Конвейер реального времени

Потоковая обработка данных

Эндрю Дж. Пселтис

ДМК ПРЕСС

MANNING

Эндрю Дж. Пселтис

Потоковая обработка данных

Конвейер реального времени

Streaming Data

UNDERSTANDING THE REAL-TIME PIPELINE

ANDREW G. PSALTIS

MANNING
SHELTER ISLAND

Потоковая обработка данных

Конвейер реального времени

Эндрю Дж. Пселтис

ДМК Пресс
издательство
Москва, 2018

УДК 004.6
ББК 32.972.13
П86

П86 Эндрю Дж. Пселтис
Потоковая обработка данных. Конвейер реального времени / пер. с англ. А. А. Слинкин – М.: ДМК Пресс, 2018. – 218 с.: ил.

ISBN 978-5-97060-606-3

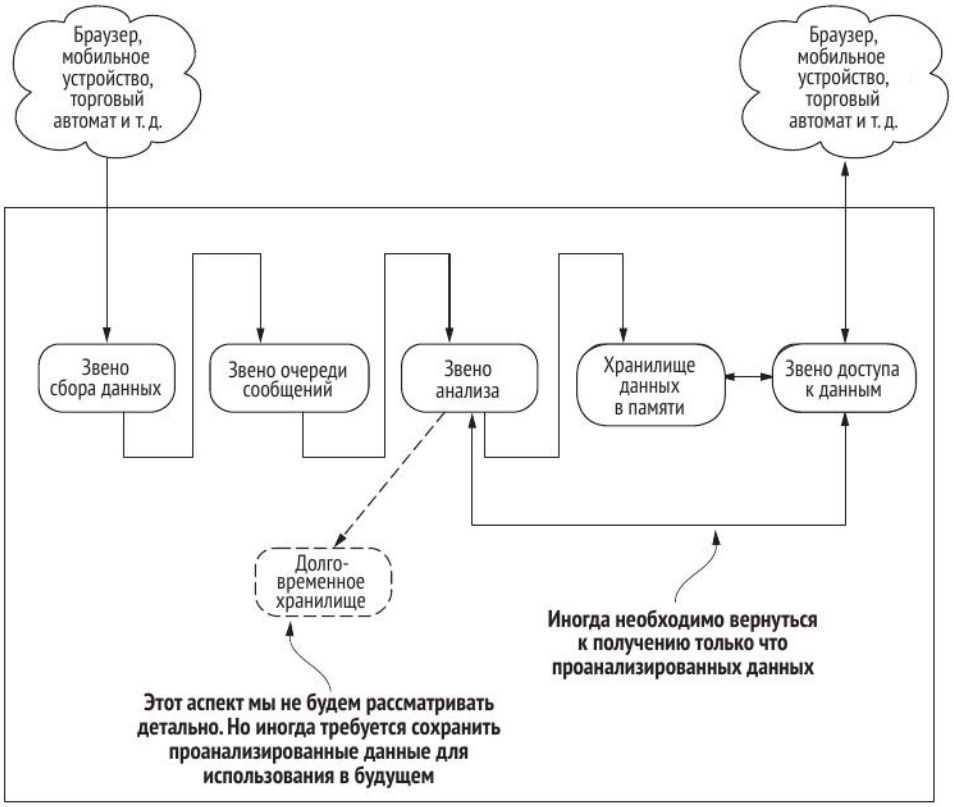
Эта насыщенная идеями книга научит вас думать об эффективном взаимодействии с быстрыми потоками данных. В ней выдержан идеальный баланс между широкой картиной и деталями реализации. На содержательных примерах и практических задачах вы узнаете о проектировании приложений, которые читают, анализируют, разделяют и сохраняют потоковые данные. Попутно вы поймете, какую роль играют такие технологии, как Spark, Storm, Kafka, Flink, RabbitMQ и другие.

Издание ориентировано на разработчиков, знакомых с концепциями реляционных баз данных.

УДК 004.6
ББК 32.972.13

ISBN 978-1-61729-228-6 (англ.)
ISBN 978-5-97060-606-3 (рус.)

Архитектурная диаграмма потоковой обработки данных



Оглавление

Предисловие

Сколько себя помню, я всегда приходил в восторг от скорости обработки данных компьютером и стремился найти способ, как решить задачу быстрее. В конце 1990-х годов я в основном занимался программированием на C++ и моим любимым ключевым словом было `__asm`, означающее, что «следующий далее блок написан на языке Ассемблера». Я понимал, что происходит на машинном уровне. В начале 2000-х я работал над программным обеспечением мобильных устройств, и тогда снова стал актуальным вопрос, как быстрее синхронизировать данные или ускорить работу устройств PalmPilot и Windows CE? В то время мы имели дело с гигантскими (по тогдашним меркам, конечно) медицинскими базами данных (объемом 25–50 МБ), для хранения которых в PalmPilot нужно было вставлять внешнюю карту памяти, и разрабатывали приложения, которые должны были обеспечить быстрый интерактивный поиск и просмотр этих данных.

По мере роста объемов данных в интересующих меня отраслях я оказался в точке, где большие наборы данных сталкиваются с необходимостью их быстрой обработки для понимания сути дела. Данные генерировались во все ускоряющемся темпе, а бизнес хотел все быстрее получать ответы на вопросы, связанные с этими данными. Как раз то, что мне было надо: большие данные и жажда скорости. Где-то в 2001 году я начал работать над приложениями для анализа рынка и электронной коммерции, когда данные непрерывно обновлялись, а нам нужно было извлекать из них смысл почти в режиме реального времени. В 2009 году я перешел в компанию Webtrends, где моя любовь к скорости и своевременной обработке данных достигла зрелости. Основным бизнесом Webtrends была аналитика, и наши идеи, касающиеся аналитики в реальном времени, как раз стали вызывать интерес у заказчиков. Первый проект, над которым я работал, заключался в том, чтобы выводить ключевые показатели на информационную панель с задержкой не более пяти минут относительно потока данных о событиях на сайте, где бы они не происходили. В то время это выходило за рамки обыденности.

В 2011 году я вошел в группу разработки новых продуктов. Нашей задачей было и дальше продвигать идею аналитики в реальном времени и попытаться взорвать нашу отрасль изнутри. Мы потратили уйму времени на исследования, создание прототипа и обдумывание следующего шага, и тут случился идеальный шторм. Мы уже поглядывали в сторону Apache Kafka, когда в сентябре 2011-го был открыт исходный код Apache Storm. Мы тут же набросились на него с бешеным энтузиазмом. Уже зимой мы продемонстрировали то, чем занимались, нескольким заинтересованным

клиентам. В то время мы не оглядывались назад и всецело сосредоточились на выполнении соглашения об уровне услуг (Service Level Agreement, SLA), звучавшем примерно так: «от щелчка мышью в любой точке света до информационной панели – три секунды!» Спустя много месяцев труда куда большей по размеру командой мы выполнили свое обещание, и нас признали технологией года в области цифровой аналитики (Digital Analytics New Technology of the Year – www.digitalanalyticsassociation.org/awards2013). Я был поглощен архитектурным проектированием и разработкой всех аспектов этого решения – от сбора данных до первого варианта пользовательского интерфейса (который ласково называл «голым скелетом», поскольку опыта в области построения UI у меня не было).

Мы не оставили свое увлечение и начали знакомиться с технологией Spark Streaming, когда она еще не вышла за стены Berkley AMPLab. С той поры я занимаюсь созданием все новых потоковых систем, стремясь к тому, чтобы полезная информация извлекалась из данных со скоростью мысли. Я выступаю на международных конференциях по этой тематике и работаю с компаниями по всему миру, помогая им проектировать и разрабатывать системы, решающие проблемы потоковой обработки данных.

Даже сегодня не все хорошо понимают, из каких частей состоит потоковая система. Найти информацию об отдельных компонентах нетрудно, но четкое понимание того, как устроен весь конвейер и как связаны между собой его звенья, встречается редко.

Поэтому я с огромным удовольствием попытался поделиться в этой книге своим практическим опытом и знаниями. Моя цель – предложить прочный фундамент, на котором можно строить и исследовать полноценные потоковые системы.

Благодарности

Прежде всего хочу поблагодарить свою семью за поддержку во время работы над книгой. На протяжении многих выходных и вечеров я только и говорил: «Извините, я не могу помочь в саду (или поиграть в лакросс, или пойти в гости)». Конечно, и детям нелегко было это слышать, и жена устала все время брать на себя мою работу. Но на всем тернистом пути поддержка со стороны семьи ни разу не пошатнулась, она была для меня постоянным источником бодрости и вдохновения. Мой долг перед женой и детьми огромен, для его выражения одного «спасибо» недостаточно.

Спасибо Карен, моему редактору-консультанту, за бесконечное терпение, понимание и готовность в любое время обговаривать со мной различные вопросы. Спасибо Робину, рецензенту со стороны издательства, который поверил в меня, пестовал идею этой книги и служил слушателем, благодаря которому поезд не сошел с рельс на ранних этапах пути. Спасибо Берту, который учил меня рассказывать историю, находить подходящий уровень глубины и педагогически правильно выстраивать техническую книгу. Спасибо техническому редактору Грегору, чьи глубокие и полезные замечания помогли сделать книгу такой, какой вы ее видите. И наконец, спасибо всему коллективу издательства Manning за фантастическую работу, благодаря которой мы все-таки дошли до конца.

Спасибо всем, кто купил и прочитал ранние варианты рукописи в рамках программы предварительного ознакомления, всем, кто оставлял сообщения на форуме автора, а также многочисленным рецензентам за их бесценные замечания, в частности: Эндрю Джибсону (Andrew Gibson), д-ру Тобиасу Бюргеру (Tobias Bürger), Джейку Маккрэри (Jake McCrary), Родриго Абрю (Rodrigo Abreu), Эндт Кеффаласу (Andy Keffalas), Джону Гатри (John Guthrie), Космасу Чатзимихалису (Kosmas Chatzimichalis), Джулиано Бертоти (Giuliano Bertoti), Карлосу Куротто (Carlos Curotto), Энди Киршу (Andy Kirsch), Дугласу Данкану (Douglas Duncan), Джеффу Смиту (Jeff Smith) и Серджио Фернандесу Гонсалесу (Sergio Fernández González), Яромиру Д. Б. Немецу (Jaromir D. B. Nemec), Хосе Самонте (Jose Samonte), Яну Ноннену (Jan Nonnen), Ромиту Сингхаи (Romit Singhai), Крису Аллену (Chris Allan), Джонатану Томсу (Jonathan Thoms), Стивену Дженкису (Steven Jenkins), Лее Джилберту (Lee Gilbert), Амандипу Хурана (Amandeep Khurana), Чарли Гейнсу (Charlie Gaines). Без вас эта книга не состоялась бы.

Еще многие принимали участие в создании книги тем или иным способом, но я не могу упомянуть всех поименно, потому что так благодарности никогда бы не закончились. Тем не менее большое спасибо всем, кто помогал мне на этом пути!

Об этой книге

Системы реального времени появились давно, но в течение многих лет режим реального времени и потоковой обработки оставался вотчиной аппаратных систем. В таких системах невыполнение SLA угрожает человеческой жизни. Но за последние десять лет появились и стали быстро распространяться системы почти реального времени. Примеры потоковой обработки встречаются повсюду: социальные сети, игры, умные города, интеллектуальные измерительные устройства, ваша новая стиральная машина – список можно продолжать долго. Подумайте вот о чем: если считать, что байт равен одному галлону воды, то сегодня среднего размера дом можно было бы заполнить за 10 секунд, а к 2020 году это займет всего 2 секунды. Чтобы извлечь смысл из такого потопа данных, нужны потоковые системы.

У этой книги, посвященной идеям потоковой обработки данных в режиме реального времени, две цели. Первая – научить вас рассуждать о конвейере в целом, чтобы вы могли не только построить потоковую систему, но и понимали, на какие компромиссы приходится идти в каждом ее звене. Вторая – заложить фундамент, опираясь на который, вы сможете глубже исследовать каждое звено, если это потребуется в интересах дела или просто ради удовлетворения любопытства.

Как работать с книгой?

Предполагалось, что книгу будут читать с начала до конца, но каждая глава содержит достаточно информации, чтобы ее можно было прочитать и понять отдельно от других. Поэтому, если вы захотите узнать о конкретном звене, можете перейти прямо к соответствующей главе, а затем воспользоваться полученной информацией в качестве основы для более глубокого изучения остальных глав.

Кому стоит прочитать эту книгу?

Эта книга рассчитана на разработчиков и архитекторов, но написана так, чтобы ее легко могли понять технические руководители и люди, принимающие решения о развитии бизнеса, – никакого предварительного знакомства с системами реального времени или потоковой обработки данных не предполагается. Единственное техническое требование – умение читать код, написанный на Java. На этом языке написаны как сами рассматриваемые системы, так и пример кода в главе 9.

Структура книги

Структура книги изображена на рис. 1.

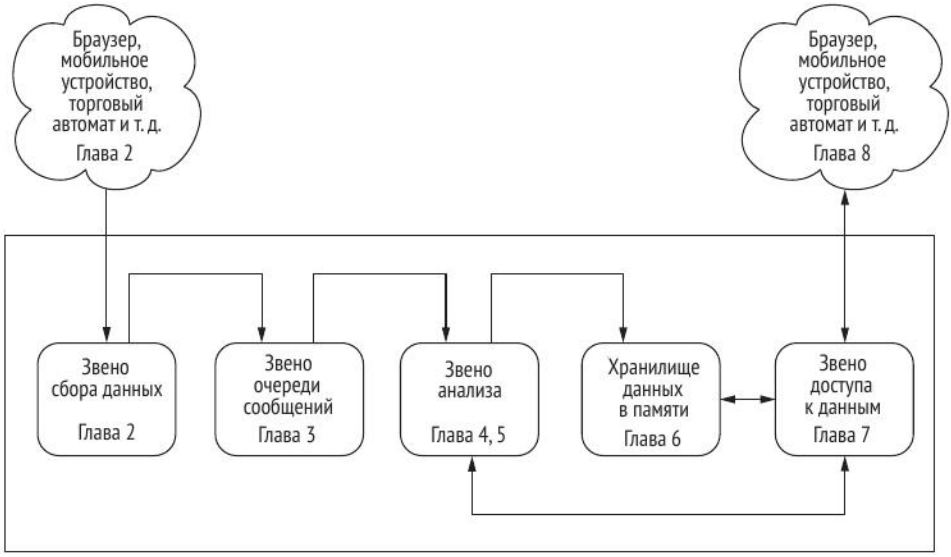


Рис. 1. Архитектурная диаграмма с разбиением на главы

Глава 1 содержит краткий план всей книги, он может послужить в качестве карты на случай, если понадобится найти нужное звено. Вслед за планом определяется, что такое система реального времени, в чем состоит различие между системой реального времени и своевременной системой, и кратко затрагивается вопрос о важности безопасности (которому можно посвятить отдельную книгу).

В главе 2 рассматриваются все аспекты сбора данных для потоковой системы: интерактивные средства, масштабирование и отказоустойчивость. Эта глава содержит все, относящееся к звену сбора данных. Прочитав ее, вы будете готовы к созданию надежного и масштабируемого звена.

Глава 3 посвящена тому, как разорвать связь между собираемыми и анализируемыми данными, разместив между ними звено очереди сообщений. Вы узнаете, зачем нужно это звено, что такое долговечность сообщений, какие бывают семантики доставки сообщений и как выбрать технологию, подходящую для решения поставленной перед вами задачи.

В главе 4 мы займемся распространенными архитектурными паттернами распределенной потоковой обработки и рассмотрим, что означает семантика доставки сообщений в этом звене, как работать с состоянием, что такое отказоустойчивость и почему она необходима.

В главе 5 мы перейдем от обсуждения архитектуры к опросу потока, проблемам, касающимся времени, и четырем популярным техникам

обобщения. Если глава 4 отвечает на вопрос, *что* такое распределенная система потоковой обработки, то в главе 5 рассматривается ответ на вопрос, *как* она устроена.

В главе 6 обсуждаются варианты хранения данных в памяти в процессе анализа и после него. Мы не будем тратить времени на обсуждение методов долгосрочного хранения данных на диске, потому что такие решения зачастую используются за рамками потокового анализа и не способны обеспечить такую же производительность, как хранилища в памяти.

В главе 7 мы начнем разговор о том, что делать с данными, которые мы собрали и проанализировали. Здесь обсуждаются варианты коммуникации и протоколы отправки данных потоковому клиенту. Попутно мы ответим на вопрос, как бизнес-требования соотносятся с различными протоколами и как выбрать подходящий протокол.

В главе 8 речь идет о концепциях, которые следует иметь в виду при построении потокового клиента. Эта глава не просто о том, как разработать веб-приложение на основе HTML, в ней рассматриваются гораздо более глубокие, низкоуровневые вопросы проектирования клиентской части потоковой системы.

Глава 9... что ж, если вы прочитали все, что ей предшествует, принимайте поздравления! В первых восьми главах было рассмотрено много материала. А в главе 9 мы претворим теорию в практику – построим полный конвейер потоковой обработки данных и обсудим, как сделать из демонстрационного примера производственное приложение.

О коде

Весь код, представленный в последней главе, можно найти в сети. Код можно скачать бесплатно с сайта издательства Manning по адресу `www.manning.com/books/streaming-data` или найти его на GitHub по адресу `https://github.com/apsaltis/StreamingData-Book-Examples`.

Код разбит на отдельные Maven-проекты, по одному для каждого рассматриваемого в главе 9 звена. Инструкции по сборке и запуску программ приводятся по ходу дела.

Исходный код в листингах и внутри текста набран `моноширинным шрифтом`. Некоторые листинги аннотированы, чтобы пояснить ключевые моменты.

Об авторе

Эндрю Пселтис одержим потоковыми системами, и все его устремления направлены на то, чтобы извлекать смысл из данных со скоростью мысли. Большую часть своего времени (когда не спит) он размышляет о потоковых системах, пишет о них или разрабатывает их. Он помогает клиентам всех мастей строить или исправлять сложные потоковые системы, расска-

зывает о потоковой обработке на всех континентах и учит других создавать потоковые системы. Время, остающееся от трудов, он посвящает прелестной жене, двоим ребятишкам и игре в лакросс – как болельщик.

Автор в сети

Приобретение книги «Потоковая обработка данных» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/books/streaming-data. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору какие-нибудь хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

Об иллюстрации на обложке

Рисунок на обложке книги называется «Традиционный костюм марокканца, зима 1695 года». Рисунок взят из четырехтомного сочинения Томаса Джеффри «Коллекция одежды различных народов от древних времен до наших дней», изданного в Лондоне между 1757 и 1772 годами. На титульной странице говорится, что это гравюры на меди, раскрашенные вручную с добавлением гуммиарабика. Томаса Джеффри (1719–1771) называли «географом короля Георга III». В свое время он был главным в Англии поставщиком карт. Он гравировал и печатал карты для правительства и других официальных учреждений, а также изготавливал самые разные коммерческие карты и атласы, особенно Северной Америки. Работа в качестве картографа возбудила в нем интерес к традиционным костюмам в тех местах, которые он наносил на карту. Эти костюмы блистательно изображены в его коллекции.

Увлечение дальними землями и путешествиями ради удовольствия в конце XVIII века было еще относительно новым явлением, и такие коллекции были весьма популярны, поскольку давали как туристам, так и любителям странствовать, не вылезая из кресла, возможность познакомиться с обитателями других стран. Разнообразие иллюстраций в четырехтомни-

ке Джеффри красноречиво свидетельствует об уникальности и индивидуальности народов мира, которое можно было наблюдать каких-то 200 лет назад. С тех пор манера одеваться сильно изменилась, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о странах или областях. Но можно взглянуть на это и с оптимизмом – мы обменяли культурное и визуальное разнообразие на иное устройство личной жизни – основанное на многостороннем и стремительном технологическом и интеллектуальном развитии.

Во времена, когда трудно отличить одну компьютерную книгу от другой, издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из собрания Джеффри.

Часть I

Новый целостный подход

Данные окружают нас повсюду, и каждый день в сети появляются новые источники данных. Если вы пока не сталкивались с созданием систем обработки данных в реальном времени, то столкнетесь в ближайшем будущем. Само существование все большего числа предприятий зависит от умения обрабатывать потоки данных и принимать решения в зависимости от результатов. В первой части книги мы рассмотрим жизненный цикл потоковой системы с момента попадания в нее данных и до момента их отображения или потребления другими системами.

В главе 1 мы познакомимся с идеей потоковой обработки данных и с применяемой терминологией. Для разных людей слова «потоковая обработка данных» и «реальное время» могут означать разные вещи. Поэтому мы уточним, что понимаем под этими терминами *мы*, и приведем архитектурную диаграмму, которую будем наполнять содержанием на протяжении всей книги. В конце главы 1 мы скажем несколько слов о безопасности в контексте потоковых систем.

Точкой входа в потоковую систему является сбор данных. Вопрос о том, как собирать данные и предотвращать их потерю, находится в центре главы 2.

Собранные данные нужно как можно быстрее поместить в очередь сообщений (иногда ее называют *буфером* сообщений). Применяемая в этом звене технология характеризуется различными уровнями долговечности, семантикой доставки и влиянием на производителей и потребителей данных. В главе 3 мы расскажем, как учитывать эти особенности, и поделимся некоторыми рекомендациями.

Глава 4 посвящена анализу потоковых данных. Основное внимание уделяется анализу данных в темпе поступления, распространенным архитектурам потоковой обработки и ключевым функциям, общим для всех движков распределенной потоковой обработки.

При работе с распределенной системой потоковой обработки нужно учитывать много нюансов. Например, время. Как следует представлять себе время в потоковой системе? Ответ на этот вопрос дается в главе 5. Здесь же рассматриваются четыре общеупотребительных метода обобщения, применяемых в процессе анализа потоковых данных.

После анализа поток данных иногда требуется сохранить. Звучит странно – зачем сохранять поток, раз мы его обрабатываем! В главе 6 объясняется, почему возникает необходимость в сохранении данных, что именно следует сохранять и как это правильно сделать в процессе обработки потока.

Когда мы доберемся до главы 7, данные уже собраны, проанализированы и, возможно, сохранены. И следующий шаг – сделать данные доступными, потому что в конечном итоге мы хотим передать результаты анализа другой системе, которая сможет предпринять какие-то действия на основе потоковых данных.

В главе 8 мы подведем итог первой части, обсудив основные принципы построения потокового клиента. Мы познакомимся с веб-клиентом и в заключение поговорим об опросе потока данных, поскольку пользователям это бывает нужно.

Глава 1

Введение в потоковую обработку данных

Краткое содержание главы:

- различие между системами реального времени и системами потоковой обработки данных;
- почему потоковая обработка данных важна;
- архитектурная диаграмма;
- безопасность в системах потоковой обработки данных.

Данные окружают нас повсеместно: благодаря телефонам, кредитным картам, оборудованным датчиками зданиям, торговым автоматам, термостатам, поездам, автобусам, самолетам, сообщениям в социальных сетях, цифровым фотографиям и видео – список можно продолжать. В отчете, датированным маем 2013 года, скандинавский исследовательский центр Sintef дал оценку, согласно которой приблизительно 90% данных, существовавших на тот день в мире, были сгенерированы в течение двух предшествующих лет. В апреле 2014 года EMC вместе с IDC выпустили седьмое ежегодное исследование цифровой вселенной (www.emc.com/about/news/press/2014/20140409-01.htm), в котором утверждалось, что цифровая вселенная удваивается каждые два года, и между 2013 и 2020 годом ее размер возрастет десятикратно – с 4,4 до 44 триллионов гигабайт. Не знаю, как вы, а я с трудом могу переварить эти цифры и соотнести их с чем-нибудь в реальном мире. Хорошее сравнение приведено в том же отчете: если считать, что один байт – это галлон воды, то для заполнения дома среднего размера в 2013 году было достаточно 10 секунд. А в 2020-м хватит и двух секунд.

Понятие «больших данных» существует уже довольно давно, но теперь у нас есть технология, позволяющая сохранить и проанализировать все собранные данные. Это не устраняет потребности использовать данные

в подходящем контексте, но стало гораздо проще задавать интересные вопросы о данных, быстрее принимать более обоснованные решения и предлагать службы, позволяющие потребителям и предприятиям использовать данные о том, что происходит вокруг.

Мы живем в мире, который все сильнее ориентирован на *сейчас*: это и социальные сети, и розничные магазины, отслеживающие перемещение покупателей по проходам, и датчики, реагирующие на изменения в окружающей среде. Нет недостатка в примерах использования данных в момент порождения. Но вот чего не хватает, так это единого способа рассуждать о таких системах – и проектировать их, – который учитывал бы не только имеющиеся, но и будущие службы.

В этой книге представлены общие архитектурные принципы, позволяющие обсуждать и проектировать системы, способные ответить на все поразительные вопросы об окружающих нас данных, которые еще только предстоит задать. Даже если вы никогда не проектировали, не разрабатывали и не работали с системами реального времени или системами для обработки больших данных, эта книга все равно послужит полезным руководством. Она посвящена важным идеям потоковой обработки данных в режиме реального времени. Но никакого предварительного опыта в этой области не предполагается, так что книга будет отличным подспорьем для разработчиков и архитекторов, желающих побольше узнать о таких системах. Она написана так, что будет полезной также техническим руководителям и лицам, принимающим решения о развитии бизнеса.

Чтобы подготовить почву, мы в этой главе познакомимся с концепцией систем потоковой обработки данных и с архитектурной диаграммой в предвкушении более глубокого изучения каждого звена в последующих главах. Но первым делом нужно понять, что вообще такое системы реального времени и потоковой обработки.

1.1. Что такое система реального времени?

Системы реального времени и *вычисления в режиме реального времени* существуют уже несколько десятков лет, но с пришествием Интернета их популярность резко возросла. Увы, вместе с популярностью пришли сомнения и споры. Из чего же состоит система реального времени?

Есть три типа систем реального времени: *жесткого реального времени*, *мягкого реального времени* и *почти реального времени*. Определения жесткого и мягкого реального времени взяты мной из книги Hermann Kopetz «Real-Time Systems» (Springer, 2011), а определение почти реального времени – со страницы http://c2.com/cgi/wiki?NearRealTime. За примером неоднозначности не надо ходить дальше определения на сайте Dictionary.com: «Обозначает или относится к системе обработки данных, работающей немного медленнее, чем система реального времени». Чтобы разобраться,

в чем тут дело, в табл. 1.1 приведена широкоупотребительная классификация систем реального времени вместе с основными характеристиками, по которым они различаются.

Таблица 1.1. Классификация систем реального времени

Классификация	Примеры	Единица измерения задержки	Терпимость к запаздыванию
Жесткая	Кардиостимулятор, антиблокировочная тормозная система	Микросекунды – миллисекунды	Нулевая – полный отказ системы, потенциальная смерть человека
Мягкая	Система резервирования авиабилетов, электронные биржевые торги, VoIP (Skype)	Миллисекунды – секунды	Низкая – без отказа системы, без угрозы жизни
Почти	Видео в Skype, домашняя автоматика	Секунды – минуты	Высокая – без отказа системы, без угрозы жизни

Опознать систему жесткого реального времени довольно просто. Почти всегда они встречаются во встраиваемых системах и характеризуются очень строгими временными ограничениями, невыполнение которых может привести к полному отказу системы. Вопрос о проектировании и реализации систем жесткого реального времени хорошо изучен в литературе, но выходит за рамки этой книги (интересующихся читателей отправляю к вышеупомянутой книге Германа Копеца).

Решить, идет ли речь о системе мягкого или почти реального времени, – интересное упражнение, поскольку частично перекрывающиеся определения создают почву для недоразумений. Приведем три примера.

- Человек, на твиттер которого вы подписаны, отправляет сообщение, и через несколько мгновений вы видите его в своем клиенте.
- Вы следите за самолетами в окрестности Нью-Йорка, пользуясь службой Live Flight Tracking, предоставляемой компанией FlightAware (http://flightaware.com/live/airport/KJFK).
- Вы пользуетесь приложением NASDAQ Real Time Quotes (www.nasdaq.com/quotes/real-time.aspx) для наблюдения за котировками своих любимых акций в режиме реального времени.

Хотя это совершенно разные системы, на рис. 1.1 показано, что между ними общего.

Во всех трех примерах разумно предположить, что допустимая временная задержка – всего несколько секунд, что угрозы жизни нет и что случайное запаздывание на несколько минут не приведет к полному отказу системы. Так? Если вы видите отправленный твит почти сразу, это мягкое или почти реальное время? А как насчет наблюдения за состоянием рейса или за котировками акций в режиме реального времени? В обоих случаях

возможно запаздывание – например, из-за медленной сети Wi-Fi в кофейне или на борту самолета. На этих примерах видно, что граница между системами мягкого и почти реального времени размыта, иногда исчезает вовсе, очень субъективна и часто зависит от потребителя данных.

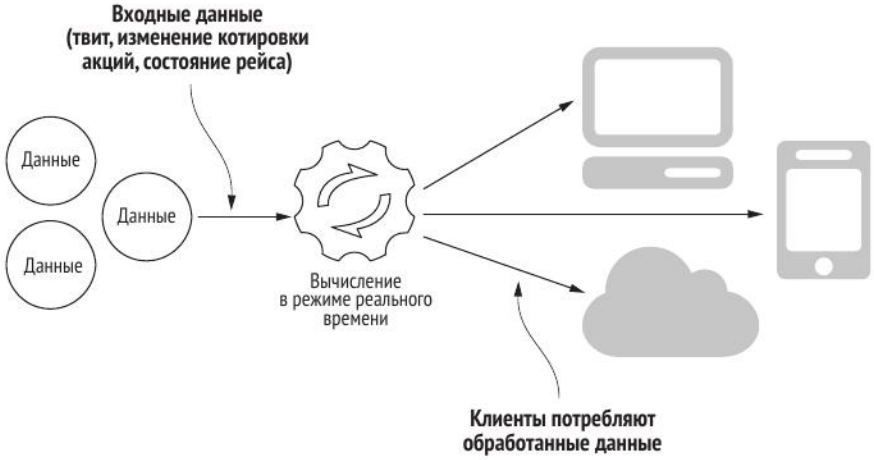


Рис. 1.1. Типичная система реального времени с потребителями

А теперь исключим из рассмотрения потребителя и сосредоточимся на самих службах.

- Твит отправляется в Твиттер.
- Служба Live Flight Tracking от компании FlightAware отслеживает авиарейсы.
- Приложение NASDAQ Real Time Quotes отслеживает котировки акций.

Мы, конечно, не знаем, как эти системы устроены внутри, но относительно любой из них можем задать один и тот же вопрос:

> *Является ли процесс доставки данных до точки, в которой их можно потребить, процессом мягкого или почти реального времени?*

Графически это изображено на рис. 1.2.

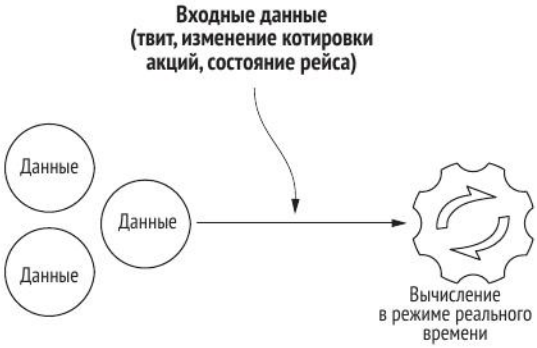


Рис. 1.2. Типичная система реального времени без потребителей

Если абстрагироваться от потребителей данных и сосредоточиться только на их обработке, то ответ изменится? Например, как бы вы классифицировали следующие процессы:

- твит отправлен в Твиттер;
- твит отправлен человеком, за которым вы наблюдаете в своем твиттер-клиенте.

Если вы классифицировали их по-разному, то почему? Из-за задержки или воспринимаемой задержки появления твита в клиенте? Проходит некоторое время – и граница между системой мягкого и почти реального времени совсем размывается. Зачастую пользователи называют их просто системами реального времени. В этой книге я собираюсь использовать более правильный способ идентификации таких систем.

1.2. Различия между системами реального времени и потоковыми системами

Сейчас уже должно быть ясно, что система считается системой мягкого или почти реального времени в зависимости от того, как воспринимают задержку пользователи. На простых примерах мы видели, что провести различие между этими типами систем реального времени не всегда просто. А если в оценке системы принимает участие несколько людей, то проблема может еще обостриться. Наша цель – выработать общий язык, на котором можно было бы описывать такие системы. Взглянув на проблему в целом, мы приходим к выводу, что пытаемся использовать один термин для определения двух частей большой системы. На рис. 1.3 показано, что такой подход обречен на неудачу: очень трудно разговаривать о системе с другими людьми, потому что у нас нет четкого определения.

На левом рисунке мы видим службу нежесткого реального времени, или *вычислительную* часть системы, а на правом – клиентов, или *потребляющую* часть системы.

Определение: система потоковой обработки данных. Во многих ситуациях вычислительная часть системы работает в режиме нежесткого реального времени, но клиенты потребляют данные не в реальном времени из-за сетевых задержек, дизайна приложения или просто потому, что клиентское приложение не запущено. Иначе говоря, мы имеем службу нежесткого реального времени и клиентов, которые потребляют данные, когда захотят. Это и называется *системой потоковой обработки данных* – система нежесткого реального времени, которая делает данные доступными, когда клиентское приложение хочет их видеть. Это не система мягкого или почти реального времени, это потоковая система.

