. Jest to wymagane w celu korygowania błędów, próbowania nowych metod analizy lub przeprowadzania audytów. Dlatego uważamy, że dzięki Kafce przetwarzanie strumieniowe odniosło tak duży sukces w nowoczesnych biznesach — pozwala na przechwytywanie i odtwarzanie strumienia zdarzeń. Bez tej funkcjonalności przetwarzanie strumieniowe nie byłoby niczym więcej niż zabawką laboratoryjną dla naukowców zajmujących się danymi.

Warto zauważyć, że ani definicja strumieni zdarzeń, ani atrybuty, które później wymieniliśmy, nie mówią nic o danych zawartych w zdarzeniach lub liczbie zdarzeń na sekundę. Dane różnią się w zależności od systemu — zdarzenia mogą być niewielkie (czasami tylko kilka bajtów) lub bardzo duże (komunikaty XML-a z wieloma nagłówkami); mogą to być również całkowicie pozbawione

struktury pary klucz-wartość, częściowo ustrukturyzowane komunikaty JSON lub ustrukturyzowane komunikaty Avry lub Protobufa. Chociaż często zakłada się, że strumienie danych są „dużymi zbiorami danych” (big data) i obejmują miliony zdarzeń na sekundę, te same techniki, które będziemy omawiać, mają zastosowanie również (i często z lepszym efektem) do mniejszych strumieni zdarzeń z zaledwie kilkoma zdarzeniami na sekundę lub minutę.

Skoro wiesz już, czym są strumienie zdarzeń, nadszedł czas, by zająć się koncepcją przetwarzania strumieniowego. Przetwarzanie strumieniowe odnosi się do trwającego przetwarzania co najmniej jednego strumienia zdarzeń. Przetwarzanie strumieniowe jest paradygmatem programowania, podobnie jak wzorzec żądanie-odpowiedź i przetwarzanie wsadowe. Porównajmy różne paradygmaty programowania, aby lepiej zrozumieć, w jaki sposób przetwarzanie strumieniowe wpasowuje się w architektury oprogramowania:

Żądanie-odpowiedź

Jest to paradygmat najmniejszych opóźnień, z czasem reakcji wynoszącym od mniej niż milisekundy do kilku milisekund, zwykle z oczekiwaniem, że czasy odpowiedzi będą bardzo spójne. Tryb przetwarzania zwykle blokuje wykonywanie — aplikacja wysyła żądanie i czeka na odpowiedź systemu przetwarzania. W świecie baz danych paradygmat ten znany jest jako **OLTP** (ang. *Online Transaction Processing*). W tym paradygmacie działają zazwyczaj systemy punktów sprzedaży, przetwarzania kart kredytowych i śledzenia czasu.

Przetwarzanie wsadowe

Jest to opcja z dużymi opóźnieniami i wysoką przepustowością. System przetwarzania budzi się o ustalonych porach — codziennie o 2:00, co godzinę itp. Odczytuje wszystkie wymagane dane wejściowe (albo wszystkie dane dostępne od ostatniego wykonywania, wszystkie dane od początku miesiąca itp.), zapisuje wszystkie wymagane dane wyjściowe i przechodzi w stan uśpienia do czasu następnego zaplanowanego uruchomienia. Czas przetwarzania waha się od minut do godzin, a podczas przeglądania wyników użytkownicy mają świadomość, że odczytują przestarzałe dane. W świecie baz danych są to hurtownie danych i systemy wywiadu biznesowego — dane są ładowane w dużych partiach raz dziennie, generowane są raporty, a użytkownicy analizują te same raporty aż do następnego załadowania danych. Ten paradygmat często charakteryzuje się dużą skutecznością i ekonomią skali, ale w ostatnich latach przedsiębiorstwa potrzebują danych dostępnych w krótszych ramach czasowych, aby proces decyzyjny był bardziej terminowy i skuteczny. Wywiera to ogromną presję na systemy, które zostały napisane w celu wykorzystania ekonomii skali, a nie w celu zapewnienia sprawozdawczości o małym opóźnieniu.

Przetwarzanie strumieniowe

Jest to opcja ciągła i nieblokująca wykonywania. Przetwarzanie strumieniowe wypełnia lukę między paradygmatem żądanie-odpowiedź, gdzie czekamy na zdarzenia, które są przetwarzane w ciągu 2 milisekund, a światem przetwarzania wsadowego, w którym dane są przetwarzane raz dziennie, a zakończenie procesu trwa 8 godzin. Większość procesów biznesowych nie wymaga natychmiastowej reakcji w ciągu milisekund, ale nie może również czekać do następnego dnia. Zwykle odbywa się to w sposób ciągły i dopóki raporty biznesowe są stale aktualizowane, a linia aplikacji biznesowych może stale reagować, przetwarzanie może być kontynuowane i nikt nie będzie musiał czekać kilku milisekund, aż otrzyma odpowiedź. Procesy biznesowe,

takie jak ostrzeżenie o podejrzanych transakcjach kredytowych lub aktywności w sieci, dostawianie cen w czasie rzeczywistym na podstawie podaży i popytu lub śledzenie dostaw paczek, w naturalny sposób nadają się do ciągłego, ale nieblokującego przetwarzania.

Należy zwrócić uwagę, że definicja nie wskazuje żadnych konkretnych frameworków, interfejsów API ani funkcjonalności. Dopóki w sposób ciągły odczytujemy dane z nieograniczonego zbioru danych, przetwarzamy je w jakiś sposób i emitujemy dane wyjściowe, jest to przetwarzanie strumieniowe. Przetwarzanie musi być jednak ciągłe i trwające cały czas. Proces, który rozpoczyna się każdego dnia o 2:00, odczytuje 500 rekordów ze strumienia, wysyła wynik i zostaje uśpiony, nie do końca spełnia założenia przetwarzania strumieniowego.

Koncepcje przetwarzania strumieniowego

Przetwarzanie strumieniowe jest bardzo podobne do wszelkiego rodzaju przetwarzania danych — piszemy kod, który odbiera dane, robi coś z danymi (kilka przekształceń, agregacji, wzbogaceń itp.), a następnie umieszczamy gdzieś wynik. Istnieją jednak pewne kluczowe koncepcje, które są unikatowe dla przetwarzania strumieniowego i często powodują dezorientację, gdy osoba z doświadczeniem w przetwarzaniu danych próbuje pierwszy raz pisać aplikacje do przetwarzania strumieniowego. Przyjrzyjmy się kilku z tych koncepcji.

Topologia

Aplikacja przetwarzania strumieniowego obejmuje co najmniej jedną topologię przetwarzania. Topologia przetwarzania rozpoczyna się od jednego lub kilku strumieni źródłowych, które są przekazywane przez graf procesorów strumieniowych połączonych strumieniami zdarzeń aż do zapisania wyników w co najmniej jednym ujściu strumieni. Każdy procesor strumieniowy jest krokiem obliczeniowym stosowanym do strumienia zdarzeń w celu przekształcenia zdarzeń. Przykładami procesorów strumieniowych, których będziemy używać w tym rozdziale, są filtr, licznik, grupowanie i połączenie lewostronne. Często wizualizujemy aplikacje przetwarzania strumieniowego, rysując węzły przetwarzania i łącząc je ze strzałkami, aby pokazać, jak zdarzenia przepływają z jednego węzła do drugiego, gdy aplikacja przetwarza dane.

Czas

Czas jest prawdopodobnie najważniejszą koncepcją w przetwarzaniu strumieniowym i często najbardziej mylącą. Abyś miał wyobrażenie o tym, jak skomplikowany może stać się czas podczas omawiania systemów rozproszonych, polecamy lekturę doskonałego artykułu Justina Sheehy'ego *There Is No Now* (<http://bit.ly/2rXXdLr>). W kontekście przetwarzania strumieniowego posiadanie wspólnego pojęcia czasu ma kluczowe znaczenie, ponieważ większość aplikacji strumieniowych wykonuje operacje w oknach czasowych. Nasza aplikacja strumieniowa może obliczać np. 5-minutową średnią kroczącą cen akcji. W takim przypadku musimy wiedzieć, co zrobić, gdy jeden z naszych producentów zostanie wyłączony na 2 godziny z powodu problemów z siecią i powróci z zaległymi danymi z tych 2 godzin — większość danych będzie miała znaczenie dla dawno minionych 5-minutowych okien, dla których wynik został już obliczony i zapisany.

Systemy przetwarzania strumieniowego zazwyczaj odnoszą się do kilku pojęć czasu. Oto one:

Czas zdarzenia

Jest to czas pojawiania się śledzonych zdarzeń i tworzenia rekordów, np.: czas wykonania pomiaru, sprzedania przedmiotu w sklepie, wejścia na stronę internetową przez użytkownika. Od wersji 0.10.0 Kafka automatycznie dodaje bieżący czas do rekordów producenta w momencie ich tworzenia. Jeśli nie jest to zgodne z pojęciem **czasu zdarzenia** używanym w aplikacji, np. gdy rekord Kafki jest tworzony na podstawie rekordu bazy danych jakiś czas po wystąpieniu zdarzenia, zalecamy dodanie czasu zdarzenia jako pola w samym rekordzie, aby oba znaczniki czasu były dostępne w późniejszym przetwarzaniu. Czas zdarzenia to zazwyczaj czas, który ma największe znaczenie podczas przetwarzania danych strumieniowych.

Czas dodania do dziennika

Jest to czas, w którym wydarzenie dotarło do brokera Kafki i zostało tam zapisane, zwany również **czasem pozyskania** (ang. *ingestion time*). Od wersji 0.10.0 brokery Kafki automatycznie dodają ten czas do otrzymanych rekordów, jeśli Kafka została tak skonfigurowana lub rekordy pochodzą od starszych producentów i nie zawierają znaczników czasu. To pojęcie czasu jest z reguły mniej istotne dla przetwarzania strumieniowego, ponieważ zwykle interesują nas czasy występowania zdarzeń. Gdy będziemy obliczać np. liczbę urządzeń produkowanych dziennie, będziemy chcieli policzyć urządzenia, które faktycznie zostały wyprodukowane tego dnia, nawet jeśli wystąpiły problemy z siecią, a zdarzenie dotarło do Kafki dopiero następnego dnia. Jednak w przypadkach, gdy rzeczywisty czas zdarzenia nie został zarejestrowany, czas dodania dziennika może być nadal spójnie stosowany, ponieważ nie zmienia się po utworzeniu rekordu i przy założeniu braku opóźnień w potoku może być rozsądnym przybliżeniem czasu zdarzenia.

Czas rozpoczęcia przetwarzania

Jest to czas, w którym aplikacja przetwarzania strumieniowego otrzymała zdarzenie w celu wykonania pewnych obliczeń. Może to mieć miejsce po upływie milisekund, godzin lub dni po wystąpieniu zdarzenia. To pojęcie czasu powoduje przypisywanie różnych znaczników czasu do tego samego zdarzenia w zależności od tego, kiedy każda aplikacja przetwarzania strumieniowego odczytała zdarzenie. Może się nawet różnić dla dwóch wątków w tej samej aplikacji! Dlatego to pojęcie czasu jest bardzo zawodne i najlepiej go unikać.

Kafka Streams przypisuje czas do każdego zdarzenia na podstawie interfejsu `TimestampExtractor`. Programiści aplikacji Kafka Streams mogą używać różnych implementacji tego interfejsu, które mogą korzystać z jednej z trzech opisanych wcześniej semantyk czasowych lub z zupełnie innego znacznika czasu, w tym ze znacznika czasu wyodrębnionego z zawartości samego zdarzenia.

Kiedy Kafka Streams zapisuje dane wyjściowe w temacie Kafki, przypisuje znacznik czasu do każdego zdarzenia, bazując na kilku zasadach:

- Gdy rekord wyjściowy mapuje się bezpośrednio na rekord wejściowy, rekord wyjściowy będzie używał tego samego znacznika czasu co rekord wejściowy.
- Gdy rekord wyjściowy jest wynikiem agregacji, znacznikiem czasu rekordu wyjściowego będzie maksymalny znacznik czasu użyty w agregacji.

- Gdy rekord wyjściowy jest wynikiem połączenia dwóch strumieni, znacznikiem czasu rekordu wyjściowego będzie największy znacznik czasu z dwóch połączonych rekordów. W przypadku łączenia strumienia i tablicy używany jest znacznik czasu z rekordu strumienia.
- Jeśli rekord wyjściowy został wygenerowany przez funkcję biblioteki Kafka Streams, taką jak `punctuate()` generująca dane w określonym harmonogramie niezależnie od danych wejściowych, wyjściowy znacznik czasu będzie zależał od bieżących czasów wewnętrznych aplikacji przetwarzania strumieniowego.

Gdy zamiast z DSL programiści korzystają z niższego poziomu API przetwarzania biblioteki Kafka Streams, mają do dyspozycji interfejsy API do bezpośredniej manipulacji znacznikami czasu rekordów, dzięki czemu mogą zaimplementować taką semantykę znacznika czasu, która będzie zgodna z wymaganą logiką biznesową aplikacji.



Zwróć uwagę na strefę czasową

Podczas pracy z czasem należy pamiętać o strefach czasowych. Cały potok danych powinien zostać znormalizowany dla jednej strefy czasowej. W przeciwnym razie wyniki operacji strumieniowych będą mylące i często bez znaczenia. Jeśli musisz obsługiwać strumienie danych z różnymi strefami czasowymi, przed rozpoczęciem wykonywania operacji na oknach czasowych upewnij się, że możesz przekonwertować zdarzenia na jedną strefę czasową. Często oznacza to zapisywanie strefy czasowej w samym rekordzie.

Stan

Dopóki wystarczy przetwarzać każde zdarzenie indywidualnie, przetwarzanie strumieniowe jest bardzo prostą czynnością. Jeśli musimy odczytywać z Kafki np. strumień transakcji zakupów internetowych, wyszukiwać transakcje powyżej 10 000 złotych i wysłać e-mail do odpowiedniego sprzedawcy, prawdopodobnie będziemy mogli zapisać to w zaledwie kilku liniach kodu przy użyciu konsumenta Kafki i biblioteki SMTP.

Przetwarzanie strumieniowe staje się naprawdę interesujące, gdy mamy operacje obejmujące wiele zdarzeń: zliczanie zdarzeń według typu, średnie kroczące, łączenie dwóch strumieni w celu tworzenia wzbogaconego strumienia informacji itp. W takich przypadkach nie wystarczy przyglądać się poszczególnym zdarzeniom indywidualnie. Trzeba śledzić więcej informacji — liczby zdarzeń każdego typu w danej godzinie, wszystkie zdarzenia wymagające połączenia, sumy, średnie itp. Te informacje nazywamy **stanem**.

Często kuszące jest zapisywanie stanu w zmiennych lokalnych dla aplikacji przetwarzającej strumień, takich jak prosta tabela mieszająca do przechowywania ruchomych liczników. W rzeczywistości robiliśmy tak w wielu przykładach w tej książce. Nie jest to jednak niezawodne podejście do zarządzania stanem w przetwarzaniu strumieniowym, ponieważ gdy aplikacja przetwarzania strumieniowego zostanie zatrzymana lub ulegnie awarii, stan zostanie utracony, co zmieni wyniki. Zwykle nie jest to pożądaný rezultat, dlatego należy zachować ostrożność, utrwalając ostatni stan i odzyskując go w przypadku zrestartowania aplikacji.

Przetwarzanie strumieniowe odwołuje się do kilku rodzajów stanu:

Stan lokalny lub wewnętrzny

Stan, do którego dostęp ma tylko określona instancja aplikacji przetwarzania strumieniowego. Jest on zwykle utrzymywany i zarządzany przez osadzoną, działającą w pamięci bazę danych uruchomioną w aplikacji. Zaletą stanu lokalnego jest to, że jest on niezwykle szybki. Jego wada polega na tym, że jesteśmy ograniczeni do ilości dostępnej pamięci. W rezultacie wiele wzorców projektowych w przetwarzaniu strumieniowym koncentruje się na sposobach podziału danych na podstrumienie, które mogą być przetwarzane przy użyciu ograniczonej ilości stanu lokalnego.

Stan zewnętrzny

Stan utrzymywany w zewnętrznym magazynie danych, często w systemie NoSQL takim jak Cassandra. Zaletami stanu zewnętrznego są jego praktycznie nieograniczony rozmiar oraz możliwość uzyskania dostępu do niego z wielu instancji aplikacji, a nawet z różnych aplikacji. Minusy to dodatkowe opóźnienie, złożoność wprowadzona wraz z dodatkowym systemem oraz dostępność — aplikacja musi obsługiwać ewentualność, że system zewnętrzny przestanie być dostępny. Twórcy większości aplikacji przetwarzania strumieniowego starają się unikać konieczności radzenia sobie z zewnętrznym magazynem lub przynajmniej ograniczyć opóźnienie poprzez buforowanie informacji w stanie lokalnym i komunikowanie się z zewnętrznym magazynem możliwie najrzadziej. Zwykle wprowadza to wyzwania dotyczące zachowania spójności między stanem wewnętrznym i zewnętrznym.

Dualność strumieniowo-tablicowa

Wszyscy znamy tablice baz danych. **Tablica** jest zbiorem rekordów, z których każdy jest identyfikowany za pomocą klucza podstawowego i zawiera zestaw atrybutów zdefiniowanych przez schemat. Rekordy tablic są mutowalne (tzn. tablice umożliwiają wykonywanie operacji aktualizowania i usuwania). Kwerendowanie tablicy pozwala sprawdzić stan danych w określonym momencie. Kwerendując np. tablicę CUSTOMERS_CONTACTS w bazie danych, spodziewamy się znaleźć aktualne dane kontaktowe dla wszystkich naszych klientów. Jeżeli tablica nie została specjalnie zaprojektowana w celu uwzględniania historii, nie znajdziemy w niej poprzednich kontaktów klientów.

W przeciwieństwie do tablic, strumienie zawierają historię zmian. **Strumień** to łańcuch zdarzeń, w którym każde zdarzenie spowodowało zmianę. Tablica zawiera aktualny stan świata, który jest wynikiem wielu zmian. Z tego opisu jasno wynika, że strumienie i tablice to dwie strony tej samej monety — świat zawsze się zmienia i czasami interesują nas zdarzenia, które spowodowały te zmiany, a innym razem interesuje nas aktualny stan świata. Systemy umożliwiające przechodzenie tam i z powrotem między dwoma sposobami patrzenia na dane są potężniejsze niż systemy, które obsługują tylko jeden sposób.