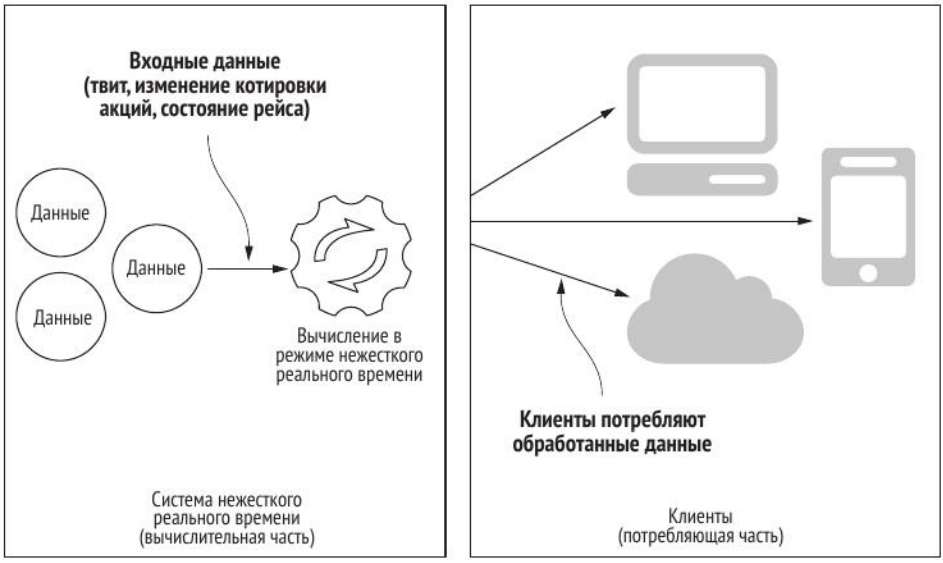


Рис. 1.3. Разделение вычисления в реальном времени и потребления

На рис. 1.4 показано, как это определение применяется к архитектуре на рис. 1.3.

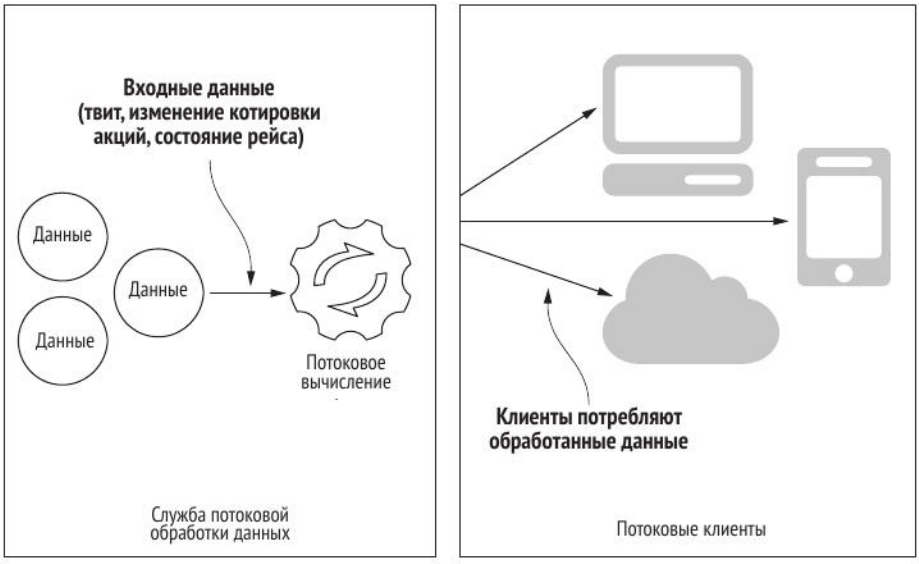


Рис. 1.4. Первый взгляд на систему потоковой обработки данных

Концепция потоковой обработки данных исключает всякие недоразумения, связанные с нечетким различием между системами мягкого реального времени, почти реального времени и вообще нереального времени, и позволяет сосредоточиться на проектировании систем, которые достав-

ляют информацию в тот момент, когда клиент ее запрашивает. Рассмотрим приведенные выше примеры с точки зрения потоковой обработки. Сможете ли вы в каждом случае отделить потоковую службу от потокового клиента?

Человек, на твиттер которого вы подписаны, отправляет сообщение, и через несколько мгновений вы видите его в своем клиенте.

Вы следите за самолетами в окрестности Нью-Йорка, пользуясь службой Live Flight Tracking, предоставляемой компанией FlightAware.

Вы пользуетесь приложением NASDAQ Real Time Quotes для наблюдения за котировками своих любимых акций в режиме реального времени.

Ну и что у вас получилось? Лично я думаю так.

- *Твиттер* – потоковая система, которая обрабатывает твиты и позволяет клиентам запрашивать последние твиты в тот момент, когда они нужны; иногда спустя несколько секунд после момента отправки, а иногда – через несколько часов.
- *FlightAware* – потоковая система, которая обрабатывает актуальную информацию о состоянии рейсов и позволяет клиентам запрашивать данные по конкретным аэропортам или рейсам.
- *NASDAQ Real Time Quotes* – потоковая система, которая обрабатывает котировки всех акций и позволяет клиентам запрашивать последнюю котировку определенных акций.

Вы обратили внимание, что теперь нет нужды задумываться о классификации системы реального времени? Важно лишь, какие данные и каким образом служба предоставляет своим клиентам, когда они посылают запрос. А раз так, то мы можем назвать такую систему *своевременной* – доставляющей данные в тот момент, когда они необходимы. Мы, конечно, не знаем, как эти системы в действительности работают, но ничего страшного. Мы научимся собирать подобные системы из компонентов с открытым исходным кодом, которые предназначены для потребления, обработки и представления потоков данных.

1.3. Архитектурная диаграмма

Разобравшись, что такое системы реального времени и потоковые системы, мы можем перейти к архитектурной диаграмме, которой будем пользоваться на протяжении всей книги. Она позволит рассуждать обо всех потоковых системах единообразным способом – на нашем языке паттернов. Эта диаграмма изображена на рис. 1.5. Познакомьтесь с ним.

По ходу изложения мы рассмотрим все звенья, показанные на рис. 1.5, не упуская из виду картину в целом. Хотя на диаграмме у каждого звена есть название, следует иметь в виду, что звенья определены не так строго и непоколебимо, как в некоторых других архитектурах. И хотя мы называ-

ем их звеньями, используются они скорее, как детали конструктора LEGO, что дает возможность спроектировать решение, отвечающее конкретной задаче. Наши звенья не подразумевают какого-то определенного сценария развертывания. Во многих случаях они даже физически находятся в различных местах.

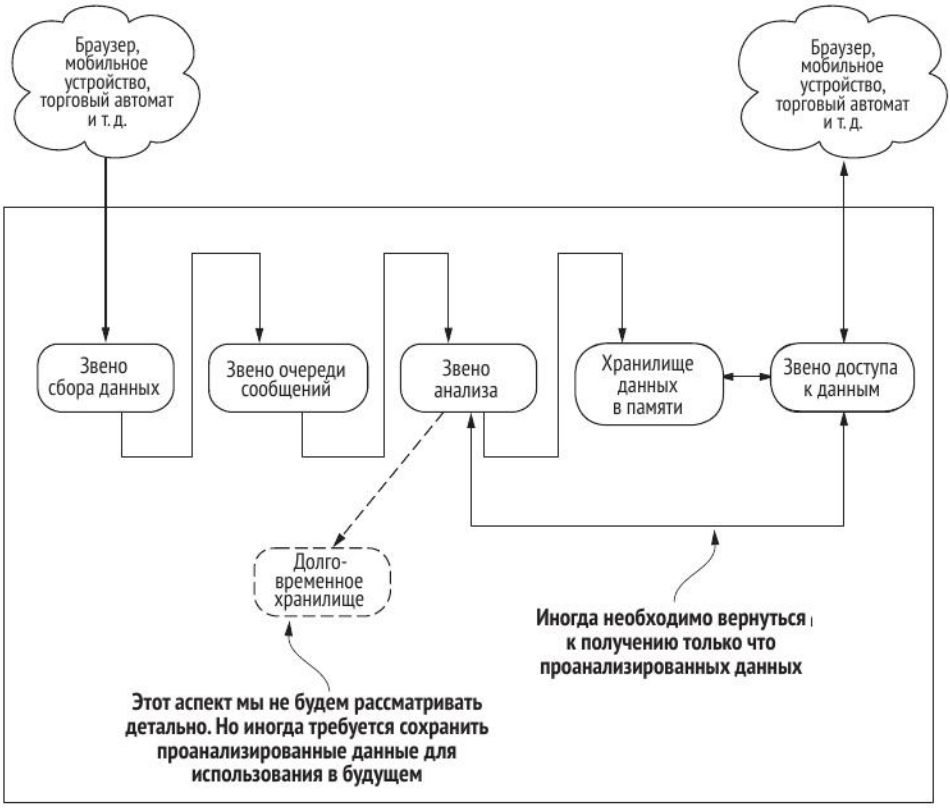


Рис. 1.5. Архитектурная диаграмма потоковой обработки данных

Обратимся к приведенным выше примерам. Посмотрим, как служба Твиттера ложится на нашу архитектуру.

- *Звено сбора данных*. Отправленные пользователями твиты собираются службами Твиттера.
- *Звено очереди сообщений*. Нет сомнений, что Твиттер задействует центры обработки данных в разных местах, и, вероятно, сбор твитов производится не там же, где анализ.
- *Звено анализа*. Я уверен, что эти 140 символов подвергаются самым разным способам обработки, но уж, как минимум, Твиттер должен определить, кто подписан на данный твит.
- *Звено долгосрочного хранения*. Хотя мы и не собираемся обсуждать этот необязательный слой, нетрудно догадаться, что раз можно уви-

деть старые твиты, значит, они хранятся в каком-то постоянном хранилище.

- *Звено хранилища данных в памяти*. Твиты, отправленные в течение нескольких последних секунд, скорее всего, хранятся в памяти.
- *Доступ к данным*. Любой твиттер-клиент должен подключаться к Твиттеру, чтобы получить доступ к службе.

Самостоятельно проведите такой же анализ для двух других примеров.

- *FlightAware* – потоковая система, которая обрабатывает актуальную информацию о состоянии рейсов и позволяет клиентам запрашивать данные по конкретным аэропортам или рейсам.
- *NASDAQ Real Time Quotes* – потоковая система, которая обрабатывает котировки всех акций и позволяет клиентам запрашивать последнюю котировку определенных акций.

Получилось? Не переживайте, если задача показалась трудной или непривычной. В последующих главах примеров будет предостаточно. Мы проработаем их вместе, обращая внимание на каждое звено, и тогда станет понятно, как из этих деталей LEGO можно собрать решения различных практических задач.

1.4. Безопасность в контексте потоковых систем

На нашей архитектурной диаграмме вы не найдете явных упоминаний безопасности. Во многих случаях безопасность важна, но ее можно естественно наложить на эту архитектуру, как показано на рис. 1.6.

Мы не будем тратить времени на детальное обсуждение вопросов безопасности, но по ходу дела я иногда буду вспоминать об этом, чтобы вы видели место безопасности в общей картине и могли поразмыслить над тем, что это может значить для решаемых вами задач. Если вас интересует углубленное рассмотрение безопасности в распределенных системах, обратитесь к книге Ross Anderson «Security Engineering: A Guide to Building Dependable Distributed Systems» (Wiley, 2008). Ее можно почитать бесплатно по адресу www.cl.cam.ac.uk/~rja14/book.html.

1.5. Как производится масштабирование?

На самом верхнем уровне есть два способа масштабирования: вертикальное и горизонтальное.

Под *вертикальным* масштабированием понимается наращивание мощности существующего оборудования (физического или виртуального) или программного обеспечения путем добавления ресурсов. При входе в ресторан иногда можно увидеть табличку, на которой указано максимальное

число мест. Когда приходят гости, можно накрыть дополнительные столы и принести еще стулья – это и есть вертикальное масштабирование. Но если заняты все места, то новых гостей усадить некуда. В конце концов, вместимость ограничена размером ресторана. В мире компьютеров примерами вертикального масштабирования могут служить оснащение сервера дополнительной памятью, процессорами или жесткими дисками. Но, как и в случае ресторана, мы ограничены максимальной емкостью системы – физической или виртуальной.

Рис. 1.6. Архитектурная диаграмма с выделением безопасности

Горизонтальное масштабирование знаменует другой подход к проблеме. Вместо того чтобы наращивать ресурсы сервера, мы увеличиваем количество серверов. Удачным примером горизонтального масштабирования может служить автострада. Возьмем двухполосную автостраду, первоначально рассчитанную на 2000 машин в час. Со временем вдоль автострады вырастают новые дома и коммерческие заведения, в результате чего поток машин возрастает до 8000 в час. Легко представить (и, возможно, вы знакомы с этим на собственном опыте), что случится: огромные пробки в часы пик и малоприятная езда в город и обратно в любое время. Для решения проблемы строятся дополнительные полосы – в итоге автострада масштабирована по горизонтали и может справиться с возросшим трафиком. Но еще эффективнее было бы, если бы автострада могла расширяться (добавление полос) и сужаться (удаление полос)

в зависимости от текущего трафика. На пунктах пропуска в аэропорт, когда пассажиропоток мал, оператор закрывает часть линий досмотра, а когда поток увеличивается, открывает. Если ваша служба размещена на серверах облачного провайдера (Google, AWS, Microsoft Azure), то вы можете воспользоваться такой эластичностью – она иногда называется *автомасштабированием*. Идея в том, что в периоды повышенного спроса на службу автоматически добавляются новые серверы, а когда спрос падает, серверы выводятся.

Современные способы проектирования системы ориентированы больше на горизонтальное масштабирование, но это не значит, что вертикальное масштабирование пора отправлять на свалку. Вертикальное масштабирование часто применяется для определения идеальной конфигурации ресурсов для службы, а затем служба масштабируется по горизонтали. Но в этой книге мы будем иметь дело в основном с горизонтальным масштабированием.

1.6. Резюме

Итак, мы познакомились с архитектурной диаграммой, и можно подвести некоторые итоги тому, что мы узнали.

- Мы определили, что такое система реального времени.
- Мы изучили различия между системами реального времени и потоковыми (своевременными) системами.
- Мы поняли, почему потоковая обработка важна.
- Мы нарисовали архитектурную диаграмму.
- Мы обсудили место безопасности в потоковых системах.

Не расстраивайтесь, если все это пока выглядит смутно или если задача применения архитектурной диаграммы к конкретным практическим проблемам кажется неподъемной. В последующих главах мы, не торопясь, рассмотрим много разных примеров. И к концу книги все эти идеи покажутся гораздо более естественными.

А сейчас мы готовы детально рассмотреть звенья, понять, из чего они состоят и как их применять к построению системы потоковой обработки данных. С какого звена начать? Взгляните на слегка модифицированную архитектурную диаграмму на рис. 1.7.

Мы будем рассматривать звенья по одному, слева направо, так что начнем со звена сбора данных.

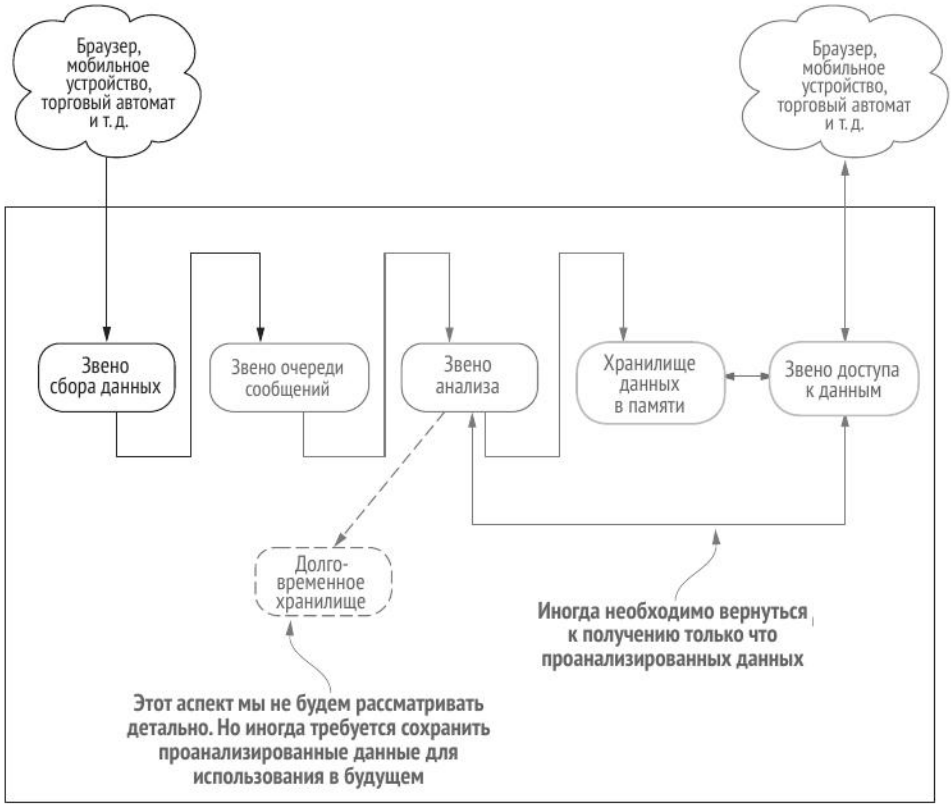


Рис. 1.7. Архитектурная диаграмма с упором на первое звено

Глава 2

Получение данных от клиентов: внесение данных

Краткое содержание главы:

- общие сведения о звене сбора данных;
- паттерны сбора данных;
- перевод звена сбора данных на новый уровень;
- защита от потери данных.

Итак, первое звено – *звено сбора данных*, являющееся точкой входа в потоковую систему. На рис. 2.1 показана немного модифицированная архитектурная диаграмма с упором на звено сбора данных.

Это звено – то место, где данные поступают в систему и начинают свое путешествие. В последующих главах мы проследим за тем, как данные распространяются по всем звеньям, а сейчас займемся звеном сбора данных. Прочитав эту главу, вы будете знать о паттернах сбора данных, о том, как масштабировать это звено и как повысить его надежность, применяя различные методы обеспечения отказоустойчивости.

2.1. Типичные паттерны взаимодействия

Вне зависимости от протокола, который клиент использует для отправки данных звену сбора данных (а в некоторых случаях само звено сбора данных играет активную роль и «вытягивает» данные), в настоящее время применяется всего несколько паттернов взаимодействия. Даже принимая во внимание протоколы, стоящие за Интернетом вещей, сами паттерны взаимодействия можно отнести к одной из следующих категорий:

- запрос-ответ;
- издатель-подписчик;

- одностороннее взаимодействие;
- запрос-подтверждение;
- поток.

Обсудим, как собираются данные с применением каждого паттерна.

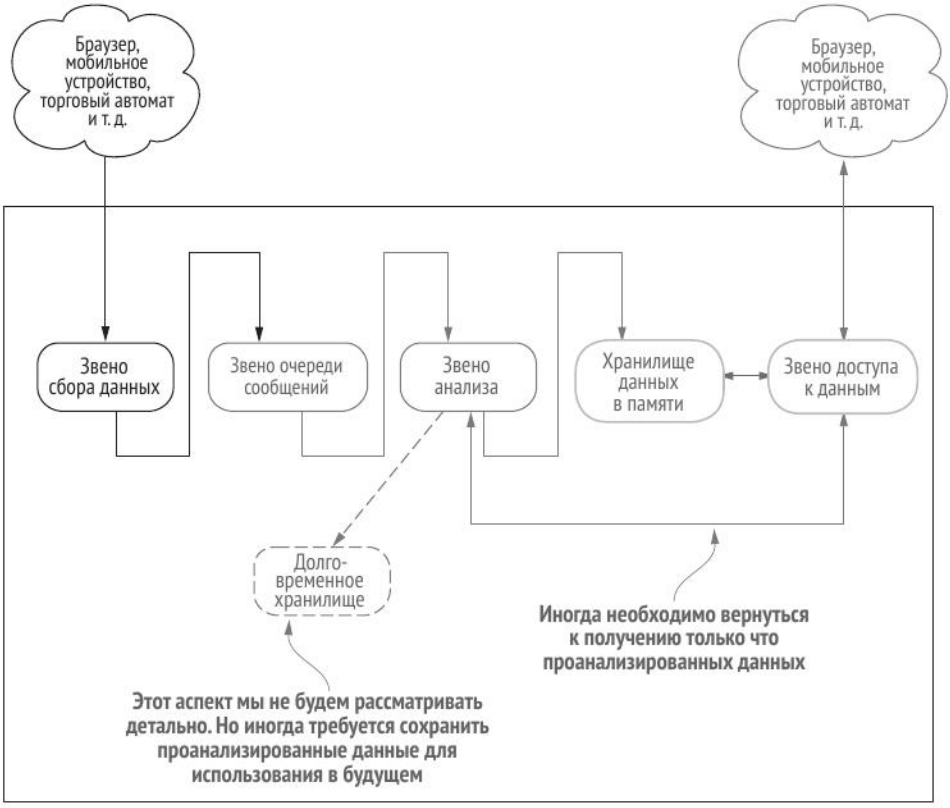


Рис. 2.1. Архитектурная диаграмма с упором на звено сбора данных

2.1.1. Запрос-ответ

Это самый простой паттерн. Он применяется, когда клиент хочет получить ответ немедленно или служба должна выполнить задание без задержки. Мы ежедневно сталкиваемся с этим паттерном, когда блуждаем в вебе, ищем информацию в сети или пользуемся своим мобильным устройством. Клиент сначала посылает запрос службе: выполнить какое-нибудь действие (например, отправить текстовое сообщение, подать заявление о приеме на работу или купить билет на самолет) или предоставить данные (например, выполнить поиск в Google или узнать прогноз погоды в своем городе). Затем служба посылает ответ клиенту. Этот паттерн показан на рис. 2.2.

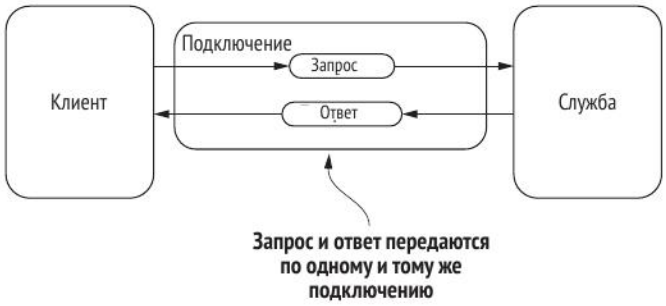


Рис. 2.2. Простой паттерн запрос-ответ

Платой за простоту синхронного паттерна запрос-ответ является тот факт, что клиент должен дождаться ответа, а служба должна ответить достаточно быстро. Для современных служб такая плата часто приводит к неприемлемому для пользователей поведению. Представьте, что вы открываете в браузере свой любимый новостной или социальный сайт, и браузер пытается запрашивать все ресурсы синхронно. Если не считать совсем уж простых служб типа запросов погоды в данный момент времени, то потенциальная задержка будет слишком велика. Во многих случаях это будет означать потерю дохода для торговца, поскольку пользователи не захотят долго ждать ответа на запрос.

Для преодоления этого ограничения есть три стратегии: на стороне клиента, на стороне службы и комбинация того и другого. Сначала рассмотрим стратегию на стороне клиента. Типичный подход – асинхронная отправка запросов – показан на рис. 2.3.

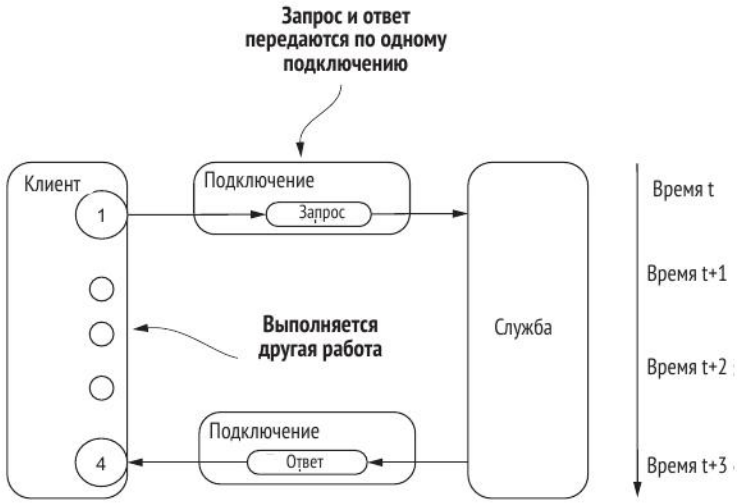


Рис. 2.3. Паттерн асинхронного запроса-ответа на стороне службы

В этом случае клиент отправляет запрос службе и продолжает заниматься своими делами, пока служба обрабатывает запрос. Этот паттерн использу-

ется во всех современных браузерах: браузер отправляет много асинхронных запросов на получение ресурсов и отображает изображения или другой контент по мере поступления. При таком способе обработки уменьшается время, затрачиваемое клиентом на ожидание ответа, и за фиксированное время клиент успевает выполнить больший объем работы. Реализация этого паттерна сегодня не представляет особых сложностей, потому что все современные языки программирования и многие каркасы поддерживают асинхронные запросы. Этот паттерн часто называют *полуасинхронным*, потому что асинхронно обрабатывается только одна часть пары запрос-ответ.

Реализация такого способа обработки на стороне службы также широко распространена и показана на рис. 2.4.

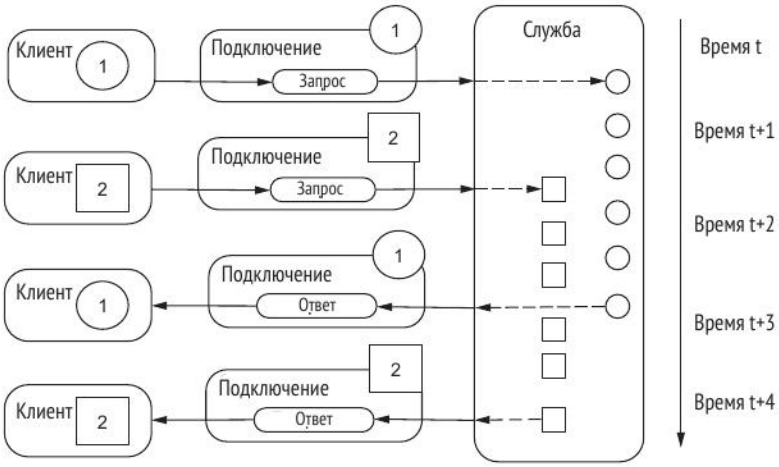


Рис. 2.4. Паттерн асинхронного запроса-ответа на стороне службы

В этом случае служба получает запрос от клиента, делегирует работу исполнителю, а по завершении отправляет клиенту ответ. Так построенные службы получаются хорошо масштабируемыми, т. е. могут обрабатывать запрос от гораздо большего числа клиентов. Этот способ широко применяется во всех каркасах для серверной разработки, существующих для всех популярных языков программирования.

В последнем варианте этого паттерна, *полноасинхронном*, и клиент, и сервер работают асинхронно; выглядит это так же, как на рис. 2.4. Многие современные клиенты и службы работают в полноасинхронном режиме.

Теперь обсудим пример использования этого паттерна при проектировании и разработке потоковой системы. Допустим, что вы работаете в транспортной отрасли и как-то, сидя за чашечкой кофе со своим другом Эриком, который связан с автомобильной промышленностью, наткнулись на идею предложить службу наблюдения за дорожной обстановкой и построения маршрутов для всех находящихся на дороге транспортных средств. Служба должна работать в режиме реального времени. Ваша ком-

пания будет отвечать за разработку службы, а компания Эрика – за создание потоковой системы, устанавливаемой в транспортное средство. Вы даже набросали эскиз решения. На рис. 2.5 показана верхнеуровневая диаграмма бортовой части системы.

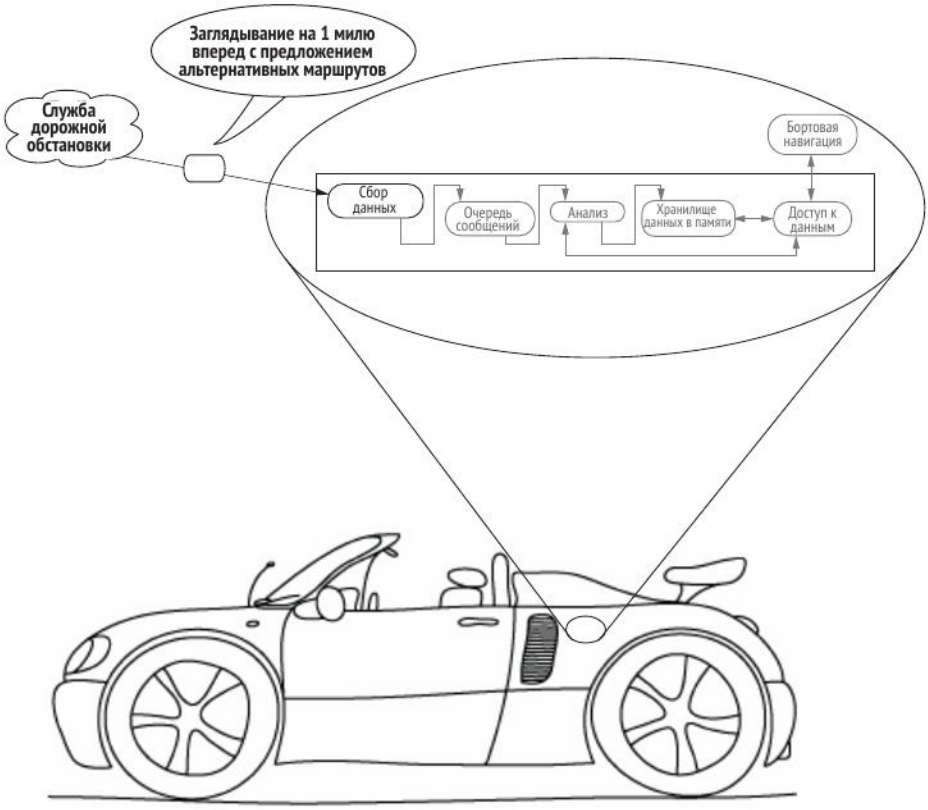


Рис. 2.5. Получение ответа на запрос о дорожной обстановке бортовой потоковой системой

Эрик собирается разработать встраиваемую потоковую систему, которая будет не только взаимодействовать с вашей службой дорожной обстановки и навигации, но и с другими службами, а возможно, и с другими транспортными средствами.

В этой ситуации вполне подходит паттерн запрос-ответ и конкретно его полноасинхронный вариант. При таком дизайне служба дорожной обстановки и навигации сможет лучше обрабатывать запросы от большого числа находящихся на дороге машин. С точки зрения бортовой потоковой системы Эрика, способность асинхронно запрашивать данные и обрабатывать их по мере поступления крайне важна. Потоковая система, придерживающаяся этого паттерна, не будет блокироваться в ожидании ответа от вашей службы и сможет выполнять другие аналитические операции в интересах транспортного средства.

Теперь Эрик может заняться бортовой частью системы, а вы – службой дорожной обстановки и построения маршрутов. Если вас заинтересовал этот паттерн, рекомендуем книгу Robert Daigneau «Service Design Patterns» (Addison-Wesley, 2011).

2.1.2. Паттерн запрос-подтверждение

Бывает так, что семантика взаимодействия похожа на паттерн запрос-ответ, но сам ответ от службы нас не интересует. Нужно лишь подтверждение того, что запрос получен. Этой ситуации соответствует паттерн запрос-подтверждение. Зачастую данные, отправленные в составе подтверждения, можно использовать для следующих запросов, например чтобы проверить статус первоначального запроса или получить окончательный ответ.

Допустим, что вы работаете в интересах маркетингового отдела компании и хотите, чтобы сайт предлагал нужный товар нужному посетителю в нужное время. Конечная цель – повысить вероятность совершения покупки при данном посещении сайта. После обсуждения с маркетологами принято решение постоянно обновлять оценку расположенности посетителя к покупке на всем протяжении посещения. Располагая такой динамически изменяющейся оценкой, сайт сможет в любой момент сделать правильное предположение, чтобы повлиять на решение о покупке. На рис. 2.6 показана общая схема такой системы.

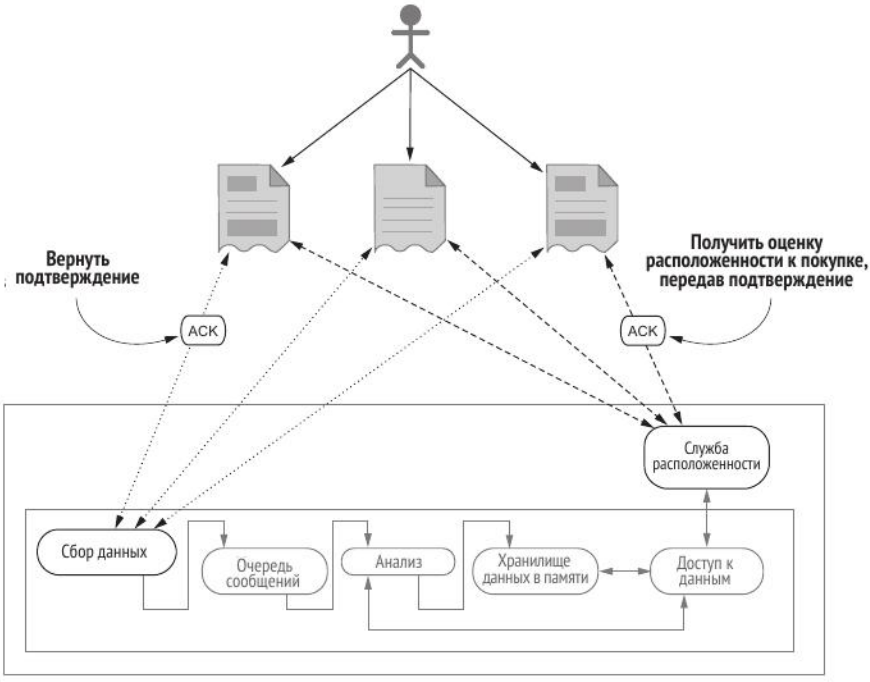


Рис. 2.6. Посетитель бродит по сайту, а в это время собираются данные и обновляется оценка расположенности к покупке

Посмотрим, как в ней устроен поток данных. Когда посетитель гуляет по нашему сайту, мы собираем данные о каждой посещенной им странице и каждом переходе по ссылке. К паттерну запрос-подтверждение относится действие, выполняемое для первой посещенной страницы. Здесь звено сбора данных возвращает подтверждение, которое можно использовать в последующих запросах. В отличие от паттерна запрос-ответ, когда может возвращаться признак успеха или неудачи, в паттерне запрос-подтверждение возвращаются данные для использования в будущих запросах. В данном случае подтверждение – это просто уникальный идентификатор, но он играет важную роль. Это подтверждение можно использовать на всех страницах, которые посетит пользователь. Обращаясь к службе расположенности, мы можем передать ей идентификатор, полученный при посещении первой страницы. Зная его, служба идентифицирует посетителя, после чего вычисляет и возвращает его текущую оценку расположенности к покупке.

Я опустил детали вычисления оценки расположенности на основе собранных данных, но все прояснится в следующих главах. Сейчас важно, что паттерн запрос-подтверждение подразумевает, что клиент просит службу выполнить некоторое действие, а в ответ получает маркер подтверждения, который можно использовать в последующих запросах. Мы сталкиваемся с этим паттерном ежедневно. Например, совершая покупку в интернет-магазине, мы часто получаем код подтверждения, по которому впоследствии можем проверить состояние заказа.

Если вы хотите узнать больше об этом паттерне, обратитесь к книге Gregor Hohpe, Bobby Woolf «Enterprise Integration Patterns» (Addison-Wesley, 2003).

2.1.3. Паттерн издатель-подписчик

Этот паттерн широко применяется в системах на основе сообщений; поток данных в нем показан на рис. 2.7.

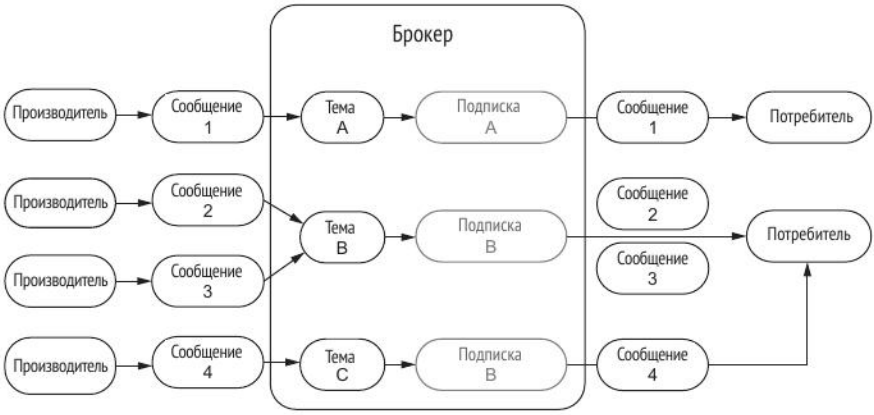


Рис. 2.7. Поток данных в паттерне издатель-подписчик

Все начинается с того, что издатель публикует сообщение, адресованное брокеру. Сообщения часто направляются в *тему*, которую можно считать логической группой сообщений. Затем сообщение рассылается всем потребителям, подписавшимся на эту тему. На этом последнем шаге есть одна тонкость, которую мы подробно обсудим в главе 3. А пока следует запомнить, что в некоторых технологиях поток данных такой, как показан на рисунке: сообщения проталкиваются потребителям. Но есть и другие технологии, в которых потребитель вытягивает сообщения от брокера. На первый взгляд, неочевидно, но иногда публикация сообщения издателем не означает, что на тему необходимо подписаться. Также не требуется, чтобы подписчик порождал какое-то сообщение.

Посмотрим на примере, как можно использовать этот протокол и как он влияет на звено сбора данных. После успешного совместного предприятия с компанией Эрика вы задумались о том, как поднять бортовую часть службы дорожной обстановки и построения маршрутов на следующий уровень. Рассмотрев несколько идей, вы остановились на том, чтобы сделать ее социальной. Помимо запроса дорожной обстановки и маршрута, она будет в реальном времени посылать службе информацию об обновлениях дорожной обстановки и сможет подписываться на отчеты о дорожной обстановке от других транспортных средств, движущихся по тому же маршруту. На рис. 2.8 показан соответствующий поток сообщений.

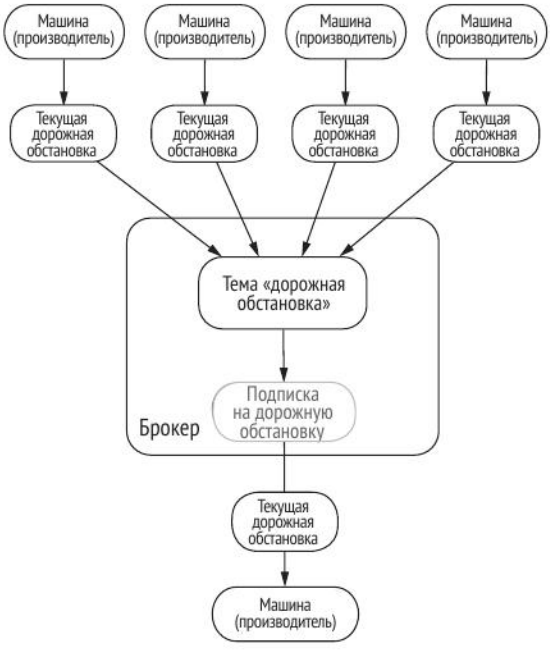


Рис. 2.8. Применение паттерна издатель-подписчик для уведомления о текущей дорожной обстановке

Для простоты мы нарисовали всего несколько машин, выступающих в роли производителей данных (они посылают сведения о текущей дорожной обстановке брокеру), и одну машину в роли потребителя. В реальной ситуации каждый производитель, наверное, также потреблял бы и анализировал все данные. Паттерн издатель-подписчик позволяет разорвать связь между отправителем информации о дорожной обстановке и ее потребителем. Если масштабировать этот простенький пример – четыре машины отправляют данные, одна потребляет – на все машины, ездящие по дорогам США, то можно представить себе, насколько важен этот разрыв. Если вас интересуют кое-какие тонкие моменты этого паттерна, то начинать имеет смысл с уже упоминавшейся книги «Enterprise Integration Patterns».

2.1.4. Паттерн одностороннего взаимодействия

Этот паттерн чаще всего встречается в ситуациях, когда система, отправляющая запрос, не нуждается в ответе. Иногда его еще называют «выстрелил и забыл». В некоторых случаях этот паттерн обладает очевидными преимуществами и, возможно, является единственным способом взаимодействия между клиентом и службой. Он похож на паттерны запрос-ответ и запрос-подтверждение в том смысле, что сообщение передается от клиента службе. А основное отличие состоит в том, что служба не отправляет никакого ответа. В других паттернах клиент знает, что запрос был получен и как-то обработан, а здесь неизвестно даже, дошел ли запрос до службы.

Возникает вопрос: при каких обстоятельствах может быть полезен паттерн, не дающий никаких гарантий доставки сообщения. Отвечаю: когда у клиента нет ресурсов или необходимости обрабатывать ответ. Взять, к примеру, данные о серверах в центре обработки данных. Мы хотели бы, чтобы сервер раз в 10 секунд отправлял данные о том, сколько памяти и процессоров используется. Но серверу нет нужды предпринимать какие-то действия или хотя бы интересоваться результатом, он просто порождает данные настолько быстро, насколько может.

Примеры этого паттерна окружают нас со всех сторон, и их число множится по мере распространения «Интернета всего» (Internet of Everything), который проникает в разные стороны нашей жизни, не исключая и спорт. Благодаря недавнему партнерству между Национальной футбольной лигой и компанией Zebra (www.zebra.com/us/en/nfl.html) игроки, принимающие участие в вечерних матчах по четвергам, прикрепляют к форме радиометку (RFID) размером с монетку в 25 центов. Каждая метка с частотой примерно 25 раз в секунду передает данные о перемещениях спортсмена, преодоленном расстоянии и скорости двадцати RFID-приемников, установленных на стадионе. Через полсекунды проанализированные данные передаются передвижным телевизионным станциям, чтобы их могли ис-

пользовать комментаторы. В данном случае радиометка, клиент, не располагает ресурсами для обработки ответа от RFID-приемника, да и не нуждается в этом.

Если в течение какой-то секунды пять из 25 отправленных порций данных потеряются и не будут получены RFID-приемником, отразится ли это на результатах анализа? Вряд ли. Это еще одна примечательная особенность данного паттерна – он пригоден (и часто применяется) в ситуациях, когда с потерей данных можно смириться в обмен на простоту, снижение потребности в ресурсах и быстродействие. Подробнее об этом паттерне можно прочитать в книге Nicolai M. Josuttis «SOA in Practice» (O'Reilly, 2007).

2.1.5. Паттерн поток

Этот способ взаимодействия сильно отличается от предыдущих. Во всех остальных паттернах клиент отправляет запрос службе, которая возвращает или не возвращает ответ. В паттерне поток все меняется местами, и служба становится клиентом. На рис. 2.9 этот паттерн сравнивается с другими.

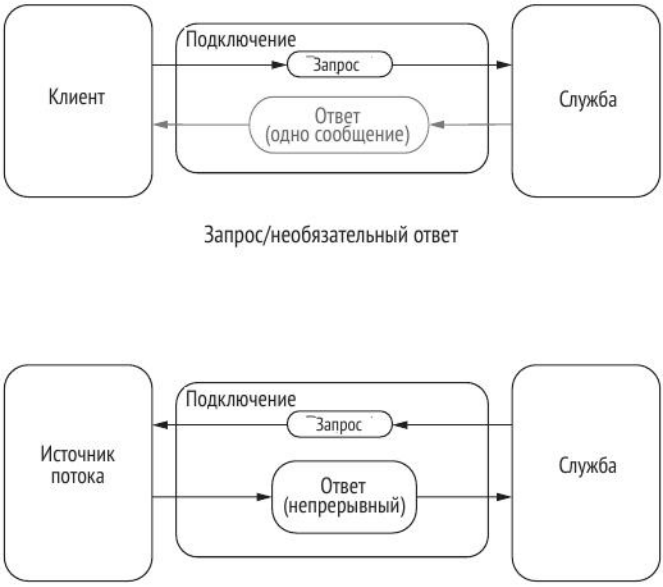


Рис. 2.9. Сравнение паттернов запрос-ответ с паттерном поток

Подчеркнем два важных отличия между предыдущими паттернами (типа запрос-ответ) и паттерном поток:

- при взаимодействии типа запрос-ответ (верхняя часть рис. 2.9) клиент передает службе данные в запросе, а служба может ответить. Этот ответ показан на рисунке серым цветом, потому что в некоторых

вариантах паттерна он необязателен. Таким образом, мы имеем один запрос и нуль или один ответ на него. Паттерн поток (нижняя часть рис. 2.9) устроен совершенно иначе: ответом на один запрос является непрерывный поток данных (или вообще никаких данных);

- в паттернах запрос-ответ клиент, внешний по отношению к потоковой системе, отправляет ей сообщение. В приведенных выше примерах это был браузер, автомобиль или телефон – клиенты, оправляющие сообщения звену сбора данных. А в паттерне поток звено сбора данных само подключается к источнику потока и вытягивает из него данные. Например, рассмотрим потоковую систему для анализа эмоциональной тональности твитов. В ней должно быть звено сбора данных, которое подключается к Твиттеру и потребляет поток твитов.

Это очень интересный и мощный паттерн: он потребляет один поток данных и порождает другой. Таким образом, можно быстро построить потоковую систему, которая потребляет общедоступные данные и создает новые потоки данных по результатам анализа. В отличие от других паттернов, где необходимо написать или найти клиента для отправки запросов службе, паттерн поток позволяет просто подключиться к источнику потока и обрабатывать получаемые данные.

Сайт Meetup.com предлагает пример потока, который можно использовать для исследования этого паттерна или в качестве входа в потоковую систему. На основе этого потока мы построим сквозную потоковую систему в главе 9. Поток состоит из данных о мероприятиях в формате JSON, генерируемых всякий раз, как кто-то размещает приглашение посетить встречу. Чтобы посмотреть, как выглядит этот поток, перейдите в браузере по адресу http://stream.meetup.com/2/rsvps. Устанавливается простое долговременное HTTP-соединение, по которому данные будут передаваться браузеру, пока вы его не закроете. Мероприятия в формате JSON выглядят, как показано в следующем листинге.

Листинг 2.1. Пример потока мероприятий в формате JSON

См. код в листингах 9.2–9.7

```
{
  "venue": {
     "venue_name": "Chicago Symphony Center",
     "lon": -87.624402,
     "lat": 41.878904,
     "venue_id": 700306
  },
  "visibility": "public",
  "response": "yes",
  "member": {
```

Информация о месте встречи

Информация об участниках

```
        "member_id": 184005505,
        "member_name": "Rifat"
    },
    "rsvp_id": 1631403261,
    "event": {
        "event_name": "Civic Orchestra Open Rehearsal w\/ Riccardo Muti -
 Brahm' s 4th Symphony",
        "event_id": "234044012",
        "time": 1474934400000
    },
    "group": {
        "group_topics": [{
            "urlkey": "symphony",
            "topic_name": "Symphony"
        }],
        "group_city": "Chicago",
        "group_country": "us",
        "group_id": 882009,
        "group_name": "Chicago Classical Music Events",
        "group_lon": -87.63,
        "group_urlname": "chicagosymphony",
        "group_state": "IL",
        "group_lat": 41.88
    }
}
```

Информация о мероприятии

Информация о группе

В листинге 2.1 приведен пример использования паттерна поток. Допустим, вы хотите объединить эти данные с данными из социальной сети, например с твитами о некоторых мероприятиях или местах проведения. Это было бы трудно сделать с помощью других паттернов. Я уверен, что при взгляде на эти данные у вас возникли и другие вопросы, помимо объединения с другими потоками. Например, можно посчитать самые популярные мероприятия или самые популярные группы в каждом городе. Но не будем забегать вперед – в главе 4 мы научимся отвечать на эти и другие вопросы. Пока же будет достаточно, если вы оцените этот паттерн и возможности, которые он открывает[1].

2.2. Масштабирование паттернов взаимодействия

Обсудив все паттерны взаимодействия, посмотрим, как можно масштабировать звено сбора данных, и поговорим о вещах, которые надо иметь в виду при его реализации. Обсуждение будем вести отдельно для двух рассмотренных выше категорий паттернов.

[1] Если вы хотите подробнее узнать об этом наборе данных, почитайте документацию по адресу www.meetup.com/meetup_api/docs/stream/2/rsvps/#polling.

2.2.1. Паттерны запрос-ответ

Обсуждать масштабирование этого общего паттерна мы будем на примере службы дорожной обстановки и построения маршрутов, обслуживающей все транспортные средства на дороге. Чтобы составить представление о масштабе задачи для США, обратимся к отчету о национальной транспортной статистике 2012 года (последний год, за который имеются полные данные, опубликованные Бюро транспортной статистики, www.rita.dot.gov). Согласно этому отчету, в 2012 году в США было зарегистрировано около 253 миллионов транспортных средств, а суммарный их пробег составил 2,966 триллиона миль. Это означает, что в любой точке на протяжении этих без малого трех триллионов миль мы можем получить от любого из 253 миллионов автомобилей запрос о текущей дорожной обстановке и вариантах маршрутов. В любой момент количество подлежащих обработке запросов может исчисляться тысячами, а то и миллионами.

Если помните, в главе 1 я говорил, что нашей конечной целью в каждом звене потоковой системы является обеспечение горизонтальной масштабируемости. В этом примере горизонтальное масштабирование будет хорошо работать по двум причинам. Во-первых, в паттерне запрос-ответ нет информации о состоянии клиента, отправившего запрос, а значит, любой клиент может подключиться и отправить запрос любому работающему экземпляру службы. Во-вторых, – и это тоже результат отсутствия информации о состоянии – мы легко можем добавить новые экземпляры службы, не изменяя тех, что уже работают. Масштабирование служб, не имеющих состояния, настолько популярно, что многие облачные провайдеры, например Amazon, предоставляют функцию автомасштабирования, которая автоматически увеличивает или уменьшает число работающих экземпляров в зависимости от спроса. Помимо горизонтальной масштабируемости, мы хотели бы также, чтобы наша служба не хранила никакого состояния, поскольку это позволит любому транспортному средству отправлять запрос любому экземпляру службы в любой момент времени. Такое свойство часто встречается в системах, где используется паттерн запрос-ответ. Принимая во внимание горизонтальную масштабируемость и отсутствие состояния, мы приходим к рис. 2.10, на котором эти два аспекта показаны вместе.

Балансировщик нагрузки используется для того, чтобы маршрутизировать запросы от автомобилей к одному из экземпляров службы. Экземпляры запускаются и останавливаются в зависимости от спроса, поэтому их состав динамически изменяется. Теперь мы довольно хорошо представляем, как будем масштабировать службу и как должен быть устроен протокол общения с клиентами.

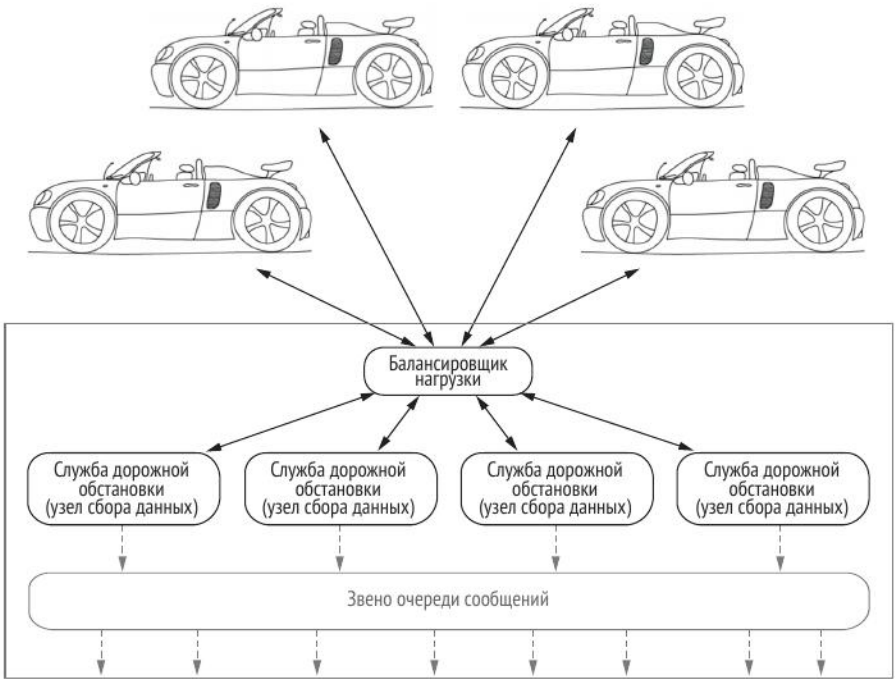


Рис. 2.10. Транспортные средства и служба дорожной обстановки с балансировщиком нагрузки

2.2.2. Масштабирование паттерна поток

У потока с сайта meetup.com, с помощью которого мы иллюстрировали паттерн Поток в разделе 2.1.5, довольно низкая скорость (менее 10 мероприятий в секунду). Понятно, что не на этом примере следует обсуждать масштабирование звена сбора данных. Допустим лучше, что Google в реальном времени предоставляет поток всех поисковых запросов; согласно сайту internetlivestats.com (www.internetlivestats.com/one-second/#googleband) это было бы примерно 46 000 событий в секунду. Напомним, что при построении каждого звена потоковой системы мы стремимся обеспечить горизонтальную масштабируемость. Как мы видели на примере сайта meetup.com, во многих потоковых протоколах существует прямое постоянное соединение между клиентом (нашим звеном сбора данных) и сервером (службой, у которой мы запрашиваем данные).

На рис. 2.11 показано, что три из четырех узлов бездействуют, поскольку между потоком поисковых запросов и узлом, обрабатывающим поток, существует прямое соединение. Для масштабирования звена сбора данных есть два варианта: масштабировать узел, потребляющий поток, или включить в звено сбора данных слой буферизации. Эти решения не являются взаимно исключающими, и в зависимости от объема и скорости потока может возникнуть необходимость в обоих. Масштабирование узла, по-

требляющего поток, имеет ограничения: в какой-то момент возможности оборудования, на котором работает узел, будут исчерпаны, и дальнейшее масштабирование станет невозможным. На рис. 2.12 показано, как выглядит звено сбора данных со слоем буферизации.

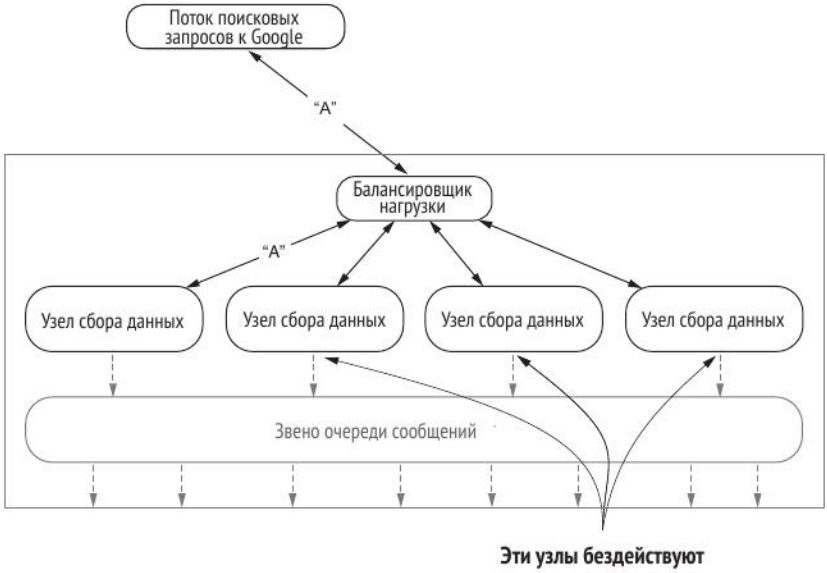


Рис. 2.11. Поток поисковых запросов с прямым подключением к узлу сбора данных

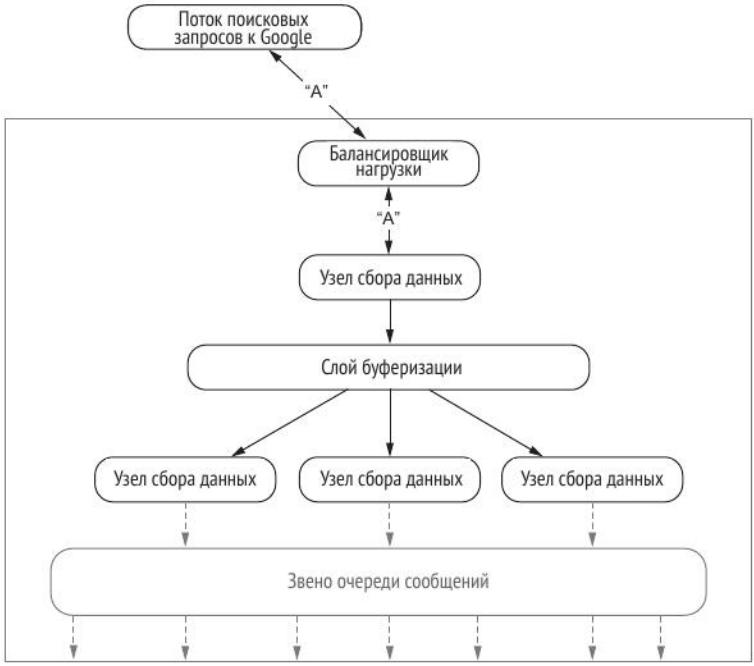


Рис. 2.12. Звено сбора данных со слоем буферизации

Чтобы в середину можно было поместить слой буферизации, необходимо, чтобы сообщения, потребляемые из потока, не подвергались никакой логической обработке, а сразу же передавались слою буферизации. Для получения сообщений из слоя буферизации и содержательной обработки может существовать дополнительный набор узлов сбора данных, который уже можно горизонтально масштабировать.

2.3. Отказоустойчивость

Какой бы паттерн взаимодействия не использовать, одно остается неизменным: в какой-то момент один или несколько узлов сбора данных могут выйти из строя. Отказ может быть вызван ошибкой в нашей или сторонней программе либо сбоем оборудования. Вне зависимости от причины мы должны компенсировать отказы и обеспечить надежность звена сбора данных. Спрашивается зачем нам об этом думать, если мы уже произвели горизонтальное масштабирование и тем самым увеличили степень резервирования. Хороший вопрос. Но ответ простой: сообщение, которое звено сбора данных получает от клиента, может оказаться невоспроизводимым. У звена сбора данных не всегда есть возможность запросить у клиента повторную передачу данных, да и клиент не всегда может это сделать, даже если бы такая просьба поступила.

Иногда с потерей данных можно смириться, но во многих случаях это не так – все зависит от конкретной задачи. В этом разделе мы рассмотрим методы обеспечения отказоустойчивости, с помощью которых можно предотвратить потерю данных и повысить надежность звена сбора. Конечная цель – сделать так, чтобы в случае выхода из строя узла сбора данных (а рано или поздно это обязательно произойдет) мы не потеряли данных и могли восстановиться, как будто никакого отказа не было. Чтобы понять, что именно нужно защищать, взгляните на рис. 2.13, где показана простейшая ситуация и отмечены места, в которых возможна потеря данных при отказе узла.

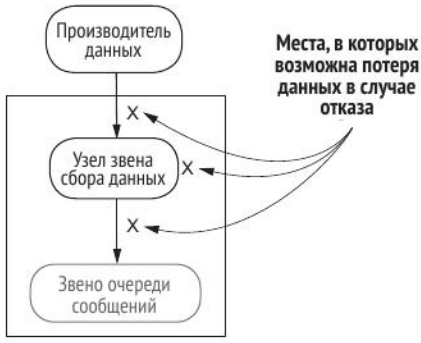


Рис. 2.13. Потенциальные точки потери данных в случае отказа узла

Два основных подхода к реализации отказоустойчивости, *контрольная точка* и *протоколирование*, призваны защитить от потери данных и обеспечить быстрое восстановление отказавшего узла. Как мы вскоре увидим, по своим характеристикам эти подходы различаются.

Сначала рассмотрим контрольные точки. В литературе описано много протоколов на основе контрольных точек, но все они обладают двумя характеристиками.

- *Глобальный снимок* – требуется, чтобы где-то во внешней памяти регулярно сохранялся мгновенный снимок глобального состояния всей системы, а не только уровня сбора данных.
- *Возможность потери данных* – гарантируется только восстановление системы в последнем сохраненном глобальном состоянии; все сообщения, которые были сгенерированы или обработаны позже, теряются.

Что понимается под глобальным снимком? Возможность получить полное состояние данных и вычислений во всех звеньях – от сбора данных до доступа к данным – и сохранить его в надежной постоянной памяти. Именно так следует понимать слова «глобальное состояние системы». Затем это состояние используется в процессе восстановления, чтобы перевести систему в последнее известное состояние. Возможность потери данных существует, если мы не можем сохранять глобальное состояние при каждом изменении данных в системе.

Пример, с которым вы, конечно, знакомы, – автосохранение в таких популярных программах редактирования документов, как Microsoft Word или Google Docs. В процессе редактирования создается снимок документа, и в случае аварийного завершения приложения мы можем восстановить последнюю контрольную точку. Скорее всего, вы, как и многие пользователи, видели, как работает механизм контрольных точек, и знаете, что в случае внезапного отказа последние изменения не сохраняются.

Рассматривая возможность применения протокола контрольных точек для реализации отказоустойчивой потоковой системы, имейте в виду две вещи: вышеупомянутую особенность и тот факт, что потоковая система состоит из многих слоев, в которых применяются разнообразные технологии. Из-за перемещения данных между слоями очень трудно создать непротиворечивый глобальный снимок на определенный момент времени, поэтому контрольные точки – не самый удачный выбор для потоковой системы. Но он вполне оправдан, если вы разрабатываете следующую версию файловой системы HDFS или, быть может, новую базу данных NoSQL. Поскольку контрольные точки плохо совместимы с потоковыми системами, мы не станем тратить время на рассмотрение подобных протоколов. Но сами по себе они очень интересны, и если вы хотите получше узнать о них, то для начала рекомендую замечательную статью Elnozahy, En Mootaz,

et al. «A Survey of Rollback-Recovery Protocols in Message-Passing Systems» (ACM Computing Surveys 34.3 (2002): 375–408)[2].

Обратимся к протоколированию, тут есть, из чего выбрать. У всех подобных протоколов общая цель: отказавшись от высоких накладных расходов и сложности контрольных точек, обеспечить возможность восстановления вплоть до последнего сообщения, полученного перед аварией. В частности, устраняется необходимость в глобальном снимке, а значит, в управлении глобальным состоянием. Вообще, в основе всех методов протоколирования лежит одна общая идея: *если любое сообщение можно воспроизвести, то система может достичь глобально непротиворечивого состояния без создания глобального снимка*.

Это означает, что каждое звено системы независимо записывает все полученные им сообщения и воспроизводит их после аварии. Реализация протоколирования освобождает нас от необходимости поддерживать глобальное состояние и позволяет сосредоточиться на том, как обеспечить отказоустойчивость звена сбора данных. Мы обсудим два классических метода: *протоколирование сообщений на стороне получателя* (receiver-based message logging, RBML) и *протоколирование сообщений на стороне отправителя* (sender-based message logging, SBML), а также новый метод – *гибридное протоколирование сообщений* (hybrid message logging, HML). Попутно поговорим о том, как и почему эти методы можно использовать в звене сбора данных.

Но сначала взгляните на рис. 2.14, где показано, как эти методы ложатся в общую картину и какие данные мы пытаемся защитить.

На рис. 2.14 мы видим один узел звена сбора данных, который получает сообщение, обрабатывает его и передает следующему звену. Как явствует из названий, задача протоколирования сообщений на стороне получателя – защитить данные, полученные узлом, а задача протоколирования сообщений на стороне отправителя – защитить данные, отправляемые следующему звену. Представьте себе, что какая-то обработка производится между двумя слоями протоколирования: один сохраняет данные до изменения, а другой – перед отправкой следующему звену. В некоторых случаях это влечет высокие накладные расходы и дублирование, поэтому HML направлен на достижение баланса между RBML и SBML. Определив контекст, перейдем к обсуждению RBML.

2.3.1. Протоколирование сообщений на стороне получателя

Метод RBML подразумевает синхронную запись каждого полученного сообщения в надежное хранилище еще до каких-либо действий с ним. Таким образом, гарантируется, что если программа «грохнется» во время обработки сообщения, то мы сможем воспроизвести его после восстанов-

[2] Статью можно скачать по адресу http://mng.bz/vUz2.

ления. На рис. 2.15 показано, как изменяется узел сбора данных после добавления RBML.

Рис. 2.14. Общая схема протоколирования сообщений на стороне получателя и на стороне отправителя

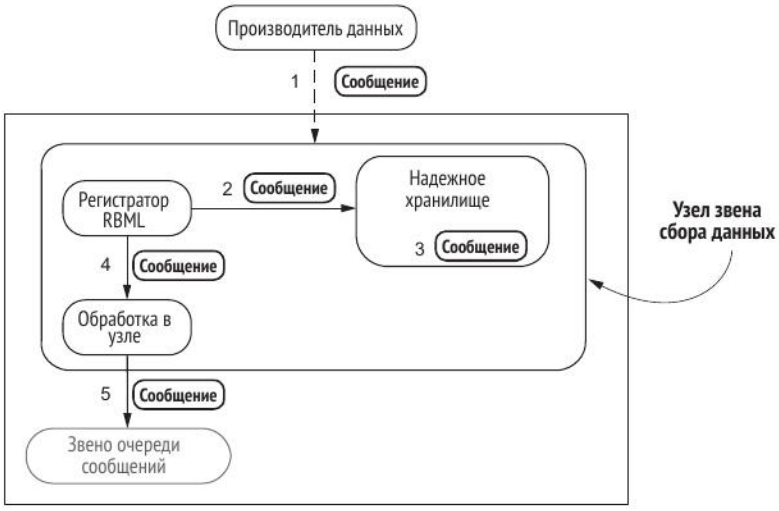


Рис. 2.15. Реализация RBML для простого узла сбора данных – безаварийный путь

На рис. 2.15 сообщение проследует по цепочке шагов от 1 до 5 – это безаварийный путь, на котором нет отказов. Восстановление мы рассмотрим чуть позже, а пока остановимся на этом пути.

1. Производитель данных (любой клиент) отправляет сообщение.
2. Новая часть программы, написанная нами для узла сбора данных, – регистратор RBML – получает сообщение от производителя данных и отправляет его в хранилище.

3. Сообщение записывается в надежное хранилище.
4. Сообщение подвергается обработке в узле; это может быть агрегирование, фильтрация или маршрутизация на основе бизнес-правил. Важно, что сообщение сохраняется сразу после получения, еще до того, как мы с ним что-то делаем.
5. Затем сообщение отправляется звену очереди сообщений – следующему звену потоковой системы.