Aby przekonwertować tablicę na strumień, musimy przechwytywać zmiany, które modyfikują tablicę: wziąć wszystkie te zdarzenia wstawiania (insert), aktualizowania (update) i usuwania (delete) i zapisać je w strumieniu. Większość baz danych oferuje rozwiązania do przechwytywania zmian danych (ang. *Change Data Capture* — CDC) w celu przechwytywania tych zmian i dostępnych jest

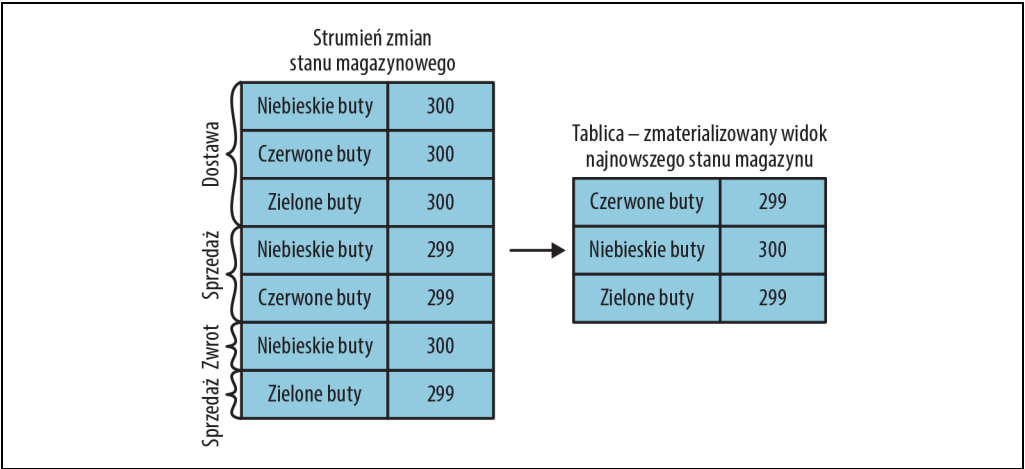
wiele konektorów Kafki, które mogą wprowadzać takie zmiany do tej platformy, gdzie będą dostępne do przetwarzania strumieniowego.

Aby przekonwertować strumień na tablicę, musimy zastosować wszystkie zmiany, które zawiera ten strumień. Nazywa się to również **materializowaniem** strumienia. Tworzymy tablicę w pamięci, w wewnętrznym magazynie stanu albo w zewnętrznej bazie danych i zaczynamy przechodzić przez wszystkie zdarzenia w strumieniu od początku do końca, zmieniając w trakcie stan. Kiedy skończymy, będziemy mieli tablicę reprezentującą stan w określonym punkcie w czasie.

Załóżmy, że sprzedajemy buty. Strumieniowa reprezentacja naszej działalności detalicznej może być strumieniem zdarzeń:

- dotarła dostawa z czerwonymi, niebieskimi i zielonymi butami;
- niebieskie buty zostały sprzedane;
- czerwone buty zostały sprzedane;
- niebieskie buty zostały zwrócone;
- zielone buty zostały sprzedane.

Jeśli chcemy wiedzieć, co w tej chwili mamy na stanie magazynowym lub ile pieniędzy zarobiliśmy do tej pory, musimy zmateralizować ten widok. Na rysunku 14.1 pokazaliśmy, że obecnie mamy 299 par czerwonych butów. Jeśli chcemy wiedzieć, jak duży ruch jest w sklepie, możemy przyjrzeć się całemu strumieniowi i zobaczyć, że dzisiaj miały miejsce cztery zdarzenia klientów. Możemy również chcieć zbadać, dlaczego niebieskie buty zostały zwrócone.



Rysunek 14.1. Materializacja zmian w stanie magazynowym

Okna czasowe

Większość operacji na strumieniach to operacje na wycinkach czasu: średnie kroczące, najlepiej sprzedawane produkty w tym tygodniu, 99. percentyl obciążenia systemu itp. Operacje łączenia dwóch strumieni to także operacje na oknach czasowych — łączymy zdarzenia, które wystąpiły w tym

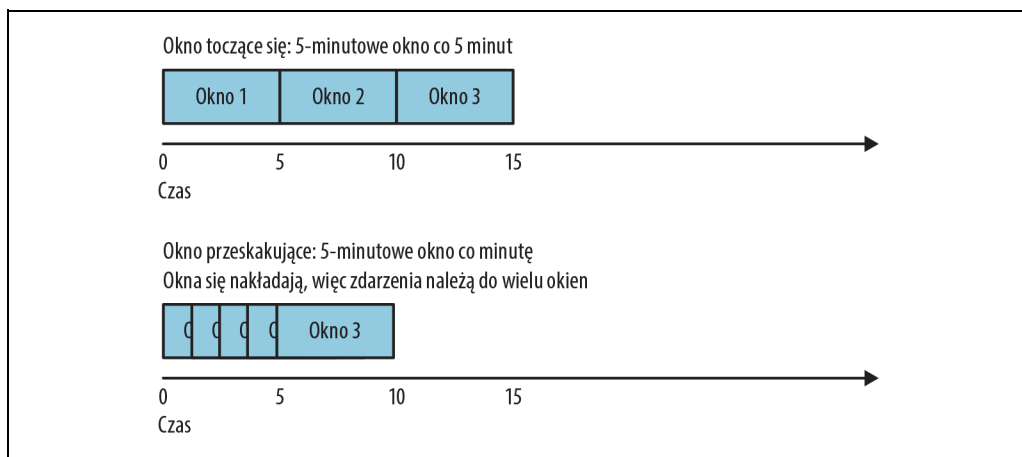
samym wycinku czasu. Bardzo niewiele osób zastanawia się, jakiego rodzaju okien potrzebują dla swojej operacji. Przy obliczaniu średnich kroczących potrzebujemy np. kilku informacji. Oto one:

Rozmiar okna

Czy chcemy obliczyć średnią wszystkich zdarzeń w każdym 5-minutowym oknie, 15-minutowym oknie, a może dla poszczególnych dni? Większe okna są bardziej wyrównane, ale mają większe opóźnienie — jeśli cena wzrośnie, zauważenie tego zajmie więcej czasu niż przy mniejszym oknie. Kafka Streams uwzględnia również **okno sesji**, gdzie wielkość okna jest definiowana przez okres bezczynności. Programista definiuje odstęp między sesjami, a wszystkie zdarzenia, które docierają w sposób ciągły w odstępach krótszych niż zdefiniowany, należą do tej samej sesji. Przerwa w zdarzeniach będzie definiowała nową sesję, a wszystkie zdarzenia nadchodzące po tej przerwie, ale przed kolejną, będą należały do nowej sesji.

W jaki sposób okno się przesuwa (interwał postępu)

Średnie 5-minutowe mogą być aktualizowane co minutę, sekundę lub za każdym razem, gdy pojawi się nowe zdarzenie. Okna, których rozmiar jest stałym przedziałem czasowym, nazywane są **oknami przeskakującymi** (ang. *hopping window*). Gdy interwał postępu jest równy rozmiarowi okna, mamy do czynienia z **oknem toczącym się** (ang. *tumbling window*). Różnicę między tymi dwoma typami okien pokazaliśmy na rysunku 14.2.



Rysunek 14.2. Porównanie okien toczących się i okien przeskakujących

Jak długo okno pozostaje aktualizowalne (okres karencji)

Nasza 5-minutowa średnia krocząca obliczyła średnią dla okna 00:00 – 00:05. Godzinę później otrzymujemy kilka kolejnych rekordów wejściowych z *czasem zdarzenia* pokazującym 00:02. Czy aktualizujemy wynik dla okresu 00:00 – 00:05, czy może zapomnimy o przeszłości? Najlepiej, gdybyśmy mogli zdefiniować pewien przedział czasu, w którym zdarzenia będą dodawane do odpowiedniego dla nich wycinka czasu. Jeśli zdarzenia będą np. opóźnione do 4 godzin, powinniśmy ponownie obliczyć wyniki i je zaktualizować. Jeżeli zdarzenia nadejdą później, możemy je zignorować.

Okna mogą być zsynchronizowane z czasem zegara — 5-minutowe okno, które przesuwają się co minutę, będzie miało pierwszy wycinek jako 00:00 – 00:05, a drugi jako 00:01 – 00:06. Można również nie synchronizować okien z zegarem i po prostu uruchamiać licznik czasu przy każdym uruchomieniu aplikacji, a wtedy pierwszym wycinkiem będzie np. 03:17 – 03:22.

Gwarancje przetwarzania

Kluczowym wymogiem dla aplikacji do przetwarzania strumieniowego jest możliwość przetwarzania każdego rekordu dokładnie raz, niezależnie od awarii. Bez gwarancji „dokładnie raz” przetwarzanie strumieniowe nie może być stosowane w przypadkach, w których potrzebne są dokładne wyniki. Jak omówiliśmy szczegółowo w rozdziale 8., Apache Kafka zapewnia wsparcie dla semantyki „dokładnie raz” za pomocą producentów transakcyjnego i idempotentnego. Kafka Streams używa transakcji Kafki do zaimplementowania gwarancji „dokładnie raz” dla aplikacji przetwarzania strumieniowego. W każdej aplikacji korzystającej z biblioteki Kafka Streams można włączyć gwarancje „dokładnie raz”, ustawiając dla właściwości `processing.guarantee` wartość `exactly_once`. Kafka Streams od wersji 2.6 zawiera bardziej efektywną implementację semantyki „dokładnie raz”, która wymaga brokerów Kafki w wersji co najmniej 2.5. Tę efektywną implementację można włączyć przez ustawienie dla `processing.guarantee` wartości `exactly_once_beta`.

Wzorce projektowe przetwarzania strumieniowego

Każdy system przetwarzania strumieniowego jest inny — od podstawowej kombinacji konsumenta, przez logikę przetwarzania i producenta po zaawansowane klastry, takie jak Spark Streaming ze swoimi bibliotekami uczenia maszynowego. Istnieją jednak pewne podstawowe wzorce projektowe, które są znanymi rozwiązaniami dla typowych wymagań architektur przetwarzania strumieniowego. Przejrzymy kilka z tych dobrze znanych wzorców i na podstawie przykładów pokażemy, w jaki sposób są one używane.

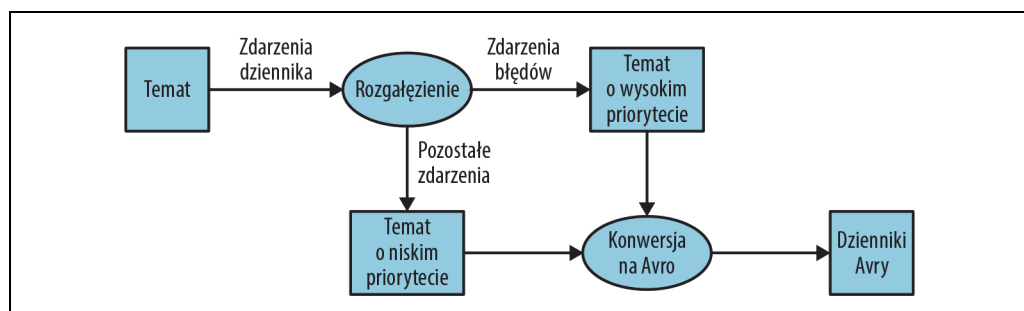
Przetwarzanie pojedynczego zdarzenia

Najbardziej podstawowym wzorcem przetwarzania strumieniowego jest przetwarzanie każdego zdarzenia oddzielnie. Jest to wzorec zwany **Mapowanie-Filtrowanie** (ang. *MapFilter*), powszechnie używany do filtrowania niepotrzebnych zdarzeń ze strumienia lub przekształcania poszczególnych zdarzeń. Określenie „mapowanie” zostało zaczerpnięte z wzorca Mapowanie-Redukowanie (ang. *MapReduce*), w którym krok mapowania przekształca zdarzenia, a krok redukcji je agreguje.

W tym wzorcu aplikacja przetwarzania strumieniowego konsumuje zdarzenia ze strumienia, modyfikuje każde zdarzenie, a następnie produkuje zdarzenia do innego strumienia. Przykładem jest aplikacja, która odczytuje komunikaty dziennika ze strumienia i zapisuje zdarzenia ERROR w strumieniu o wysokim priorytecie, a pozostałe zdarzenia w strumieniu o niskim priorytecie. Innym przykładem jest aplikacja, która odczytuje zdarzenia ze strumienia i konwertuje je z formatu JSON na Avro. Takie aplikacje nie muszą utrzymywać stanu w aplikacji, ponieważ każde zdarzenie może być obsługiwane niezależnie. Oznacza to, że odzyskiwanie sprawności po awarii aplikacji lub

równoważeniu obciążenia jest niezwykle łatwe, ponieważ nie ma potrzeby odzyskiwania stanu; możemy po prostu przekazać zdarzenia do przetworzenia innej instancji aplikacji.

Ten wzorzec można łatwo obsługiwać za pomocą prostego producenta i konsumenta, jak pokazaliśmy na rysunku 14.3.



Rysunek 14.3. Topologia przetwarzania pojedynczego zdarzenia

Przetwarzanie ze stanem lokalnym

Większość aplikacji przetwarzania strumieniowego zajmuje się agregacją informacji, zwłaszcza agregacją okien. Przykładem może być wyszukiwanie minimalnych i maksymalnych cen akcji dla każdego dnia obrotu i obliczanie średniej kroczącej.

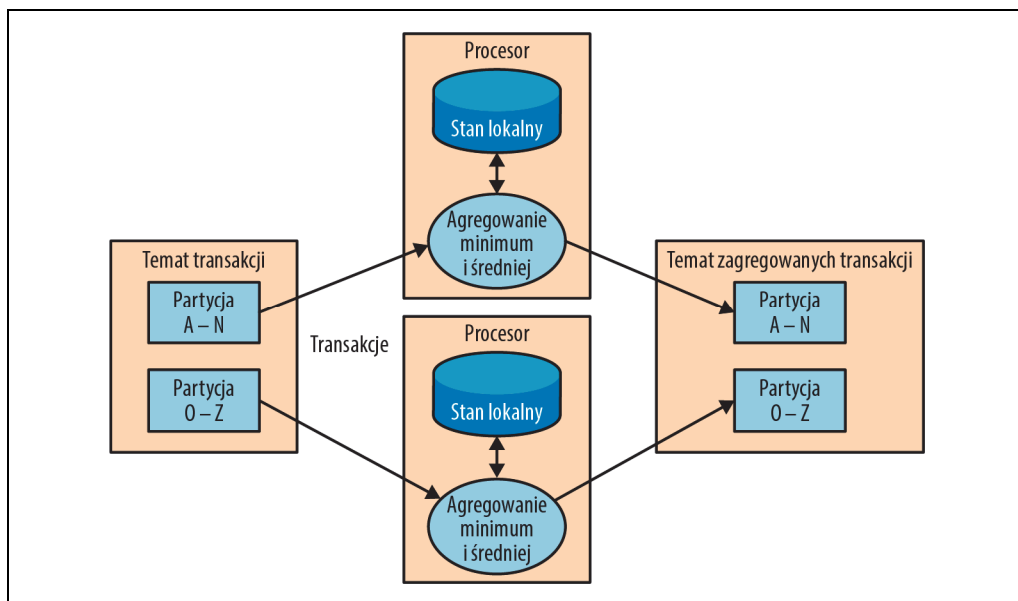
Agregacje te wymagają utrzymywania *stanu*. Aby w naszym przykładzie obliczyć minimalną i średnią cenę dla każdego dnia, musimy zapisać minimalną wartość, sumę oraz liczbę rekordów do danego momentu.

Wszystko to można zrobić przy użyciu stanu **lokalnego** (a nie współdzielonego), ponieważ każda operacja w naszym przykładzie jest agregowaniem typu *group by*. Oznacza to, że wykonujemy agregację według symbolu spółki giełdowej, a nie ogólnie dla całej giełdy. Używamy partycjonera Kafki, aby mieć pewność, że wszystkie zdarzenia z określonym symbolem spółki będą zapisywane na tej samej partycji. Następnie każda instancja aplikacji otrzyma wszystkie zdarzenia z przypisanych do niej partycji (jest to gwarancja konsumenta Kafki). Oznacza to, że każda instancja aplikacji może utrzymywać stan dla podzbioru symboli spółek zapisywanych na przypisanych do niej partycjach. Pokazaliśmy to na rysunku 14.4.

Aplikacje przetwarzania strumieniowego stają się znacznie bardziej skomplikowane, gdy mają stan lokalny. W aplikacjach przetwarzania strumieniowego trzeba rozwiązać kilka problemów:

Użycie pamięci

Stan lokalny najlepiej przechowywać w pamięci dostępnej dla instancji aplikacji. Niektóre lokalne magazyny umożliwiają wykorzystywanie dysku twardego jako obszaru wymiany (ang. *disk spilling*), ale ma to znaczący wpływ na wydajność.



Rysunek 14.4. Topologia przetwarzania zdarzeń ze stanem lokalnym

Utrwalanie

Musimy mieć pewność, że stan nie zostanie utracony po zamknięciu instancji aplikacji i będzie mógł zostać odzyskany po ponownym uruchomieniu instancji lub zastąpieniu jej przez inną instancję. Kafka Streams obsługuje to bardzo dobrze — stan lokalny jest przechowywany w pamięci przy użyciu wbudowanej bazy danych RocksDB, która utrzuła również dane na dysku w celu szybkiego odzyskiwania po restartach. Jednak wszystkie zmiany w stanie lokalnym są ponadto wysyłane do tematu Kafki. Jeśli węzeł strumienia ulegnie awarii, stan lokalny nie zostanie utracony — można go łatwo odtworzyć przez ponowne odczytanie zdarzeń z tematu Kafki. Jeżeli stan lokalny zawiera np. „aktualna cena minimalna dla IBM = 167,19”, zapisujemy to w Kafce, byśmy później mogli ponownie zapełnić lokalną pamięć podręczną na podstawie tych danych. Kafka używa kompaktowania dzienników dla tych tematów, aby nie rosły one w nieskończoność, a odtworzenie stanu było zawsze wykonalne.

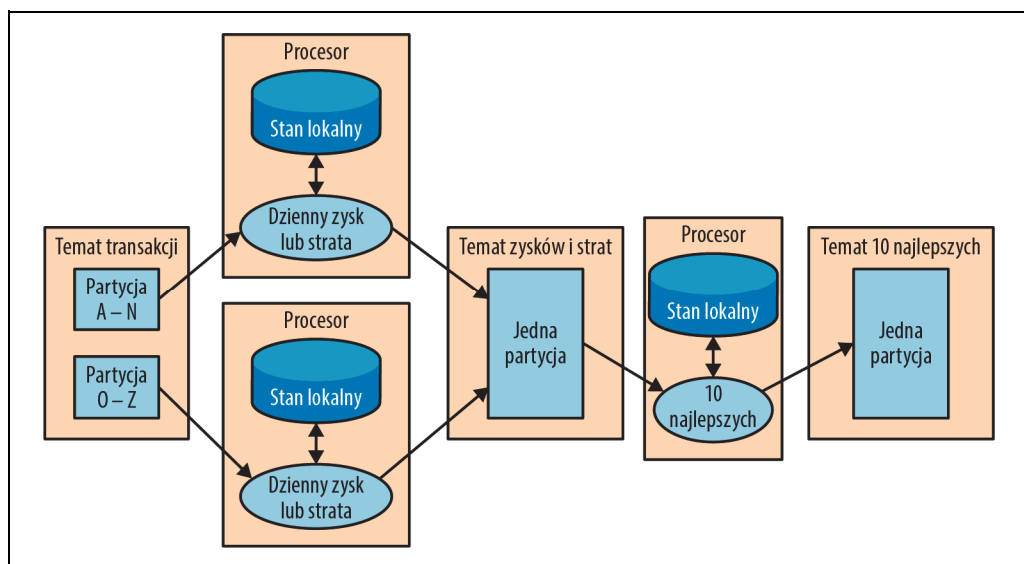
Równoważenie obciążenia

Partycje czasami są ponownie przypisywane do innego konsumenta. W takim przypadku instancja, która traci partycję, musi zapisać ostatni dobry stan, a instancja otrzymująca partycję musi wiedzieć, że ma przywrócić poprawny stan.