Подчеркнем, что в зависимости от типа надежного хранилища шаги 2 и 3 могут привести к снижению пропускной способности узла сбора данных, иногда это рассматривается как один из недостатков техники протоколирования. Гибридное протоколирование, которое рассматривается в разделе 2.3.3, частично решает эти проблемы. Но пока не будем усложнять – в конечном итоге побеждают простота и возможность восстановления, обеспечиваемая RBML в узле сбора данных.

Разобравшись, что происходит с данными при нормальной работе, обратимся к рис. 2.16, где показано, как выглядит поток данных в случае восстановления после сбоя.

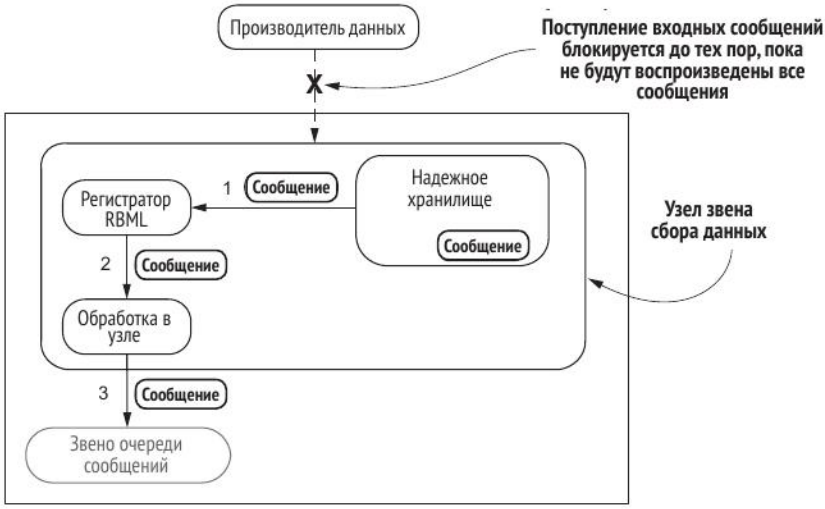


Рис. 2.16. Поток данных для RBML в случае восстановления после сбоя

На рис. 2.16 нужно обратить внимание на несколько моментов. Во-первых, после аварии поступление входных сообщений на этот узел прекращается. Поскольку узлов сбора данных несколько и все они находятся за балансировщиком нагрузки, этот узел выводится из ротации. Во-вторых, регистратор RBML читает еще не обработанные сообщения из надежного хранилища и передает их обработчику, как будто ничего не случилось. В-третьих, после того как все сохраненные ранее сообщения обработаны, узел считается восстановленным и возвращается в ротацию; возобновляется порядок следования данных, показанный на рис. 2.15.

2.3.2. Протоколирование сообщений на стороне отправителя

Метод SBML подразумевает запись сообщения в надежное хранилище перед его отправкой. Если в случае RBML сохраняются сообщения, поступающие узлу сбора данных через переднюю дверь, и тем самым мы защищаемся от собственных отказов, то SBML означает протоколирование исходящих из узла сообщений, т. е. защищает от отказа в следующем звене или потери связности сети. На рис. 2.17 изображен поток данных в случае SBML.

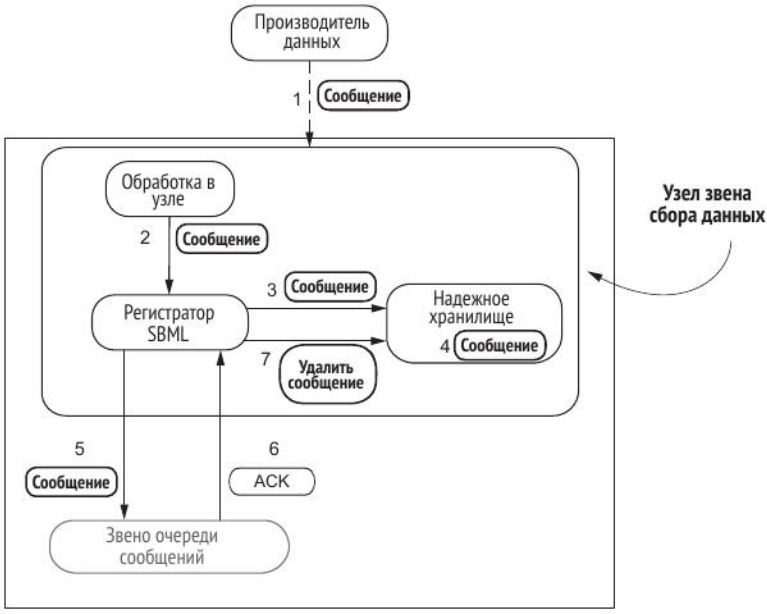


Рис. 2.17. Поток данных при нормальном выполнении в случае SBML

После того как мы рассмотрели RBML, поток данных в методе SBML, наверное, не вызывает вопросов до шага 5. Отличие в том, что в случае RBML сообщение записывается сразу после получения, еще до того как мы успели что-то с ним сделать, а в случае SBML – до отправки его следующему звену. Регистратор RBML сохраняет исходные поступающие данные, а регистратор SBML – данные, уже обработанные узлом (и, возможно, как-то пополненные), но еще не отправленные дальше.

Помимо этого нюанса есть еще одна мелочь, о которой не следует забывать. Как в процессе восстановления мы можем узнать, было ли обработано сообщение, которое мы воспроизводим, следующим звеном? Есть несколько способов. Один из них, показанный на рис. 2.17, заключается в том, чтобы звено очереди сообщений возвращало подтверждение того, что сообщение получено. Получив подтверждение, мы можем либо пометить

сообщение в надежном хранилище как уже воспроизведенное, либо удалить его, поскольку в процессе восстановления воспроизводить его не надо. Если выбранная технология реализации звена очереди сообщений не поддерживает возврата подтверждений, то мы оказываемся в ситуации, когда сам факт отсутствия ошибки при отправке сообщения звену очереди сообщений означает, что сообщение нужно удалить из надежного хранилища.

Поток данных в процессе восстановления, показанный на рис. 2.18, несколько сложнее, чем случае RBML.

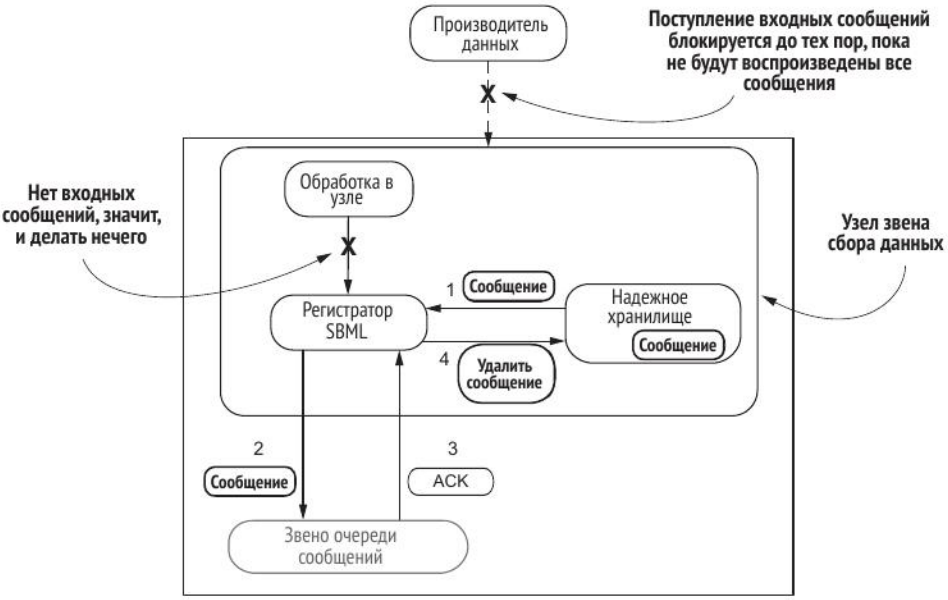


Рис. 2.18. Поток данных для SBML в случае восстановления после сбоя

Полагаю, вы согласитесь, что хотя восстановление в случае SBML немного сложнее, чем в случае RBML, никаких неожиданностей здесь нет.

2.3.3. Гибридное протоколирование сообщений

См. код в листингах 9.1–9.8

Если бы мы на этом остановились, то остались бы с двумя решениями для предотвращения потери данных и повышения надежности: RBML – для протоколирования входных сообщений и SBML – для выходных сообщений. Но, как уже отмечалось, запись в надежное хранилище может негативно повлиять на производительность узла сбора данных. Реализация одновременно RBML и SBML означает, что писать в надежное хранилище придется, по меньшей мере, дважды в ходе нормального выполнения. Кто-то скажет, что мы больше протоколируем, чем обрабатываем данные, и рис. 2.19 показывает, что это не так уж далеко от истины.

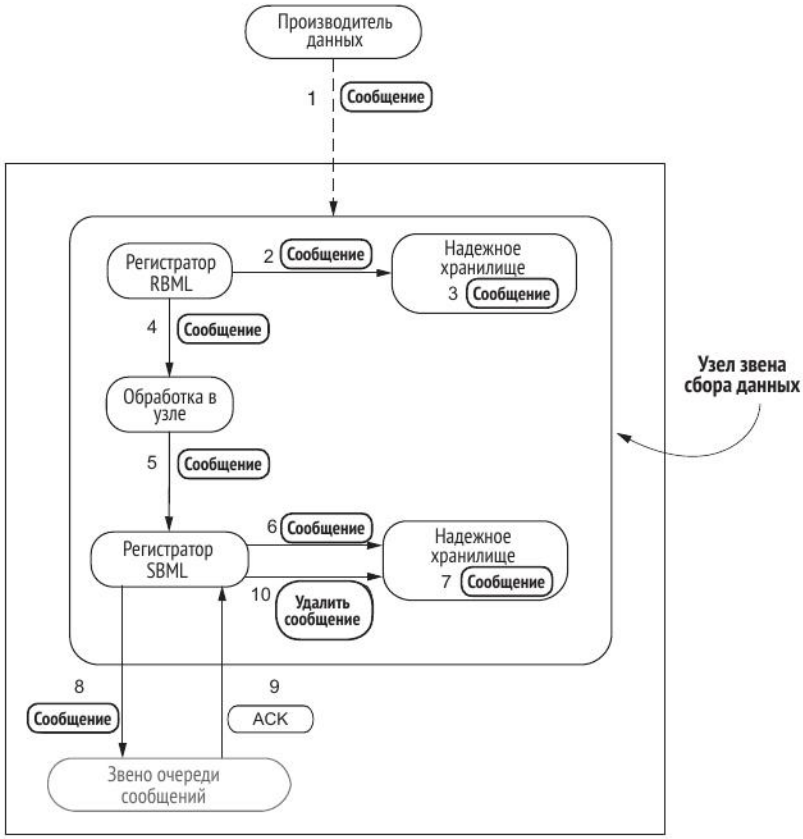


Рис. 2.19. Реализация RBML и SBML в узле сбора данных

Чтобы справиться с этой проблемой, было предложено гибридное протоколирование сообщения, которое снижает вредные последствия протоколирования в процессе нормального выполнения. С этой целью используется всё лучшее, что есть в RBML и SBML, ценой незначительного усложнения. При проектировании HML ставилась задача обеспечить такой же уровень защиты данных и способности к восстановлению, как в RBML, SBML и других методах протоколирования. Существует несколько способов реализации HML, самый употребительный показан на рис. 2.20.

При сравнении рису. 2.19 и 2.20 становится ясно, что подход на основе HML немного проще, чему способствует несколько факторов. Первый, не вызывающий удивления, состоит в том, что два экземпляра надежного хранилища объединены. Это мелкое изменение, но благодаря ему уменьшается общее число компонентов. Что касается второго изменения – асинхронной записи в хранилище, – то тут различие более тонкое. Можно предположить, что оно сильнее сказывается на сложности и производительности реализации. Сложность связана с необходимостью корректно обрабатывать ошибки, а повышение производительности – с наличием

нескольких ядер, позволяющих одновременно выполнять более одной задачи.

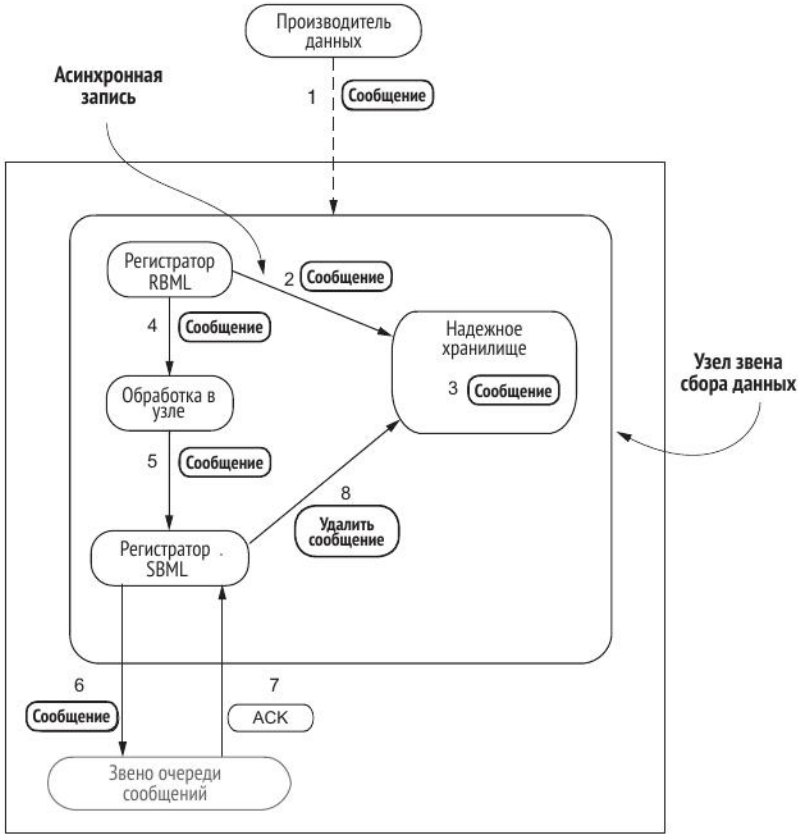


Рис. 2.20. Поток данных в случае HML

Все остальное в этом потоке данных вам уже знакомо. Если дополнительная сложность вас не пугает и имеется выбор между реализацией HML или стандартных методов RBML и SBML, то лучше остановиться на HML, поскольку при этом влияние протоколирования на производительность не столь велико, а отказоустойчивость и защищенность, свойственные RBML и SBML, сохраняются. Если вы хотите узнать больше о HML, рекомендую начать со статьи Хьюго Мейера с соавторами[3]. В ней показано, как удалось добиться снижения накладных расходов на 43% по сравнению с RBML.

2.4. Опустимся на грешную землю

Расскажу забавную историю, чтобы соотнести все разговоры о масштабировании и отказоустойчивости с реальностью. Как-то раз я работал над

[3] Meyer H., Rexachs D., and Luque E., «Hybrid Message Logging. Combining Advantages of Sender-based and Receiver-based Approaches» (Procedia Computer Science 29 (2014): 2380–90), http://mng.bz/5ZUc.

потоковой системой, которая выводила данные на хитроумные информационные панели для маркетологов. В ней было все, что душе угодно: масштабирование, отказоустойчивость, мониторинг, уведомления – полный набор. И при всем при том мы не могли потерять никаких данных, поскольку заказчик не принял бы решения с неполными данными. После того как система была запущена, я из любопытства поинтересовался, как справляются наши информационные панели, потребляющие данные по протоколу WebSockets. Оказалось, что многие заказчики поспевают только за 60% посылаемого им потока, а остальные 40% отбрасываются, потому что люди не могут читать с такой скоростью. Когда я рассказал об этом коллегам, они были потрясены и даже не сразу поверили, потому что и заказчикам, и руководителям очень нравилось то, что они видели.

Итак, созданные нами информационные панели отражали картину бизнеса заказчиков, не искаженную отсутствием данных. Но с моей точки зрения, разница такая же, как между высококлассным и средненьким телевидением высокой четкости: качество, наверное, чуть получше, но картинка-то не меняется. Я не хочу сказать, что о масштабируемости и отказоустойчивости вообще не нужно думать, но всегда стоит трезво оценить положение, а затем сопоставить требование «функция *xyz* должна быть обязательно» с реальностью.

2.5. Резюме

В этой главе мы рассмотрели различные аспекты сбора данных в потоковой системе: от паттернов взаимодействия до методов масштабирования и обеспечения отказоустойчивости. По ходу дела мы:

- узнали о звене сбора данных;
- познакомились с различными паттернами сбора данных;
- имели возможность посмотреть на реальную систему;
- узнали, что понимается под масштабируемостью звена сбора данных;
- узнали о распространенных методах обеспечения отказоустойчивости.

Глава 3

Транспортировка данных из звена сбора данных: расчленение конвейера данных

Краткое содержание главы:

- зачем нужно звено очереди сообщений;
- что такое долговечность сообщения;
- поддержка периодически отключающихся потребителей;
- семантика доставки сообщений;
- выбор подходящей технологии.

До сих пор я рассказывал о роли обработки входных данных, а не о выводе из звена сбора данных. В этой главе мы займемся транспортировкой данных от звена сбора до остальных частей потокового конвейера. Хотя местами упоминаются звенья сбора данных и анализа, на самом деле обсуждаются только вывод данных из первого и доставка второму с помощью звена очереди сообщений. На рис. 3.1 показана наша потоковая архитектура и выделена та ее часть, которая нас будет интересовать.

Прочитав эту главу, вы будете хорошо понимать, зачем нужно звено очереди сообщений, каковы основные функции продуктов, обычно применяемых в этом звене, и как решить, какие функции важны конкретно для вашей системы.

3.1. Зачем нужно звено очереди сообщений

Если вам не доводилось раньше строить конвейеры данных и особенно потоковые системы, то, взглянув на рис. 3.1, вы, наверное, подумали: «Ну хорошо, я вижу, что выход звена сбора данных подается на вход звена оче-

реди сообщений, а затем данные волшебным образом перетекают в звено анализа. Ну и что? Зачем вообще нужно это звено очереди сообщений?» Это правильные вопросы. Но давайте на минутку представим, что этого звена нет, на рис. 3.2 изображена архитектура без него.

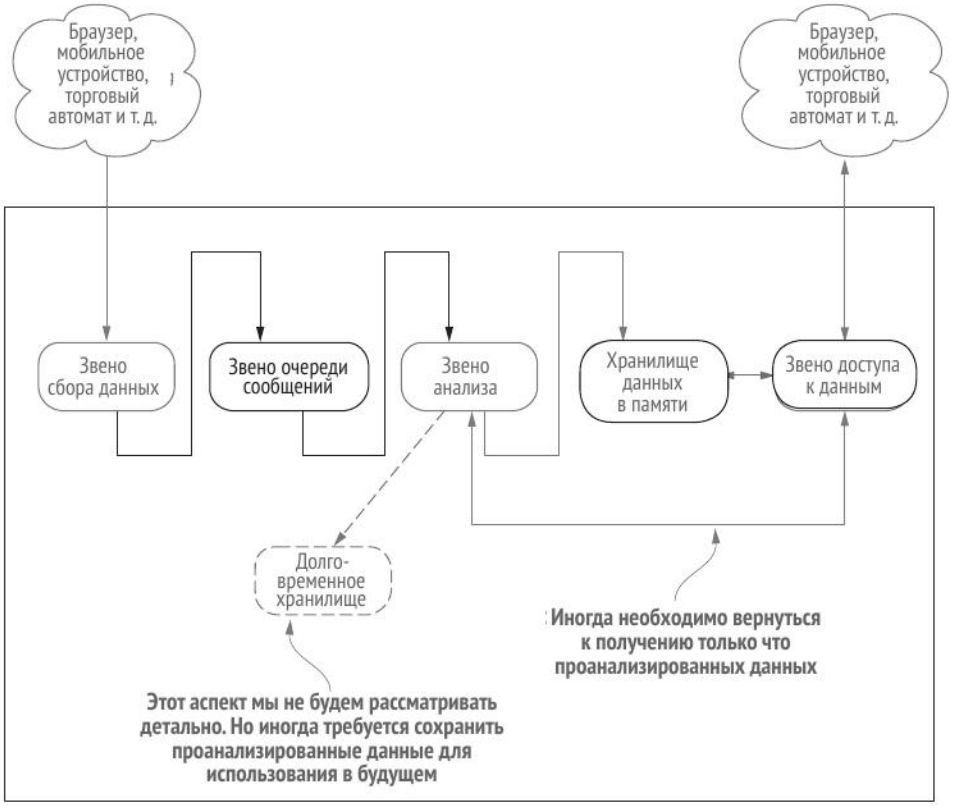


Рис. 3.1. Звено очереди сообщений, а также его вход и выход

На первый взгляд, стало проще, и работать должно правильно, но не поддавайтесь искушению взять эту якобы упрощенную архитектуру на вооружение. Конечно, иногда можно собрать всю потоковую систему на одной машине, так что каждое звено будет вызывать следующее непосредственно, но обычно система размещается на нескольких машинах. При проектировании любой программной системы следует стремиться к разрыву связей между компонентами. И потоковая система – не исключение: мы хотим разорвать связи между компонентами одного звена, а также – и это даже важнее – связи между самими звеньями. Сердцем потоковой системы, как и любой распределенной системы, является взаимодействие между машинами, на которых эта система развернута. Познакомившись с литературой на тему *межпроцессного взаимодействия*[1], вы

[1] С обзором межпроцессного взаимодействия можно познакомиться в статье https://en.wikipedia.org/wiki/Межпроцессное_взаимодействие.

встретите многочисленные модели; в этой главе нас будет интересовать *модель очереди сообщений*. Следуя предписаниям этой модели, мы сможем разорвать связь между звеном сбора данных и звеном анализа. А это даст возможность работать на более высоком уровне абстракции – передавать сообщения, а не выполнять явные вызовы следующего звена. Это свойство желательно в любой системе, а не только в распределенной потоковой. В этой и в последующих главах мы увидим, что разрыв связей между звеньями обладает рядом ценных преимуществ.

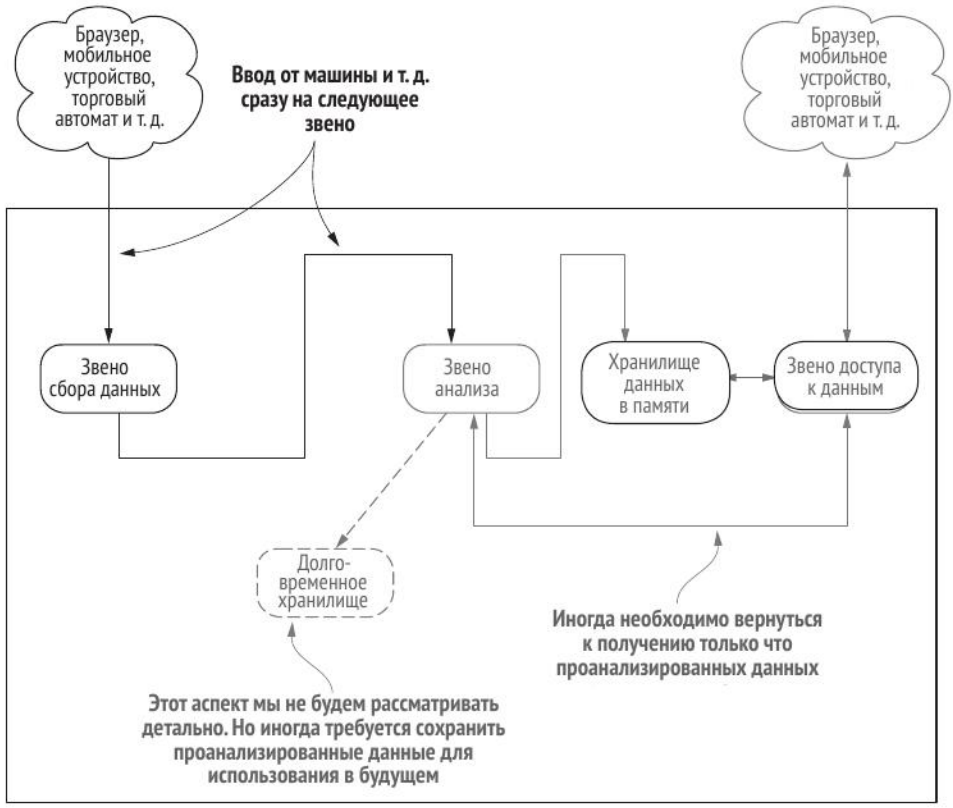


Рис. 3.2. От звена сбора данных непосредственно к звену анализа

3.2. Основные концепции

Рассмотрим характеристики продуктов, реализующих очереди сообщений, особенно важные для успеха нашей потоковой системы. Но сначала покончим с одной формальностью – определением того, что мы понимаем под *очередями сообщений*. Я использую этот термин в широком смысле, относя его ко всему спектру служб обмена сообщениями: от традиционных (RabbitMQ, ActiveMQ, HornetQ и т. д.) до более современных (NSQ, ZeroMQ, Apache Kafka). Проект Apache Kafka стал чем-то большим, чем

система сообщений, и я хочу привлечь ваше внимание к тому, что он позволяет публиковать потоки сообщений и подписываться на них. По своей функциональности это напоминает очередь сообщений или корпоративную систему обмена сообщениями.

В этом разделе мы обсудим место этого звена в общей потоковой архитектуре и поговорим об основных функциях, которые нужно рассматривать при выборе программной реализации очередей сообщений, – но только тех, на которые следует обращать внимание при проектировании потоковой системы. Вооруженные этими сведениями, вы сможете объективно рассуждать о решаемой задаче в контексте того, что важно для бизнеса, и подобрать наиболее подходящий для решения инструмент.

Но сначала убедимся, что мы одинаково понимаем, из чего состоит продукт, реализующий очереди сообщений, и как его компоненты ложатся на нашу потоковую архитектуру.

3.2.1. Производитель, брокер и потребитель

В мире очередей сообщений есть три главных компонента: *производитель*, *брокер* и *потребитель*. Каждый из них играет важную роль в функционировании и проектировании продукта. На рис. 3.3 показано, как они связаны между собой в простейшем случае.

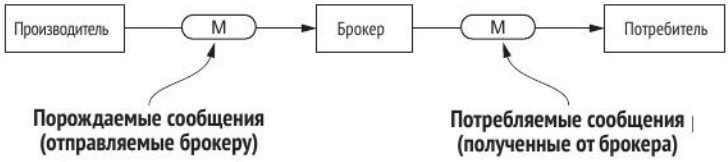


Рис. 3.3. Три главные части системы очередей сообщений

Как видим, задачи производителя и потребителя точно отражены в их названиях: производитель порождает сообщения, а потребитель их потребляет. Обратите внимание, что на рис. 3.3 используется термин *брокер*, а не *очередь сообщений*. С чего бы такое изменение? Ну, вообще-то это не столько изменение, сколько абстракция, а ввели мы ее потому, что брокер может управлять несколькими очередями. На рис. 3.4 показано, что очередь сообщений никуда не делась, но просто абстрагирована брокером.

Теперь поток данных начинает обретать смысл. На рис. 3.4 выполняются следующие действия (в порядке слева направо):

- производитель отправляет сообщение брокеру;
- брокер помещает сообщение в очередь;
- потребитель запрашивает сообщение у брокера.

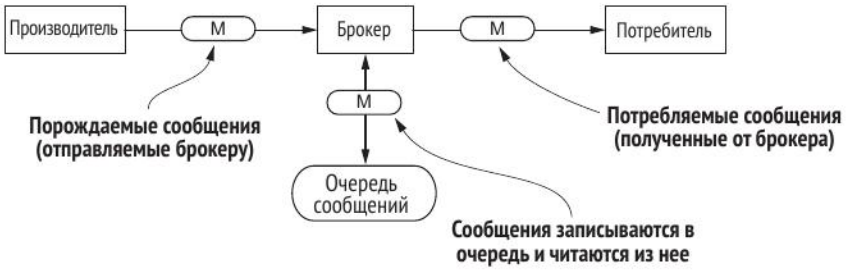


Рис. 3.4. Брокер и стоящая за ним очередь сообщений

На рис. 3.5 эти компоненты показаны на фоне общей потоковой архитектуры.

Согласитесь, что выглядит все это довольно просто и понятно, но, как говорится, дьявол кроется в деталях. Именно эти детали – тонкие взаимодействия между производителем, брокером и потребителем, а также различные виды поведения самого брокера – и будут нас интересовать.

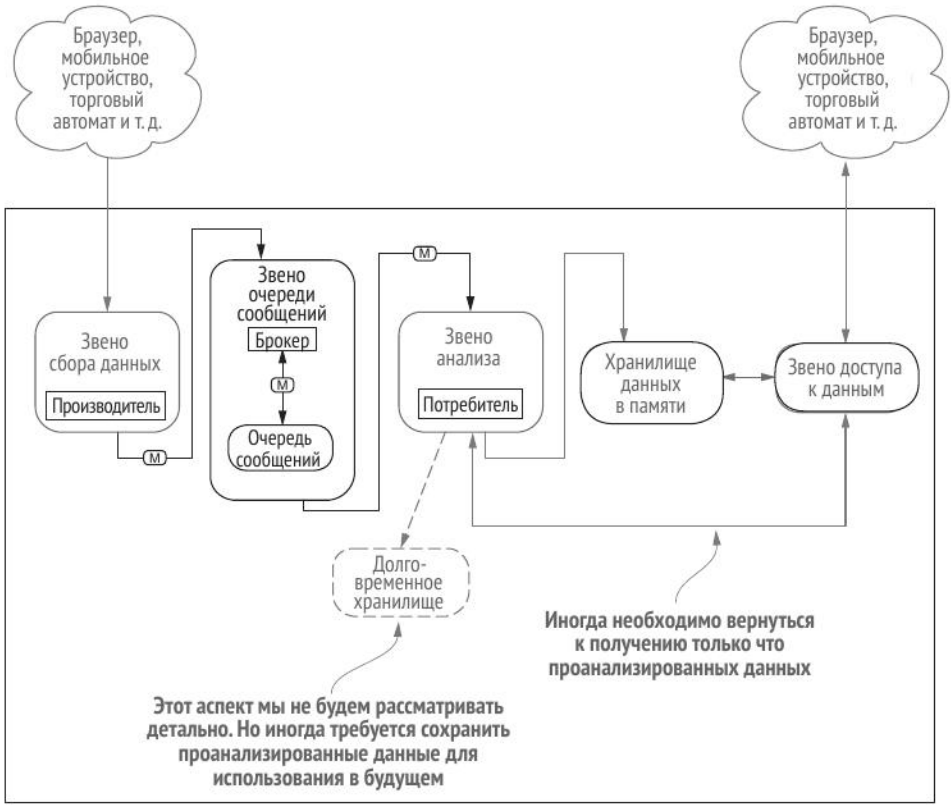


Рис. 3.5. Потоковая архитектура с развернутыми компонентами звена очереди сообщений

3.2.2. Изоляция производителей от потребителей

Как уже отмечалось, одна из наших целей – разорвать связь между различными звеньями. Звено очереди сообщений делает это для звеньев сбора данных и анализа. Рассмотрим несколько причин желанности этого свойства и те преимущества, которые оно несет.

В зависимости от характера задачи и дизайна потоковой системы может оказаться так, что производитель (звено сбора данных) порождает сообщения быстрее, чем потребитель (звено анализа) может их потребить. Часто это связано с тем, что в звене анализа производится больше вычислений, и потому оно не может обрабатывать данные с необходимой скоростью. И как же в таком случае предотвратить захлебывание звена анализа данными, порождаемыми звеном сбора?

Мне это напоминает старый мультик о шланге, один конец которого заткнут, а в другой поступает вода из крана. Он раздувается, раздувается и в конце концов лопается, не выдержав давления. На рис. 3.6 показана аналогичная картина, только роль воды играют данные, текущие от звена сбора к звену анализа.

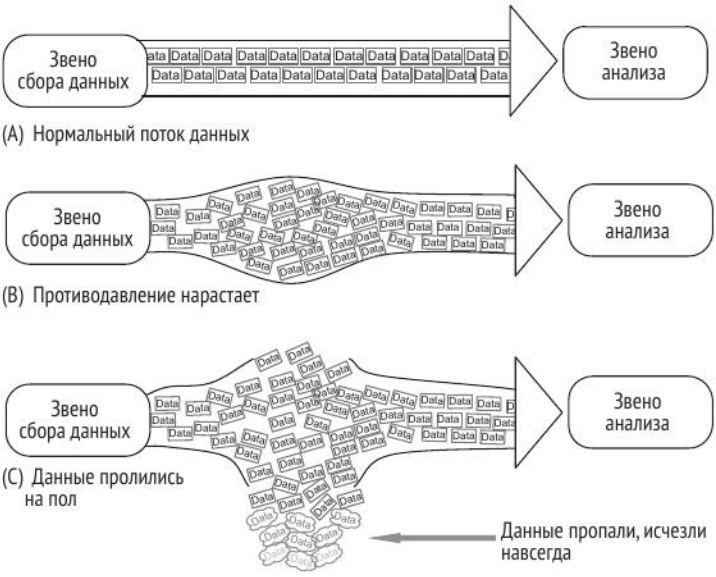


Рис. 3.6. Три стадии в жизни данных, текущих без очереди сообщений. Шаг C нам явно не нужен

Рассмотрим рис. 3.6 поэтапно.

- *Шаг A* – все выглядит хорошо, именно так, как мы хотели бы видеть.
- *Шаг B* – что-то не в порядке, противодавление нарастает.
- *Шаг C* – «шланг», по которому текут данные, не выдержал давления и лопнул, данные пролились на пол и навсегда исчезли.

Ой! Это никуда не годится – мы теряем данные, а в некоторых практических задачах это считается катастрофой. На первый взгляд, это кажется проблемой потребителя, и стоить добавить дополнительных потребителей или сделать их быстрее, чтобы они справлялись с навязанным темпом, как жизнь наладится. Но на самом деле потребители тут ни при чем; во многих случаях считается вполне допустимым, когда потребители читают медленно или время от времени вообще выходят из сети. Например, в пакетной задаче потребитель может читать все накопившиеся сообщения один раз в час, а потом отключаться.

Имейте в виду, что не все системы очередей сообщений поддерживают управление производителем, оставляя на усмотрение разработчика приложения контроль над темпом порождения сообщения со стороны звена сбора данных. Если вы пренебрежете этим, то брокер и потребитель могут захлебнуться. Поддержка медленных и периодически отключающихся потребителей имеется в тех продуктах, где реализованы долговечные сообщения.

3.2.3. Долговечные сообщения

Какое отношение долговечные сообщения и отключающиеся потребители имеют к книге, посвященной системам потоковой обработки данных? Отличный вопрос! Давайте посмотрим, что еще мы получаем в обмен на использование очередей сообщений, поддерживающих отключающихся потребителей. Представьте, что ЦОДы, используемые вашим предприятием, территориально разнесены: один находится в Амстердаме, а другой в Сан-Диего в Калифорнии, как показано на рис. 3.7.

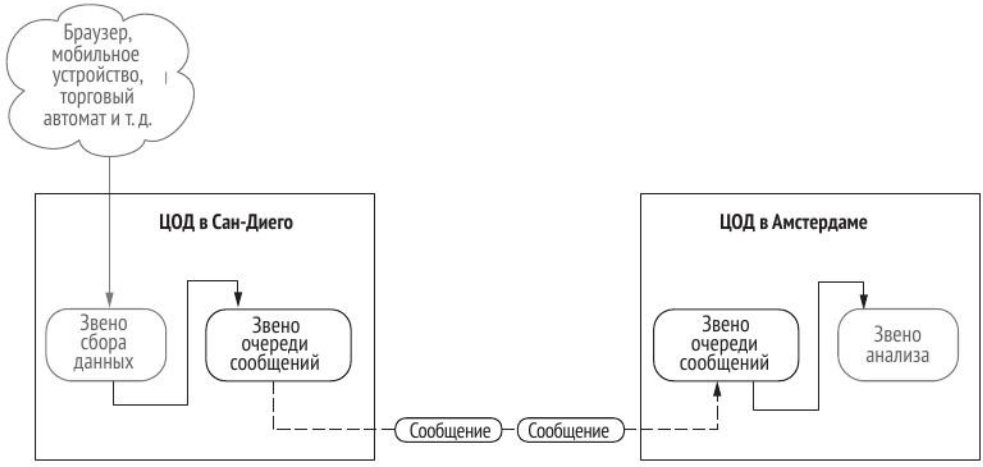


Рис. 3.7. Два центра обработки данных и поток данных между ними

В ЦОДе, который находится в Сан-Диего, работает звено сбора данных, а в амстердамском ЦОДе – звено анализа. Звено анализа нуждается в данных

от звена сбора. Все замечательно, мы собираем данные в Сан-Диего, а анализируем их в Амстердаме. Но однажды не повезло – вечер пятницы, вы как раз предвкушаете выходной, а тут растяпа-экскаваторщик ковшом перебил волоконно-оптический кабель, в результате чего связь между ЦОДами оборвалась (рис. 3.8).

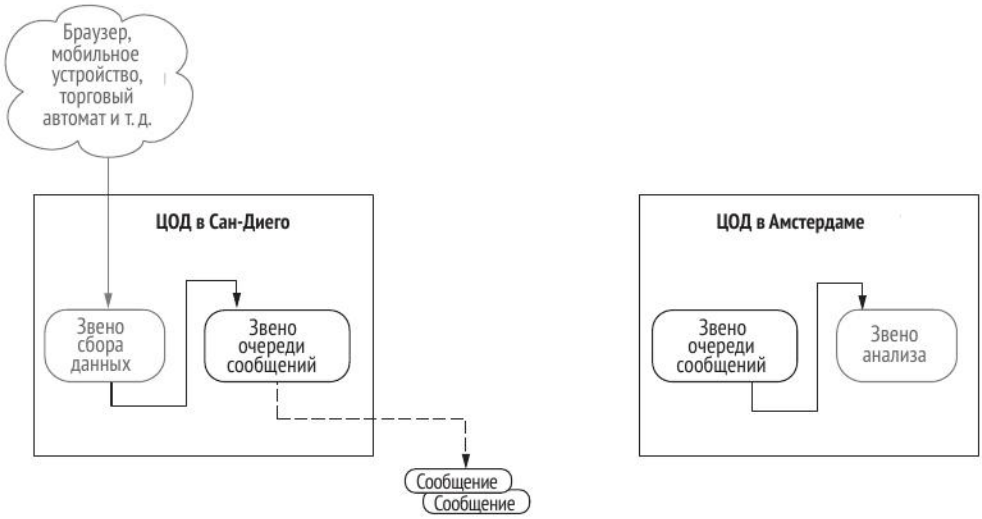


Рис. 3.8. Два центра обработки данных и поток данных, вытекающий в океан

Телекоммуникационная компания, владеющая линией, говорит, что на ремонт уйдет два-три дня. И как это отразится на вашем бизнесе? Сколько данных от звена сбора можно потерять без катастрофических последствий? Если потеря данных за несколько дней нетерпима, то выбирайте такую технологию очередей, которая способна хранить сообщения в течение длительного времени. На рис. 3.9 показаны место долговечного хранения сообщений в этом звене и некоторые варианты реализации.

Долговечные сообщения не только в какой-то мере обеспечивают отказоустойчивость, а значит, и аварийное восстановление, но и открывают возможность для поддержки отключающихся потребителей. Допустим, вы создаете систему, которая в реальном времени строит транспортные маршруты и, позволяет водителям получать обновления на смартфон и перестраивать маршрут в зависимости от текущей дорожной обстановки. Спустя три месяца руководство захотело вывести на рынок программу воспроизведения прошлой дорожной обстановки – пользователь выбирает город и воспроизводит данные за указанные день, неделю или месяц. Если архитектура системы такая, как на рис. 3.10, то сообщения, потребленные звеном анализа, удаляются из очереди сообщений, т. е. пропадают навсегда, так что воспроизвести прошлую дорожную обстановку не получится.

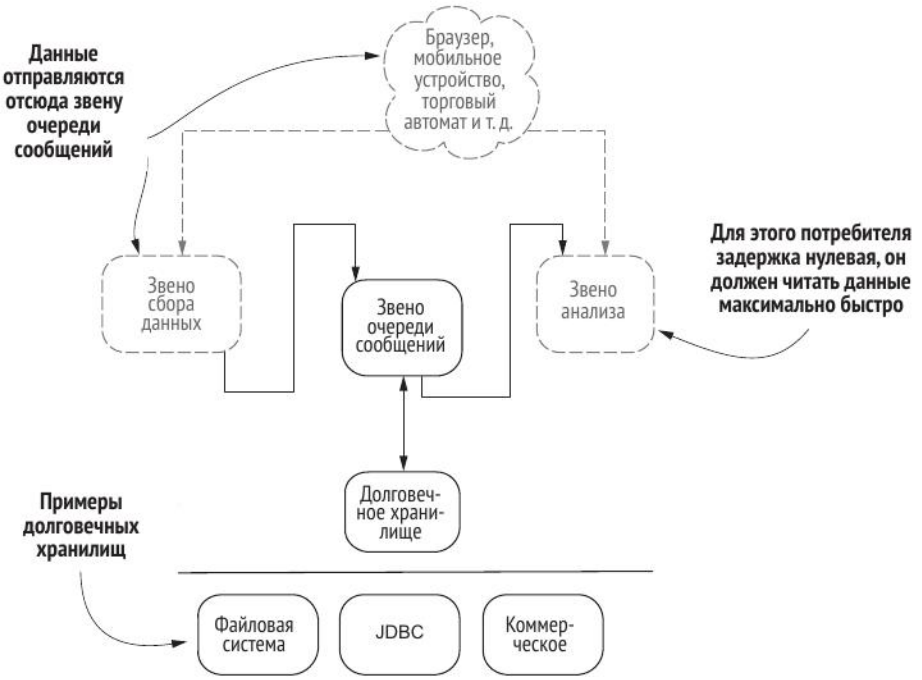


Рис. 3.9. Долговечные сообщения – где их место и как их можно хранить

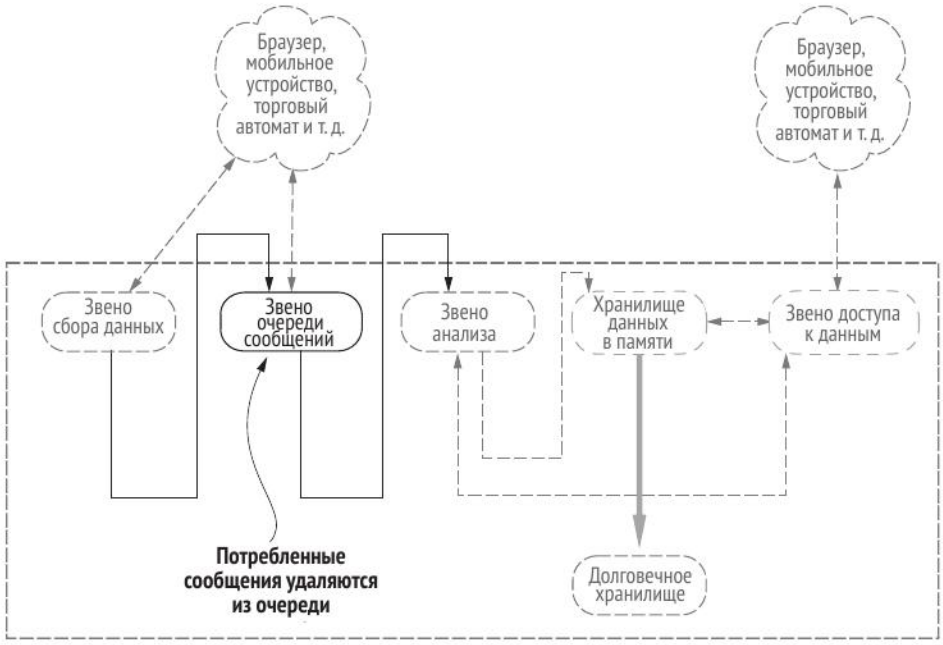


Рис. 3.10. Недолговечные сообщения удаляются после потребления звеном анализа

Для решения этой проблемы нужна архитектура, изображенная на рис. 3.11.

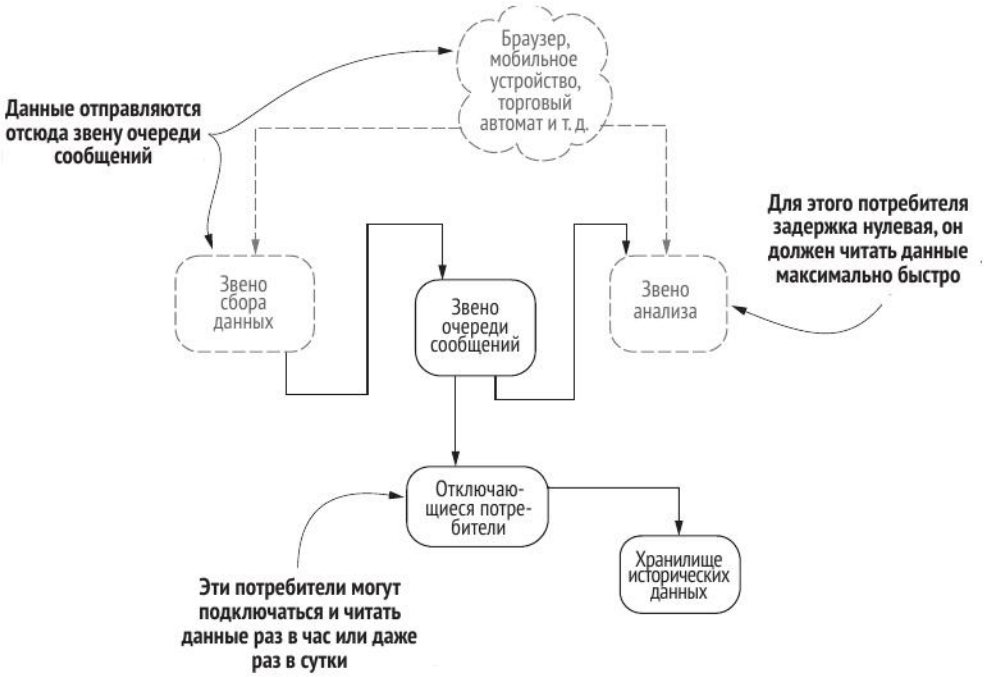


Рис. 3.11. Отключающиеся потребители сохраняют данные для анализа истории и отчетов

Если в вашей задаче требуется поддержка отключающихся потребителей или нужно, чтобы связь между производителями и потребителями была полностью разорвана, то ищите продукт, поддерживающий долговечные сообщения.

3.2.4. Семантика доставки сообщений

Напомним, что производитель отправляет сообщения брокеру, а потребитель запрашивает их у брокера. Это очень общее описание того, как работает механизм доставки сообщений. Теперь пойдем дальше и посмотрим, какие семантические гарантии дают продукты для обмена сообщениями. Есть три вида таких гарантий.

- *Не более одного раза* – сообщение может потеряться, но ни в каком случае потребитель не прочтет его дважды.
- *Не менее одного раза* – сообщение не может потеряться, но может быть несколько раз прочитано потребителем.
- *Ровно один раз* – сообщение не может потеряться, и потребитель читает его один и только один раз.

Какое решение выбрать? Если вы считаете, что «ровно один раз», то вы не одиноки. Большинство людей хочет иметь систему, в которой сообщения никогда не теряются и каждое доставляется потребителю один и только один раз. Конечно, кто бы от такого отказался? Увы, все не так просто – подобные системы изобилуют подводными камнями и рисками. На рис. 3.12 показаны точки возможного отказа.

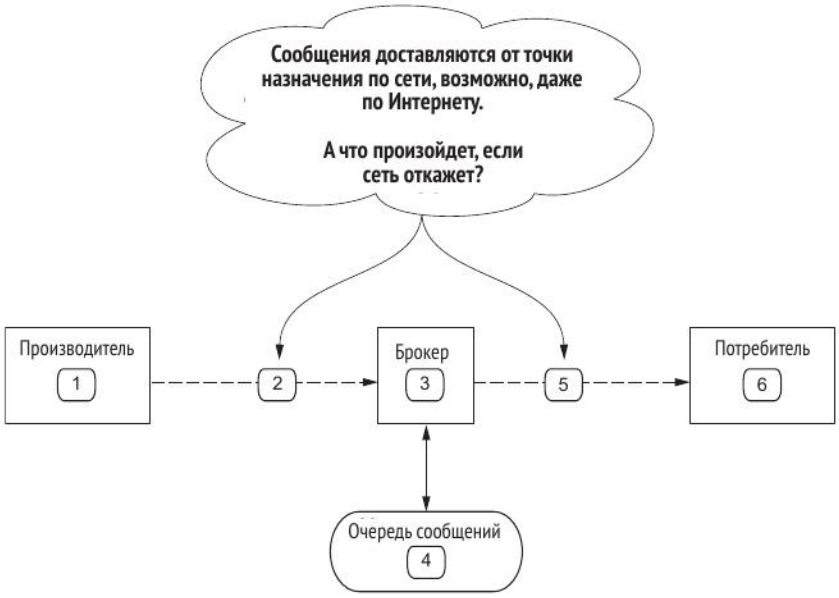


Рис. 3.12. Точки отказа, которые следует учитывать

Вот так-то! Похоже, чуть ли не каждое место диаграммы является точкой возможного отказа. Но не пугайтесь, не все так мрачно. Рассмотрим эти точки более пристально и разберемся, в чем заключается опасность.

1. *Производитель* – если производитель отказывает после того, как сообщение сгенерировано, но до того, как оно отправлено по сети брокеру, то сообщение будет потеряно. Есть также шанс, что производитель откажет, ожидая подтверждения приема сообщения от брокера, а после восстановления еще раз отправит то же самое сообщение.
2. *Сеть между производителем и брокером* – если выходит из строя сеть между производителем и брокером, то сообщение, отправленное производителем, не дойдет до брокера или брокер его получит, но производитель не увидит отправленного брокером подтверждения. В обоих случаях производитель может отправить сообщение повторно.
3. *Брокер* – если брокер отказывает, когда сообщения еще находятся в оперативной памяти и не сохранены в надежном хранилище, то сообщения будут потеряны. Если брокер отказывает, не успев послать

подтверждение производителю, то производитель может отправить сообщение повторно. Аналогично, если брокер следит за тем, какие сообщения были прочитаны потребителями, и отказывает, не успев сохранить эту информацию, то потребитель может прочитать одно и то же сообщение несколько раз.

4. *Очередь сообщений* – если очередь сообщений абстрагирует постоянное хранилище, то в случае отказа при попытке записи на диск сообщения могут быть потеряны.
5. *Сеть между потребителем и брокером* – если выходит из строя сеть между потребителем и брокером, то возможна ситуация, когда брокер отправил сообщение и запомнил этот факт, а потребитель так и не получил сообщения. Брокер ждет подтверждения о приеме, но не получает его, поэтому отправляет сообщение повторно.
6. *Потребитель* – если потребитель отказывает, не успев зафиксировать факт обработки сообщения – путем отправки подтверждения брокеру или записи в надежное хранилище, – то он может запросить то же самое сообщение повторно. Другая вариация на ту же тему – когда есть несколько потребителей и одно и то же сообщение доставляется двум или более.

Я понимаю, что такое изобилие вариантов может ошеломить, но не расстраивайтесь. Мы еще не раз будем обсуждать эти семантики. В контексте системы очередей сообщений мы должны помнить о сценариях отказа; тогда, если в документации по системе сказано, что она обеспечивает доставку ровно один раз, мы будем понимать, в какой мере это действительно так. Как часто бывает, при выборе технологии приходится идти на компромиссы, описанные в табл. 3.1.

Таблица 3.1. Компромиссы при выборе системы очередей сообщений

Меньшая сложность, повышенная производительность, более слабые гарантии	или	Большая сложность, пониженная производительность, более сильные гарантии

Что выбрать, зависит от конкретной задачи. Если вы создаете потоковый продукт для веб-аналитики, то потеря некоторых сообщений не сильно повлияет на качество продукта. Если же речь идет о потоковой системе обнаружения мошеннических операций, то пропускать сообщения крайне нежелательно.

При изучении систем сообщений может выясниться, что та, которая вам больше всего нравится, например Apache Kafka или Apache ActiveMQ, не дает гарантий доставки ровно один раз. Но не отчаивайтесь. Часто система предоставляет достаточно метаданных о сообщениях, чтобы семантику однократной доставки можно было реализовать самостоятельно, добавив координацию производителей и потребителей.

На рис. 3.13 представлены методы решения этой проблемы.

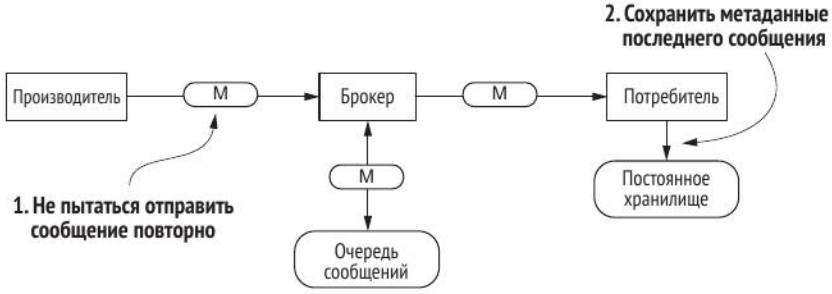


Рис. 3.13. Два способа обеспечить семантику однократной доставки в случае, когда система сообщений ее не гарантирует

- *Не пытаться отправлять сообщения повторно.* Это первое, что следует сделать. Для этого нужно каким-то образом запоминать, какие сообщения производитель (или производители) отправляет брокеру (или брокерам). Если ответ от брокера не поступил или сетевое соединение между производителем и брокером оборвалось, то мы можем прочитать данные от брокера и проверить, было ли получено сообщение, для которого не пришло подтверждение. Этот механизм гарантирует, что производитель отправляет каждое сообщение ровно один раз.
- *Сохранять метаданные последнего сообщения* – в этом случае мы должны сохранять некоторые данные о последнем прочитанном сообщении. Какие метаданные сохранять, зависит от системы сообщений. Если система основана на технологии JMS, то сохранять можно JMSMessageID, а если это Apache Kafka – то смещение сообщения. Так или иначе, нужны такие данные о сообщении, которые позволят удостовериться, что потребитель не обрабатывает сообщение повторно. На рис. 3.13 видно, что метаданные запоминаются в постоянном хранилище. Но нужно еще решить, что делать, если при сохранении метаданных произойдет ошибка.

Реализовав оба приема, вы можете гарантировать семантику однократной доставки сообщений. Возможно, вы не обратили внимания, но попутно вы получаете два небольших бонуса. Ну, не в прямом смысле, конечно, а в виде качества и надежности системы. Взгляните еще раз на рис. 3.13 и прочитайте обсуждение. Как вы думаете, о каких бонусах идет речь? Я имею в виду аудит сообщений и обнаружение дубликатов, хотя, возможно, есть еще какие-то.

Если говорить об аудите, то мы уже отслеживаем на уровне метаданных сообщения, отправляемые производителем. На стороне потребителя эти

метаданные можно использовать, чтобы учитывать не только сами поступающие сообщения, но также максимальное, минимальное и среднее время обработки сообщения. Быть может, так удастся выявить медленного производителя или медленного потребителя. Что касается обнаружения дубликатов, то мы уже решили, что производитель будет проверять, что сообщение было отправлено брокеру ровно один раз. А потребитель проверяет, было ли сообщение уже обработано.

И еще один совет: сделайте так, чтобы потребитель не просто запоминал метаданные, раскрываемые самой системой сообщений (некоторые из них делают доступным идентификатор сообщения, зная который, можно проверить, было ли сообщение обработано). Сохраняйте также метаданные, которые позволят однозначно идентифицировать полезную нагрузку сообщения. Это упростит дедупликацию сообщений и аудит данных.

Теперь вы знаете, как гарантировать семантику однократной доставки, а также реализовать аудит сообщений и обнаружение дубликатов. Мы еще встретимся с этими концепциями на страницах этой книги, а возможно, вы найдете и другие способы воспользоваться аудитом сообщений в потоковой архитектуре.

3.3. Безопасность

До сих пор нас интересовало только, как не потерять данные и не затопить сообщениями брокера или потребителей, а также как восстановиться после отказа. И теперь мы знаем, как построить надежное звено очереди сообщений. Но есть еще одна проблема: безопасность. В наши дни важно не только обезопасить данные в процессе передачи и в месте хранения, но также следить за тем, разрешено ли производителю порождать сообщения, а потребителю – потреблять их. На рис. 3.14 показаны все места, на которые следует обратить внимание, обеспечивая безопасность звена очереди сообщений.

Выходит, нам предстоит решить непростую задачу обеспечения безопасности на уровне звена очереди сообщений. Как минимум, вы должны продумать эти вопросы, а если работаете совместно с группой безопасности, то хорошо было бы подключить ее к защите этого и других звеньев. На верхнем уровне все это выглядит не слишком сложно, но, как я уже говорил, дьявол кроется в деталях. В частности, нужно как-то обойти ограничения в случае, если выбранная система очередей сообщений не поддерживает все, что вам нужно. Если вы хотите глубоко изучить проблему безопасности в распределенных системах, то рекомендую начать с книги Ross Anderson «Security Engineering: A Guide to Building Dependable Distributed Systems» (Wiley, 2008).

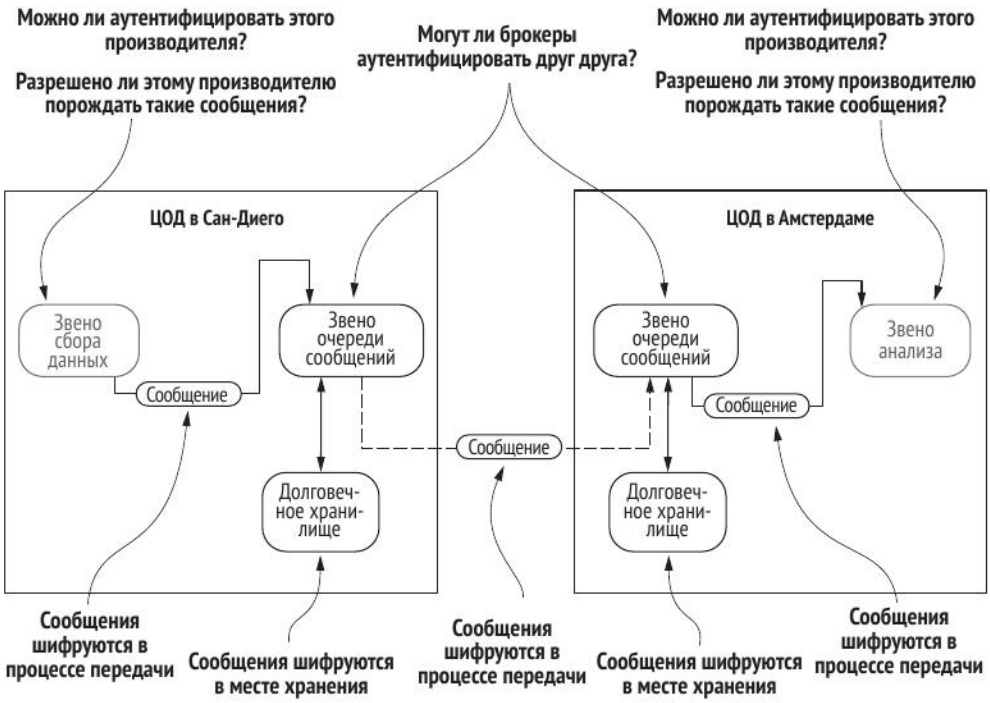


Рис. 3.14. Точки звена передачи сообщений, в которых следует обеспечить безопасность

3.4. Отказоустойчивость

Итак, звенья сбора данных и анализа изолированы друг от друга, сообщения между ними опосредованы звеном очереди сообщений. Теперь надо понять, что произойдет с данными, когда случится что-то нехорошее. Подчеркиваю: не *если* случится, а *когда* случится. Сколько вы сможете указать мест в архитектуре с несколькими ЦОДами, где возможна потеря данных? Взгляните на рис. 3.15 и сравните со своими выводами.

Некоторые проблемы, показанные на рис. 3.15, уже знакомы нам по главе 2, в частности обработка сетевых отказов производителем. Потеря связи между ЦОДами – реальность, последствия которой можно сгладить за счет долговечного хранилища в составе брокера. Если в системе очередей сообщений, которая удовлетворяет требованиям бизнеса, нет долговечного хранилища, то придется либо смириться с потерей данных, либо найти другой способ уменьшить риск потери связности сети. На рисунке также показан отказ сети на стороне потребителя в составе звена анализа. Эта проблема обсуждается в главе 4.

Посмотрим более внимательно, что может случиться в этом звене. На рис. 3.16 представлена детальная картина звена очереди сообщений.

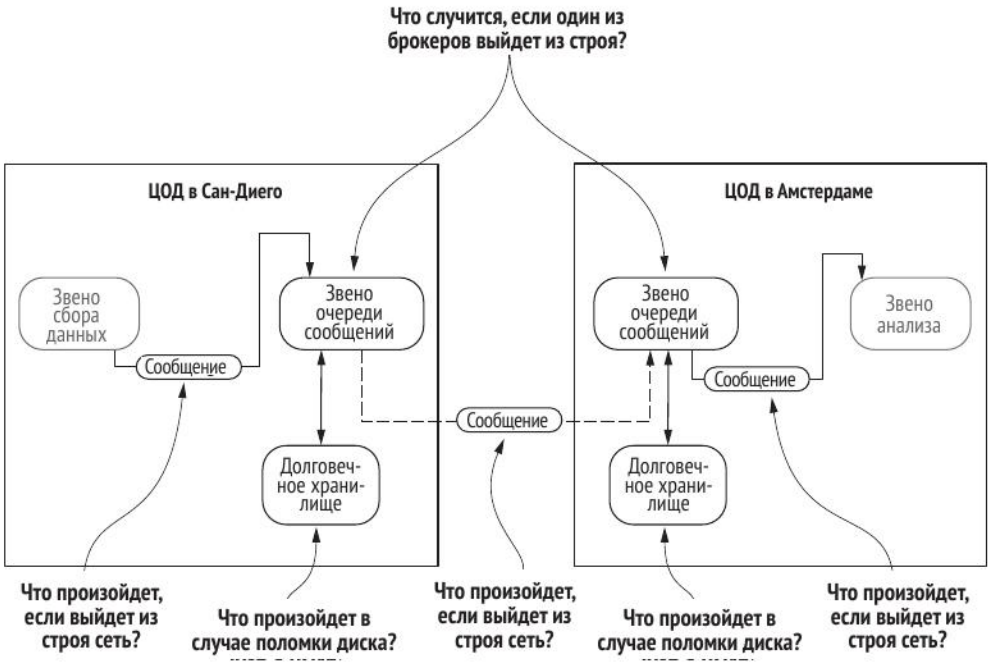


Рис. 3.15. Общая картина возможных отказов

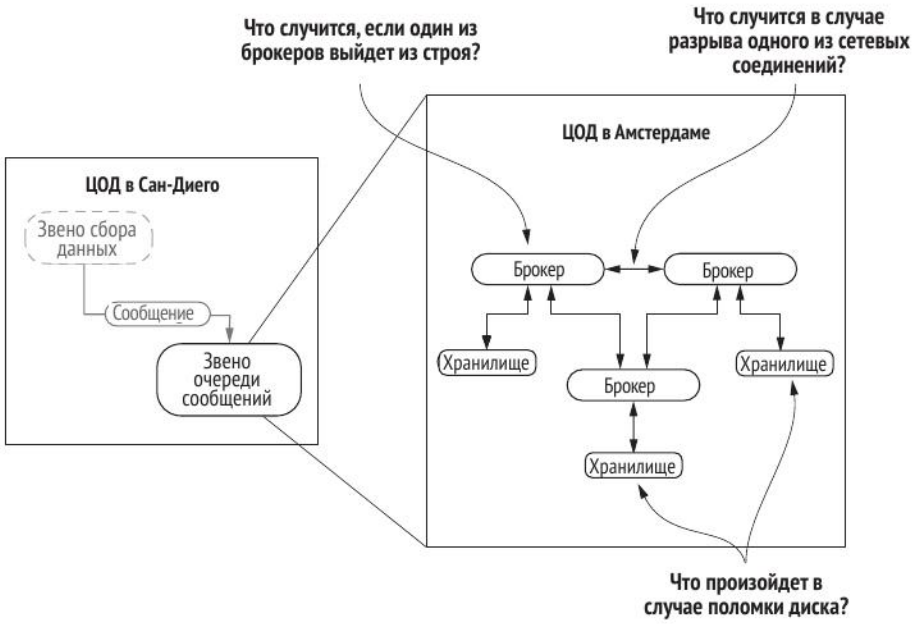


Рис. 3.16. Детальное представление брокеров и возможных отказов

Здесь показаны три брокера, но важно не столько их число, сколько то, что происходит между ними и где может случиться неприятность. Когда

дело дойдет до выбора конкретного продукта, количество развернутых брокеров будет иметь значение, и вам предстоит хорошенько подумать, как лучше распорядиться выбранным продуктом. А пока перед нами другая задача – понять, что может сломаться и какие вопросы нужно задать себе при оценке различных продуктов.

Попробуем ответить на вопросы, поставленные на рис. 3.16. И сам перечень вопросов, и последующее их обсуждение нельзя назвать исчерпывающими, но это неплохая отправная точка для более детального обдумывания проблемы.

- *Что случится, если один из брокеров выйдет из строя?* Интересная проблема. Если брокер использует долговечное хранилище, то можно надеяться, что риску подвергаются только те сообщения, которые находились в оперативной памяти в момент аварии брокера. Есть несколько способов снизить этот риск.
  - Можно воспользоваться уроками главы 2: чтобы гарантировать доставку сообщений, производитель должен дождаться подтверждения записи данных на диск.
  - Можно заставить систему очередей сообщений реплицировать сообщение нескольким брокерам. В этом случае риск остается, но он значительно меньше, потому что проблема возникает, только когда два или более брокеров одновременно «падают», не успев записать сообщения на диск.
  - Можно настроить брокер, так чтобы он хранил в памяти как можно меньше данных. Этот метод чреват снижением производительности, т. е. мы выбираем между производительностью и долговечностью.
- *Что случится в случае разрыва одного из сетевых соединений?* При использовании многих систем очередей сообщений, поддерживающих репликацию, ваши данные в безопасности, потому что хранятся в нескольких брокерах. Кроме того, после восстановления связности сети брокер снова присоединится к кластеру и синхронизируется с другими брокерами, получив от них недостающие сообщения. Выбирая продукт, получите ответы на следующие смежные вопросы:
  - выбирается ли в качестве новой реплики другой брокер?
  - что происходит после восстановления связности сети?
  - существует ли возможность настроить тайм-аут, при превышении которого считается, что сеть «упала»?
  - что произойдет с данными, если связность сети не восстановится?
  - что произойдет с данными, если брокер отсоединился от кластера в момент, когда производитель отправлял ему сообщение?