Frameworki przetwarzania strumieniowego różnią się tym, w jakim stopniu pomagają programistom zarządzać potrzebnym im stanem lokalnym. Jeśli nasza aplikacja wymaga utrzymywania stanu lokalnego, powinniśmy sprawdzić frameworki i ich gwarancje. Na końcu rozdziału znajdziesz krótki przewodnik porównawczy, ale jak wszyscy wiemy, oprogramowanie szybko się zmienia, a frameworki przetwarzania strumieniowego jeszcze szybciej.

Przetwarzanie wielofazowe (repartycjonowanie)

Stan lokalny nadaje się doskonale, jeśli potrzebujemy agregacji typu *group by*. Ale co zrobić, jeżeli potrzebujemy rezultatu, który będzie wykorzystywał wszystkie dostępne informacje? Załóżmy, że chcemy publikować 10 najlepszych spółek każdego dnia — 10 spółek, których akcje w poszczególnych dniach odnotowały największy wzrost od otwarcia do zamknięcia giełdy. Oczywiście nic, co będziemy robić lokalnie w każdej instancji aplikacji, nie będzie wystarczające, ponieważ te 10 najlepszych spółek może znajdować się w partycjach przypisanych do innych instancji. Potrzebujemy podejścia dwufazowego. Najpierw obliczamy dzienny zysk (lub stratę) dla każdej spółki. Możemy to zrobić w każdej instancji za pomocą stanu lokalnego. Następnie zapisujemy wyniki w nowym temacie z pojedynczą partycją. Ta partycja zostanie odczytana przez pojedynczą instancję aplikacji, która będzie mogła następnie znaleźć 10 najlepszych spółek danego dnia. Drugi temat, który zawiera tylko podsumowanie dzienne dla każdej spółki, jest oczywiście znacznie mniejszy i charakteryzuje się dużo słabszym natężeniem ruchu niż tematy zawierające same transakcje, zatem może być przetwarzany przez pojedynczą instancję aplikacji. Czasami potrzeba więcej kroków, aby wygenerować wynik. Pokazaliśmy to na rysunku 14.5.



Rysunek 14.5. Topologia, która obejmuje zarówno stan lokalny, jak i kroki repartycjonowania

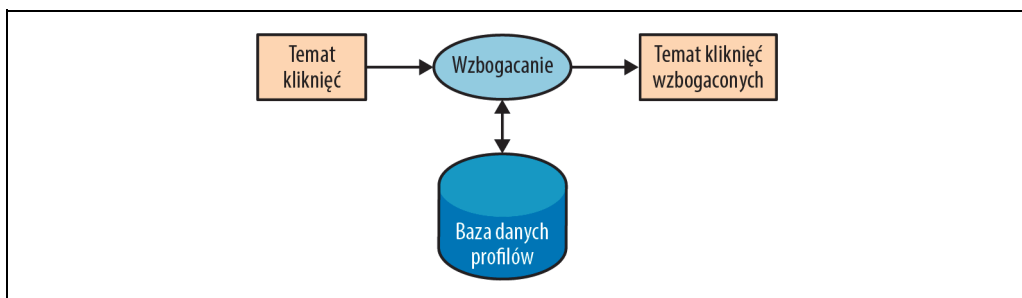
Ten typ przetwarzania wielofazowego jest dobrze znany programistom, którzy piszą kod Map-Reduce, gdzie często trzeba uciekać się do wielu faz redukcji. Jeśli kiedykolwiek pisałeś kod Map-Reduce, zapewne pamiętasz, że do każdego kroku redukcji potrzebowałeś osobnej aplikacji. W przeciwieństwie do MapReduce większość frameworków przetwarzania strumieniowego umożliwia uwzględnienie wszystkich kroków w pojedynczej aplikacji, przy czym framework obsługuje kwestie dotyczące tego, która instancja aplikacji (lub węzeł roboczy) uruchomi każdy etap.

Przetwarzanie z wyszukiwaniem zewnętrznym

— łączenie strumieniowo-tablicowe

Czasami przetwarzanie strumieniowe wymaga integracji z danymi zewnętrznymi w stosunku do strumienia — walidacji transakcji względem zbioru reguł przechowywanych w bazie danych lub wzbogacania informacji ze strumienia kliknięć o dane klikających użytkowników.

Oczywistym pomysłem na wykonanie zewnętrznego wyszukiwania w celu wzbogacenia danych może być mniej więcej coś takiego: dla każdego zdarzenia kliknięcia w strumieniu wyszukujemy użytkownika w bazie danych profili i zapisujemy w kolejnym temacie zdarzenie, które zawiera oryginalne kliknięcie oraz wiek i płeć użytkownika. Pokazaliśmy to na rysunku 14.6.



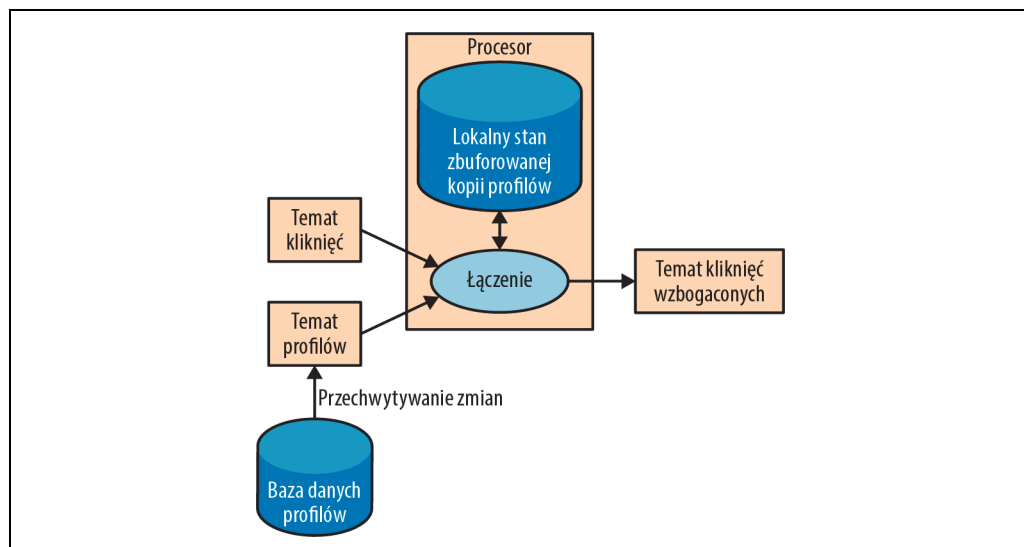
Rysunek 14.6. Przetwarzanie strumieniowe, które obejmuje zewnętrzne źródło danych

Problem z tym oczywistym pomysłem polega na tym, że zewnętrzne wyszukiwanie znacznie zwiększa opóźnienie w przetwarzaniu każdego rekordu — zwykle od 5 do 15 milisekund. W wielu przypadkach nie jest to wykonalne. Często nieakceptowalne jest również dodatkowe obciążenie zewnętrznego magazynu danych — systemy przetwarzania strumieniowego mogą obsługiwać od 100 do 500 tys. zdarzeń na sekundę, ale baza danych przy rozsądnej wydajności może obsługiwać np. tylko 10 tys. zdarzeń na sekundę. Zwiększa się także złożoność związana z dostępnością — nasza aplikacja będzie musiała poradzić sobie z sytuacjami, w których zewnętrzna baza danych nie będzie dostępna.

Aby uzyskać dobrą wydajność i dostępność, musimy buforować informacje z bazy danych w naszej aplikacji przetwarzania strumieniowego. Zarządzanie tą pamięcią podręczną może być jednak trudne — jak zapobiec zdezaktualizowaniu się informacji w pamięci podręcznej? Jeżeli będziemy zbyt często odświeżać zdarzenia, nadal będziemy obciążać bazę danych, a pamięć podręczna nie będzie zbyt pomocna. Jeśli będziemy zbyt długo czekać na pobranie nowych zdarzeń, będziemy wykonywać przetwarzanie strumieniowe z przestarzałymi informacjami.

Jeśli jednak będziemy mogli przechwytywać do strumienia zdarzeń wszystkie zmiany w tablicy bazy danych, nasze zadanie przetwarzania strumieniowego będzie mogło nasłuchiwać tego strumienia i aktualizować pamięć podręczną na podstawie zdarzeń zmian w bazie danych. Przechwytywanie zmian w bazie danych jako zdarzeń w strumieniu nazywa się **przechwytywaniem zmian danych** (ang. *change data capture* — CDC), a Kafka Connect ma wiele konektorów, które mogą obsługiwać CDC i konwertować tablice baz danych na strumień zdarzeń zmian. Pozwala nam to

zachować własną prywatną kopię tablicy i otrzymywać powiadomienia o każdym zdarzeniu zmiany bazy danych, abyśmy mogli odpowiednio zaktualizować własną kopię. Pokazaliśmy to na rysunku 14.7.



Rysunek 14.7. Topologia, która łączy tablicę i strumień zdarzeń, eliminując potrzebę angażowania w przetwarzanie strumieniowe zewnętrznego źródła danych

Gdy teraz będziemy otrzymywać zdarzenie kliknięcia, będziemy mogli sprawdzać `user_id` na naszym stanie lokalnym i wzbogacać zdarzenie. A skoro używamy stanu lokalnego, skaluje się to znacznie lepiej i nie wpływa na bazę danych i inne korzystające z niej aplikacje. Nazywamy to **łączeniem strumieniowo-tablicowym**, ponieważ jeden ze strumieni reprezentuje zmiany w lokalnie buforowanej tablicy.

Łączenie tablicowo-tablicowe

W poprzednim punkcie napisaliśmy, że tablica i strumień zdarzeń aktualizacji mogą być równoważne. Omówiliśmy szczegółowo, jak działa łączenie strumienia i tablicy. Nie ma powodu, dla którego nie moglibyśmy mieć tych zmaterializowanych tablic po obu stronach operacji łączenia.

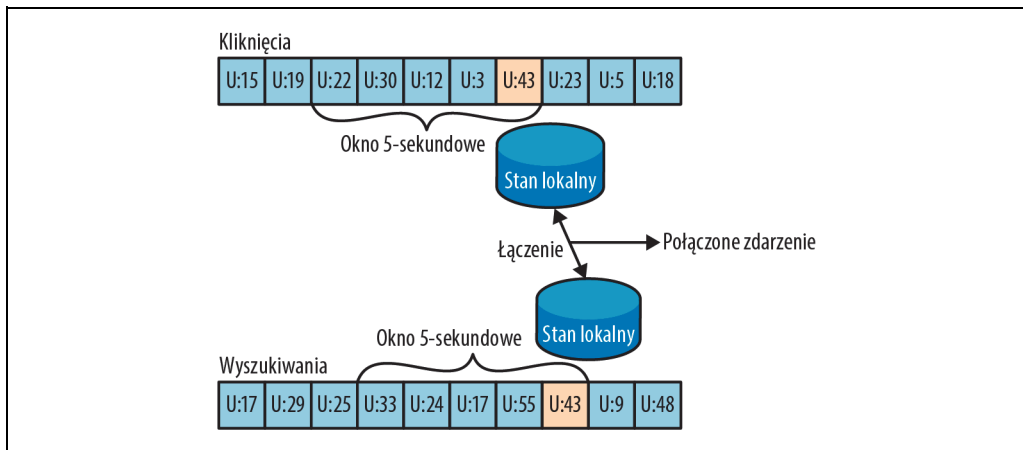
Łączenie dwóch tablic nigdy nie odbywa się dla określonego okna czasowego i powoduje połączenie aktualnego stanu tych tablic w momencie wykonywania operacji. Za pomocą Kafka Streams możemy wykonać operację `equi-join`, gdzie obie tablice mają ten sam klucz, który jest partycjonowany w ten sam sposób, a zatem operacja łączenia może być efektywnie dystrybuowana między dużą liczbę instancji aplikacji i maszyn.

Kafka Streams obsługuje również łączenie dwóch tablic za pomocą klucza obcego — klucz jednego strumienia lub tablicy jest łączony z dowolnym polem z innego strumienia lub tablicy. Więcej na temat sposobu działania tego łączenia dowiesz się z wykładu *Crossing the Streams* (<https://oreil.ly/f34U6>) z konferencji Kafka Summit 2020 lub z posta (<https://oreil.ly/hlKNz>) na blogu Confluenta.

Łączenie strumieniowe

Czasami zamiast jednego strumienia chcemy połączyć z tablicą dwa prawdziwe strumienie zdarzeń. Co sprawia, że strumień jest „prawdziwy”? Jak wspominaliśmy na początku rozdziału, strumienie są nieograniczone. Kiedy używamy strumienia do reprezentowania tablicy, możemy zignorować większość historii ze strumienia, ponieważ zależy nam tylko na bieżącym stanie z tablicy. Jednak kiedy łączymy dwa strumienie, łączymy całą historię, próbując dopasować zdarzenia z jednego strumienia do zdarzeń z drugiego strumienia, które mają ten sam klucz i zaszły w tym samym oknie czasowym. Dlatego łączenie strumieniowe nazywane jest również **łączeniem z uwzględnieniem okna czasowego**.

Załóżmy, że mamy jeden strumień z zapytaniami wyszukiwania, które użytkownicy wpisywali na naszej stronie, i drugi strumień z kliknięciami obejmującymi kliknięcia wyników wyszukiwania. Chcemy dopasować zapytania wyszukiwania do klikniętych wyników, abyśmy wiedzieli, które wyniki są najbardziej popularne dla poszczególnych zapytań. Oczywiście chcemy dopasowywać wyniki na podstawie wyszukiwanego hasła, ale tylko w określonym oknie czasowym. Zakładamy, że wynik jest klikany 5 sekund po wprowadzeniu zapytania do naszej wyszukiwarki. Na każdym strumieniu wstrzymujemy więc niewielkie, kilkusekundowe okno i dopasowujemy wyniki z każdego okna. Pokazaliśmy to na rysunku 14.8.

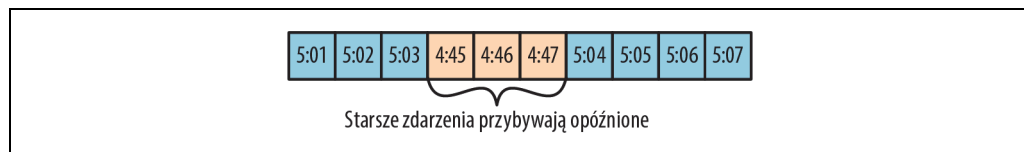


Rysunek 14.8. Łączenie dwóch strumieni zdarzeń; te łączenia zawsze obejmują ruchome okno czasowe

Biblioteka Kafka Streams obsługuje łączenia `equi-join`, gdzie strumień, zapytania i kliknięcia są partycjonowane na te same klucze, które są również kluczami łączenia. W ten sposób wszystkie zdarzenia kliknięcia dla użytkownika `user_id:43` znajdą się w partycji 5 tematu kliknięć, a wszystkie zdarzenia wyszukiwania dla użytkownika `user_id:43` znajdą się w partycji 5 tematu wyszukiwania. Następnie Kafka Streams zapewni przypisanie partycji 5 obu tematów do tego samego zadania. To zadanie będzie więc widziało wszystkie istotne zdarzenia dla `user_id:43`. Okno łączenia dla obu tematów będzie utrzymywane przez bibliotekę Streams we wbudowanym magazynie stanu RocksDB i dzięki temu będzie ona mogła wykonać łączenie.

Zdarzenia poza kolejnością

Obsługa zdarzeń, które docierają do strumienia w niewłaściwym czasie, jest wyzwaniem nie tylko w przetwarzaniu strumieniowym, ale również w tradycyjnych systemach ETL. W scenariuszach IoT zdarzenia poza kolejnością zdarzają się dość często i są spodziewane (rysunek 14.9). Urządzenie mobilne traci np. sygnał wi-fi na kilka godzin i gdy się ponownie połączy, wysyła zdarzenia z tych kilku godzin. Dzieje się tak również podczas monitorowania sprzętu sieciowego (wadliwy przełącznik nie wysyła sygnałów diagnostycznych, dopóki nie zostanie naprawiony) lub produkcji (łączność sieciowa w zakładach produkcyjnych jest bardzo często zawodna, zwłaszcza w krajach rozwijających się).



Rysunek 14.9. Zdarzenia poza kolejnością

Nasze aplikacje przetwarzania strumieniowego muszą być w stanie obsługiwać te scenariusze. Zazwyczaj oznacza to, że aplikacja musi mieć kilka funkcjonalności:

- Aplikacja musi rozpoznawać, że zdarzenie jest poza kolejnością — wymaga to zbadania czasu zdarzenia i wykrycia, że jest ono starsze w stosunku do aktualnego czasu.
- Aplikacja musi zdefiniować okres, w którym spróbuje przeprowadzić rekonyliację zdarzenia poza kolejnością. Rekonyliacja może obejmować np. trzygodzinne opóźnienie, a zdarzenia sprzed ponad trzech godzin można odrzucić.
- Aplikacja musi mieć możliwość przeprowadzania rekonyliacji zdarzeń w trakcie normalnego przetwarzania. Jest to główna różnica między aplikacjami przetwarzania strumieniowego a zadaniami wsadowymi. Jeśli mamy codzienne zadanie wsadowe i kilka zdarzeń dotrze po zakończeniu zadania, zwykle możemy po prostu powtórzyć wczorajsze zadanie i zaktualizować zdarzenia. W przypadku przetwarzania strumieniowego nie ma „powtórki wczorajszego zadania” — ten sam ciągły proces musi przez cały czas obsługiwać zarówno stare, jak i nowe zdarzenia.
- Aplikacja musi być w stanie aktualizować wyniki. Jeżeli wyniki przetwarzania strumieniowego są zapisywane w bazie danych, do aktualizacji wyników wystarczy operacja put lub update. Jeśli aplikacja strumieniowa wysyła wyniki e-mailem, aktualizacje mogą być trudniejsze.

Kilka frameworków przetwarzania strumieniowego, w tym Dataflow firmy Google i Kafka Streams, ma wbudowane wsparcie dla pojęcia czasu zdarzenia niezależnego od czasu przetwarzania i możliwość obsługi zdarzeń z czasami, które są wcześniejsze lub późniejsze niż bieżący czas przetwarzania. Zwykle odbywa się to poprzez utrzymywanie wielu okien agregacji dostępnych do aktualizacji w stanie lokalnym i umożliwienie programistom konfigurowania, jak długo te agregaty okien będą dostępne do aktualizacji. Oczywiście im dłuższe okna agregacji są dostępne do aktualizacji, tym więcej pamięci jest wymagane do utrzymywania stanu lokalnego.

Kafka Streams API zawsze zapisuje wyniki agregacji w tematach wyników. Są to zazwyczaj *tematy skompaktowane*, co oznacza, że zachowywana jest tylko najnowsza wartość dla każdego klucza. W przypadku gdy rezultaty okna agregacji muszą zostać zaktualizowane w wyniku opóźnionego zdarzenia, Kafka Streams zapisuje po prostu nowy wynik dla tego okna agregacji, które w efekcie zastępuje poprzedni wynik.