- *Что произойдет в случае поломки диска?* Хотя такая ситуация чревата катастрофической потерей данных, эксплуатационники знают, как с этим бороться. Если задействовано несколько брокеров, задайтесь следующими вопросами:
  - существуют ли реплики потерянных данных?
  - что, если реплицируемые данные не успели записать на диск до поломки: будут ли эти данные потеряны?
  - как восстановить брокер?

Надеюсь, что вы сумеете поставить эти вопросы и получить на них ответы, когда будете оценивать пригодность конкретного продукта для решения своей задачи.

3.5. Применение базовых концепций в конкретных задачах

Рассмотрев базовые концепции, относящиеся к звену очереди сообщений, попробуем применить их к различным сценариям.

Финансы: обнаружение мошенничества

Компания Пола предоставляет службу для обнаружения мошенничества в режиме реального времени. Для этого необходимо собирать сведения об операциях по кредитным картам во всем вебе, применять к данным хитроумные алгоритмы и отправлять клиентам сообщения об одобрении или отклонении операции в момент совершения покупки. С учетом всего, что мы обсуждали выше, в табл. 3.2 перечислены вопросы, которые Пол должен задать себе при проектировании архитектуры приложения.

Таблица 3.2. Вопросы к приложению для обнаружения мошеннических операций

Вопрос	Обсуждение
Как повлияет на бизнес Пола длительное отсутствие связи между звеном сбора данных и звеном анализа?	Это катастрофа. Компания Пола не сможет предоставлять обслуживание, а это может негативно сказаться на бизнесе его заказчиков
Данные за сколько дней может потерять без последствий бизнес Пола?	Ни за сколько. Более того, с учетом характера приложения и типа обрабатываемых данных я бы сказал, что потери данных вообще не должно быть – никакой
Должен ли, по вашему мнению, Пол хранить старые данные?	Я бы не удивился, если бы хотя бы один заказчик или кто-то из руководства компанией запросил детальный отчет о функционировании службы за период времени
Какая семантика доставки сообщений нужна в бизнесе Пола?	Я считаю, что нужна семантика в точности однократной доставки. Иначе можно пропустить сообщение, а значит, и мошенническую операцию. Можно ли обойтись семантикой «не менее одного раза»? Не исключено. Это увеличит сложность потребителя но, в принципе, возможно

Это было интересно. А теперь возьмем два совершенно других приложения и посмотрим, сможете ли вы ответить на те же вопросы.

Интернет вещей: твитингующий автомат для продажи кока-колы

Компании Фрэнка принадлежат тысячи автоматов для продажи кока-колы, и он хотел бы сделать их «социальными». Он хочет, что его автоматы отправляли твиты и push-уведомления со специальными предложениями находящимся поблизости потребителям. А если и этого недостаточно, то вот вам еще одна фишка: если в ближайшем автомате кончился товар, то он должен рекомендовать другой находящийся рядом автомат, который может предложить выгодную сделку. Как бы вы ответили на вопросы из табл. 3.3? В столбец «Обсуждение» я добавил несколько дополнительных вопросов для обдумывания. Поскольку бизнес-требования в данном случае кардинально отличаются, не удивляйтесь, если ваши ответы окажутся не такими, как в предыдущем примере. Ваша задача – потратить некоторое время на всестороннее рассмотрение проблемы.

Таблица 3.3. Вопросы к приложению для Интернета вещей

Вопрос	Обсуждение
Как повлияет на бизнес Фрэнка длительное отсутствие связи между звеном сбора данных и звеном анализа?	Очевидно, что Фрэнк может потерять деньги, если социальные автоматы составляют значительную долю в выручке. А что, если он использует их еще и для управления запасами? Будет ли влияние более сильным?
Данные за сколько дней бизнес Фрэнка может потерять без последствий?	Это зависит от того, как устроена система управления запасами, и от популярности автоматов. Возможно, также от места расположения
Должен ли, по вашему мнению, Фрэнк хранить старые данные?	Возможно – если он хочет получать отчеты о работе своих социальных ответов в динамике
Какая семантика доставки сообщений нужна в бизнесе Фрэнка?	В данном случае семантики «не менее одного раза» достаточно, хотя я думаю, что у вас найдутся аргументы в пользу семантики «не более одного раза». Что, по-вашему, может случиться, если одно сообщение будет обработано несколько раз? А если сообщение будет пропущено?

Электронная коммерция: рекомендование товаров

Рекс владеет компанией электронной торговли модной одеждой и думает о том, как увеличить коэффициент обращаемости посетителей сайта в покупателей. Он полагает, что в этом могут помочь социальные методики. С этой целью Рекс попросил вас спроектировать систему, которая будет показывать посетителям сайта, что другие посетители недавно положили в свою корзину или купили. Если я рассматриваю джинсы, а другие покупатели вместе с этими джинсами положили в корзину ботинки или купили вместе с ними рубашку, то я должен увидеть рекомендацию, пред-

лагающую купить также ботинки и рубашку. Сообщение может выглядеть как-то так: «Десять человек только что купили эти [наименование товара] вместе с джинсами, которыми вы заинтересовались» или «Пять человек добавили в корзину вместе с этими джинсами еще и [наименование товара]». Короче говоря, вы должны удовлетворить следующие требования:

- отслеживать все покупки в реальном времени;
- отслеживать корзины всех покупателей в реальном времени;
- на странице каждого товара показывать, какие товары недавно были куплены вместе с ним;
- на странице каждого товара показывать, какие товары находятся вместе с ними в корзинах других покупателей.

Имея в виду эти требования, ответьте на вопросы в табл. 3.4, чтобы мы могли правильно спроектировать систему для Рекса. Я добавил в столбец «Обсуждение» свои соображения, которые, возможно, помогут вам в анализе.

Таблица 3.4. Вопросы к приложению для электронной коммерции

Вопрос	Обсуждение
Как повлияет на бизнес Рекса длительное отсутствие связи между звеном сбора данных и звеном анализа?	Если отсутствуют маркетинговые возможности для продажи похожих и сопутствующих товаров, то эта функция просто не будет работать. Представьте, что вы зашли на сайт Amazon, а там не показывают рекомендации. Количественно оценить финансовые потери Рекса трудно, но нет сомнений, что он упустит выгоду и продаст меньше, чем мог бы
Данные за сколько дней бизнес Рекса может потерять без последствий?	Быть может, мы могли бы предложить старомодную пакетную рекомендательную систему в качестве резервной, но актуальность данных играет важную роль
Должен ли, по вашему мнению, Рекс хранить старые данные?	Возможно – для отчетности и уточнения алгоритмов
Какая семантика доставки сообщений нужна в бизнесе Рекса?	Семантики «не менее одного раза» достаточно. Семантика «ровно один раз» в этом случае излишество. Как вы думаете, поможет ли семантика «не более одного раза»?

3.6. Резюме

В этой главе мы разобрались, как разорвать связь между звеньями сбора и анализа данных путем размещения между ними звена очереди сообщений. Попутно мы:

См. раздел 9.2.1.

- поняли, зачем нужно звено очереди сообщений;
- узнали, что такое долговечность сообщения;

- научились поддерживать периодически отключающихся потребителей;
- узнали о различных семантиках доставки сообщений;
- поняли, как выбирать технологию, отвечающую решаемой задаче.

По мере чтения последующих глав рассмотренные только что концепции встретятся еще не раз, так что не расстраивайтесь, если не все поняли, – мы еще поговорим о семантике доставки сообщений. Если вы хотите узнать больше о многочисленных аспектах систем сообщений и различных способах интеграции, обратитесь к книге Gregor Hohpe, Bobby Woolf «Enterprise Integration Patterns» (Addison-Wesley, 2003).

Ну а мы готовы сделать что-нибудь интересное с собранными данными. Следующая глава посвящена звену анализа – нашему потребителю сообщений.

Глава 4

Анализ потоковых данных

Краткое содержание главы:

- анализ данных в движении;
- типичная архитектура потоковой обработки;
- основные функции каркасов потоковой обработки.

В главе 3 мы говорили о важности звена очереди сообщений. Смысл этого звена в том, чтобы получать данные от звена сбора данных и делать их доступными другим компонентам потоковой архитектуры. В этой точке данные готовы к потреблению и ожидают, когда над ними произведут разные магические действия. В этой главе мы познакомимся со звеном анализа, а в главе 5 узнаем о том, как с его помощью волшебным образом преображать данные. Памятуя об этом плане, взглянем еще раз на архитектурную диаграмму на рис. 4.1, чтобы вспомнить, как устроен поток данных.

Отметим, что, в отличие от главы 3, где мы обсуждали ввод и вывод данных, сейчас нас будет интересовать только ввод. Разговор о том, куда попадают данные после этого звена, мы отложим до следующей главы. Прочитав эту главу, вы будете знать о базовых концепциях, встречающихся во всех современных инструментах, которые используются в этом звене, и подготовитесь к выполнению различных операций над данными.

Налейте себе чашечку кофе – и приступим.

4.1. Анализ данных в движении

Для понимания обсуждаемой в этой главе функциональности необходимо усвоить концепцию *данных в движении* (in-flight data) и *непрерывных запросов*. Под данными в движении понимаются все кортежи в системе, от источника (звена очереди сообщений) до клиента на выходе (следующего звена). Данные находятся в движении, а не в покое, т. е. не записываются в постоянное хранилище. (Под данными *в покое* понимаются данные,

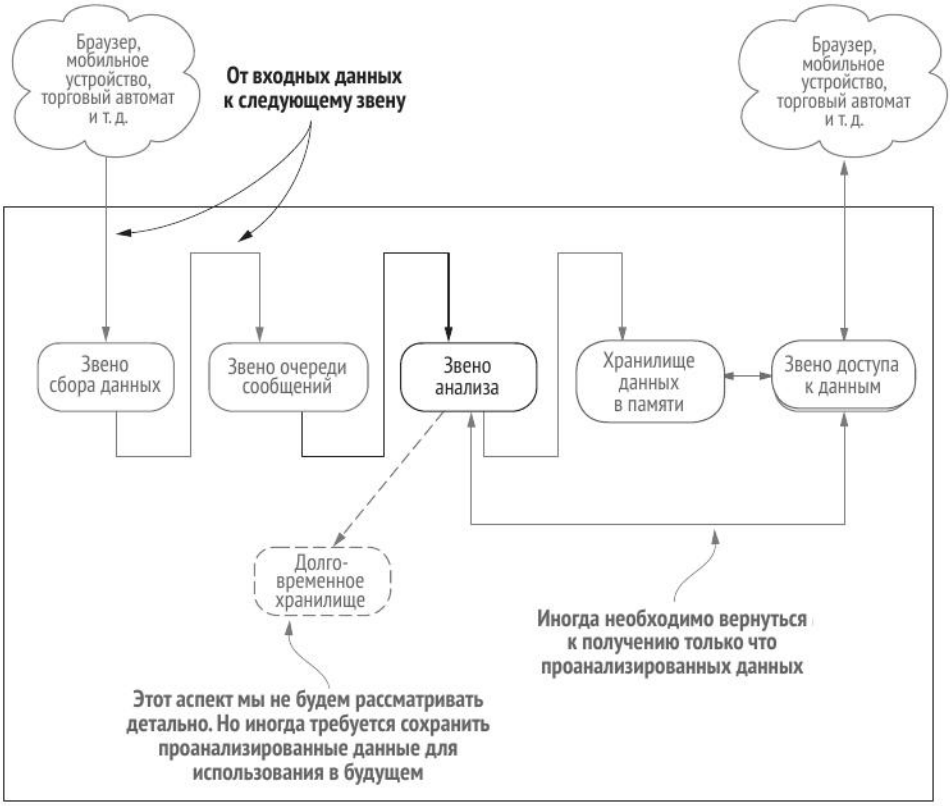


Рис. 4.1. Архитектура потоковой обработки данных – выделено звено анализа

записанные на диск или иное устройство хранения.) На рис. 4.2 показано, как это выглядит в нашей потоковой архитектуре.

Из рисунка видно, что наша цель – максимально быстро вытянуть данные из звена очереди сообщений; в идеале звено анализа должно выдерживать тот темп, в котором звено сбора проталкивает данные звену очереди сообщений. В чем тут отличие от непотоковой системы, например построенной на основе традиционной СУБД (реляционной, Hadoop, HBase, Cassandra и т. д.)? В непотоковой системе данные находятся в покое, и мы формулируем запросы, на которые система дает ответы. В потоковой системе все переворачивается с ног на голову – под фиксированный запрос подаются данные. Эта модель называется *моделью непрерывного запроса*, т. е. мы постоянно вычисляем ответ на запрос по мере поступления новых данных.

Представьте, что вы управляете крупным новостным агентством и хотите знать, пользуется ли статья популярностью или не побилась ли ссылка. Это нужно, для того чтобы скорректировать маркетинговую кампанию или исправить сайт. При использовании традиционных технологий на основе СУБД нужно было бы сделать следующее:

Рис. 4.2. Звено сбора проталкивает данные, а звено анализа вытягивает их

1. Собрать данные с сайта.
2. Загрузить эти данные в СУБД.
3. Выполнить запрос и узнать, побилась ли ссылка или пользуется ли статья популярностью.
4. Предпринять соответствующее действие.
5. Повторять через каждые *x* минут, а скорее часов.

Сравните с действиями, выполняемыми в потоковой системе:

1. Собрать поток данных.
2. Запустить запрос, который определяет, побилась ли ссылка или пользуется ли статья популярностью.
3. Предпринять соответствующее действие.

Думаю, вы согласитесь, что современному бизнесу было бы трудно реагировать на изменения, если он пользуется традиционной системой, тогда как в потоковой системе запрос всегда вычисляется применительно к данным, и на проблемы или обнаруженные тенденции можно реагировать в режиме реального времени. В потоковой системе пользователь (или приложение) регистрирует запрос, который выполняется при каждом поступлении данных или с заранее заданным интервалом. Результат выполнения проталкивается клиенту.

Отметим ключевые различия.

- В традиционных системах управления базами данных пользователь (активная сторона), желающий получить ответ на вопрос, отправляет системе (пассивной стороне) запрос, а та возвращает ответ. Ответ всегда основан на данных, загруженных в систему до отправки запроса, т. е. набор данных, по существу, статический.
- В потоковой системе запрос запускается и непрерывно (по таймеру или по другому событию) выполняется для поступающих в систему

данных. Ответ на запрос проталкивается следующему звену, в роли которого может выступать пользователь или приложение.

Тем самым традиционная модель управления данными инвертируется: пользователь становится пассивной стороной, а система управления – активной. На рис. 4.3 это изображено графически.

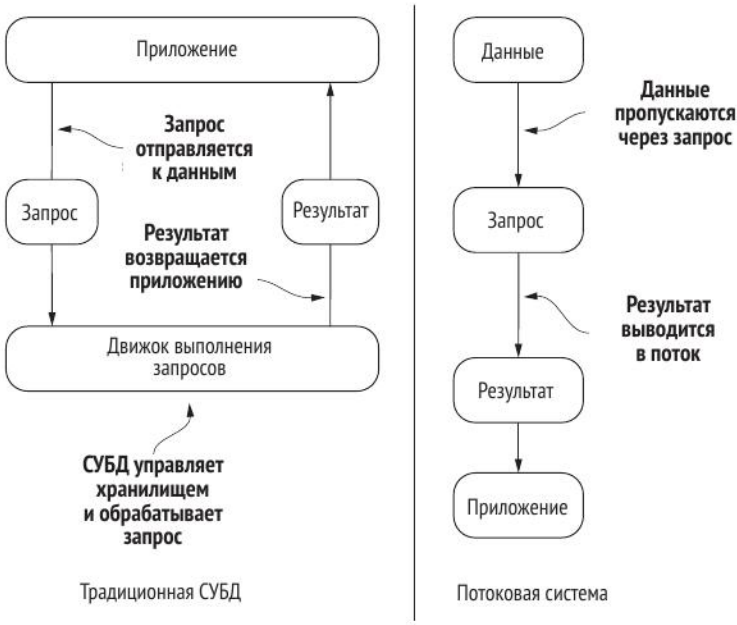


Рис. 4.3. Переворачивание с ног на голову: традиционная и потоковая системы

В левой части показана традиционная СУБД – запрос отправляется к данным и выполняется, а результат возвращается приложению. В потоковой системе, показанной справа, модель совершенно иная – данные пропускаются через запрос, а результат посылается приложению. В потоковой системе данные проталкиваются или вытягиваются, образуя никогда не завершающийся поток; это, безусловно, отражается как на дизайне системы, так и на способе предъявления ей запросов. Чтобы лучше прочувствовать разницу, в табл. 4.1 показаны основные отличия традиционной СУБД от потоковой системы.

Таблица 4.1. Сравнение традиционной СУБД с потоковой системой

	СУБД	Потоковая система
Модель запроса	Запросы основаны на одноразовой модели в предположении непротиворечивого состояния данных. В одноразовой модели пользователь выполняет запрос, получает ответ, после чего запрос забывается. Это модель вытягивания.	Запрос постоянно выполняется для данных, протекающих через систему. Пользователь один раз регистрирует запрос, и результаты регулярно проталкиваются клиенту

	СУБД	Потоковая система
Изменение данных	Когда система не работает, данные не могут измениться	Многие потоковые приложения продолжают генерировать данные, даже когда звено анализа не работает, поэтому после сбоя может понадобиться наверстать упущенное
Состояние запроса	Если система «падает» в момент выполнения запроса, то он забывается. Повторное выполнение запроса после восстановления системы – задача приложения (или пользователя)	Зарегистрированные непрерывные запросы иногда нужно продолжать с прерванного места, а иногда не нужно. Часто ситуация выглядит так, будто они никогда и не останавливались

Подумайте о данных, ежеминутно окружающих вас, – от сонмища подключенных устройств до ваших действий в сети. Сколько вопросов вы могли бы задать и сколько проблем решить, если бы удалось подвергнуть их потоковому анализу! Приведем лишь несколько примеров, чтобы возбудить ваше воображение.

- *Отслеживание поведения*. Представьте, что вы можете показывать персонализированное рекламное объявление в зависимости от местоположения пользователя, погоды и запомненных ранее покупательских привычек и предпочтений. Макдональдс делает это с помощью платформы VMob. В одном исследовании показано, что в Нидерландах Макдональдсу удалось увеличить количество откликов на предложения на 700%, причем пользователи, установившие приложение, возвращались в два раза чаще и тратили в среднем на 47% больше (http://mng.bz/1p0t).
- *Повышение безопасности и эффективности дорожного движения*. Согласно отчету Европейской комиссии (http://ec.europa.eu/transport/themes/urban/urban_mobility/index_en.htm), транспортные пробки в Европейском союзе (ЕС) в городах и их окрестностях ежегодно обходятся примерно в 100 миллиардов евро, или 1% ВВП ЕС. Согласно отчету Федерального управления шоссейных дорог (http://ops.fhwa.dot.gov/program_areas/reduce-non-cong.htm), 25% транспортных пробок происходит из-за ДТП. Представьте, что мы смогли воспользоваться дорожными датчиками транспортных средств (см. вводный обзор по адресу www.fhwa.dot.gov/policyinformation/pubs/vdstits2007/03.cfm); на основе анализа данных о дорожном движении мы могли бы предлагать водителям обновляемые сведения о дорожной обстановке и перестраивать маршруты для более эффективного перемещения. Реальные примеры см. на сайте Blip Systems (www.blipsystems.com/traffic/), где описывается, как в некоторых городах решены многочисленные транспортные проблемы.

- *Обнаружение мошенничества в режиме реального времени.* При каждой оплате покупки кредитной картой необходимо выполнить сложную последовательность действий, чтобы определить, является ли операция честной или мошеннической. Согласно отчету компании FICO (www.fico.com/en/node/8140?file=5582), потери от мошенничества с кредитными картами в США снизились на 70% после внедрения анализа операций в реальном времени.

Эти примеры – лишь верхушка айсберга, и мы надеемся, что они разожгли у вас аппетит к изучению других возможностей. А заодно помогли понять, что умение создавать подобные системы для работы с многочисленными потоками данных, доступными в современном мире, становится весьма востребованным навыком.

Но не будем забегать вперед – сейчас мы должны заняться изучением базовой функциональности звена анализа. Мы начнем с обсуждения общей архитектуры системы потоковой обработки данных, а затем перейдем к ключевым функциям и посмотрим, какую роль они играют в решении о выборе конкретного продукта.

4.2. Архитектуры распределенной обработки потоков

Звено анализа может работать и на одном компьютере, но объем и скорость поступления данных рано или поздно поставят крест на этом режиме. Представьте, к примеру, что вас интересуют не популярность статьи и поиск битых ссылок, а анализ работы газовой турбины в реальном времени с целью обнаружения признаков неполадок. Согласно данным компании General Electric, одна турбина порождает приблизительно 1 ТБ данных в час. А раз так, то давайте сразу займемся инструментами и технологиями, позволяющими строить распределенное звено анализа.

На рынке существует немало технологий для обработки потоков. На момент написания этой книги самыми популярными из продуктов с открытым исходным кодом были Spark Streaming, Storm, Flink и Samza – все они являются проектами Apache. Мы не будем вдаваться в детали каждого продукта, но кратко обсудим их, предварительно познакомившись с обобщенной потоковой архитектурой, чтобы понять, какое место они в ней занимают.

Обобщенная архитектура

Spark, Storm, Flink и Samza обладают рядом общих черт.

- Компонент, которому отправляется ваше потоковое приложение, – по аналогии с тем, как работает Hadoop MapReduce. Приложение отправляется на тот узел кластера, который будет его выполнять.

- Отдельные узлы кластера выполняют написанные вами потоковые алгоритмы.
- Входом для потоковых алгоритмов являются источники данных.

С учетом этих идей и соответствующих архитектур мы можем спроектировать общую архитектуру, показанную на рис. 4.4.

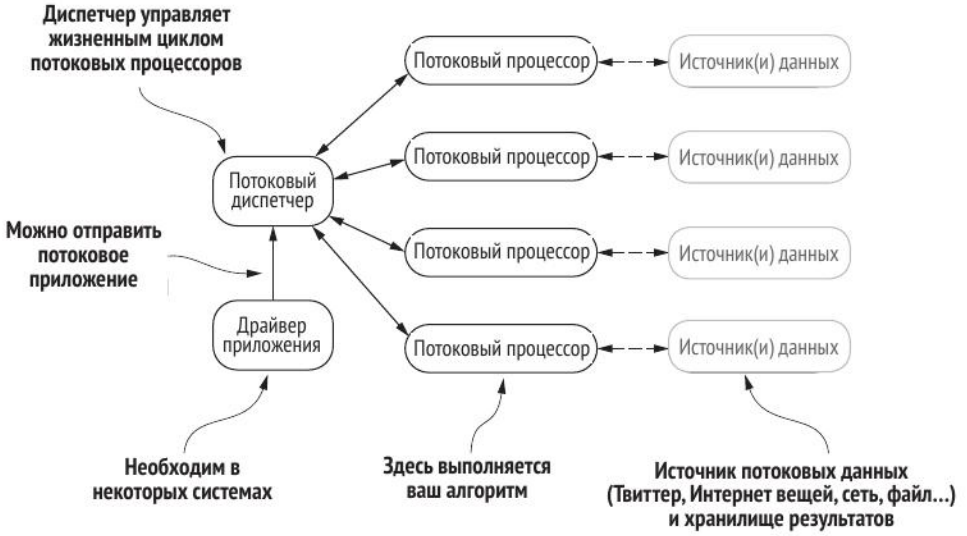


Рис. 4.4. Общая архитектура потокового анализа, встречающаяся во многих продуктах, представленных на рынке

На рынке имеются и другие потоковые системы, пока не достигшие той же популярности, что и вышеупомянутые, и без сомнения в будущем появятся новые. Зачастую другие продукты укладываются в ту же архитектуру, что облегчает их освоение.

Драйвер приложения. В некоторых потоковых системах это клиентский код, который определяет, как программируется потоковая обработка, и взаимодействует с потоковым диспетчером. Например, в технологии Spark Streaming ваш клиентский код разбивается на две логические части: драйвер и потоковый алгоритм, или задача. Драйвер отправляет задачу потоковому диспетчеру, быть может, собирает результаты в самом конце и управляет временем жизни задачи.

- *Потоковый диспетчер.* Этот компонент несет ответственность за передачу вашей потоковой задачи одному или нескольким потоковым процессорам; в некоторых случаях он запрашивает ресурсы, необходимые потоковым процессорам, и управляет ими.
- *Потоковый процессор.* Именно здесь происходит самое интересное – выполняется ваша задача. Конкретная форма зависит от платформы, но суть всегда одинакова – выполнить полученную задачу.

- *Источник(и) данных.* Представляют входные и, возможно, выходные данные потоковой задачи. Одни платформы позволяют подавать на вход задачи данные из нескольких источников, другие ограничиваются единственным источником. В показанной архитектуре не вполне ясно, куда деваются результаты задач. Иногда данные собирает ваш драйвер, а иногда они записываются в другой источник данных, чтобы их могла использовать другая система или другая задача.

Теперь, когда мы свели различные архитектуры к единой, вернемся к примеру мониторинга работы газовой турбины и посмотрим, как эта задача ложится на общую архитектуру. Это показано на рис. 4.5 (где общая архитектура упрощена, чтобы не загромождать рисунок деталями).

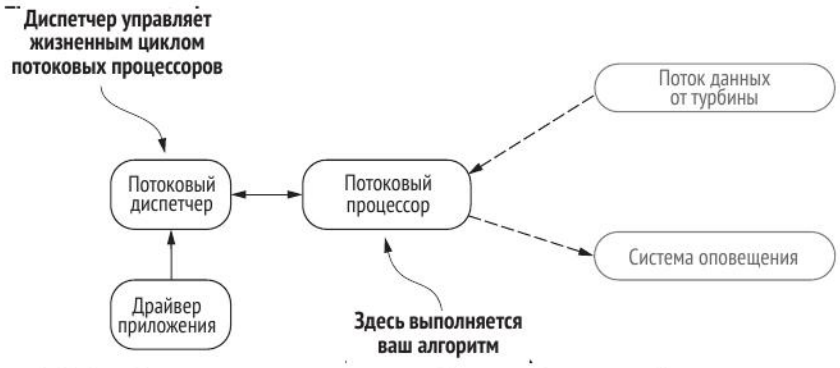


Рис. 4.5. Отображение мониторинга работы турбины на общую архитектуру

Я понимаю, что нужно время, чтобы уложить все это в сознании, поэтому поразмыслите над тем, как отобразить на предложенную архитектуру свою собственную прикладную задачу. Когда будете готовы, можно переходить к обсуждению архитектуры трех основных потоковых систем на рынке. А затем мы поговорим о ключевых функциях, которые следует рассматривать при выборе конкретного продукта.

Apache Spark Streaming

Система Apache Spark Streaming, которую часто называют просто Spark Streaming, построена поверх Apache Spark, как показано на рис. 4.6.

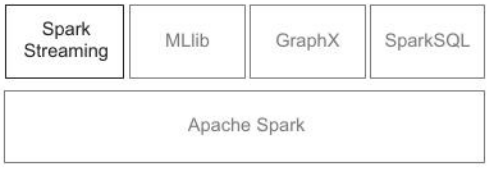


Рис. 4.6. Apache Spark Streaming и базовый стек Spark

Как видите, поверх Apache Spark надстроены и другие системы, так что она де-факто становится универсальной платформой распределенных вычислений. Эта платформа поддерживает несколько языков (Java, Scala,

Python и R), и на момент написания книги над ней были надстроены следующие высокоуровневые средства: Spark Streaming, MLlib (машинное обучение), SparkR (интеграция с R) и GraphX (для обработки графов). Помимо официальной документации проекта, для изучения Spark рекомендуем воспользоваться замечательной книгой Marko Bonaći, Petar Zečević «Spark in Action» (www.manning.com/books/spark-in-action). На рис. 4.7 показана общая архитектура Spark Streaming.

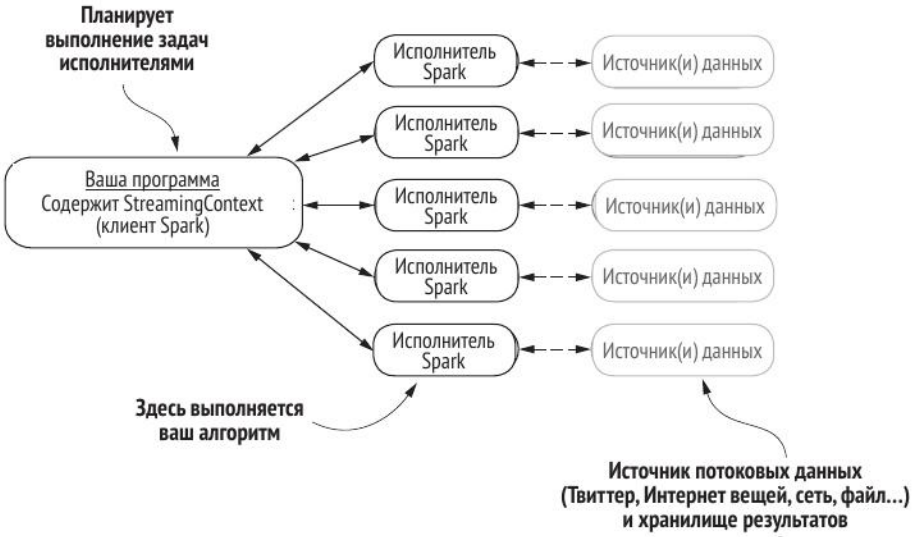


Рис. 4.7. Верхнеуровневая архитектура Spark Streaming

В левой части рисунка показана наша программа, содержащая так называемый потоковый контекст Spark – StreamingContext; все вместе это образует *драйвер*. Не вдаваясь в детали, скажем, что StreamingContext содержит логику, необходимую для отслеживания поступающих данных, подготовки потоковых задач, планирования их размещения на исполнителях Spark и выполнения. Вы, вероятно, обратили внимание, что мы говорим о задачах, а не о потоке. Spark, а вместе с ним и Spark Streaming оперирует пакетными работами. В случае Spark Streaming пакет представляет данные за период времени и может планироваться с периодичностью менее полсекунды. Под *задачей* в Spark Streaming понимается логика вашей программы, упакованная и переданная исполнителям Spark. Идея та же, что в технологии Hadoop MapReduce. В средней части рис. 4.7 изображены исполнители Spark, работающие на разных компьютерах (числом от одного до нескольких тысяч). Именно здесь выполняется ваша задача (написанный вами алгоритм обработки потока). Исполнители получают данные из внешнего источника и взаимодействуют с контекстом Spark StreamingContext, который является частью драйвера.

Apache Storm

Apache Storm – система потоковой обработки, работающая по принципу «один кортеж за раз». Она спроектирована для обработки потоков данных в режиме реального времени. У Storm так много функций, что рассмотреть их подробно я здесь не могу. Если хотите узнать больше о Storm, обратитесь к книге Sean Allen, Peter Pathirana, Matthew Jankowski «Storm Applied» (Manning, 2013). Общая архитектура Storm показана на рис. 4.8.

См. раздел 9.3.1 главы 9

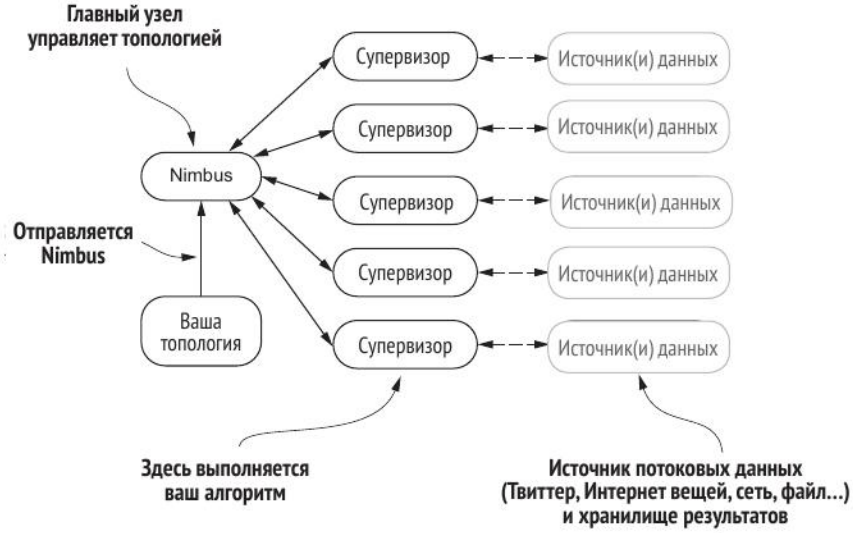


Рис. 4.8. Верхнеуровневая архитектура Storm – показаны Nimbus, супервизоры и источники данных

Как видите, на верхнем уровне Storm напоминает Spark Streaming или, быть может, кластер Hadoop, если заменить Nimbus трекером задач, а супервизоры – узлами данных. В Hadoop и в Spark термином *задача* (job) описывается единица работы; в Storm применяется термин *топология*. Разница в том, что *задача* рано или поздно заканчивается, а *топология* работает вечно. Но не будем долго задерживаться на семантике; в конечном итоге обе системы обеспечивают развертывание вашей программы на узлах исполнителей. Памятуя об этом, рассмотрим различные компоненты, изображенные на рис. 4.8.

В левой нижней части мы видим топологию, которая отправляется компоненту под названием Nimbus. Этот компонент решает, как развернуть топологию на супервизорах, назначает супервизорам задания и ведет мониторинг отказов во всей системе. В средней части рисунка показаны узлы *супервизоров*, где исполняется топология, а в правой части – *источники данных*, поставляющие данные топологии.