Ponowne przetwarzanie

Ostatnim ważnym wzorcem jest ponowne przetwarzanie zdarzeń. Istnieją dwa warianty tego wzorca:

- Mamy ulepszoną wersję naszej aplikacji przetwarzania strumieniowego. Chcemy uruchomić tę nową wersję aplikacji na tym samym strumieniu zdarzeń, na którym działa stara wersja, wyprodukować nowy strumień wyników, który nie zastąpi pierwszej wersji, porównać wyniki tych dwóch wersji i w pewnym momencie przenieść klienty, by korzystały z nowych wyników zamiast z istniejących.
- Istniejąca aplikacja przetwarzania strumieniowego zawiera błędy. Naprawiamy błędy i chcemy ponownie przetworzyć strumień zdarzeń i jeszcze raz obliczyć nasze wyniki.

Pierwszy przypadek użycia jest prosty, ponieważ Apache Kafka przechowuje strumień zdarzeń w całości przez długi czas w skalowalnym magazynie danych. Oznacza to, że posiadanie dwóch wersji aplikacji przetwarzania strumieniowego, zapisujących dwa strumienie wynikowe, wymaga wykonania tylko kilku czynności:

- Uruchomienie nowej wersji aplikacji jako nowej grupy konsumentów.
- Skonfigurowanie nowej wersji do rozpoczęcia przetwarzania od pierwszego przesunięcia tematów wejściowych (aby pobrała własną kopię wszystkich zdarzeń ze strumienia wejściowych).
- Umożliwienie kontynuowania przetwarzania nowej aplikacji i przełączenie aplikacji klienckich na nowy strumień wyników, gdy nowa wersja zadania przetwarzania nadrobi zaległości.

Drugi przypadek użycia jest trudniejszy — wymaga „zresetowania” istniejącej aplikacji, aby rozpocząć przetwarzanie od początku strumienia wejściowych, zresetowania stanu lokalnego (by nie pomieścić wyników z dwóch wersji aplikacji) i ewentualnego wyczyszczenia poprzedniego strumienia wyjściowego. Chociaż Kafka Streams ma narzędzie do resetowania stanu dla aplikacji przetwarzania strumieniowego, zalecamy użycie pierwszej metody, gdy istnieje wystarczająca pojemność, aby uruchomić dwie kopie aplikacji i wygenerować dwa strumienie wyników. Pierwsza metoda jest znacznie bezpieczniejsza — umożliwia przełączanie się między wieloma wersjami i porównywanie wyników, a przy tym nie ryzykuje się utraty kluczowych danych ani wprowadzenia błędów podczas procesu czyszczenia.

Zapytania interaktywne

Jak wspomnieliśmy wcześniej, aplikacje przetwarzania strumieniowego mają stan i stan ten może być rozproszony na wiele instancji aplikacji. Najczęściej użytkownicy aplikacji przetwarzania strumieniowego uzyskują wyniki przetwarzania przez odczytywanie ich z tematu wyjściowego. Jednak w niektórych przypadkach pożądanym jest pójście na skróty i odczytywanie wyników z samego

magazynu stanu. Jest to powszechne, gdy wynikiem jest tablica (np. 10 najlepiej sprzedających się książek), a strumień wyników jest tak naprawdę strumieniem aktualizacji tej tablicy — znacznie szybciej i łatwiej jest po prostu odczytać tabelę bezpośrednio ze stanu aplikacji przetwarzania strumieniowego.

Kafka Streams zawiera elastyczne interfejsy API do kwerendowania stanu aplikacji przetwarzania strumieniowego (<https://oreil.ly/pCGeC>).

Kafka Streams na przykładach

Aby zademonstrować implementowanie tych wzorców w praktyce, pokażemy kilka przykładów wykorzystujących Apache Kafka Streams API. Stosujemy ten konkretny interfejs API, ponieważ jest on dość prosty w użyciu i dostarczany z platformą Apache Kafka, do której mamy już dostęp. Należy pamiętać, że te wzorce można zaimplementować w dowolnym frameworku i dowolnej bibliotece przetwarzania strumieniowego — wzorce są uniwersalne, ale przedstawione przykłady są charakterystyczne dla wybranego przez nas API.

Apache Kafka ma dwa strumieniowe interfejsy API: niskopoziomowy Processor API i wysokopoziomowy Streams DSL. W naszych przykładach wykorzystamy Kafka Streams DSL. DSL pozwala definiować aplikacje przetwarzania strumieniowego przez określenie łańcucha przekształceń (transformacji) dla zdarzeń w strumieniach. Mogą to być proste transformacje jak filtr lub złożone jak łączenie strumienia ze strumieniem. API niższego poziomu umożliwia tworzenie własnych transformacji. Więcej informacji na temat niskopoziomowego Processor API znajdziesz w przewodniku programisty (<https://oreil.ly/bQ5nE>), a doskonałym wprowadzeniem jest prezentacja *Beyond the DSL* (<https://oreil.ly/4vson>).

Aplikacja, która korzysta z interfejsu DSL API, zawsze zaczyna się od użycia klasy `StreamsBuilder` do utworzenia **topologii** przetwarzania — skierowanego grafu acyklicznego (ang. *Directed Acyclic Graph* — DAG) transformacji, które są stosowane do zdarzeń w strumieniach. Następnie na podstawie topologii tworzymy obiekt wykonania `KafkaStreams`. Uruchomienie obiektu `KafkaStreams` spowoduje zainicjowanie wielu wątków, z których każdy zastosuje do zdarzeń w strumieniu daną topologię przetwarzania. Przetwarzanie zakończy się po zamknięciu obiektu `KafkaStreams`.

Przyjrzymy się kilku przykładom, w których biblioteka Kafka Streams jest wykorzystywana do zaimplementowania wzorców projektowych, które właśnie omówiliśmy. Prosty przykład zliczania słów zostanie użyty do zademonstrowania wzorca `MapFilter` i prostych agregacji. Następnie przejdziemy do przykładu, w którym będziemy obliczać różne statystyki dotyczące transakcji giełdowych, co pozwoli nam zademonstrować agregacje okien. Na koniec użyjemy wzbogacania strumieni kliknięć jako przykładu łączenia strumieni.

Licznik słów

Przeanalizujmy skrócony przykład liczenia słów dla Kafka Streams. Pełny przykład znajdziesz w repozytorium na stronie <https://ftp.helion.pl/przyklady/kafpra.zip>.

Podczas tworzenia aplikacji przetwarzania strumieniowego na początek należy skonfigurować Kafka Streams. Ta biblioteka kliencka ma dużą liczbę możliwych konfiguracji, których nie będziemy tutaj omawiać, ale więcej informacji na ten temat znajdziesz w dokumentacji (<http://bit.ly/2t7obPU>). Ponadto dodając dowolny producent lub klient do obiektu `Properties`, można skonfigurować producent i konsument wbudowany w Kafka Streams:

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
            "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092"); ❷
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName());
```

- ❶ Każda aplikacja Kafka Streams musi mieć identyfikator aplikacji. Służy on do koordynowania instancji aplikacji, a także do nazywania wewnętrznych magazynów lokalnych i związanych z nimi tematów. Nazwa ta musi być unikatowa dla każdej aplikacji Kafka Streams pracującej z danym klastrem Kafki.
- ❷ Aplikacja Kafka Streams zawsze odczytuje dane z tematów Kafki i zapisuje swoje wyniki w tematach Kafki. Jak zobaczysz później, aplikacje Kafka Streams używają Kafki również do koordynacji. Lepiej więc poinformować naszą aplikację, gdzie znaleźć Kafkę.
- ❸ Podczas odczytu i zapisu danych aplikacja będzie musiała serializować i deserializować, dlatego zapewniamy domyślne klasy `Serde`. Jeśli zajdzie taka potrzeba, możemy nadpisać te domyślne klasy później podczas budowania topologii strumieni.

Skoro mamy konfigurację, zbudujmy topologię strumieni:

```
StreamsBuilder builder = new StreamsBuilder(); ❶

KStream<String, String> source = builder.stream("wordcount-input");

final Pattern pattern = Pattern.compile("\\W+");

KStream<String, String> counts = source.flatMapValues(value->
    Arrays.asList(pattern.split(value.toLowerCase()))) ❷
    .map((key, value) -> new KeyValue<String,String>(value, value))
    .filter((key, value) -> (!value.equals("the"))) ❸
    .groupByKey() ❹
    .count().mapValues(value->Long.toString(value)).toStream(); ❺
counts.to("wordcount-output"); ❻
```

- ❶ Tworzymy obiekt `StreamsBuilder` i zaczynamy definiować strumień przez wskazanie tematu, którego będziemy używać jako danych wejściowych.
- ❷ Każde zdarzenie, które odczytujemy z tematu źródłowego, jest linią słów. Za pomocą wyrażenia regularnego dzielimy tę linię na wiele pojedynczych słów. Następnie bierzemy każde

słowo (obecnie wartość rekordu zdarzenia) i umieszczamy je w kluczu rekordu zdarzenia, aby można było użyć go w operacji `groupBy`.

- ❸ Odfiltrowujemy słowo *the*, aby pokazać, jak łatwe jest filtrowanie.
- ❹ Grupujemy według klucza, mamy więc teraz kolekcję zdarzeń dla każdego unikatowego słowa.
- ❺ Liczymy, ile zdarzeń mamy w każdej kolekcji. Wynikiem zliczania jest typ danych `Long`. Konwertujemy go na `String`, aby wyniki były czytelne dla ludzi.
- ❻ Została tylko jedna rzecz — zapisanie wyników z powrotem w Kafce.

Ponieważ zdefiniowaliśmy przepływ transformacji, które uruchomi nasza aplikacja, musimy ją po prostu uruchomić:

```
KafkaStreams streams = new KafkaStreams(builder.build(), props); ❶

streams.start(); ❷

// Zazwyczaj aplikacja przetwarzania strumieniowego działa w nieskończoność,
// ale w tym przykładzie pozwolimy jej działać jakiś czas, a potem ją zatrzymamy
Thread.sleep(5000L);

streams.close(); ❸
```

- ❶ Definiujemy obiekt `KafkaStreams` na podstawie naszej topologii i zdefiniowanych przez nas właściwości.
- ❷ Uruchamiamy `Kafka Streams`.
- ❸ Po jakimś czasie zatrzymujemy aplikację.

To wszystko! W kilku krótkich liniach pokazaliśmy, jak łatwo jest zaimplementować wzorec przetwarzania pojedynczego zdarzenia (zastosowaliśmy do zdarzeń mapowanie i filtr). Przeprowadziliśmy repartycjonowanie danych przez dodanie operatora `groupBy`, a następnie utrzymywaliśmy prosty stan lokalny, gdy liczyliśmy rekordy, które mają każde słowo jako klucz. Potem utrzymywaliśmy prosty stan lokalny, gdy liczyliśmy, ile razy pojawiło się każde słowo.

Zalecamy uruchomienie pełnego przykładu. Plik *README* dostępny w repozytorium pobranym pod adresem <https://ftp.helion.pl/przyklady/kafpra.zip> zawiera instrukcje dotyczące uruchamiania przykładu.

Zwróć uwagę, że możemy uruchomić na naszej maszynie cały przykład bez instalowania cokolwiek prócz platformy Apache Kafka. Gdyby nasz temat wejściowy zawierał wiele partycji, moglibyśmy uruchomić wiele instancji aplikacji `WordCountExample` (wystarczyłoby uruchomić aplikację w kilku różnych oknach terminala) i otrzymalibyśmy nasz pierwszy klastr przetwarzania `Kafka Streams`. Instancje aplikacji `WordCountExample` komunikują się ze sobą i koordynują pracę. Chociaż w przypadku niektórych frameworków przetwarzania strumieniowego tryb lokalny jest bardzo łatwy w użyciu, jednak aby uruchomić klastr produkcyjny, trzeba zainstalować menedżer `YARN` lub `Mesos`, potem zainstalować dany framework przetwarzania strumieniowego na wszystkich tych maszynach, a następnie nauczyć się dodawać aplikację do klastra. Jeśli chodzi o `Kafka Streams API`, wystarczy uruchomić wiele instancji aplikacji i otrzymujemy klastr. Ta sama aplikacja jest uruchamiana zarówno na naszej maszynie programistycznej, jak i w środowisku produkcyjnym.

Statystyki giełdowe

Następny przykład jest bardziej skomplikowany — będziemy odczytywać strumień zdarzeń giełdowych, które obejmują symbol giełdowy, cenę wywoławczą i kwotę zabezpieczeń. W transakcjach giełdowych **cena wywoławcza** (ang. *ask price*) jest ceną proponowaną przez sprzedawcę, natomiast **cena ofertowa** (ang. *bid price*) jest tym, co sugeruje kupujący. **Kwota bezpieczeństwa** (ang. *ask size*) to liczba akcji, które sprzedawca jest skłonny sprzedać po danej cenie. W celu uproszczenia przykładu całkowicie zignorujemy oferty. Nie będziemy również umieszczać w naszych danych znacznika czasu; zamiast tego będziemy polegać na czasie zdarzeń wypełnianym przez nasz producent Kafki.

Następnie utworzymy strumienie wyjściowe, które będą zawierać kilka statystyk dla okien czasowych:

- najlepsza (tj. minimalna) cena wywoławcza dla każdego 5-sekundowego okna;
- liczba transakcji dla każdego 5-sekundowego okna;
- średnia cena wywoławcza dla każdego 5-sekundowego okna.

Wszystkie statystyki będą aktualizowane co sekundę.

Żałujemy dla uproszczenia, że nasza giełda ma tylko 10 spółek. Ustawienia i konfiguracja są bardzo podobne do tych, których użyliśmy w aplikacji liczenia słów w poprzednim punkcie:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BootstrapServersConfig, Constants.BROKER);
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    TradeSerde.class.getName());
```

Główną różnicą są używane klasy Serde. W liczeniu słów stosowaliśmy łańcuchy znaków zarówno dla klucza, jak i wartości, a zatem używaliśmy dla obu klasy Serdes.String() jako serializatora i deserializatora. W tym przykładzie klucz jest nadal łańcuchem znaków, ale wartością jest obiekt Trade, który zawiera symbol giełdowy, cenę wywoławczą i kwotę bezpieczeństwa. Do serializacji i deserializacji tego obiektu (oraz kilku innych obiektów, które wykorzystaliśmy w tej małej aplikacji) zastosowaliśmy bibliotekę Gson firmy Google, aby z naszego obiektu Javy wygenerować serializator i deserializator JSON. Następnie zdefiniowaliśmy niewielką funkcję opakowującą, która utworzyła z nich obiekt Serde. Oto jak utworzyliśmy Serde:

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(), new JsonDeserializer<Trade>(Trade.class));
    }
}
```

Nic wymyślnego, ale pamiętaj, aby dostarczyć obiekt Serde dla każdego obiektu, który chcesz przechowywać w Kafce — danych wejściowych, wyjściowych, a w niektórych przypadkach wyników pośrednich. Aby to ułatwić, zalecamy generowanie tych obiektów Serde za pośrednictwem bibliotek, takich jak Gson, Avro, Protobuf itp.

Skoro mamy wszystko skonfigurowane, nadszedł czas, aby zbudować naszą topologię:

```
KStream<Windowed<String>, TradeStats> stats = source
    .groupByKey() ❶
    .windowedBy(TimeWindows.of(Duration.ofMillis(windowSize))
        .advanceBy(Duration.ofSeconds(1))) ❷
    .aggregate( ❸
        () -> new TradeStats(),
        (k, v, tradestats) -> tradestats.add(v), ❹
        Materialized.<String, TradeStats, WindowStore<Bytes, byte[]>>
            as("trade-aggregates") ❺
            .withValueSerde(new TradeStatsSerde())) ❻
    .toStream() ❼
    .mapValues((trade) -> trade.computeAvgPrice()); ❽

stats.to("stockstats-output",
    Produced.keySerde(
        WindowedSerdes.timeWindowedSerdeFrom(String.class, windowSize))); ❾
```

- ❶ Zaczynamy od odczytu zdarzeń z tematu wejściowego i wykonania operacji `groupByKey()`. Pomimo nazwy ta operacja nie wykonuje żadnego grupowania. Zapewnia raczej partycjonowanie strumienia zdarzeń na podstawie klucza rekordu. Ponieważ zapisaliśmy dane w temacie za pomocą klucza i nie zmodyfikowaliśmy klucza przed wywołaniem `groupByKey()`, dane są nadal partycjonowane według ich klucza — więc w tym przypadku ta metoda nic nie robi.
- ❷ Określamy okno — w tym przypadku okno 5-sekundowe przesuwające się co sekundę.
- ❸ Po zapewnieniu prawidłowego partycjonowania i konfiguracji okien rozpoczynamy agregację. Metoda `aggregate` rozdzieli strumień na zachodzące na siebie okna (5-sekundowe okno co sekundę), a następnie zastosuje metodę agregacji dla wszystkich zdarzeń w oknie. Pierwszym parametrem przyjmowanym przez tę metodę jest nowy obiekt, który będzie zawierał wyniki agregacji — w naszym przypadku `Tradestats`. Utworzyliśmy ten obiekt, aby zawierał wszystkie statystyki, którymi jesteśmy zainteresowani dla każdego okna czasowego — cenę minimalną, cenę średnią i liczbę transakcji.
- ❹ Następnie zapewniamy metodę rzeczywistego agregowania rekordów — w tym przypadku metoda `add` obiektu `Tradestats` służy do aktualizacji ceny minimalnej, liczby transakcji i cen całkowitych w oknie z nowym rekordem.
- ❺ Jak wspomnieliśmy w podrozdziale „Wzorce projektowe przetwarzania strumieniowego”, agregacja okien wymaga utrzymania stanu w lokalnym magazynie. Ostatnim parametrem metody agregacji jest konfiguracja magazynu stanu. `Materialized` jest obiektem konfiguracji magazynu, a my konfigurujemy nazwę magazynu jako `trade-aggregates`. Może to być dowolna unikatowa nazwa.
- ❻ W ramach konfiguracji magazynu stanu zapewniamy także obiekt `Serde` do serializacji i deserializacji wyników agregacji (obiekt `Tradestats`).
- ❼ Wynikiem agregacji jest *tablica* z symbolem giełdowym i oknem czasowym jako kluczem podstawowym oraz wynikiem agregacji jako wartością. Przekształcamy tablicę z powrotem w strumień zdarzeń.

- 8 Ostatnim krokiem jest aktualizacja średniej ceny — teraz wyniki agregacji obejmują sumę cen i liczbę transakcji. Przechodzimy przez te rekordy i wykorzystujemy istniejące statystyki do obliczenia średniej ceny, byśmy mogli uwzględnić ją w strumieniu wyjściowym.
- 9 I na koniec zapisujemy wyniki w strumieniu `stockstats-output`. Ponieważ wyniki są częścią operacji okienkowania, tworzymy obiekt `WindowedSerde`, który przechowuje wyniki w formacie danych okienkowych zawierających znacznik czasu okna. Rozmiar okna jest przekazywany jako część `Serde`, nawet jeśli nie jest on używany w serializacji (deserializacja wymaga rozmiaru okna, ponieważ tylko czas rozpoczęcia okna jest przechowywany w temacie wyjściowym).

Po zdefiniowaniu przepływu używamy go do wygenerowania obiektu `KafkaStreams` i uruchomienia go, tak jak to zrobiliśmy w aplikacji liczenia słów w poprzednim punkcie.

Ten przykład pokazuje, jak wykonać na strumieniu agregację z oknami czasowymi — jest to prawdopodobnie najpopularniejszy przypadek użycia dla przetwarzania strumieniowego. Należy zwrócić uwagę, jak mało pracy wymagało utrzymywanie lokalnego stanu agregacji — wystarczy zapewnić `Serde` i nazwać magazyn stanu. Ta aplikacja nadal będzie się skalowała do wielu instancji i automatycznie odzyskiwała sprawność po awarii dowolnej instancji przez przeniesienie przetwarzania niektórych partycji do jednej z zachowanych instancji. Więcej informacji na ten temat znajdziesz w podrozdziale „Strumienie Kafki — przegląd architektury” dalej w tym rozdziale.

Jak zwykle pełny przykład oraz instrukcje dotyczące jego uruchamiania znajdziesz w repozytorium pod adresem <https://ftp.helion.pl/przyklady/kafpra.zip>.

Wzbogacanie strumienia kliknięć

Ostatni przykład zademonstruje łączenie strumieni przez wzbogacenie strumienia kliknięć na stronie internetowej. Wygenerujemy strumień symulowanych kliknięć, strumień aktualizacji fikcyjnej tablicy bazy danych profili oraz strumień wyszukiwań internetowych. Następnie połączymy wszystkie trzy strumienie, aby uzyskać kompletny wgląd w każdą aktywność użytkownika. Czego szukali użytkownicy? Co kliknęli w rezultatach wyszukiwania? Czy zmienili swoje „zainteresowania” w profilu użytkownika? Tego typu łączenia zapewniają bogaty zbiór danych do celów analitycznych. Na takich informacjach często oparte są rekomendacje produktów — użytkownik szukał rowerów, kliknął linki dla rowerów Treka i interesuje się podróżami, możemy więc reklamować rowery firmy Trek, kaski i wycieczki rowerowe do egzotycznych miejsc, np. do Nebraski.

Ponieważ konfiguracja aplikacji jest podobna do poprzednich przykładów, pominiemy tę część i przyjrzymy się topologii łączenia wielu strumieni:

```
KStream<Integer, PageView> views =  
    builder.stream(Constants.PAGE_VIEW_TOPIC,  
        Consumed.with(Serdes.Integer(), new PageViewSerde())); ❶  
KStream<Integer, Search> searches =  
    builder.stream(Constants.SEARCH_TOPIC,  
        Consumed.with(Serdes.Integer(), new SearchSerde()));  
KTable<Integer, UserProfile> profiles =  
    builder.table(Constants.USER_PROFILE_TOPIC,  
        Consumed.with(Serdes.Integer(), new ProfileSerde())); ❷  
  
KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
```

```

        (page, profile) -> {
            if (profile != null)
                return new UserActivity(
                    profile.getUserID(), profile.getUserName(),
                    profile.getZipcode(), profile.getInterests(),
                    "", page.getPage()); ❷
            else
                return new UserActivity(
                    -1, "", "", null, "", page.getPage());
        });

KStream<Integer, UserActivity> userActivityKStream =
    viewsWithProfile.leftJoin(searches, ❸
        (userActivity, search) -> {
            if (search != null)
                userActivity.updateSearch(search.getSearchTerms()); ❹
            else
                userActivity.updateSearch("");
            return userActivity;
        },
        JoinWindows.of(Duration.ofSeconds(1)).before(Duration.ofSeconds(0)), ❺
        StreamJoined.with(Serdes.Integer(), ❻
            new UserActivitySerde(),
            new SearchSerde()));

```

- ❶ Najpierw tworzymy obiekty dla dwóch strumieni, które chcemy połączyć — kliknięć i wyszukiwań. Gdy tworzymy obiekt strumienia, przekazujemy temat wejściowy oraz klucz i wartość Serde, które będą używane podczas konsumowania rekordów z tematu i deserializacji ich do postaci obiektów wejściowych.
- ❷ Definiujemy również KTable dla profili użytkowników. KTable to zmaterializowany magazyn, który jest aktualizowany przez strumień zmian.
- ❸ Następnie wzbogacamy strumień kliknięć o informacje z profilu użytkownika, łącząc strumień zdarzeń z tablicą profili. W połączeniu strumieniowo-tablicowym każde zdarzenie w strumieniu otrzymuje informacje z buforowanej kopii tablicy profili. Wykonujemy łączenie lewostronne, więc zachowywane będą kliknięcia bez znanego użytkownika.
- ❹ Jest to metoda join — przyjmuje dwie wartości: jedną ze strumienia, a drugą z rekordu, i zwraca trzecią wartość. Inaczej niż w bazach danych decydujemy, jak połączyć te dwie wartości w jeden wynik. W tym przypadku utworzyliśmy jeden obiekt activity, który zawiera zarówno szczegóły użytkownika, jak i wyświetloną stronę.
- ❺ Następnie chcemy połączyć informacje o kliknięciach z wyszukiwaniami wykonanymi przez tego samego użytkownika. Jest to nadal łączenie lewostronne, ale teraz łączymy dwa strumienie, a nie strumień z tablicą.
- ❻ Jest to metoda join — dodajemy po prostu wyszukiwane hasła do wszystkich pasujących odsłon strony.
- ❼ To jest ta interesująca część — **łączenie strumieniowo-strumieniowe** jest łączeniem z oknem czasowym. Połączenie wszystkich kliknięć i wyszukiwań dla każdego użytkownika nie ma większego sensu — chcemy połączyć każde wyszukiwanie z powiązanymi z nim kliknięciami, czyli kliknięciami, które zaszły w krótkim przedziale czasu po wyszukiwaniu. Definiujemy więc

jednosekundowe okno łączenia. Wywołujemy `of` w celu utworzenia jednosekundowego okna przed każdym wyszukiwaniem i po nim, a następnie wywołujemy `before` z interwałem zero-sekundowym, aby upewnić się, że łączymy tylko kliknięcia, które zachodzą jedną sekundę po każdym wyszukiwaniu, a nie przed nim. Wyniki będą zawierały odpowiednie kliknięcia, wyszukiwane hasła i profil użytkownika. Umożliwi to pełną analizę wyszukiwań i ich wyników.

- 8 Tutaj definiujemy `Serde` wyniku łączenia. Obejmuje to `Serde` dla klucza wspólnego dla obu stron łączenia oraz `Serde` dla obu wartości, które zostaną uwzględnione w wyniku łączenia. W tym przypadku kluczem jest identyfikator użytkownika, więc używamy prostego `Serde` typu `Integer`.

Po zdefiniowaniu przepływu używamy `go` do wygenerowania obiektu `KafkaStreams` i uruchomienia `go`, tak jak to zrobiliśmy w aplikacji liczenia słów.

Ten przykład pokazuje dwa różne wzorce łączenia możliwe w przetwarzaniu strumieniowym. Jeden łączy strumień z tablicą, aby wzbogacić wszystkie zdarzenia strumieniowe o informacje z tablicy. Jest to podobne do łączenia tablicy faktów z wymiarami podczas uruchamiania kwerend hurtowni danych. Drugi przykład łączy dwa strumienie na podstawie okna czasowego. Ta operacja jest uni-katowa dla przetwarzania strumieniowego.

Jak zwykle pełny przykład wraz z instrukcjami dotyczącymi jego uruchamiania znajdziesz w repozytorium pod adresem <https://ftp.helion.pl/przyklady/kafpra.zip>.

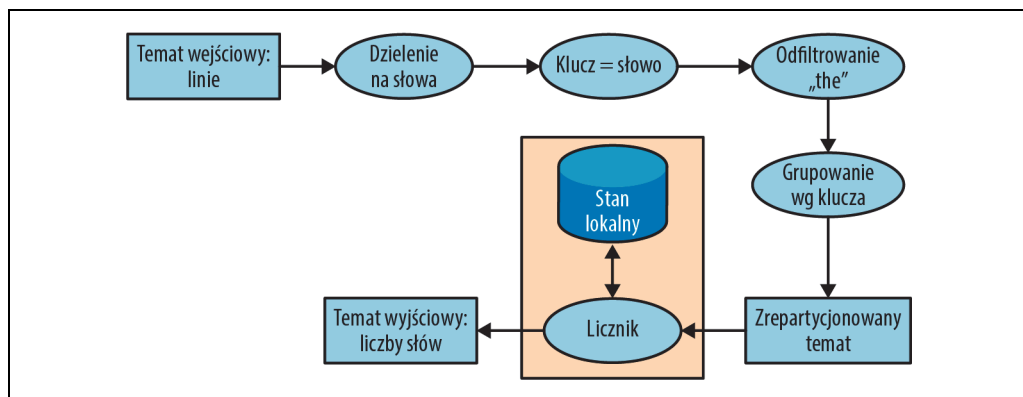
Strumienie Kafki — przegląd architektury

W przykładach w poprzednim podrozdziale pokazaliśmy, jak wykorzystać `Kafka Streams API` do zaimplementowania kilku dobrze znanych wzorców projektowych przetwarzania strumieniowego. Jednak aby lepiej zrozumieć, jak działa i skaluje się biblioteka `Kafka Streams`, musimy zajrzeć za kulisy i przeanalizować niektóre zasady projektowe kryjące się za tym interfejsem `API`.

Budowanie topologii

Każda aplikacja strumieniowa implementuje i wykonuje jedną **topologię**. Topologia (w innych frameworkach przetwarzania strumieniowego nazywana również skierowanym grafem acyklicznym) jest zbiorem operacji i zmian stanu, przez które każde zdarzenie przechodzi od punktu wejścia do punktu wyjścia. Na rysunku 14.10 przedstawiliśmy topologię aplikacji liczenia słów z poprzedniego podrozdziału.

Nawet prosta aplikacja ma nietrywialną topologię. Topologia składa się z procesorów — są to węzły na grafie topologii (reprezentowane przez okręgi na naszym diagramie). Większość procesorów implementuje operacje na danych: filtrowanie, mapowanie, agregowanie itp. Istnieją również procesory źródłowe, które konsumują dane z tematu i przekazują je dalej, oraz procesory ujściowe, które pobierają dane od wcześniejszych procesorów i produkują je do tematu. Topologia zawsze zaczyna się od co najmniej jednego procesora źródłowego, a kończy na przynajmniej jednym procesorze ujścia danych.



Rysunek 14.10. Topologia dla przykładu przetwarzania strumieniowego zliczania słów

Optymalizacja topologii

Domyślnie Kafka Streams wykonuje aplikacje, które zostały zbudowane za pomocą interfejsu DSL API przez mapowanie poszczególnych metod DSL niezależnie na odpowiednik niższego poziomu. W wyniku niezależnej ewaluacji każdej metody DSL traci się możliwości optymalizacji ogólnej topologii wynikowej.

Należy jednak pamiętać, że wykonanie aplikacji Kafka Streams jest procesem trzystopniowym:

1. Topologia logiczna jest definiowana przez utworzenie obiektów `KStream` i `KTable` oraz wykonywanie na nich operacji DSL, takich jak filtrowanie i łączenie.
2. `StreamsBuilder.build()` generuje fizyczną topologię na podstawie topologii logicznej.
3. `KafkaStreams.start()` wykonuje topologię — tutaj dane są konsumowane, przetwarzane i produkowane.

Ogólne optymalizacje planu można zastosować w drugim kroku, w którym fizyczna topologia jest generowana na podstawie definicji logicznych.

Obecnie Apache Kafka zawiera tylko kilka optymalizacji, głównie dotyczących ponownego wykorzystania tematów, o ile to możliwe. Można je włączyć przez ustawienie `StreamsConfig.TOPLOGY_OPTIMIZATION` na `StreamsConfig.OPTIMIZE` i wywołanie `build(props)`. Jeśli wywołasz `build()` bez przekazania konfiguracji, optymalizacja pozostanie wyłączona. Zaleca się przetestowanie aplikacji z optymalizacją i bez optymalizacji oraz porównanie czasu wykonywania i ilości danych zapisanych w Kafce, a ponadto sprawdzenie, czy wyniki są identyczne w różnych znanych scenariuszach.

Testowanie topologii

Generalnie należy przetestować oprogramowanie przed użyciem go w scenariuszach, w których jego pomyślne wykonanie jest ważne. Zautomatyzowane testowanie jest uważane za złoty standard. Powtarzalne testy, które są oceniane za każdym razem, gdy wprowadzana jest zmiana w aplikacji lub bibliotece, umożliwiają szybką iterację i łatwiejsze rozwiązywanie problemów.

Chcemy zastosować ten sam rodzaj metodologii do naszych aplikacji Kafka Streams. Oprócz automatycznych testów kompleksowych (ang. *end-to-end*), które uruchamiają aplikację przetwarzania strumieniowego w środowisku przejściowym z wygenerowanymi danymi, chcemy również uwzględnić szybsze, lżejsze i łatwiejsze do debugowania testy jednostkowe i integracyjne.

Głównym narzędziem testowym dla aplikacji Kafka Streams jest `TopologyTestDriver`. Od czasu jego wprowadzenia w wersji 1.1.0 do jego API dodano istotne ulepszenia, a wersje od 2.4 są wygodne i łatwe w użyciu. Te testy wyglądają jak normalne testy jednostkowe. Definiujemy dane wejściowe, produkujemy je do atrap tematów wejściowych, uruchamiamy topologię ze sterownikiem testowym, odczytujemy wyniki z atrap tematów wyjściowych i sprawdzamy rezultat, porównując go z oczekiwanymi wartościami.

Zalecamy używanie sterownika `TopologyTestDriver` do testowania aplikacji przeprowadzających przetwarzanie strumieniowe, ponieważ jednak nie symuluje on zachowania buforowania Kafka Streams (optymalizacji nieomówionej w tej książce, całkowicie niezwiązanej z magazynem stanu, który jest symulowany przez ten framework), istnieją całe klasy błędów, których nie wykryje.

Testy jednostkowe są zazwyczaj uzupełniane testami integracyjnymi, a dla Kafka Streams istnieją dwa popularne frameworki testów integracyjnych: `EmbeddedKafkaCluster` i `Testcontainers`. Pierwszy z nich uruchamia brokery Kafki wewnątrz JVM, która uruchamia testy, natomiast drugi uruchamia kontenery Dockera z brokerami Kafki (i wiele innych komponentów, w zależności od potrzeb testów). Zalecany jest framework `Testcontainers`, ponieważ dzięki użyciu Dockera w pełni izoluje Kafkę, jej zależności i wykorzystanie zasobów od aplikacji, którą próbujemy przetestować.

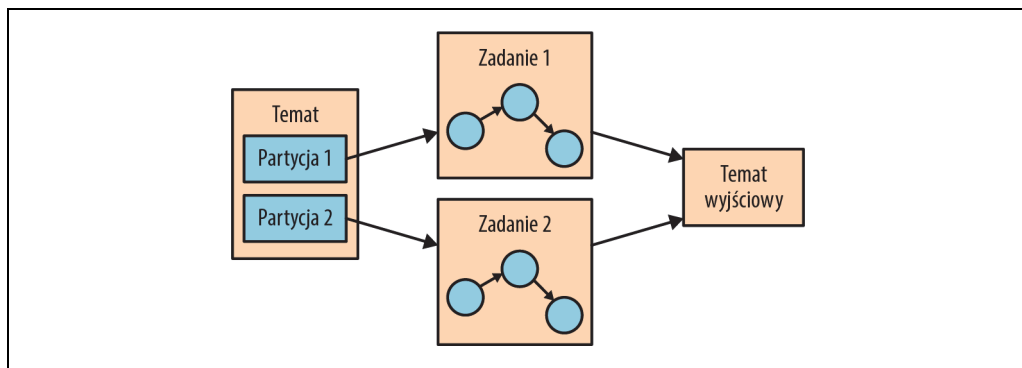
To tylko krótki przegląd metodologii testowania Kafka Streams. Obszerniejszy opis i szczegółowe przykłady topologii i testów znajdziesz w poście *Testing Kafka Streams — A Deep Dive* (<https://oreil.ly/RvTIA>) na blogu Confluenta.

Skalowanie topologii

Kafka Streams skaluje się przez uruchamianie wielu wątków wykonywania w jednej instancji aplikacji i obsługę równoważenia obciążenia między rozproszonymi instancjami aplikacji. Możemy uruchomić aplikację Streams na jednym komputerze z wieloma wątkami lub na wielu komputerach; w obu przypadkach wszystkie aktywne wątki w aplikacji zrównoważą pracę związaną z przetwarzaniem danych.

Silnik Streams zrównolegla wykonywanie topologii, dzieląc ją na zadania. Liczba zadań jest określana przez silnik Streams i zależy od liczby partycji w tematach przetwarzanych przez aplikację. Każde zadanie jest odpowiedzialne za podzbiór partycji: zadanie będzie subskrybowało te partycje i konsumowało z nich zdarzenia. Przed ostatecznym zapisaniem wyniku do ujścia danych dla każdego konsumowanego zdarzenia zadanie wykona po kolei wszystkie etapy przetwarzania, które mają zastosowanie do tej partycji. Zadania te są podstawową jednostką równoległości w Kafka Streams, ponieważ każde zadanie może być wykonywane niezależnie od innych. Pokazaliśmy to na rysunku 14.11.

Programista aplikacji może wybrać liczbę wątków, które będą wykonywane przez każdą instancję aplikacji. Jeśli dostępnych jest wiele wątków, każdy wątek będzie wykonywał podzbiór zadań

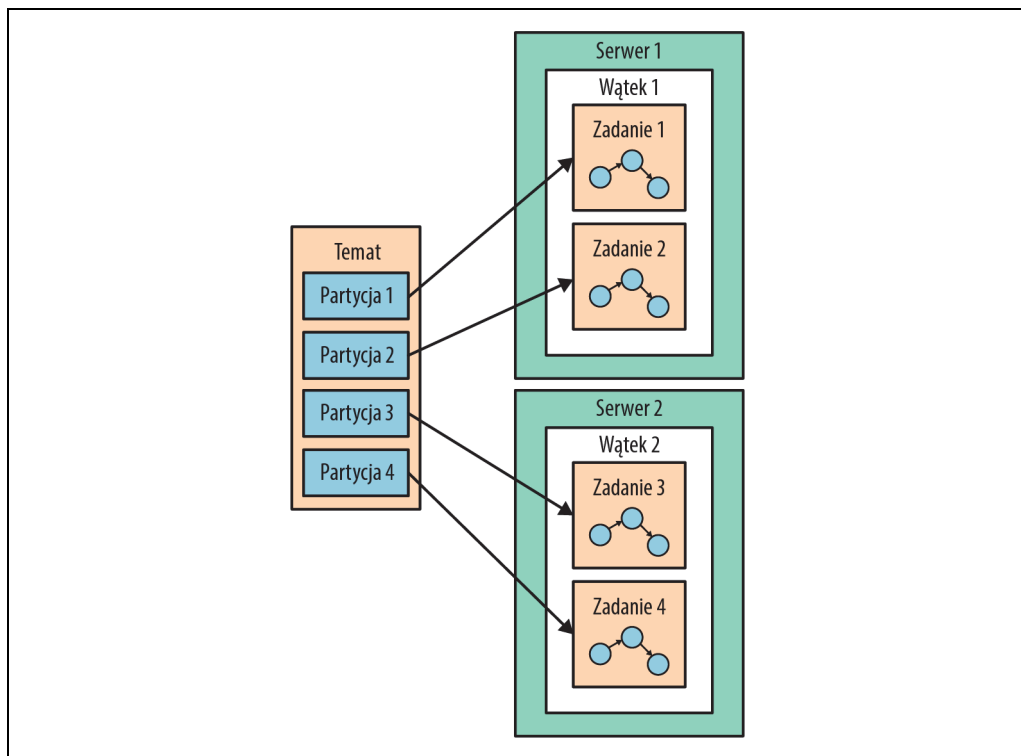


Rysunek 14.11. Dwa zadania uruchamiające tę samą topologię — po jednym dla każdej partycji z tematu wejściowego

tworzonych przez aplikację. Jeżeli wiele instancji aplikacji działa na wielu serwerach, dla każdego wątku na każdym serwerze będą wykonywane różne zadania. W ten sposób skalują się aplikacje streamin-gowe: będziemy mieli tyle zadań, ile partycji w przetwarzanych tematach. Gdy chcemy przetwarzać szybciej, dodajemy więcej wątków. Jeśli zabraknie zasobów na serwerze, uruchamiamy kolejną instancję aplikacji na innym serwerze. Kafka będzie automatycznie koordynować pracę — każdemu zadaniu przydzieli jego własny podzbiór partycji, a każde zadanie będzie niezależnie przetwarzało zdarzenia z tych partycji i utrzymywało własny stan lokalny z odpowiednimi agregatami, jeśli tego wymaga topologia. Pokazaliśmy to na rysunku 14.12.

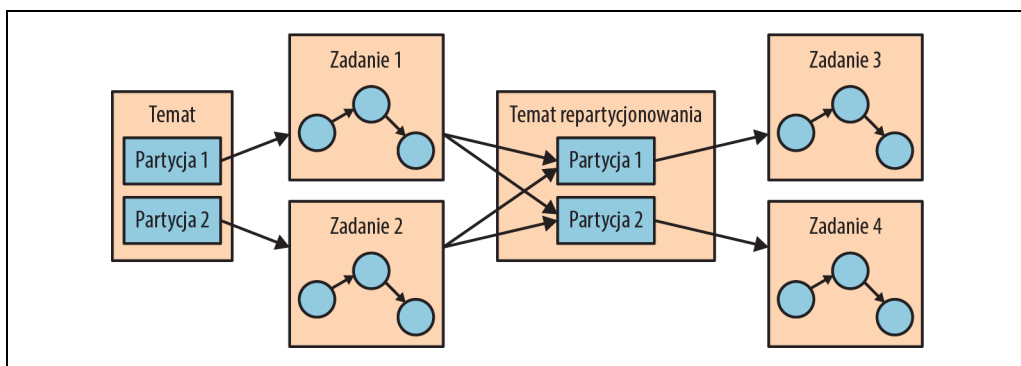
Czasami etap przetwarzania może wymagać danych wejściowych z wielu partycji, które mogą tworzyć zależności między zadaniami. Jeżeli połączymy np. dwa strumienie (jak zrobiliśmy to w przykładzie wzbogacania strumieni kliknięć w poprzednim podrozdziale), to zanim będziemy mogli wyemitować wynik, będziemy potrzebować danych z partycji z każdego strumienia. Kafka Streams radzi sobie z tą sytuacją, przypisując wszystkie partycje potrzebne do jednego łączenia do tego samego zadania, aby to zadanie mogło konsumować ze wszystkich odpowiednich partycji i wykonywać łączenie niezależnie. Dlatego Kafka Streams wymaga aktualnie, aby wszystkie tematy, które biorą udział w operacji łączenia, miały taką samą liczbę partycji i były partycjonowane na podstawie klucza łączenia.

Kolejnym przykładem zależności między zadaniami jest sytuacja, gdy aplikacja wymaga repartycjonowania. W przykładzie strumieni kliknięć wszystkie zdarzenia są np. grupowane według identyfikatora użytkownika. Ale co zrobić, gdybyśmy chcieli generować statystyki dla poszczególnych stron albo kodów pocztowych? Kafka Streams dokona repartycjonowania danych według kodu pocztowego i uruchomi agregację danych z nowymi partycjami. Jeżeli zadanie 1 będzie przetwarzało dane z partycji 1 i dotrze do procesora, który repartycjonuje dane (operacja `groupBy`), konieczne będzie **przetasowanie** (ang. *shuffle*) lub wysłanie zdarzeń do innych zadań. W przeciwieństwie do innych frameworków przetwarzania strumieniowego Kafka Streams repartycjonuje przez zapisywanie zdarzeń w nowym temacie z nowymi kluczami i partycjami. Następnie drugi zestaw zadań odczytuje zdarzenia z nowego tematu i kontynuuje przetwarzanie. Kroki repartycjonowania dzielą naszą topologię na dwie podtopologie, z których każda ma swoje własne zadania. Drugi zbiór zadań zależy od pierwszego, gdyż przetwarza wyniki pierwszej podtopologii. Jednak pierwszy i drugi zestaw



Rysunek 14.12. Zadania przetwarzania strumieniowego mogą być uruchamiane w wielu wątkach i na wielu serwerach

zadań mogą nadal działać niezależnie i równolegle, ponieważ pierwszy zapisuje dane w temacie we własnym tempie, a drugi konsumuje z tego tematu i samodzielnie przetwarza zdarzenia. Między zadaniami nie ma komunikacji ani współdzielonych zasobów i zadania nie muszą działać w tych samych wątkach lub na tych samych serwerach. To jedna z bardziej pożytecznych rzeczy, które robi Kafka — zmniejsza zależności między różnymi częściami potoku. Pokazaliśmy to na rysunku 14.13.



Rysunek 14.13. Dwa zestawy zadań przetwarzających zdarzenia, które są repartycjonowane przy użyciu osobnego tematu

Obsługa awarii

Ten sam model, który umożliwia skalowanie aplikacji, pozwala również płynnie obsługiwać awarie. Przede wszystkim Kafka zapewnia wysoką dostępność, a zatem dane utrwalane w Kafce są również wysoce dostępne. Dlatego jeśli aplikacja ulegnie awarii i będzie wymagała zrestartowania, może sprawdzić swoją ostatnią pozycję w strumieniu Kafki i kontynuować przetwarzanie od ostatniego przesunięcia, które zatwierdziła przed awarią. Zwróć uwagę, że jeśli lokalny magazyn stanu zostanie utracony (np. dlatego, że musieliśmy wymienić serwer, na którym był przechowywany), aplikacja przetwarzania strumieniowego może zawsze odtworzyć go z dziennika zmian przechowywanego w Kafce.