Apache Flink

Apache Flink – *потокоориентированная* система, в которой всё рассматривается как поток. Программа для Flink состоит из строительных блоков двух видов: потоки и преобразования. *Поток* можно рассматривать как промежуточный результат движения данных от источника к стоку. *Преобразованием* называется любая операция, которая принимает один или несколько потоков и порождает на выходе один или несколько потоков. Приложение для Flink, составленное из этих частей, интерпретируется как поток данных, который начинается с поступления данных из одного или нескольких источников, обычно содержит одно или несколько преобразований и завершается записью данных в один или несколько стоков. Приложения Flink работают в распределенной среде, состоящей из одного главного узла и нескольких исполнителей. На рис. 4.9 показана верхнеуровневая архитектура. Здесь мы лишь слегка коснулись поверхности, для более подробного ознакомления с Apache Flink горячо рекомендую книгу Sameer Wadkar, Hari Rajaram «Flink In Action» (Manning, 2016).

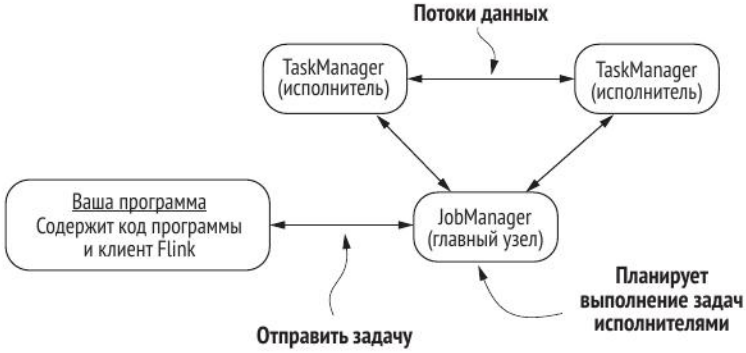


Рис. 4.9. Верхнеуровневая архитектура Apache Flink

Apache Samza

Потоковая модель в Apache Samza несколько отличается в том плане, что предоставляет поэтапный каркас обработки потока. Для этого используются две хорошо известные в мире больших данных технологии: Apache Yarn и Apache Kafka. Мы не станем тратить много времени на их рассмотрение, но скажем несколько слов о том, как они соотносятся с архитектурой Samza, изображенной на рис. 4.10.

Yarn – это диспетчер кластера, предназначенный для управления ресурсами, а также для планирования и мониторинга задач. Звучит сложно, но суть вот в чем: компонент управления ресурсами отвечает за выделение ресурсов (процессоров, памяти, дисков, сети и т. д.) всем приложениям, работающим в вычислительном кластере. Компонент планирования и мониторинга задач отвечает за выполнение задачи в кластере. На рис. 4.10

видно, что клиент Samza Yarn просит диспетчер ресурсов Yarn выделить ресурсы для выполнения приложения Samza. После согласования ресурсов на различных узлах кластера запускаются исполнители заданий Samza. Картина намеренно упрощена, поскольку подробный рассказ о специфике Yarn ничего не добавляет нашему обсуждению, к тому же по мере развития проекта материал устаревает. В центральной части рисунка показаны задания Samza. В данном случае вход и выход заданий опосредован Apache Kafka – технологией, которая четко укладывается в рамки обсуждения звена очереди сообщений (глава 3) и к которой мы еще вернемся в последующих главах. А пока можете считать, что это высокоскоростное хранилище, откуда наши потоковые задания читают данные и куда записывают результаты. Прекрасными источниками информации о Yarn могут служить книги Alex Holmes «Hadoop in Practice», второе издание (Manning, 2014) и Chuck Lam, Mark Davis, Ajit Gaddam «Hadoop in Action», второе издание (Manning, 2014). Актуальную информацию о Apache Samza см. по адресу http://samza.apache.org.

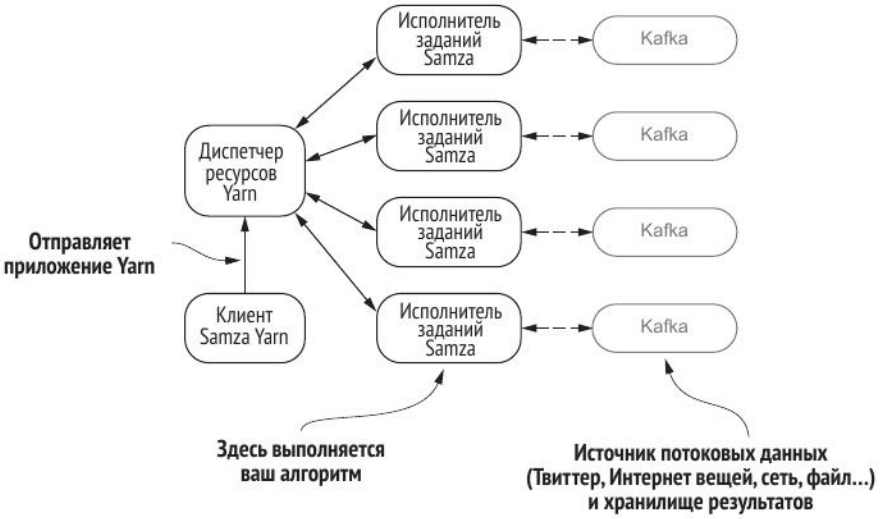


Рис. 4.10. Верхнеуровневая архитектура Apache Samza

4.3. Ключевые функции систем потоковой обработки

В звене анализа потоковой архитектуры можно использовать различные системы потоковой обработки. Мы хотим обратить особое внимание на несколько ключевых функций, которые следует принимать во внимание при сравнении таких систем и выборе наиболее подходящей для решения поставленной задачи.

4.3.1. Семантика доставки сообщений

В главе 3 мы узнали о семантике доставки сообщений в контексте звена очереди сообщений, производителей, брокеров и потребителей. На этот раз мы хотим рассмотреть семантику доставки сообщений в звене анализа. Определения остаются прежними, но следствия будут несколько иными.

Во-первых, напомним, какие гарантии дают различные семантики.

- *Не более одного раза* – сообщение может потеряться, но никогда не будет обработано дважды.
- *Не менее одного раза* – сообщение не может потеряться, но может быть обработано несколько раз.
- *Ровно один раз* – сообщение не может потеряться и обрабатывается ровно один раз.

Это чуть более общие формулировки, чем при обсуждении звена очереди сообщений. В главе 3 нас интересовало, что означает каждая семантика применительно к порождению и потреблению сообщений. А сейчас, обсуждая средства потоковой обработки, мы говорим о продолжении потребителя, которого рассматривали в звене очереди сообщений. Как эти определения проявляются в средствах потоковой обработки? Давайте наложим их на диаграмму потоков данных и посмотрим, что они означают.

На рис. 4.11 показаны семантика «не более одного раза» и две возможные ошибки: отбрасывание сообщения и отказ потокового процессора. В последнем случае сообщения еще и теряются до тех пор, пока не будет поднят подменный процессор.

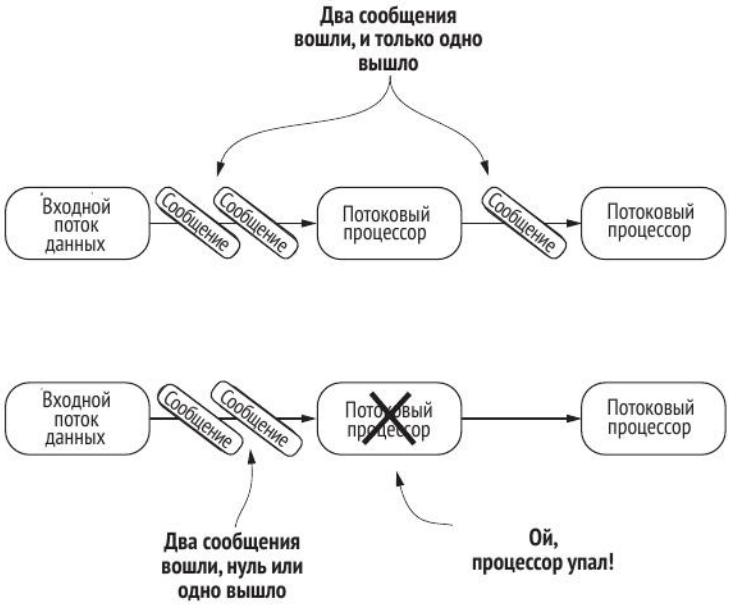


Рис. 4.11. Доставка сообщения не более одного раза и поток данных

Семантика доставки «не более одного раза» – самая простая из того, что может предложить система; нигде не требуется никакой специальной логики. Если сообщение отбрасывается, потоковый процессор «падает» или машина, на которой работает потоковый процессор, выходит из строя, то сообщение теряется.

Семантика «не менее одного раза» увеличивает сложность, поскольку потоковая система должна отслеживать каждое сообщение, отправленное потоковому процессору, и подтверждение его получения. Если потоковый диспетчер решит, что сообщение не было обработано (поскольку оно потерялось или потоковый процессор не ответил в течение заранее оговоренного времени), то сообщение отправляется снова. Имейте в виду, что при таком уровне гарантии ваша потоковая задача может получить одно и то же сообщение несколько раз. Поэтому задача должна быть *идемпотентной*, т. е. многократная обработка одного и того же сообщения должна давать один и тот же результат. Если вы не будете забывать об этом при проектировании потоковых задач, то сможете успешно обработать сообщения-дубликаты.

Семантика «ровно один раз» еще немного увеличивает сложность системы потоковой обработки. Помимо внутреннего учета всех отправленных сообщений, она должна обнаруживать и игнорировать дубликаты. При таком уровне гарантии задаче уже не нужно думать о том, что делать с сообщениями-дубликатами, – ей остается только вернуть признак успеха или ошибки после обработки сообщения. Хотя идемпотентность потоковой задачи в случае этой семантики доставки не обязательна, я настоятельно рекомендую делать все потоковые задачи идемпотентными. Тогда рассуждать о них и отлавливать ошибки будет гораздо проще.

Какая гарантия нужна в вашем случае, зависит от характера решаемой задачи. Возьмем предыдущий пример – систему мониторинга работы турбины. Напомним, что мы хотим постоянно анализировать работу турбины, чтобы можно было предсказать отказ и произвести профилактическое обслуживание. Я говорил, что турбина ежечасно порождает приблизительно 1 ТБ данных – и это только одна турбина, а нам нужно вести мониторинг тысяч таких турбин. Так ли необходимо исключить всякую возможность потери сообщений? Может быть, но имеет смысл выяснить, все ли данные нужны алгоритму прогнозирования отказов. Если он способен работать в условиях отсутствия части данных, то я бы для начала выбрал наименее сложную гарантию.

А что, если характер задачи таков, что на основе потокового запроса принимается решение о финансовой операции? Быть может, вы эксплуатируете рекламную сеть и выставляете клиентам счета в режиме реального времени? В этом случае требуется, чтобы выбранная потоковая система обеспечивала семантику «ровно один раз».

Надеюсь, вы поняли суть проблемы и сможете применить эти знания к собственной задаче. А мы сейчас перейдем к вопросу об управлении состоянием.

Управление состоянием

Если потоковый алгоритм анализа становится сложнее, чем операции с одним лишь текущим сообщением без учета зависимостей от прошлых сообщений или внешних данных, то возникает необходимость в запоминании состояния и, вполне вероятно, понадобится служба управления состоянием, предоставляемая выбранной системой. Приведем простой пример, который поможет понять суть вопроса.

Представьте, что вы отвечаете за маркетинг в крупном интернет-магазине и хотите знать, сколько страниц в час просматривает каждый посетитель. Догадываюсь, что вы подумали: «Целый час – ну, это можно решить с помощью пакетного процесса». Но пока оставим это в стороне, а подумаем, какое состояние нужно запоминать, чтобы решить поставленную задачу. На рис. 4.12 показано, как следовало бы организовать потоковые процессоры для ответа на вопрос.

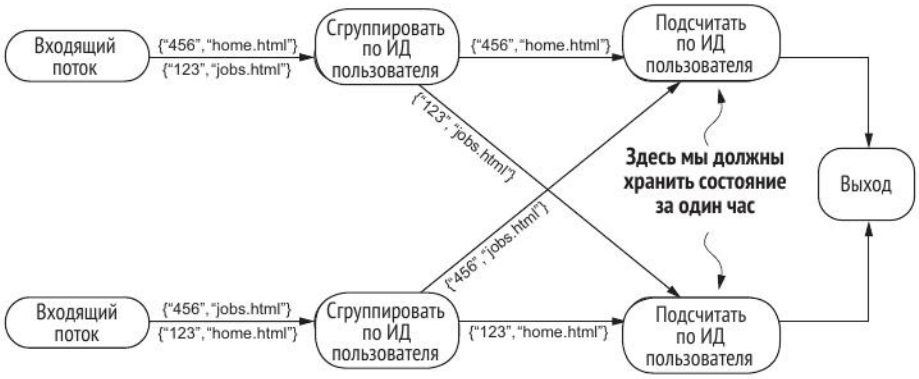


Рис. 4.12. Простой пример подсчета страниц, просмотренных пользователями в течение часа

Из рисунка должно быть понятно, где нужно хранить состояние, – в том потоковом процессоре, где ваша задача производит подсчет по идентификатору пользователя. Если выбранное средство потокового анализа не поддерживает управления состоянием сама, то один из возможных вариантов – хранить данные в памяти и сбрасывать их каждый час. Это будет работать при условии, что вы готовы смириться с полной потерей данных, если потоковый процессор или задача потерпят аварию. Ну и, конечно, получится так, что до поры до времени задача будет работать без нареканий, а в один прекрасный день начнет падать на 59-й минуте каждого часа. В зависимости от характера бизнеса риск потери данных вследствие хранения всего состояния в памяти может оказаться приемлемым. Но во

многих случаях жизнь устроена не так просто, и об управлении состоянием надо специально заботиться. Поэтому различные системы потоковой обработки предоставляют соответствующие механизмы, которыми вы можете воспользоваться.

Предлагаемые различными системами средства управления состоянием естественно располагаются по оси сложности, как показано на рис. 4.13.

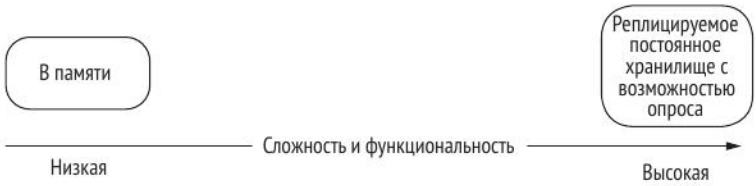


Рис. 4.13. Сложность управления состоянием в системах потоковой обработки

В левой части оси находится наивное решение с хранением состояния в памяти, похожее на то, что мы рассматривали выше, а на другом конце спектра – системы, предлагающие реплицируемое постоянное хранилище с возможностью опроса. Если вам кажется, что это два совершенно разных взгляда на управление состоянием, то так оно и есть. Простые решения в левой части оси годятся только для запоминания состояния вычислений, не страшащегося сбоев, например текущего счетчика.

Системы на другом конце спектра позволяют отвечать на гораздо более разнообразные и сложные вопросы, например соединять два потока данных. Представьте, что вы эксплуатируете рекламную систему и хотите отслеживать две вещи: показы рекламы и клики по рекламе. Понятно, что в результате получаются два потока данных: показы и клики. На рис. 4.14 показаны эти потоки, а ваша потоковая задача должна их обрабатывать.

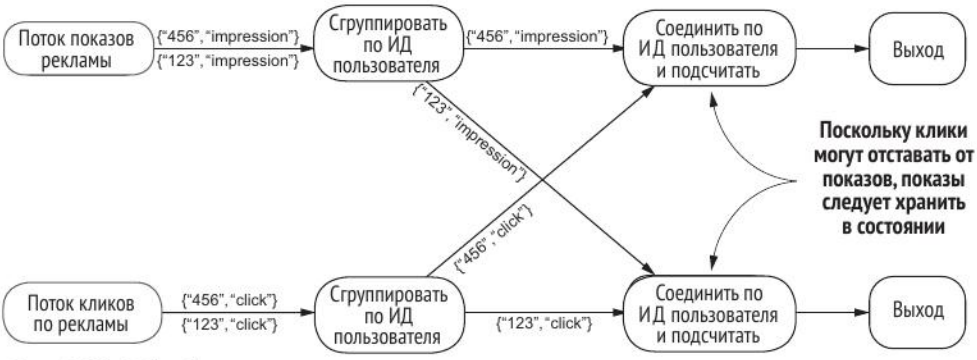


Рис. 4.14. Обработка потоков показов и кликов с использованием состояния потока

В этом примере данные о показах рекламы и кликах поступают в виде двух отдельных потоков, мы соединяем эти потоки по идентификатору пользователя, а затем подсчитываем. Клики отстают от показов, но, используя систему потоковой обработки, которая хранит состояние в реп-

лицируемом постоянном хранилище с возможностью опроса, мы можем соединить потоки и вернуть один результат. Думаю, вы согласитесь, что умение соединять потоки с помощью системных средств управления состоянием – не совсем то же самое, что сохранение текущего результата агрегирования.

Еще немного поразмыслив над этим примером, вы наверняка придумаете, как соединить более двух потоков. Мы вернемся к этой увлекательной теме в главе 5. А пока обратимся к еще одной функции, которую нужно учитывать при выборе системы потоковой обработки.

Отказоустойчивость

Как хорошо было бы жить в мире, где ничего не ломается. Но реальность жестока: вопрос не в том, сломается или нет, а в том, *когда* это произойдет. Способность системы потоковой обработки продолжать функционирование в условиях отказов – прямое следствие реализованных в ней механизмов отказоустойчивости. На рис. 4.15 показаны все точки возможных отказов.

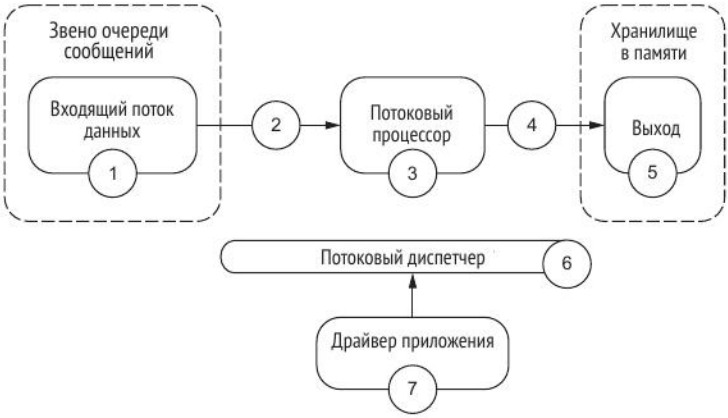


Рис. 4.15. Точки отказа потоковой обработки в контексте всей потоковой архитектуры

Таких точек семь. Рассмотрим их и разберемся, что требуется от системы потоковой обработки в плане отказоустойчивости.

1. *Входной поток данных.* Конечно, звено очереди сообщений не контролируется системой потоковой обработки, но оно может отказать, и система должна корректно обрабатывать отказ, а не «падать» из-за недоступности данных или ресурса.
2. *Сеть передачи входного потока.* Эту часть система потоковой обработки не контролирует, но должна корректно обрабатывать разрыв связи.

3. *Потоковый процессор.* Здесь выполняется ваш код, и выполнение должно контролироваться системой потоковой обработки. Если что-то пойдет не так – например, возникнет авария в вашей программе или выйдет из строя машина, на которой она работает, – то потоковый диспетчер должен предпринять действия для перезапуска процессора или переноса обработки на другую машину.
4. *Связь с местом назначения.* Потоковый диспетчер не может контролировать сеть до места назначения результатов, но должен управлять потоком данных от последнего потокового процессора, чтобы тот не захлебнулся из-за противодавления в сети и не упал в случае недоступности места назначения или сети.
5. *Место назначения.* Не контролируется потоковым диспетчером, но его отказ может повлиять на обработку потока, и это следует иметь в виду.
6. *Потоковый диспетчер.* Если он откажет, то мы останемся без координатора – потоковые процессоры продолжат работать без наблюдения со стороны диспетчера. Некому будет запустить новые потоковые процессоры и восстановить отказавшие.
7. *Драйвер приложения.* Они бывают двух видов. В первом случае драйвер не делает ничего, кроме отправки потоковой задачи диспетчеру, – это нам не интересно. Во втором случае драйвер приложения логически включает в себя потоковый диспетчер и, следовательно, подвержен тем же рискам.

Посмотрим, как можно решить эти проблемы. Во-первых, немного упростим ситуацию. Из предыдущего списка можно исключить входящий поток и доступность места назначения как не имеющие отношения к потоковой системе. Само собой разумеется, что потоковая система не должна выходить из строя в случае отказов на входе или на выходе. Поэтому будем считать, что эти аспекты не касаются отказоустойчивости. Если объединить оставшиеся элементы списка по логической близости, то получится вот что:

- *потеря данных.* Сюда относится потеря данных в сети, а также авария потокового процессора или вашей задачи, в результате чего теряются данные, находившиеся в этот момент в памяти;
- *утрата контроля над ресурсом.* Сюда относится авария потокового диспетчера и драйвера приложения, если таковой имеется.

Обсуждая отказоустойчивость в главе 3, мы говорили о различных способах предотвратить потерю данных. В системах потоковой обработки все распространенные методы обработки ошибок сводятся к той или иной форме репликации и координации. Обычно потоковый диспетчер реплицирует состояние вычисления (вашей потоковой задачи) на различных

потоковых процессорах. В случае отказа потоковый диспетчер должен координировать реплики, чтобы правильно восстановиться после ошибки. Механизмы отказоустойчивости проектируются в расчете на максимальное число одновременных отказов, поэтому часто можно услышать выражение *k-отказоустойчивая система*.

В общем случае существуют два подхода к репликации и координации в потоковой системе: конечный автомат и восстановление путем отката. Оба предполагают, что алгоритм обработки потока идемпотентный, т. е. если два правильно работающих потоковых процессора получают одинаковые входные данные в одном и том же порядке, то они породят одинаковые результаты в одном и том же порядке.

В случае подхода на основе *конечного автомата* потоковый диспетчер реплицирует задачу на нескольких независимых узлах и координирует реплики, посылая всем одни и те же входные данные в одном и том же порядке. Для этого необходимо в $k + 1$ раз больше ресурсов, чем для одной реплики, где k – число потоковых процессоров, исполняющих нашу задачу. Но зато можно быстро перехватить управление при отказе с минимальным временем простоя. Для таких приложений, как система обнаружения вторжений, когда в любой момент запаздывание должно быть мало, дополнительные затраты на ресурсы оправданы.

В случае *восстановления путем отката* потоковый процессор периодически упаковывает состояние вычисления в так называемую *контрольную точку* и копирует ее на другой узел потокового процессора или в постоянное хранилище, например на диск. Между контрольными точками процессор должен хранить состояние вычисления. Поскольку диски характеризуются большим временем задержки, их включение в процесс увеличивает время обработки потока. Поэтому не столь странно выглядит альтернативное решение: копировать сохраненное в контрольной точке состояние на другие процессорные узлы с одновременным хранением журналов в памяти. В этом случае если потоковый процессор упадет, то потоковый диспетчер должен будет восстановить состояние из последней контрольной точки и накатить журнал – тогда потоковая задача будет переведена в состояние, предшествующее аварии. По сравнению с первым подходом у этого решения ниже накладные расходы, но оно обходится дороже с точки зрения времени восстановления после отказа. Однако оно полезно в ситуациях, когда отказоустойчивость важна, а с редкими умеренными задержками можно смириться.

Изучая различные системы потоковой обработки для решения своих задач, вы увидите, что если система вообще предлагает средства отказоустойчивости, то в виде варианта одного из этих двух подходов. Если хотите познакомиться с ними более глубоко, то рекомендую следующие статьи: Elnozahy, Alvisi, Wang, Johnson «A Survey of Rollback-Recovery Protocols

in Message-Passing Systems»[1] и Schneider «Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial»[2].

4.4. Резюме

В этой главе мы рассмотрели общую архитектуру распространенных систем потоковой обработки и указали ключевые функции, на которые следует обратить внимание.

Вот что мы узнали:

- общая архитектура систем потоковой обработки;
- что означает семантика доставки сообщений в этом звене;
- что такое состояние и как им можно управлять;
- что такое отказоустойчивость и зачем она нужна.

Я понимаю, что кое-что из перечисленного может показаться расплывчатым или несколько абстрактным, но не стоит переживать по этому поводу. В главе 5 мы увидим, как выполняется анализ и (или) опрос данных, протекающих через систему потоковой обработки. Кто-то скажет, что тамто и начинается самое интересное, но чтобы эффективно задавать вопросы о данных, нужно было овладеть материалом, изложенным в этой главе.

Готовы задавать вопросы о данных? Отлично! Переверните страницу – и приступим.

[1] *ACM Computing Surveys*, 34(3):375–408 (2002), www.cs.utexas.edu/~lorenzo/papers/SurveyFinal.pdf

[2] *ACM Computing Surveys*, 22(4):299–319 (1990), www.cs.cornell.edu/fbs/publications/smsurvey.pdf.

Глава 5

Алгоритмы анализа данных

Краткое содержание главы:

- опрос потока;
- к вопросу о времени;
- методы обобщения.

В главе 4 были рассмотрены потоки данных во многих системах потоковой обработки, семантика доставки и отказоустойчивость. В этой главе мы отойдем от вопросов архитектуры и обсудим алгоритмическую сторону потоковой обработки, которая часто называется *потоковой аналитикой*, или *потоковой добычей данных*. Нас будут интересовать ответы на вопросы «что» и «почему», а иногда и детальный разбор ответа на вопрос «как». Не расстраивайтесь, не обнаружив подробных математических обоснований и кода алгоритмов, – будет дано достаточно ссылок на дополнительные ресурсы, чтобы можно было продолжить изучение.

Но сначала я хочу поговорить о том, как с помощью описываемых инструментов выполняются запросы. Вообще говоря, в потоковой системе встречаются запросы двух типов:

- *ситуативные запросы*. Это вопросы о потоке, которые задаются однократно, например: каково максимальное встретившееся в потоке значение? Такие запросы аналогичны выполняемым в реляционной СУБД;
- *непрерывные запросы*. Ответы на такие запросы вычисляются все время, пока поток существует, например: каждые пять минут вычислять наибольшее значение, встретившееся в потоке, и генерировать оповещение, если это значение превышает заданный порог.

К сожалению, среди многочисленных современных систем потоковой обработки не найдется двух, предлагающих одинаковый язык запросов, и зачастую система не предлагает SQL-подобного языка, а настаивает на

программном выражении алгоритма. В табл. 5.1 описано текущее положение дел с поддержкой языков запросов в популярных системах потоковой обработки (оно может измениться, поскольку многие из этих проектов активно развиваются и становятся более зрелыми).

Таблица 5.1. Поддержка языков запросов в системах потоковой обработки

Система	Поддержка языков запросов
Apache Storm	В версии 1.1.0 имеется поддержка SQL (http://storm.apache.org/releases/1.1.0/storm-sql.html). На момент написания книги она еще считалась экспериментальной и не готовой к использованию в производственной системе
Apache Samza	Начиная с версии 0.9 в JIRA висит предложение добавить поддержку языка запросов. На момент написания книги оно еще было открыто: https://issues.apache.org/jira/browse/SAMZA-390
Apache Flink	Табличный API поддерживает SQL-подобные выражения (http://ci.apache.org/projects/flink/flink-docs-release-0.9/libs/table.html)
Apache Spark Streaming	Поддерживается язык SparkSQL/Hive (http://spark.apache.org/docs/latest/sql-programming-guide.html)

Принимая во внимание современное положение дел в области поддержки SQL-подобных языков, я не стану останавливаться на деталях реализации, поскольку в разных продуктах они сильно различаются. Но приведу некоторые соображения по поводу реализации алгоритмов в разных системах. Однако сначала обсудим некоторые ограничения.

5.1. Ограничения и их ослабление

Из предыдущих глав мы знаем, что одна из характерных особенностей потоковой системы состоит в том, что невозможно сохранить весь поток, т. к. он не ограничен и никогда не кончается. Наша цель – непрерывно выдавать результаты запросов в онлайновом режиме. Когда данные попадают в звено анализа, результаты следует пересчитать и, возможно, предъявить. На первый взгляд кажется, что отвечать на такие запросы легко, но при проектировании алгоритмов обработки потоков следует иметь в виду ряд ограничений.

- *Однопроходность.* Помните, что данные не архивируются, и у вас имеется только один шанс их обработать. Это может оказать существенное влияние на конструкцию алгоритма. Так, многие традиционные алгоритмы добычи данных итеративны и подразумевают несколько проходов по данным. Для работы в потоковой системе такие алгоритмы следует модифицировать. Повторю – на любой элемент данных можно взглянуть только один раз.
- *Изменение концепции.* Этот феномен может оказать влияние на прогностическую модель. По мере поступления новых данных могут из-

меняться их статистические свойства. Нужно ли принимать это во внимание, зависит от вида анализа и разрабатываемой прогностической модели.

- *Ограниченность ресурсов.* Для многих потоков данных мы почти или совсем не контролируем темп поступления данных. Иногда из-за кратковременного пика скорости или объема данных располагаемых ресурсов оказывается недостаточно, и алгоритм вынужден отбрасывать кортежи, которые не успел обработать. Это называется *сбрасыванием нагрузки*. Такое ограничение присутствует в потоковых системах практически всегда, но удивительно, как мало алгоритмов учитывают его. Существуют два типа алгоритмов: рандомизированные и семантические; в последнем случае алгоритм принимает во внимание свойства потока и параметры качества обслуживания[1].
- *Ограничения предметной области.* Если остальные ограничения относятся почти ко всем потокам, то эти характерны для конкретной предметной области. Например, если в социальной сети 100 000 000 пользователей и мы хотим произвести анализ всех почтовых сообщений друг другу, то понадобилась бы память для хранения пар почтовых адресов. Требования к памяти могли бы выражаться многими петабайтами. Даже подсчет простой статистики, скажем, уникальных значений в таком потоке, – непростая задача. Может показаться, что это ограничение проходит по классу нехватки ресурсов, но на самом деле оно связано с характером данных в задаче.

Именно из-за этих ограничений практически в каждом потоковом алгоритме используется та или иная форма конспектирования данных. Существуют различные виды конспектов, и, как мы увидим, от того, какой выбран, сильно зависит характер возможных вопросов. Но прежде чем перейти собственно к анализу данных, поговорим о временны́х аспектах потоковой обработки.

5.2. К вопросу о времени

Если вам доводилось работать с системами, где данные статические, например Hadoop или РСУБД, то, наверное, вы иногда включали в запросы время. В задаче MapReduce или Spark, в запросе на языке Hive, SQL или еще каком-то можно указать во фразе *where* временной диапазон и быть уверенным, что будут возвращены все данные, загруженные в течение этого времени. С другой стороны, в потоковой системе данные текут непрерывно. Они могут приходить не в том порядке, в каком порождались, или за-

[1] Дополнительные сведения см. в работе «Load Shedding in a Data Stream Manager», опубликованной в сборнике *Proceedings of the 29th International Conference on Very Large Data Bases* (2003, стр. 309–320), http://dl.acm.org/citation.cfm?id=1315479.

паздывать – и мы никак не можем опросить все данные сразу, поскольку поток бесконечен. Но не все так страшно. Я расскажу о подходах к учету времени и о решении типичных проблем, возникающих в процессе анализа потока данных.

Время потока и время события

Время *потока* – это время, когда событие попало в потоковую систему. Время *события* – это время, когда событие произошло. Представьте, что мы собираем данные с фитнес-монитора, например Fitbit, и данные передаются в нашу потоковую систему. Тогда время потока – это время, когда события монитора достигают звена анализа, а время события – время, когда устройство зафиксировало событие. В контексте архитектуры в целом время потока – это время, когда событие поступает звену анализа. Если потоковый анализ опирается на время события, то следует помнить, что оно далеко не всегда совпадает со временем потока. Разность между ними – *временной сдвиг* – может быть велика (см. рис. 5.1).

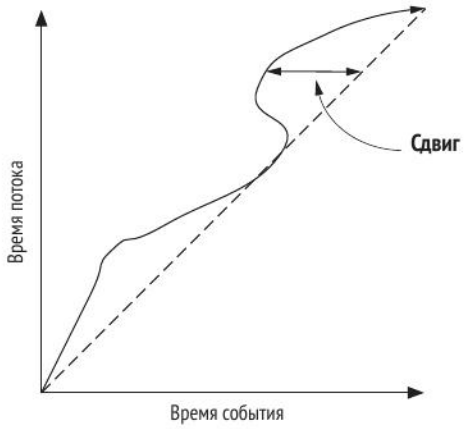


Рис. 5.1. Временной сдвиг между временем события и временем потока

И как это может повлиять на анализ данных в нашем примере? Отразится ли сдвиг на вычислении средней скорости бегущего? Ответ на эти вопросы напрямую связан со следующей темой: оконными методами в системах потоковой обработки. По ходу ее обсуждения держите в уме концепцию временного сдвига.

Временные окна

Поскольку поток бесконечен, потоковая система не может хранить его в памяти целиком, а значит, традиционные методы пакетной обработки к потокам неприменимы. Тогда как же производить вычисления? С помощью *окон данных*. На рис. 5.2 показано окно, содержащее небольшое количество данных в окрестности некоторого момента времени.