Kafka Streams wykorzystuje ponadto koordynację konsumentów Kafki, aby zapewnić wysoką dostępność zadań. Jeżeli zadanie nie powiodło się, ale istnieją aktywne wątki lub instancje aplikacji przetwarzania strumieniowego, zadanie zostanie zrestartowane w jednym z dostępnych wątków. Przypomina to sposób, w jaki grupy konsumentów obsługują awarię jednego konsumenta w grupie, przypisując partycje któremuś z pozostałych konsumentów. Kafka Streams skorzystała z ulepszeń w protokole koordynacji grup konsumentów Kafki, takich jak statyczne członkostwo w grupach i kooperatywne równoważenie obciążenia (opisane w rozdziale 4.), a także z ulepszeń semantyki „dokładnie raz” Kafki (opisanej w rozdziale 8.).

Chociaż omówione tutaj metody zapewniania wysokiej dostępności sprawdzają się w teorii, rzeczywistość wprowadza pewną złożoność. Jedną z ważnych kwestii jest szybkość odzyskiwania sprawności. Gdy wątek musi rozpocząć przetwarzanie zadania wykonywanego przez wątek, który uległ awarii, najpierw musi odzyskać jego zapisany stan, np. bieżące okna agregacji. Często odbywa się to przez ponowne odczytywanie wewnętrznych tematów z Kafki w celu zainicjowania magazynów stanu Kafka Streams. W czasie potrzebnym do przywrócenia stanu zadania, które się nie powiodło, praca przetwarzania strumieniowego nie poczyni postępów w zakresie tego podzbioru danych, co doprowadzi do zmniejszenia dostępności i zdezaktualizowania danych.

Dlatego skrócenie czasu odzyskiwania sprawności często sprowadza się do skrócenia czasu przywrócenia stanu. Kluczowe jest ustawienie niskiej wartości `min.compaction.lag.ms` i rozmiaru segmentu 100 MB zamiast domyślnego 1 GB (pamiętaj, że ostatni segment w każdej partycji, czyli aktywny segment, nie jest kompaktowany), aby zapewnić skonfigurowanie wszystkich tematów Kafka Streams do agresywnego kompaktowania.

W celu jeszcze szybszego odzyskiwania sprawności zalecamy konfigurację **repliki rezerwowej** — są to zadania, które po prostu mirrorują aktywne zadania w aplikacji przetwarzania strumieniowego i utrzymują bieżący stan na innym serwerze. Gdy nastąpi przełączenie awaryjne, mają już najbardziej aktualny stan i są gotowe do kontynuowania przetwarzania praktycznie bez żadnego przestoju.

Więcej informacji na temat skalowalności i wysokiej dostępności w Kafka Streams znajdziesz w poście na blogu Confluenta (<https://oreil.ly/mj9Ca>) oraz w prelekcji z konferencji Kafka Summit (<https://oreil.ly/cUvKa>).

Przypadki użycia dla przetwarzania strumieniowego

W tym rozdziale nauczyłeś się, jak wykonywać przetwarzanie strumieniowe — przeszliśmy od ogólnych koncepcji i wzorców do konkretnych przykładów z wykorzystaniem biblioteki Kafka Streams. Na tym etapie warto przyjrzeć się powszechnym przypadkom użycia dla przetwarzania strumieniowego. Jak wyjaśniliśmy na początku rozdziału, przetwarzanie strumieniowe — inaczej: przetwarzanie ciągle — jest przydatne w sytuacjach, gdy chcemy, aby zdarzenia były przetwarzane szybko, a nie czekały godzinami do następnej partii, ale nie spodziewamy się także odpowiedzi w milisekundach. To wszystko jest prawdziwe, ale również bardzo abstrakcyjne. Przyjrzyjmy się kilku realnym scenariuszom, które można obsłużyć za pomocą przetwarzania strumieniowego:

Obsługa klienta

Załóżmy, że właśnie zarezerwowaliśmy pokój w dużej sieci hotelowej i oczekujemy przesłania e-mailem potwierdzenia i rachunku. Kilka minut po dokonaniu rezerwacji, gdy potwierdzenie jeszcze nie dotarło, dzwoniemy do działu obsługi klienta, aby potwierdzić naszą rezerwację. Przyjmijmy, że biuro obsługi klienta poinformowało nas, iż nie widzą zamówienia w swoim systemie, ale zadanie wsadowe, które ładuje dane z systemu rezerwacji do hoteli i biura obsługi klienta, jest uruchamiane tylko raz dziennie, więc mamy zadzwonić jutro, natomiast e-mail powinien przyjść w ciągu 2 – 3 dni roboczych. Nie wygląda to na dobrą obsługę, ale już nie raz odbyliśmy podobną konwersację z dużą siecią hotelową. Tak naprawdę chcemy, aby każdy system w sieci hotelowej, w tym centrum obsługi klienta, hotel, system wysyłający potwierdzenia e-mailowe, strona internetowa itd., otrzymywał aktualne informacje o nowej rezerwacji w kilka sekund lub minut po dokonaniu tej rezerwacji. Chcemy również, by centrum obsługi klienta było w stanie natychmiast pobrać wszystkie szczegóły dotyczące naszych poprzednich wizyt w dowolnym hotelu z tej sieci, a recepcja w hotelu wiedziała, że jesteśmy lojalnym klientem i być może należy nam się jakiś bonus. Zbudowanie wszystkich tych systemów za pomocą aplikacji przetwarzania strumieniowego pozwoli im odbierać i przetwarzać aktualizacje w czasie zbliżonym do rzeczywistego, co zapewni lepsze doświadczenie klienta. W przypadku takiego systemu klient w ciągu kilku minut otrzyma wiadomość e-mail z potwierdzeniem, jego karta kredytowa zostanie od razu obciążona, rachunek zostanie wysłany, a biuro obsługi będzie mogło natychmiast odpowiedzieć na jego pytania dotyczące rezerwacji.

Internet rzeczy

Internet rzeczy (ang. *Internet Of Things* — IoT) to pojemne określenie — od domowych urządzeń do regulacji temperatury i zamawiania uzupełnień detergentu do prania po kontrolę jakości produkcji farmaceutycznej w czasie rzeczywistym. Gdy stosuje się przetwarzanie strumieniowe do czujników i urządzeń, bardzo częstym przypadkiem użycia jest przewidywanie, kiedy konieczna będzie zapobiegawcza konserwacja. Przypomina to monitorowanie aplikacji, ale zastosowane do sprzętu, i jest to powszechne w wielu branżach, w tym w produkcji, telekomunikacji (do identyfikacji uszkodzonych masztów telefonii komórkowej), telewizji kablowej (do identyfikacji uszkodzonych dekodów, zanim użytkownicy zgłoszą problem) itd. Każdy przypadek ma swój własny wzorec, ale cel jest podobny: przetwarzanie zdarzeń przychodzących z urządzeń na dużą skalę i identyfikacja wzorców, które sygnalizują, że urządzenie wymaga konserwacji. Do takich wzorców można zaliczyć np.: pakiety porzucane przez przełącznik,

większą siłę wymaganą do dokręcania śrub w produkcji lub częstsze restartowanie dekodерów telewizji kablowej przez użytkowników.

Wykrywanie oszustw

Znane również jako wykrywanie anomalii. Jest to bardzo szeroka dziedzina, która koncentruje się na łapaniu „oszustów” lub złych aktorów w systemie. Przykłady aplikacji wykrywających oszustwa obejmują wykrywanie oszustw związanych z kartami kredytowymi, obrotem akcjami, grami wideo i zagrożeń związanych z cyberbezpieczeństwem. We wszystkich tych dziedzinach jak najwcześniejsze wykrycie oszustwa zapewnia duże korzyści, dlatego system działający w czasie zbliżonym do rzeczywistego, który jest w stanie szybko reagować na zdarzenia — np. zatrzymać złe transakcje, zanim zostaną nawet zatwierdzone — jest znacznie bardziej preferowany niż praca wsadowa, która wykrywa oszustwo np. trzy dni po fakcie, gdy posprzątanie staje się znacznie bardziej skomplikowane. To ponownie problem identyfikacji wzorców w strumieniu zdarzeń o dużej skali.

W cyberbezpieczeństwie istnieje metoda znana jako **beaconing**. Kiedy haker umieszcza wewnątrz organizacji złośliwe oprogramowanie, czasami komunikuje się ono ze światem zewnętrznym, aby odbierać polecenia. Wykrywanie tej aktywności może być trudne, ponieważ może ona zachodzić w dowolnym momencie i z dowolną częstotliwością. Zazwyczaj sieci są dobrze chronione przed atakami zewnętrznymi, ale bardziej narażone na to, że ktoś z wewnątrz organizacji będzie komunikował się na zewnątrz. Przetwarzając duży strumień zdarzeń połączeń sieciowych i rozpoznając wzorzec komunikacji jako nieprawidłowy (np. wykrywając, że dany host zazwyczaj nie uzyskuje dostępu do konkretnych adresów IP), organizacja ds. bezpieczeństwa może zostać powiadomiona wcześniej, zanim wyrządzone zostanie więcej szkód.

Jak wybrać framework przetwarzania strumieniowego

Przy wyborze frameworku przetwarzania strumieniowego ważne jest, aby wziąć pod uwagę typ aplikacji, którą planujesz napisać. Różne rodzaje aplikacji wymagają odmiennych rozwiązań przetwarzania strumieniowego:

Przyjmowanie danych

Celem jest pobieranie danych z jednego systemu do drugiego z pewnymi modyfikacjami tych danych, aby dostosować się do systemu docelowego.

Działania liczone w milisekundach

Każda aplikacja, która wymaga niemal natychmiastowej reakcji. W tym przedziale mieszczą się niektóre przypadki użycia dla wykrywania oszustw.

Mikrouслуги asynchroniczne

Te mikrouслуги wykonują proste czynności na rzecz większego procesu biznesowego, np. aktualizację stanu magazynowego. Aplikacje te mogą wymagać utrzymywania lokalnego stanu buforowania zdarzeń jako sposobu na poprawę wydajności.

Analiza danych w czasie zbliżonym do rzeczywistego

Te aplikacje strumieniowe wykonują złożone agregacje i złączenia, aby powycinać i posegmentować dane oraz wygenerować interesujące, istotne dla biznesu spostrzeżenia.

System przetwarzania strumieniowego, który wybierzesz, będzie w dużej mierze zależał od problemu, jaki masz do rozwiązania:

- Jeśli próbujesz rozwiązać problem z przyjmowaniem danych, powinieneś ponownie rozważyć, czy potrzebujesz systemu przetwarzania strumieniowego, czy prostszego systemu skupionego na przyswajaniu danych, takiego jak Kafka Connect. Gdy jesteś pewien, że chcesz mieć system przetwarzania strumieniowego, musisz zagwarantować zarówno dobry wybór konektorów, jak i wysokiej jakości konektory dla systemów docelowych.
- Jeśli próbujesz rozwiązać problem, który wymaga działań liczonych w milisekundach, również powinieneś ponownie rozważyć wybór strumieni. Często do tego zadania lepiej dostosowane są wzorce typu żądanie-odpowiedź. Jeśli na pewno chcesz mieć system przetwarzania strumieniowego, musisz wybrać taki, który obsługuje model zdarzenie po zdarzeniu przy niskim opóźnieniu, a nie taki, który koncentruje się na mikropartiach wsadowych.
- Jeśli budujesz mikrousługi asynchroniczne, potrzebujesz systemu przetwarzania strumieniowego, który dobrze integruje się z wybraną magistralą komunikatów (mamy nadzieję, że będzie to Kafka), ma funkcjonalności przechwytywania zmian, które łatwo dostarczają upstreamowo zmiany w stanie lokalnym mikrousługi, i zapewnia dobrą obsługę lokalnego magazynu, który może służyć jako pamięć podręczna lub zmaterializowany widok danych mikrousługi.
- Jeśli budujesz złożony silnik analityczny, również potrzebujesz systemu przetwarzania strumieniowego z dobrą obsługą magazynu lokalnego sklepu — tym razem nie do konserwacji lokalnych pamięci podręcznych i zmaterializowanych widoków, ale raczej do obsługi zaawansowanych agregacji, okien i złączeń, które w przeciwnym razie są trudne do wdrożenia. Interfejsy API powinny obsługiwać niestandardowe agregacje, operacje na oknach i wiele typów złączeń.

Oprócz zagadnień dotyczących konkretnych przypadków należy wziąć pod uwagę kilka czynników globalnych:

Funkcjonalność systemu

Czy łatwo jest wdrożyć go w środowisku produkcyjnym? Czy monitorowanie i rozwiązywanie problemów jest łatwe? Czy w razie potrzeby łatwo jest skalować w górę i w dół? Czy dobrze integruje się z istniejącą infrastrukturą? Co zrobić w przypadku wystąpienia błędu i konieczności ponownego przetworzenia danych?

Użyteczność interfejsów API i łatwość debugowania

Spotykałem się z różnicami rzędów wielkości, jeśli chodzi o czas potrzebny na napisanie wysokiej jakości aplikacji w różnych wersjach tego samego frameworku. Czas rozwoju oprogramowania i czas wprowadzenia na rynek to istotne kwestie, musisz więc wybrać system, który sprawi, że będziesz wydajny.

Ułatwianie wykonywania trudnych czynności

Prawie wszyscy twórcy systemów będą twierdzić, że ich system może wykonywać zaawansowane agregacje na oknach czasowych i utrzymywać lokalne magazyny, ale nasuwa się jedno pytanie: „Czy ułatwiają Ci to?”. Czy obsługują praktyczne szczegółowe kwestie dotyczące skali i odzyskiwania sprawności, czy może dostarczają wyciekające abstrakcje i sprawiają, że musisz

zajmować się tym całym bałaganem? Im bardziej system udostępnia czyste interfejsy API i abstrakcje oraz samodzielnie obsługuje praktyczne szczegóły, tym bardziej produktywni będą programiści.

Spółeczność

Większość aplikacji przetwarzania strumieniowego, które będziesz brał pod uwagę, będzie aplikacjami udostępnianymi na licencji *open source*, a nic nie zastąpi dynamicznej i aktywnej społeczności. Dobra społeczność oznacza, że regularnie otrzymujesz nowe i ekscytujące funkcjonalności, jakość jest stosunkowo dobra (nikt nie chce pracować nad złym oprogramowaniem), błędy są szybko naprawiane, a użytkownicy otrzymują odpowiedzi na pytania w odpowiednim czasie. Oznacza to również, że jeśli pojawi się dziwny błąd i wygooglujesz go, znajdziesz informacje na jego temat, ponieważ inne osoby korzystają z tego systemu i widzą te same problemy.

Podsumowanie

Rozpoczęliśmy rozdział od wyjaśnienia, czym jest przetwarzanie strumieniowe. Podaliśmy formalną definicję i omówiliśmy typowe atrybuty paradygmatu przetwarzania strumieniowego. Porównaliśmy go również z innymi paradygmatami programowania.

Następnie opisaliśmy ważne koncepcje przetwarzania strumieniowego. Koncepcje te zademonstrowaliśmy za pomocą trzech przykładowych aplikacji napisanych przy użyciu Kafka Streams.

Po omówieniu wszystkich szczegółów tych przykładowych aplikacji przedstawiliśmy przegląd architektury biblioteki Kafka Streams i wyjaśniliśmy, jak działa ona za kulisami. Zakończyliśmy ten rozdział (i całą książkę) kilkoma przykładami przypadków użycia dla przetwarzania strumieniowego i poradami dotyczącymi sposobów porównywania różnych frameworków przetwarzania strumieniowego.

Instalowanie Kafki w innych systemach operacyjnych

Apache Kafka to przede wszystkim aplikacja Javy i dlatego powinna działać na dowolnym systemie, na którym można zainstalować JRE. Platforma ta została jednak zoptymalizowana pod kątem systemów operacyjnych opartych na Linuksie, tam więc będzie działała najlepiej. Uruchamianie na innych systemach operacyjnych może spowodować powstawanie charakterystycznych dla nich błędów. Dlatego podczas używania Kafki do celów programistycznych lub testowych w standardowym systemie operacyjnym komputerów stacjonarnych warto rozważyć uruchomienie jej na maszynie wirtualnej zgodnej z ewentualnym środowiskiem produkcyjnym.

Instalowanie w systemie Windows

W systemie Microsoft Windows 10 można teraz uruchomić Kafkę na dwa sposoby. Tradycyjnym sposobem jest użycie natywnej instalacji Javy. Użytkownicy Windows 10 mają również możliwość skorzystania z windowsowego podsystemu dla Linuksa. Ta druga metoda jest preferowana, ponieważ zapewnia znacznie prostszą konfigurację, która bardziej odpowiada typowemu środowisku produkcyjnemu, więc jej przyjrzymy się najpierw.

Korzystanie z windowsowego podsystemu dla Linuksa

Jeśli korzystasz z systemu Windows 10, możesz zainstalować natywną obsługę Ubuntu przy użyciu windowsowego podsystemu dla Linuksa (ang. *Windows Subsystem for Linux* — WSL). W momencie gdy piszemy tę książkę, Microsoft nadal traktuje WSL jak funkcjonalność eksperymentalną. Choć WSL działa podobnie do maszyny wirtualnej, nie wymaga zasobów pełnej maszyny wirtualnej i zapewnia bogatszą integrację z systemem operacyjnym Windows.

Aby zainstalować WSL, postępuj zgodnie z instrukcjami podanymi w artykule *What is the Windows Subsystem for Linux?* w dokumentacji Microsoftu (<https://oreil.ly/dULqm>). Gdy wykonasz te czynności, będziesz musiał zainstalować JDK za pomocą menedżera pakietów apt (przy założeniu, że zainstalowałeś pakiet systemu Ubuntu dla WSL):

```
$ sudo apt install openjdk-16-jre-headless
[sudo] password for username:
Reading package lists... Done
Building dependency tree
Reading state information... Done
[...]
done.
$
```

Po zainstalowaniu JDK możesz przystąpić do instalacji platformy Apache Kafka, postępując zgodnie z instrukcjami podanymi w rozdziale 2.

Korzystanie z natywnej Javy

W starszych wersjach systemu Windows (lub jeśli nie chcesz korzystać ze środowiska WSL) możesz uruchomić Kafkę natywnie za pomocą środowiska Javy dla systemu Windows. Należy jednak pamiętać, że może to wprowadzać błędy charakterystyczne dla środowiska Windows. Społeczność programistów Apache Kafki może nie poświęcać tym błędom tyle samo uwagi, co podobnym problemom związanym z systemem Linux.

Przed zainstalowaniem ZooKeepera i Kafki musisz skonfigurować środowisko Javy. Powinieneś zainstalować najnowszą wersję Java 16 od Oracle'a, którą znajdziesz na stronie pobierania Oracle Java SE (<https://jdk.java.net>). Pobierz pełny pakiet JDK, aby mieć wszystkie dostępne narzędzia Javy, i postępuj zgodnie z instrukcjami instalacji.



Ostrożnie ze ścieżkami

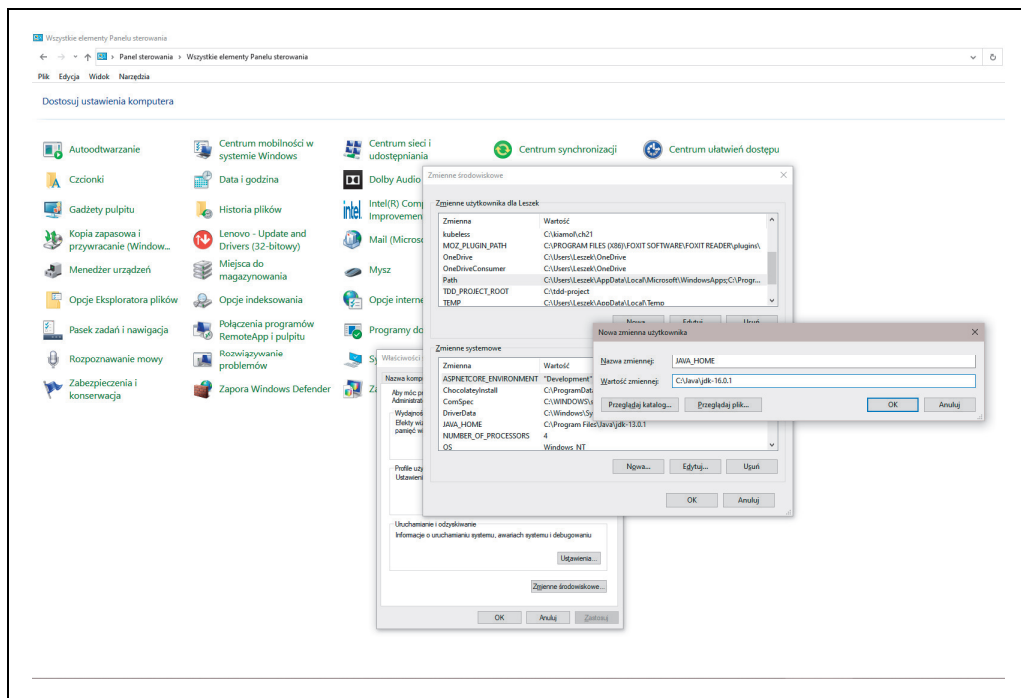
Podczas instalowania Javy i Kafki zdecydowanie zaleca się trzymanie się ścieżek instalacyjnych, które nie zawierają spacji. Chociaż system Windows dopuszcza spacje w ścieżkach, aplikacje zaprojektowane do pracy w środowiskach uniksowych nie są skonfigurowane w ten sposób i określenie ścieżek będzie utrudnione. Instalując Javę, upewnij się, że ustawiłeś ścieżkę instalacji, mając to na uwadze. W przypadku instalacji pakietu JDK 16.0.1 dobrym wyborem będzie użycie np. ścieżki `C:\Java\jdk-16.0.1`.

Po zainstalowaniu Javy skonfiguruj zmienne środowiskowe, by można było z niej korzystać. W systemie Windows można to zrobić w Panelu sterowania, chociaż dokładna lokalizacja zależy od wersji systemu operacyjnego. W systemie Windows 10 należy wykonać kilka czynności:

1. Wybierz *System*.
2. Wybierz *Zaawansowane ustawienia systemu*, co spowoduje otwarcie okna *Właściwości systemu*.
3. Na karcie *Zaawansowane* kliknij przycisk *Zmienne środowiskowe*.

Użyj tej sekcji, aby dodać nową zmienną użytkownika o nazwie `JAVA_HOME` (rysunek A.1) i ustawić dla niej ścieżkę, w której zainstalowałeś Javę. Następnie dokonaj edycji zmiennej systemowej `Path` i dodaj nowy wpis `%JAVA_HOME%\bin`. Zapisz te ustawienia i wyjdź z Panelu sterowania.

Teraz możesz przystąpić do instalacji platformy Apache Kafka. Ta instalacja obejmuje ZooKeeper, nie musisz więc instalować go oddzielnie. Aktualną wersję Kafki możesz pobrać ze strony internetowej <https://oreil.ly/xpwY1>. Gdy piszemy tę książkę, jest to wersja 2.8.0 działająca pod Scalą w wersji 2.13.0. Pobrany plik będzie skompresowany w formacie *gzip* i spakowany za pomocą narzędzia



Rysunek A.1. Dodawanie zmiennej JAVA_HOME

tar, dlatego do jego rozpakowania będziesz musiał użyć aplikacji windowsowej takiej jak 8 Zip. Podobnie jak w przypadku instalacji w Linuksie, musisz wybrać katalog, do którego chcesz wypakować Kafkę. W tym przykładzie założymy, że wypakowujemy Kafkę do `C:\kafka_2.13-2.8.0`.

Uruchamianie ZooKeepera i Kafki w systemie Windows nieco się różni, ponieważ musisz używać plików wsadowych przeznaczonych dla Windowsa, a nie skryptów powłoki dla innych platform. Te pliki wsadowe nie obsługują ponadto uruchamiania aplikacji w tle, będziesz więc potrzebował osobnej powłoki dla każdej aplikacji. Najpierw uruchom ZooKeeper:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> bin\windows\zookeeper-server-start.bat C:\kafka_2.13-2.8.0\config\zookeeper.properties
[2021-07-18 17:37:12,917] INFO Reading configuration from: C:\kafka_2.13-2.8.0\config\zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[...]
[2021-07-18 17:37:13,135] INFO PrepRequestProcessor (sid:0) started, reconfigEnabled=false (org.apache.zookeeper.server.PrepRequestProcessor)
[2021-07-18 17:37:13,144] INFO Using checkIntervalMs=60000 maxPerMinute=10000 (org.apache.zookeeper.server.ContainerManager)
```

Po uruchomieniu ZooKeepera możesz otworzyć kolejne okno, aby uruchomić Kafkę:

```
PS C:\> cd kafka_2.13-2.8.0
PS C:\kafka_2.13-2.8.0> .\bin\windows\kafka-server-start.bat C:\kafka_2.13-2.8.0\config\server.properties
[2021-07-18 17:39:46,098] INFO Registered kafka:type=kafka.Log4jControllerMBean (kafka.utils.Log4jControllerRegistration$)
```

```
[...]
[2021-07-18 17:39:47,918] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
[2021-07-18 17:39:48,009] INFO [broker-0-to-controller-send-thread]: Recorded new
controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

Instalowanie w systemie macOS

macOS korzysta z dystrybucji Darwin, która jest uniksowym systemem operacyjnym wywodzącym się po części z FreeBSD. Oznacza to, że sprawdza się wiele oczekiwań związanych z uruchamianiem systemu operacyjnego Unix, a instalacja aplikacji zaprojektowanych dla Uniksa, takich jak Apache Kafka, nie jest zbyt trudna. Instalacja może być prosta, jeśli będziesz korzystał z menedżera pakietów (takiego jak Homebrew), albo można zainstalować Javę i Kafkę ręcznie, by uzyskać większą kontrolę nad wersjami.

Korzystanie z Homebrew

Jeśli masz już zainstalowany menedżer pakietów Homebrew (<https://brew.sh>) dla macOS, możesz użyć go do zainstalowania Kafki w jednym kroku. Dzięki temu będziesz miał pewność, że najpierw zainstalujesz Javę, a następnie platformę Apache Kafka 2.8.0 (lub nowszą).

Jeśli nie zainstalowałeś jeszcze Homebrew, zrób to w pierwszej kolejności, postępując zgodnie ze wskazówkami zamieszczonymi na stronie dotyczącej instalacji (<https://oreil.ly/ZVEvc>). Następnie możesz zainstalować samą Kafkę. Menedżer pakietów Homebrew zapewni, że najpierw zostaną zainstalowane wszystkie zależności, w tym Java:

```
$ brew install kafka
==> Installing dependencies for kafka: openjdk, openssl@1.1 and zookeeper
==> Installing kafka dependency: openjdk
==> Pouring openjdk--16.0.1.big_sur.bottle.tar.gz
[...]
==> Summary
/usr/local/Cellar/kafka/2.8.0: 200 files, 68.2MB
$
```

Homebrew zainstaluje Kafkę w lokalizacji `/usr/local/Cellar`, ale pliki będą zlinkowane z innymi katalogami:

- pliki binarne i skrypty będą się znajdowały w katalogu `/usr/local/bin`;
- konfiguracje Kafki będą się znajdowały w katalogu `/usr/local/etc/kafka`;
- konfiguracje ZooKeepera będą się znajdowały w katalogu `/usr/local/etc/zookeeper`;
- dla konfiguracji `log.dirs` (lokalizacji danych dla Kafki) zostanie ustawiona ścieżka `/usr/local/var/lib/kafka-logs`.

Po zakończeniu instalacji możesz uruchomić ZooKeeper i Kafkę (ten przykład powoduje uruchomienie Kafki na pierwszym planie):

```
$ /usr/local/bin/zkServer start
ZooKeeper JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
```

```
Starting zookeeper ... STARTED
$ /usr/local/bin/kafka-server-start /usr/local/etc/kafka/server.properties
[2021-07-18 17:52:15,688] INFO Registered kafka:type=kafka.Log4jControllerMBean
(kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 17:52:18,187] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
[2021-07-18 17:52:18,232] INFO [broker-0-to-controller-send-thread]: Recorded new
controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

Instalacja ręczna

Podobnie jak w przypadku ręcznej instalacji dla systemu operacyjnego Windows, podczas instalacji Kafki w systemie macOS należy najpierw zainstalować pakiet JDK. Użyj tej samej strony pobierania Oracle Java SE (<https://jdk.java.net>), aby uzyskać odpowiednią wersję dla macOS. Następnie możesz pobrać Apache Kafkę, podobnie jak dla systemu Windows. W tym przykładzie założymy, że pobrany plik instalacyjny Kafki został rozpakowany do katalogu `/usr/local/kafka_2.13-2.8.0`.

Uruchamianie ZooKeepera i Kafki wygląda tak jak uruchamianie ich w systemie Linux, ale najpierw musisz się upewnić, że został ustawiony katalog `JAVA_HOME`:

```
$ export JAVA_HOME="/usr/libexec/java_home -v 16.0.1"
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home

$ /usr/local/kafka_2.13-2.8.0/bin/zookeeper-server-start.sh -daemon /usr/local/kafka_2.13-
2.8.0/config/zookeeper.properties
$ /usr/local/kafka_2.13-2.8.0/bin/kafka-server-start.sh /usr/local/kafka_2.13-
2.8.0/config/server.properties
[2021-07-18 18:02:34,724] INFO Registered kafka:type=kafka.Log4jControllerMBean
(kafka.utils.Log4jControllerRegistration$)
[...]
[2021-07-18 18:02:36,873] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
[2021-07-18 18:02:36,915] INFO [broker-0-to-controller-send-thread]: Recorded new
controller, from now on will use broker 192.168.0.2:9092 (id: 0 rack: null)
(kafka.server.BrokerToControllerRequestThread)((("macOS, installing Kafka on",
startref="ix_macOS")))((("operating systems", "other than Linux, installing Kafka on",
startref="ix_OSinstall"))))
```


Dodatkowe narzędzia Kafki

Spółeczność platformy Apache Kafka stworzyła solidny ekosystem narzędzi i platform, które znacznie ułatwiają uruchamianie i używanie Kafki. Chociaż nie jest to wyczerpująca lista, w tym dodatku przedstawiliśmy kilka z bardziej popularnych narzędzi, które pomogą Ci się wdrożyć do pracy z Kafką.



Niech kupujący się strzeże

Chociaż autorzy są powiązani z niektórymi firmami i projektami umieszczonymi na tej liście, ani oni, ani wydawnictwo O'Reilly nie przedkładają żadnego konkretnego narzędzia nad inne. Pamiętaj, aby przeprowadzić własne rozpoznanie w kwestii przydatności tych platform i narzędzi do pracy, którą musisz wykonać.

Kompleksowe platformy

Kilka firm oferuje w pełni zintegrowane platformy do pracy z Kafką. Obejmuje to zarządzanie wdrożeniami wszystkich komponentów, aby można było skupić się na korzystaniu z Kafki, a nie na sposobie jej uruchamiania. Może to stanowić idealne rozwiązanie dla przypadków użycia, w których nie są dostępne (lub nie chcesz ich przeznaczać) zasoby do nauki prawidłowej pracy z Kafką i wymaganą dla niej infrastrukturą. Niektóre firmy zapewniają również narzędzia takie jak zarządzanie schematami, interfejsy RESTful, a w określonych przypadkach obsługę bibliotek klienckich, dzięki czemu można mieć pewność, że komponenty będą poprawnie współpracować.