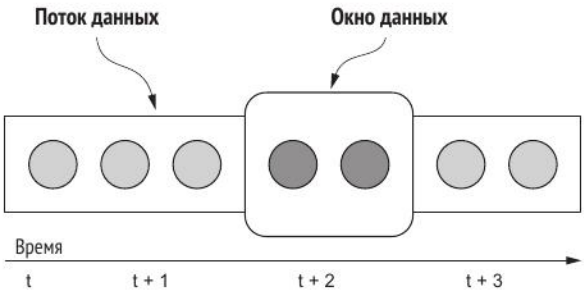


Рис. 5.2. Окно данных относительно остального потока

Как видим, окно представляет собой небольшую часть потока данных. Ситуация в действительности несколько сложнее, но не намного. Сложность проистекает из двух характеристик, присущих всем оконным методам: политики срабатывания и политики вытеснения. Политика *срабатывания* определяет правила, с помощью которых система потоковой обработки уведомляет наш код о том, что пора обработать находящиеся в окне данные. Политика *вытеснения* определяет правила, с помощью которых система решает, что элемент данных нужно удалить из окна. В основе той и другой лежит либо время, либо объем данных в окне. Различие между политиками и вопрос о том, как в игре участвует время или число элементов, прояснится в процессе обсуждения оконных методов, из которых наибольшее практическое значение имеют скользящее окно и прыгающее окно.

5.2.1. Скользящее окно

В методе *скользящего окна* политики вытеснения и срабатывания основаны на времени. Роль в них играют протяженность окна и интервал скольжения (см. рис. 5.3).

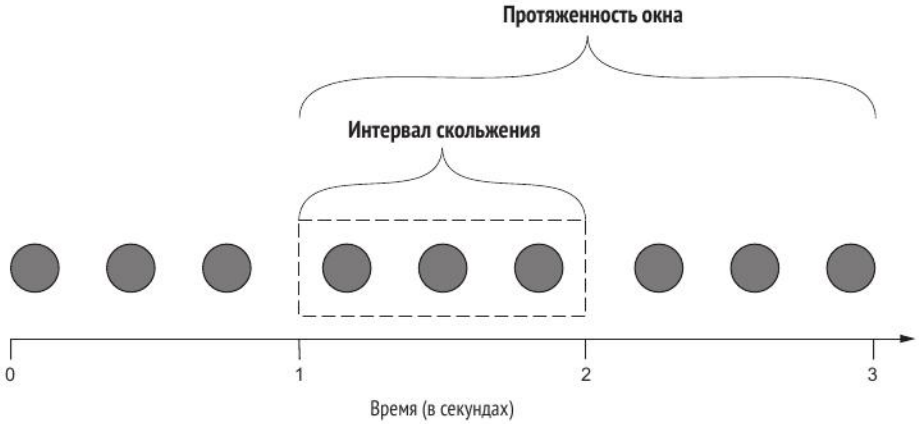


Рис. 5.3. Скользящее окно – показаны длина окна и интервал скольжения

Протяженность окна определяет политику вытеснения; это время, в течение которого данные сохраняются и доступны для обработки. На рис. 5.3 протяженность окна составляет две секунды; по мере поступления новых данных данные, хранящиеся более двух секунд, вытесняются. Интервал скольжения определяет политику срабатывания; на рис. 5.3 он равен одной секунде. Это означает, что каждую секунду вызывается наш код, так что мы сможем обработать данные в интервале скольжения или во всем окне.

Пример использования

Возвращаясь к примеру устройства Fitbit, вспомним, что данные постоянно втекают в нашу систему. Начальник отдела маркетинга попросил сделать информационную панель, на которой показывается скорость всех бегунов, усредненная по возрастным группам, например: 12–17, 18–24, 25–34 и т. д. Данные на панели должны обновляться раз в 5 секунд, а усреднение должно производится за последние 30 минут. Абстрагируемся от внешнего вида самой панели и сосредоточимся на потоковом анализе. Как решить задачу с применением метода скользящего окна?

Нам нужно окно протяженностью 30 минут с интервалом скольжения 5 секунд. Не забывайте о различии между временем потока и временем события. Будет ли ваш анализ иметь смысл, если в определении протяженности окна и интервала скольжения участвует время потока?

Поддержка в имеющихся системах

Не все современные системы потоковой обработки поддерживают скользящие окна, а уровень поддержки не одинаков. В табл. 5.2 показано, как скользящие окна поддерживаются в популярных системах.

Таблица 5.2. Поддержка скользящих окон в популярных системах потоковой обработки

Система	Скользящие окна	Время события или время потока	Комментарии
Spark Streaming	Да	Время потока	Пользовательские политики не поддерживаются
Storm	Нет	–	Встроенной поддержки скользящих окон нет, но ее можно реализовать с помощью таймеров
Flink	Да	То и другое	Пользователь может самостоятельно определить политики срабатывания и вытеснения
Samza	Нет	–	Прямой поддержки скользящих окон нет

Подробное описание поддержки окон в Spark Streaming и Flink имеется в официальной документации. Отметим, что Spark Streaming поддерживает только время потока. Если для приложения важно различие между

временем потока и временем события, то это следует учесть в алгоритмах и при выборе размера окна.

В Apache Storm и Apache Samza встроенной поддержки скользящих окон нет, но ее можно реализовать самостоятельно. Поэтому вам предстоит много дополнительной работы, а результат может оказаться менее эффективным по сравнению со встроенной поддержкой. Детали реализации такой поддержки выходят за рамки книги. Если она вам абсолютно необходима, следите за последними обновлениями, а также за обсуждением поддержки окон в JIRA и в списках рассылки. Поскольку все это проекты с открытым исходным кодом, то вы и сами можете внести в них свой вклад.

5.2.2. Прыгающие окна

Прыгающее окно устроено несколько иначе. Политика вытеснения в нем всегда основана на заполнении окна, а политика срабатывания – на количестве элементов в окне или на времени. Сначала рассмотрим прыгающие окна со срабатыванием по счетчику. Как они работают, показано на рис. 5.4.

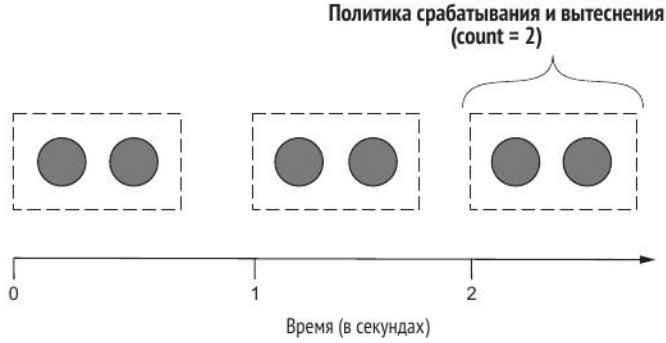


Рис. 5.4. Прыгающее окно со срабатыванием по счетчику, в котором количество элементов в политиках срабатывания и вытеснения равно 2

Здесь количество элементов в политиках срабатывания и вытеснения равно 2: как только в окне окажется два элемента, срабатывает триггер, и окно опустошается. Это поведение не зависит от времени – сколько бы времени ни потребовалось для заполнения окна, одна секунда или пять часов, политики срабатывания и вытеснения активируются, когда будет достигнуто заданное значение счетчика.

На рис. 5.5 показано прыгающее окно со срабатыванием по времени, когда интервал времени равен 2 секундам.

Здесь обе политики основаны на двухсекундном интервале времени. Не имеет значения, сколько элементов в окне: три или пять. Как только вре-

мя истекает, политики срабатывания и вытеснения активируются, и окно опустошается. Очевидно, что это отличается от описанного выше поведения скользящего окна.

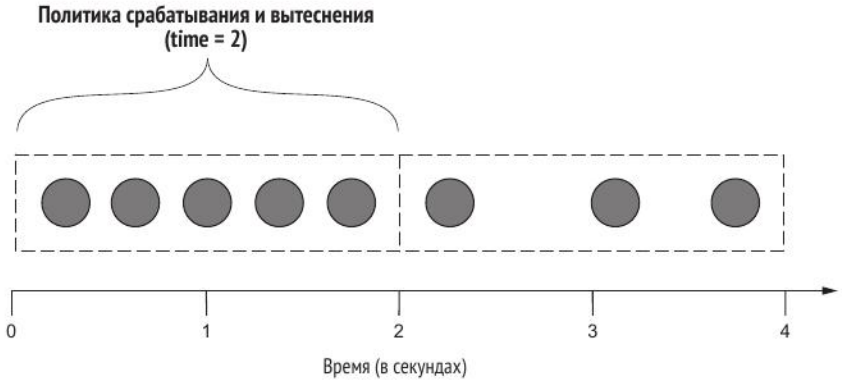


Рис. 5.5. Прыгающее окно со срабатыванием по времени, в котором время в политиках срабатывания и вытеснения равно 2 секундам

Пример использования

Допустим, что мы производим велосипеды, оснащенные различными датчиками, которые выдают данные о координатах (полученных от системы GPS), текущей скорости, текущем направлении, температуре и влажности окружающей среды. На основе этих данных мы хотим рассчитать два показателя. Во-первых, нас интересует средняя скорость всех велосипедистов, вычисляемая каждые 30 секунд на протяжении всего дня. Можно было бы усреднять с учетом местонахождения, но пока достаточно и глобального счетчика. Во-вторых, мы хотим получать уведомления всякий раз, как в городе оказывается больше 100 человек на наших велосипедах. Подумайте, как бы вы решили эти задачи с помощью прыгающих окон.

Ну, что получилось? Для вычисления первого показателя наш код должен выполняться раз в 30 секунд. Для этого я бы создал поток, содержащий только результаты измерения скорости, и настроил прыгающее окно с интервалом времени 30 секунд. В момент вызова нашего кода мы вычислим среднее по всем данным в окне за этот период. Если впоследствии мы захотим усреднять с учетом местоположения, то есть два варианта. Первый – не подвергать поток предварительной фильтрации, оставляя только данные измерения скорости, а сохранить полные сообщения от велосипеда. Тогда каждые 30 секунд мы будем иметь скорость и координаты, что позволит сегментировать данные с учетом территории.

Второй вариант – наоборот, добавить фильтрацию. В этом случае мы сделали бы так, чтобы звено сбора данных сначала распределяло данные по местоположению, а затем отправляло их другим звеньям. Тогда у нас был бы специфический поток: скорость вместе с местоположением. Это

чреватое проблемами и не очень гибкое решение. Нам пришлось бы заранее определить границы областей сегментации и придумать, как быть, если они изменятся.

Теперь рассмотрим второй показатель: все моменты времени, когда в городе находятся 100 человек на наших велосипедах. Чтобы поддержать его, нужно сделать две вещи. Во-первых, создаем поток (он может исходить от звена сбора данных, но это необязательно) для каждого нового встретившегося в данных города и настраиваем прыгающее окно размера 100 со срабатыванием по счетчику. Когда выполняется политика срабатывания, мы будем иметь все данные для каждого города, в котором число велосипедистов достигло 100.

Мы проработали два относительно простых примера. Теперь посмотрим, как прыгающие окна поддерживаются в существующих системах.

Поддержка в существующих системах

Не все представленные на рынке системы потоковой обработки поддерживают прыгающие окна, а если и поддерживают, то по-разному. В табл. 5.3 показан уровень поддержки в каждой из популярных систем.

Таблица 5.3. Поддержка прыгающих окон в популярных системах потоковой обработки

Система	По счетчику	По времени	Комментарии
Spark Streaming	Нет	Нет	Вы должны добавить поддержку самостоятельно
Storm	Да	Да	Хотя в Storm нет встроенной поддержки окон, ее легко реализовать самостоятельно
Flink	Да	Да	Имеется встроенная поддержка для прыгающих окон обоих типов
Samza	Нет	Да	Встроенной поддержки прыгающих окон со срабатыванием по счетчику нет

На момент написания книги Apache Flink была единственной системой со встроенной поддержкой прыгающих окон со срабатыванием по счетчику и по времени. В остальных системах ее нужно реализовывать самостоятельно, и сложность этой работы разнится. Как всегда бывает, функциональность, доступная на момент оценка программного обеспечения, впоследствии может измениться, поэтому если в вашей задаче прыгающие окна необходимы, проверьте возможности выбранной системы еще раз.

Мы рассмотрели два наиболее распространенных типа окон в современных системах потоковой обработки. Эта информация понадобится при обсуждении методов обобщения ниже.

5.3. Методы обобщения

В этом разделе мы рассмотрим четыре метода обобщения, лежащих в основе различных видов анализа и добычи данных. Спрашивается: зачем вообще заводить разговор об обобщении и почему приходится удовлетворяться неточными ответами на вопросы? Все дело в характере потоковой обработки. Напомним, что мы не знаем, закончится ли поток, и не можем сохранить его целиком в памяти. Поэтому получить точные ответы на вопросы о данных в потоке крайне сложно. Во многих случаях вполне хватает высокой степени уверенности в правильности или почти правильности ответа. Конечно, бывают ситуации, когда нужен точный ответ, но для его получения придется либо пожертвовать скоростью обработки, либо изменить принципы реализации. Столкнувшись с требованием предоставить точные цифры, необходимо разобраться внимательно и выяснить, не будет ли достаточно хорошего приближения.

> **Примечание.** Когда-то я работал над проектом потоковой аналитики, в котором заказчик потребовал точных цифр, потому что так всегда было в прошлом (еще до внедрения потоков). Но метод потребления данных клиентами не допускал точного вычисления показателей. И знаете, что случилось? Да ничего – в действительности картина бизнеса никак не поменялась. Мы, люди, хорошо умеем выявлять закономерности, и если данные, выдаваемые приложением потоковой обработки, хорошо представляют события, имеющие место в бизнесе, но обобщены, так что данных стало меньше, то на стадии визуализации общая картина останется такой же.

Некоторые из рассматриваемых ниже методов довольно сложны. Не торопитесь и при необходимости прочитайте текст медленно, раздел за разделом. Готовы? Тогда начнем с первого метода обобщения: случайной выборки.

5.3.1. Случайная выборка

Часто требуется произвести случайную выборку из потока. Допустим, что мы разрабатываем рекламную сеть, и наши серверы каждую минуту получают 10 миллионов запросов на показ объявлений. Это отлично, но дальше – больше, и нам понадобился статистический анализ показа объявлений в режиме реального времени. На первый взгляд, ничего сложного, но при ближайшем рассмотрении мы понимаем, что данные поступают быстро, поступление никогда не прекращается, и все данные в памяти не помещаются. Возможное решение – произвести случайную выборку из потока поступающих данных. А как произвести случайную выборку из набора данных, которого нет ни в памяти, ни на диске? И как узнать, случайна ли она?

Типичный подход к решению проблемы – *резервуарная выборка*. Идея состоит в том, чтобы хранить заранее определенное число значений из потока (резервуар) и при поступлении нового принимать вероятностное решение: поместить его в резервуар или использовать в качестве случайного примера. На рис. 5.6 показана общая схема резервуарной выборки; новые данные пропускаются через алгоритм выборки, и определяется случайный пример.

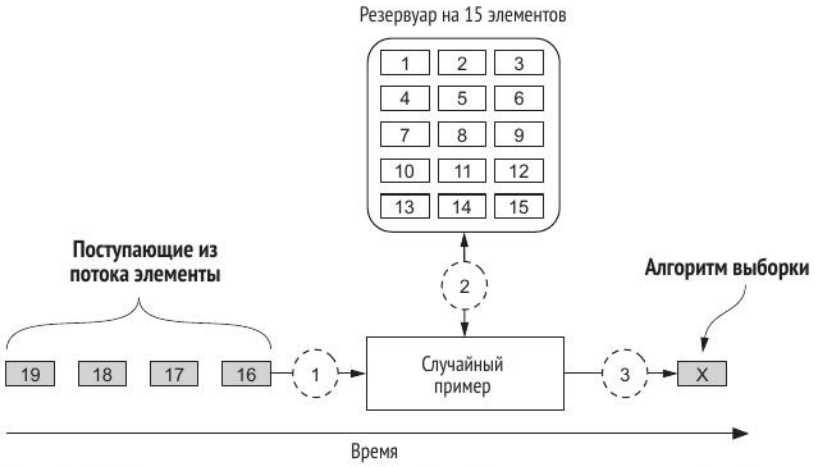


Рис. 5.6. Общая схема резервуарной выборки: предстоит обработать новый элемент данных

Разберемся, что происходит на каждом шаге. Напомним, в чем состоит наша цель: гарантировать, что после обработки 16-го элемента данных резервуар содержит случайную выборку из всех виденных ранее данных и что мы выбрали случайное значение. Независимо от того, сколько элементов потреблено из потока, у каждого элемента вероятность включения в резервуар одинакова. Имейте в виду, что на рис. 5.6 показано состояние резервуара после обработки первых 15 элементов. Мы взяли число 15, но правило общее – резервуар всегда заполняется первыми x значениями из потока, где x – размер резервуара. После того как резервуар заполнен и приложение проработало некоторое время, мы ожидаем, что в резервуаре окажется случайный набор данных, взятых из разных мест потока.

Памятуя об этом, обсудим шаги, обозначенные на рис. 5.6.

1. Когда поступает 16-й элемент, мы помещаем его в резервуар с вероятностью k/n, где k – размер резервуара, а n – номер обрабатываемого элемента данных. В данном случае вероятность помещения элемента в резервуар составит 15/16, потому что размер резервуара равен 15, и мы обрабатываем 16-й элемент.
2. Для решения о том, поместить ли в резервуар 16-й элемент, мы генерируем случайное число от 0 до 1. Если оно меньше или равно 15/16,

то помещаем элемент в резервуар, заменяя один из находящихся в нем. Если же случайное число больше 15/16, то 16-й элемент становится нашим случайным примером.

3. Если на 2-ом шаге 16-й элемент следует поместить в резервуар, то мы случайным образом выбираем элемент из резервуара и заменяем его 16-м элементом. А выбранный элемент становится случайным примером.

Мы описали алгоритм резервуарной выборки. Следующий шаг – интегрировать его в выбранную систему потокового анализа. В настоящее время этот алгоритм не встроен ни в одну из обсуждаемых систем (Spark Streaming, Storm, Samza, Flink), но реализовать его не составляет особого труда. Дополнительные сведения о резервуарной выборке см. в оригинальной статье Джеффри Виттера «Random Sampling with a Reservoir» (*Association for Computing Machinery Transactions on Mathematical Software*, 1985), доступной по адресу www.cs.umd.edu/~samir/498/vitter.pdf.

5.3.2. Подсчет уникальных элементов

Иногда требуется подсчитать количество уникальных элементов в потоке, но следует помнить, что память ограничена и сохранить в ней весь поток невозможно. В этом разделе мы продолжим рассматривать пример из раздела 5.3.1 – рекламную сеть, обслуживающую 10 миллионов показов в минуту. Мы хотим ответить на вопрос: сколько различных объявлений было показано в последнюю минуту?

В предыдущем разделе было продемонстрировано, как произвести случайную выборку из потока данных, но как поступить, если нужно подсчитывать число уникальных объявлений в минуту? Возможно, вы подумали: «Это же всего 10 миллионов элементов – сохраню их в хэш-таблице или еще какой-нибудь структуре данных, допускающей поиск, и проблема будет решена». Ну да, для нашего рекламного сервера это может сработать, но что, если мы разрабатываем систему обнаружения вторжений в сеть, которая должна работать со скоростью 40 Гб/с (~78 миллионов 64-байтовых пакетов)? В этом случае, да и всегда, когда нет возможности сохранить поток целиком, для вычисления счетчиков уникальных объектов приходится прибегать к вероятностным алгоритмам.

Для решения этой задачи существуют алгоритмы, которые можно отнести к двум общим категориям.

- *На основе битовых комбинаций.* Такие алгоритмы основаны на рассмотрении комбинаций битов, встречающихся в начале двоичного значения элемента потока. С помощью этой комбинации, а точнее, количества нулей в начале двоичного представления хэша элемента потока, определяется количество элементов. К таким алгоритмам относятся, в частности, LogLog, HyperLogLog и HyperLogLog++.

- *На основе порядковых статистик.* Алгоритмы этого класса основаны на порядковых статистиках, например на наименьших значениях, встретившихся в потоке. Примерами могут служить алгоритмы MinCount и Бар-Йоссефа.

На практике сейчас чаще используются алгоритмы на основе битовых комбинаций, их мы и будем рассматривать далее[2]. Наиболее популярными представителями этого класса являются HyperLogLog и HyperLogLog++. Концептуально они одинаковы, поэтому я объединю их под названием HyperLogLog. На рис. 5.7 показан порядок обработки нового элемента в алгоритме HyperLogLog. Рассмотрим его сверху вниз.

Рис. 5.7. Обработка одного элемента потока в алгоритме HyperLogLog

- На шаге 1 мы видим идентификатор просмотренного рекламного объявления. В данном случае я использовал UUID, но это совершенно необязательно – можно использовать любые идентификаторы.
- На шаге 2 строка, показанная на шаге 1, передается функции хэширования, которая порождает хэш-код (шаг 3).

[2] Желающим узнать больше об алгоритмах на основе порядковых статистик рекомендуем начать со статей Ziv Bar-Yossef «Counting Distinct Elements in a Data Stream» (*Randomization and Approximation Techniques*, 2002) по адресу https://link.springer.com/chapter/10.1007/3-540-45726-7_1 и Frederic Giroire «Order Statistics and Estimating Cardinalities of Massive Data Sets» (*International Conference on Analysis of Algorithms*, 2005) по адресу www.emis.ams.org/journals/DMTCS/pdfpapers/dmAD0115.pdf.

- На шаге 4 начинается самое интересное. Мы определяем, в какой регистр, или накопитель, попадает вычисленный хэш-код и как нужно обновить находящееся там значение. Для вычисления номера регистра используются шесть младших битов хэш-кода. Количество битов называется *точностью*, число шесть выбрано мной произвольно. Если будете использовать этот алгоритм в своих программах, убедитесь, что понимаете, в чем смысл точности. Двоичное число 100010 равно 34. Следовательно, мы будем обновлять значений в регистре с индексом 34.
- Зная индекс обновляемого регистра, посчитаем количество нулей в оставшейся части битовой строки и прибавим к нему 1. В данном случае начальных нулей нет, поэтому получается число 0 + 1, и мы записываем в 34-й регистр значение 1.
- Теперь для определения приближенного числа уникальных элементов нужно вычислить среднее гармоническое всех значение в регистрах.

Применяя этот алгоритм, следует помнить, что для оценки количества элементов в потоке подсчитываются начальные нули в битовой строке. Затем для повышения точности и устранения смещения производится усреднение по большому числу оценок; среднее гармоническое уменьшает влияние выбросов. Этот алгоритм и все его усовершенствования восходят к оригинальной работе Philippe Flajolet, G. Nigel Martin «Probabilistic Counting Algorithms» (*Journal of Computer and Systems Science*, 1985) и более поздней работе Durand, Flajolet «LogLog Counting of Large Cardinalities» (*Annual European Symposium on Algorithms*, 2003).

Алгоритм HyperLogLog++ содержит несколько усовершенствований HyperLogLog в части уменьшения объема памяти и повышения точности при вычислении количества элементов. Я описал их работу на концептуальном уровне, чтобы вы знали, как их интерпретировать и использовать[3]. На практике реализовать этот алгоритм не сложно, и возможно, даже вы сможете найти его реализацию на языке, которым пользуетесь.

Говоря об алгоритме HyperLogLog, следует еще упомянуть, что он потребляет мало памяти и допускает распределенную реализацию. Согласно авторам вышеупомянутых работ, для подсчета уникальных значений в наборе данных, содержащем миллиард элементов, с точностью 2% достаточно 1,5 КБ памяти – впечатляющий результат. Что касается распределенных вычислений, так можно легко вычислить объединение двух структур Ну-

[3] Тем, кто хочет разобраться во внутреннем устройстве этих алгоритмов, рекомендую работы Flajolet, Fusy, Gandouet, Meunier «HyperLogLog: The Analysis of a Near-optimal Cardinality Estimation Algorithm» (*Conference on Analysis of Algorithms*, 2007) по адресу http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf и Huele, Nunkesser, Hall «HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm» (*Proceedings of the EDBT*, 2013 Conference) по адресу https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf.

perLogLog. При выполнении потокового анализа это дает возможность произвести обобщение на каждом узле анализа, а затем объединить результаты для получения суммарного счетчика уникальных элементов.

Этот алгоритм интегрируется с любой из рассматриваемых систем потоковой обработки. Таким образом, мы теперь умеем приближенно вычислять счетчики уникальных элементов в потоке. В следующем разделе мы рассмотрим алгоритм, позволяющий ответить на чуть более сложный вопрос.

5.3.3. Частота

В предыдущем разделе мы обсуждали, как вычислить количество уникальных элементов в потоке. А сейчас попробуем ответить на вопрос сколько раз в потоке встретился элемент X?

Самый популярный алгоритм для ответа на вопросы такого типа называется Count-Min Sketch[4]. Его можно использовать всякий раз, как требуется вычислить сводные сведения о потоке, выражаемые в виде счетчиков. Общий алгоритм Count-Min Sketch предназначен для получения приблизительных ответов на вопросы следующего вида:

См. раздел 9.3.2 и листинг 9.25 в главе 9

- *точечные запросы* – интересуют сведения об одном элементе потока;
- *запросы по диапазону* – интересуют частоты в заданном диапазоне;
- *запросы по внутреннему произведению* – интересует размер соединения двух эскизов. В примере рекламной системы это можно использовать для ответа на вопрос, какие товары были просмотрены после показа объявления.

Эти вопросы играют фундаментальную роль во многих потоковых приложениях. В случае рекламной системы можно спросить, как часто показывалось объявление X. Аналогичные вопросы важны в системе мониторинга и анализа сетевого трафика, которая должна обрабатывать миллионы пакетов в секунду и по возможности предотвращать злонамеренные действия типа DOS-атак[5]. Уверен, что вы найдете много других применений алгоритма Count-Min Sketch, а пока посмотрим, как он работает.

Алгоритм Count-Min Sketch, как следует из самого названия, проектировался сначала для подсчета, а затем для вычисления минимума.

---

[4] Graham Cormode и S. Muthu Muthukrishnan впервый опубликовали этот алгоритм в *Journal of Algorithm* (2004) в статье под названием «An Improved Data Stream Summary: The Count-Min Sketch and Its Applications», с которой можно познакомиться по адресу http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf.

[5] Чтобы составить представление о том, как алгоритмы такого типа используются в системах мониторинга и анализа сетевого трафика, познакомьтесь с работой Cormode, Muthukrishnan «What's New: Finding Significant Differences in Network Data Streams» (INFOCOM, 2004) по адресу http://infocom2004.ieee-infocom.org/Papers/33_1.PDF.

Прежде чем рисовать диаграммы, дадим несколько определений. В Count-Min Sketch используется набор числовых массивов, называемых *счетчиками*, число таких массивов определяется шириной *w*, а размер каждого – длиной *m*. Массивы индексируются с 0, т. е. индексы изменяются в диапазоне {0...*m* – 1}. С каждым счетчиком ассоциируется своя функция хэширования, причем эти функции должны быть попарно независимы, иначе алгоритм не будет работать, как задумано. Схематически всё это изображено на рис. 5.8.

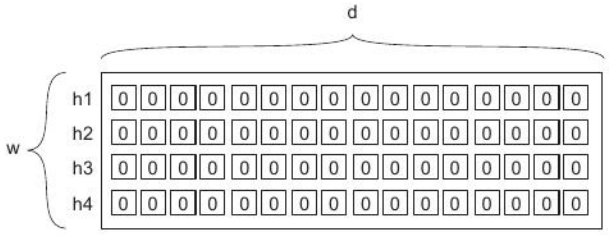


Рис. 5.8. Организация данных в алгоритме Count-Min Sketch

На рис. 5.8 мы видим двумерный массив, все элементы которого инициализированы нулями и с каждой строкой ассоциирована отдельная функция хэширования. Использование различных функций хэширования повышает точность обобщения и одновременно снижает вероятность получения плохой оценки, поскольку уменьшается вероятность коллизий хэширования. В нашем примере рекламной сети эскиз (sketch) представляет вероятностную оценку количества показов объявления. Если бы эскиз выглядел, как показано на рис. 5.8, то мы имели бы двумерный массив 4×16. Все его строки независимы и представляют собой массивы, которые мы будем использовать для хранения счетчиков.

Теперь рассмотрим процесс обновления эскиза при получении из потока данных о показе объявления (рис. 5.9).

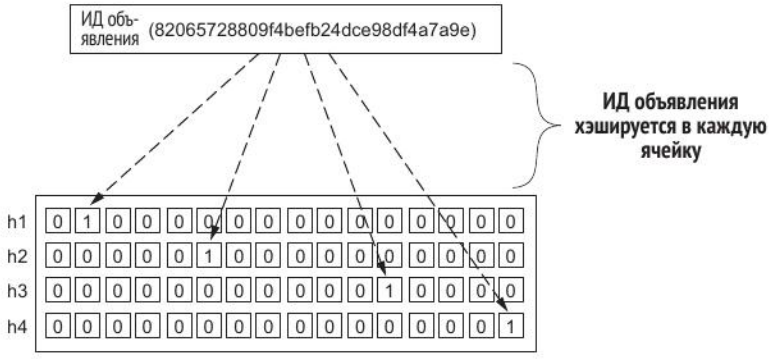


Рис. 5.9. Процесс обновления в алгоритме Count-Min Sketch

На рис. 5.9 показан идентификатор объявления, который мы хотим добавить в эскиз. Первым делом к этому значению применяется каждая

функция хэширования, в результате чего вычисляется индекс ячейки в соответствующей строке. Затем мы увеличиваем на 1 счетчик в этой ячейке. В нашем примере все значения первоначально равны 0, поэтому после этой операции в некоторые ячейки будет записано значение 1. По мере получения новых данных из потока процесс обновления эскиза повторяется. По прошествии времени мы хотим оценить, сколько раз показывалось объявление с указанным на рис. 5.9 идентификатором. Частота оценивается по такой формуле:

ESTIMATED COUNT =
min(h1(82065728809f4befb24dce98df4a7a9e),h2(82065728809f4befb24dce98df4a7a9e),
h3(82065728809f4befb24dce98df4a7a9e),h4(82065728809f4befb24dce98df4a7a9e)).

Здесь мы хэшируем идентификатор объявления и получаем индексы четырех ячеек. Затем вычисляется минимум значений, хранящихся в этих ячейках. Это и есть приближенная оценка количества показов объявления. Отметим, что этот алгоритм может давать завышенную оценку, но никогда не дает заниженной. Насколько точна оценка? Авторы оригинальной статьи показали, что при ширине 8 и длине 128 (двумерный массив 8×128) относительная ошибка составляет приблизительно 1,5%, а вероятность того, что относительная ошибка не больше 1,5%, равна 99,6%.

Лично меня поражает, что это можно сделать при таком небольшом объеме памяти и вычислений. Этот довольно прямолинейный алгоритм можно использовать для ответа на многочисленные вопросы. Если вы хотите познакомиться с его обоснованием, прочитайте удостоенную награды статью Cormode, Muthukrishnan «An Improved Data Stream Summary: The Count-Min Sketch and Its Applications» (*Journal of Algorithm*, 2004)[6].

Далее мы поговорим об эскизе, тесно связанном с алгоритмом Count-Min Sketch, но отвечающем на вопрос, встречался ли некоторый элемент потока раньше.

5.3.4. Вопрос о вхождении

Вопрос ставится так: «Встречался ли данный элемент в потоке ранее?»

На первый взгляд, задача трудновыполнима. Мы уже не раз отмечали, что сохранить весь поток в памяти невозможно, – нельзя даже сохранить идентификаторы всех встречавшихся ранее элементов. Ну и как же выкручиваться из этого положения? А просто. Мы построим структуру данных, к которой будем обращаться, чтобы ответить на вопрос о вхождении в множество; *фильтр Блума*. Как и прочие алгоритмы, рассматриваемые в этой главе, фильтр Блума дает вероятностный ответ и, естественно, допускает конфигурирование.

[6] Статья опубликована по адресу http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf.

Специфическая особенность фильтра Блума состоит в том, что он может давать ложноположительный ответ, но никогда не дает ложноотрицательных. Что это означает? Если фильтр говорит, что данный элемент в потоке не встречался, то это наверняка так и есть. Если же фильтр сообщает, что элемент встречался, то бабушка надвое сказала. В литературе описаны различные улучшенные варианты фильтра Блума, но для нашего обсуждения хватит и самого простого. Поняв, как он работает, вы сможете перейти к более сложным.

Фильтр Блума содержит битовый массив длины m, с которым ассоциирован набор независимых функций хэширования. Знакомо звучит? Напомним, что в алгоритме Count-Min Sketch тоже используется w массивов длины m. Для перехода от одного алгоритма к другому много усилий не потребуется. Как и в Count-Min Sketch, индексы элементов массива изменяются от 0 до $(m - 1)$, а поскольку массив битовый, то для его хранения необходимо $m/8$ байтов.

На рис. 5.10 показан принцип работы алгоритма.

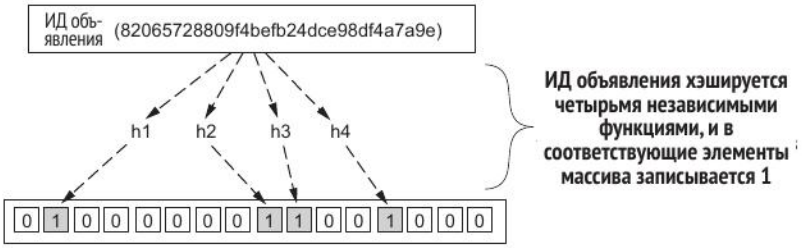


Рис. 5.10. Применение фильтра Блума для обработки одного элемента

Как видите, все очень