Nazwa	Confluent Cloud
URL	https://www.confluent.io/confluent-cloud
Opis	Tak się składa, że firma założona przez kilku programistów, którzy opracowali i wspierają Kafkę, zapewnia zarządzane rozwiązanie. Confluent Cloud łączy w jednej ofercie wiele niezbędnych narzędzi, w tym zarządzanie schematami, klienty, interfejs RESTful i monitorowanie. Jest dostępny na wszystkich trzech głównych platformach chmurowych (AWS, Microsoft Azure i Google Cloud Platform) i ma wsparcie zapewniające przez znaczną część głównych kontrybutorów Apache Kafki zatrudnionych przez firmę Confluent. Wiele komponentów zawartych w platformie, takich jak rejestr schematów i REST proxy, jest dostępnych jako samodzielne narzędzia na licencji Confluent Community License (https://oreil.ly/IAFga), która ogranicza niektóre przypadki użycia.

Nazwa	Aiven
URL	https://aiven.io
Opis	Aiven zapewnia zarządzane rozwiązania dla wielu platform danych, również dla Kafka. W tym celu firma opracowała Karapace (https://karapace.io), czyli rejestr schematów i REST proxy z interfejsami API kompatybilnymi z komponentami Confluenta, ale wspieranymi na licencji Apache 2.0 (https://oreil.ly/a96F0), która nie ogranicza przypadków użycia. Oprócz trzech głównych dostawców chmury, Aiven obsługuje również DigitalOcean (https://www.digitalocean.com) i UpCloud (https://upcloud.com).
Nazwa	CloudKafka
URL	https://www.cloudkafka.com
Opis	CloudKafka koncentruje się na dostarczaniu zarządzanego rozwiązania Kafka z integracją dla popularnych usług infrastrukturalnych, takich jak DataDog czy Splunk. Obsługuje używanie na swojej platformie rejestru schematów i serwera REST proxy Confluenta, ale tylko w wersji 5.0 przed zmianami w licencji wprowadzonymi przez Confluent. CloudKafka świadczy swoje usługi zarówno na platformie AWS, jak i Google Cloud.
Nazwa	Amazon Managed Streaming for Apache Kafka (Amazon MSK)
URL	https://aws.amazon.com/msk
Opis	Amazon również oferuje własną zarządzaną platformę Kafka, obsługiwaną tylko przez AWS. Wsparcie dla schematu zapewnia integracja z AWS Glue (https://oreil.ly/hvjoV), natomiast REST proxy nie jest bezpośrednio obsługiwany. Amazon promuje korzystanie z narzędzi społecznościowych (takich jak Cruise Control, Burrow i REST proxy Confluenta), ale nie wspiera ich bezpośrednio. W związku z tym MSK jest nieco mniej zintegrowane niż inne oferty, ale nadal może zapewnić podstawowy klaster Kafka.
Nazwa	Azure HDInsight
URL	https://azure.microsoft.com/en-us/services/hdinsight
Opis	Microsoft także oferuje zarządzaną platformę dla Kafka w usłudze HDInsight, która obsługuje ponadto Hadoop, Spark i inne komponenty wielkich zbiorów danych. Podobnie jak MSK, HDInsight koncentruje się na podstawowym klastrze Kafka, pozostawiając zapewnienie wielu innych komponentów (w tym rejestru schematów i REST proxy) użytkownikowi. Niektóre zewnętrzne firmy dostarczyły szablony do wykonywania tych wdrożeń, ale nie są one wspierane przez firmę Microsoft.
Nazwa	Cloudera
URL	https://www.cloudera.com/products/open-source/apache-hadoop/apache-kafka.html
Opis	Cloudera jest stałym elementem społeczności Kafka niemal od samego początku i zapewnia zarządzaną Kafkę jako komponent danych strumieniowych swojego ogólnego produktu Customer Data Platform (CDP). CDP koncentruje się jednak nie tylko na Kafce i działa w środowisku chmury publicznej, ale zapewnia także opcje prywatne.

Wdrażanie klastra i zarządzanie nim

Podczas uruchamiania Kafka poza zarządzaną platformą będziesz potrzebował kilku rzeczy, które pomogą Ci w poprawnym uruchomieniu klastra. Obejmuje to pomoc w alokacji zasobów i we wdrażaniu, w równoważeniu danych i wizualizowaniu klastrów.

Nazwa	Strimzi
URL	https://strimzi.io
Opis	Strimzi zapewnia operatory Kubernetesa do wdrażania klastrów Kafka, aby ułatwić konfigurację Kafka w środowisku Kubernetesa. Nie oferuje usług zarządzanych, za to ułatwia uruchamianie w chmurze, publicznej lub prywatnej. Zapewnia również Strimzi Kafka Bridge, który jest implementacją serwera REST proxy wspieraną na licencji Apache 2.0 (https://oreil.ly/a96F0). Aktualnie Strimzi nie ma wsparcia dla rejestru schematów ze względu na kwestie związane z licencjami.

Nazwa	AKHQ
URL	https://akhq.io
Opis	AKHQ to graficzny interfejs użytkownika do interakcji z klastrami Kafka i zarządzania nimi. Obsługuje zarządzanie konfiguracją, w tym użytkownikami i listami ACL, a ponadto zapewnia częściowe wsparcie dla komponentów takich jak rejestr schematów i Kafka Connect. Oferuje również narzędzia do pracy z danymi w klastrze jako alternatywę dla narzędzi konsolowych.
Nazwa	JulieOps
URL	https://github.com/kafka-ops/julie
Opis	JulieOps (dawniej Kafka Topology Builder) zapewnia zautomatyzowane zarządzanie tematami i listami ACL przy użyciu modelu GitOps. JulieOps oferuje nie tylko przegląd stanu bieżącej konfiguracji, ale także możliwość deklaratywnej konfiguracji oraz kontroli zmian tematów, schematów, list ACL, a z czasem będą dochodzić do tego kolejne elementy.
Nazwa	Cruise Control
URL	https://github.com/linkedin/cruise-control
Opis	Cruise Control to odpowiedź LinkedIna na to, jak zarządzać setkami klastrów z tysiącami brokerów. Początkowo narzędzie to było rozwiązaniem problemu zautomatyzowanego przywracania równowagi danych w klastrach, ale ewoluowało, aby obejmować również wykrywanie anomalii i operacje administracyjne, takie jak dodawanie i usuwanie brokerów. Jest to niezbędnik każdego operatora Kafki, który planuje coś więcej niż tylko uruchamianie klastra testowego.
Nazwa	Conduktor
URL	https://www.conduktor.io
Opis	Chociaż Conduktor nie jest udostępniany na licencji <i>open source</i> , jest popularnym narzędziem do zarządzania klastrami Kafki i interakcji z nimi. Obsługuje wiele zarządzanych platform (w tym Confluent, Aiven i MSK) i wiele różnych komponentów (takich jak Connect, kSQL i Streams). Umożliwia również interakcję z danymi w klastrach zamiast korzystania z narzędzi konsolowych. Bezpłatna licencja jest przeznaczona do użytku programistycznego dla współpracy z pojedynczym klastrem.

Monitorowanie i eksploracja danych

Kluczowym elementem uruchamiania Kafki jest upewnienie się, że klastr i klienci są w dobrej kondycji. Podobnie jak wiele innych aplikacji, Kafka udostępnia liczne wskaźniki i inną telemetrię, ale ogarnięcie tego wszystkiego może być wyzwaniem. Wiele większych platform monitoringowych, takich jak Prometheus (<https://prometheus.io>), może łatwo pobierać wskaźniki z brokerów i klientów Kafki. Dostępnych jest również sporo narzędzi, które pomogą w zrozumieniu tych wszystkich danych.

Nazwa	Xinfra Monitor
URL	https://github.com/linkedin/kafka-monitor
Opis	Xinfra Monitor (dawniej Kafka Monitor) został opracowany przez firmę LinkedIn w celu monitorowania dostępności klastrów i brokerów Kafki. Robi to za pomocą zestawu tematów do generowania syntetycznych danych przepuszczanych przez klastr i mierzenia opóźnień, dostępności i kompletności. Jest to cenne narzędzie do pomiaru kondycji wdrożenia Kafki bez konieczności bezpośredniej interakcji z klientami.
Nazwa	Burrow
URL	https://github.com/linkedin/burrow
Opis	Burrow to kolejne narzędzie stworzone przez LinkedIn, które gwarantuje holistyczne monitorowanie opóźnień konsumendkich w ramach klastrów Kafki. Zapewnia wgląd w kondycję konsumentów bez konieczności bezpośredniej interakcji z nimi. Burrow jest aktywnie wspierany przez społeczność i ma własny ekosystem narzędzi (https://oreil.ly/yNPRQ) do łączenia go z innymi komponentami.

Nazwa	Kafka Dashboard
URL	https://www.datadoghq.com/dashboards/kafka-dashboard
Opis	Jeśli używasz do monitorowania narzędzia DataDog, będziesz mieć do dyspozycji doskonały dashboard Kafki, ułatwiający integrację jej klastrów ze stosem monitorowania. Został zaprojektowany, aby zapewnić widok klastra Kafki w jednym panelu, upraszczając podgląd wielu wskaźników.
Nazwa	Streams Explorer
URL	https://github.com/bakdata/streams-explorer
Opis	Streams Explorer to narzędzie do wizualizacji przepływu danych przez aplikacje i konektory we wdrożeniu Kubernetesa. Chociaż w dużym stopniu opiera się na strukturyzacji wdrożeń przy użyciu biblioteki Kafka Streams lub Faust z wykorzystaniem narzędzi firmy bakdata, może zapewnić łatwy do zrozumienia podgląd tych aplikacji oraz ich wskaźników.
Nazwa	kcat
URL	https://github.com/edenhill/kafkacat
Opis	kcat (dawniej kafkacat) to bardzo lubiana alternatywa dla konsolowego producenta i konsumenta, które są częścią głównego projektu Apache Kafka. Jest to niewielkie, szybkie i napisane w języku C narzędzie, nie ma więc narzutu maszyny wirtualnej Javy. Obsługuje również ograniczone widoki statusu klastra, wyświetlając wyjście metadanych dla klastra.

Biblioteki klienckie

Projekt Apache Kafka zapewnia biblioteki klienckie dla aplikacji Javy, ale jeden język nigdy nie jest wystarczający. Dostępnych jest wiele implementacji klienta Kafki z popularnymi językami, takimi jak Python, Go i Ruby, które mają kilka opcji. Ponadto serwery REST proxy (takie jak te z Confluenta, Strimzi lub Karapace'a) mogą obejmować różne przypadki użycia. Oto kilka implementacji klienta, które przetrwały próbę czasu.

Nazwa	librdkafka
URL	https://github.com/edenhill/librdkafka
Opis	librdkafka jest napisaną w C implementacją biblioteki klienta Kafki, uważanej za jedną z najlepiej działających dostępnych bibliotek. Jest w rzeczywistości tak dobra, że Confluent obsługuje klienty dla Go, Pythona i .NET, które zostały stworzone jako biblioteki opakowujące librdkafkę. Biblioteka librdkafka jest udostępniana na 2-klauzulowej licencji BSD (https://oreil.ly/dLoe8), co ułatwia korzystanie z niej w dowolnej aplikacji.
Nazwa	Sarama
URL	https://github.com/Shopify/sarama
Opis	Firma Shopify opracowała klient Sarama jako natywną implementację w języku Golang. Sarama został wydany na licencji MIT (https://oreil.ly/sajdS).
Nazwa	kafka-python
URL	https://github.com/dpkp/kafka-python
Opis	kafka-python to kolejna natywna implementacja klienta, tym razem w Pythonie. Została wydana na licencji Apache 2.0 (https://oreil.ly/a96F0).

Przetwarzanie strumieniowe

Chociaż projekt Apache Kafka zawiera bibliotekę Kafka Streams do budowania aplikacji, nie jest to jedyny wybór do przetwarzania strumieniowych danych z Kafki.

Nazwa	Samza
URL	https://samza.apache.org
Opis	Apache Samza to framework do przetwarzania strumieniowego, który został zaprojektowany specjalnie dla Kafki. Choć powstał wcześniej niż Kafka Streams, oba frameworki mają wielu wspólnych twórców, a w rezultacie mają także sporo wspólnych koncepcji. Jednak inaczej niż Kafka Streams, Samza obsługuje Yarn i zapewnia pełny framework do uruchamiania aplikacji.
Nazwa	Spark
URL	https://spark.apache.org
Opis	Spark to kolejny projekt Apache zorientowany na wsadowe przetwarzanie danych. Obsługuje strumienie, traktując je jak szybkie mikropartie. Oznacza to, że opóźnienie jest nieco większe, ale odporność na błędy jest zapewniana po prostu przez ponowne przetwarzanie partii, a architektura Lambda jest prosta. Spark korzysta również z szerokiego wsparcia społeczności.
Nazwa	Flink
URL	https://flink.apache.org
Opis	Apache Flink jest zorientowany konkretnie na przetwarzanie strumieniowe i działa z bardzo niewielkimi opóźnieniami. Podobnie jak Samza, obsługuje Yarn, ale współpracuje także z Mesosem, Kubernetesem lub samodzielnymi klastrami. Obsługuje ponadto Pythona i R za pomocą dostarczonych wysokopoziomowych interfejsów API.
Nazwa	Beam
URL	https://beam.apache.org
Opis	Apache Beam nie zapewnia bezpośrednio przetwarzania strumieniowego, ale jest promowany jako ujednolicony model programowania zarówno dla przetwarzania wsadowego, jak i strumieniowego. Wykorzystuje platformy, takie jak Samza, Spark i Flink, do uruchamiania komponentów w całym potoku przetwarzania.

O autorach

Gwen Shapira jest głównym inżynierem w firmie Confluent i zarządza zespołem Kafki dla natywnej chmury, który jest odpowiedzialny za wydajność, elastyczność i wielodzierżawność Kafki. Posiada 15-letnie doświadczenie w pracy z kodem i klientami przy tworzeniu skalowalnych architektur danych. Gwen jest częstą prelegentką na konferencjach branżowych i członkinią PMC projektu Apache Kafka.

Todd Palino jest głównym inżynierem SRE w firmie LinkedIn, zajmującym się wyzwaniem związanym z zarządzaniem pojemnością i wydajnością całej platformy. Wcześniej odpowiadał za architekturę, codzienne operacje oraz rozwój narzędzi dla Kafki i ZooKeepera w LinkedInie, w tym za stworzenie zaawansowanego systemu monitorowania i powiadamiania. Todd jest programistą udostępnianego na licencji *open source* projektu Burrow, narzędzia do monitorowania konsumentów Kafki. Można go spotkać na konferencjach branżowych i prelekcjach technicznych, gdzie dzieli się swoimi doświadczeniami związanymi z SRE. Todd pracuje w branży technologicznej od ponad 20 lat, uruchamiając usługi infrastrukturalne. Pracował jako inżynier systemów w Verisign, rozwijając automatyzację zarządzania usługami dla systemu DNS, sieci i sprzętu, a także zarządzając standardami dotyczącymi sprzętu i oprogramowania w całej firmie.

Rajini Sivaram jest głównym inżynierem w Confluencie, projektującym i rozwijającym funkcje replikacji międzyklastrowej dla Kafki i funkcjonalności bezpieczeństwa dla Confluent Platform i Confluent Cloud. Jest kontrybutorką projektu Apache Kafka i członkinią komisji Apache Kafka Program Management Committee. Zanim dołączyła do firmy Confluent, pracowała w Pivotalu, budując wysokowydajny reaktywny interfejs API dla Kafki z wykorzystaniem biblioteki Project Reactor. Wcześniej Rajini pracowała w firmie IBM nad platformą Kafka-as-a-Service dla IBM Bluemix. Jej doświadczenie rozciąga się od systemów równoległych i rozproszonych po maszyny wirtualne Javy i systemy komunikacyjne.

Krit Petty jest menedżerem SRE ds. Kafki w firmie LinkedIn. Zanim został menedżerem, pracował jako inżynier SRE w zespole, rozszerzając i rozwijając Kafkę, aby pokonać przeszkody związane ze skalowaniem Kafki do nigdy wcześniej niewidzianych rozmiarów, w tym podejmując pierwsze kroki w celu przeniesienia posiadanych przez LinkedIn dużych wdrożeń Kafki do chmury Azure Microsoftu. Krit ma tytuł magistra informatyki i wcześniej pracował przy zarządzaniu systemami Linux oraz jako inżynier oprogramowania tworzący oprogramowanie do wysokowydajnych projektów obliczeniowych w przemyśle naftowym i gazowym.

Kolofon

Zwierzę przedstawione na okładce książki *Kafka w praktyce* to niebieskoskrzydła kookaburra (*Dacelo leachii*). Należy do rodziny *Alcedinidae* i zamieszkuje rejony południowej Nowej Gwiney oraz mniej suchy obszar północnej Australii. Ptaki te uważa się za zimorodki.

Samiec kookaburry ma kolorowe upierzenie. Dolne skrzydło i pióra ogona są niebieskie, stąd jego nazwa, ale ogony samic są czerwono-brązowe z czarnymi paskami. Obie płcie mają kremowe tułowie ze smugami brązu i białe tęczówki oczu. Dorosłe kookaburry są mniejsze od innych zimorodków, mierzą zaledwie od 38 do 43 cm i ważą średnio od 260 do 330 gramów.

Dieta niebieskoskrzydłej kookaburry jest bogata w mięso, a pożywienie różni się nieznacznie w zależności od zmiennych pór roku. Latem w pożywieniu tych ptaków występuje np. więcej jaszczurek, owadów i żab, jednak podczas bardziej suchych miesięcy wprowadzają do swojej diety więcej raków, ryb, gryzoni, a nawet mniejszych ptaków. Nie są wyjątkami, jeśli chodzi o zjadanie innych ptaków, ponieważ w sezonie same stanowią menu rdzawostrzębia jednoskrzydłego oraz sowicy maskowej.

Okres lęgowy dla kookaburry niebieskoskrzydłej przypada od września do grudnia. Gniazda budowane są w dziuplach w wysokich partiach drzew. Wychowywanie młodych to wysiłek społeczności; mamie i tacie pomaga przynajmniej jeden ptasi pomocnik. Składanych jest od 3 do 4 jaj, które następnie są wysiadywane przez ok. 26 dni. Pisklęta nabierają upierzenia ok. 36 dni po wylęgu, jeśli przeżyją. Po wylęgu zdarza się bowiem, że w pierwszym tygodniu życia młodsze rodzeństwo jest zabijane przez starsze, co jest elementem walki i instynktu przetrwania. Pisklęta, które nie staną się ofiarami bratobójstwa lub nie zdechną wskutek innych przyczyn, są szkolone przez rodziców do polowania przez 6 – 10 tygodni.

Kolorowa ilustracja Karen Montgomery została oparta na czarno-białej rycinie zaczerpniętej z pozycji *English Cyclopaedia*.

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - wideokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL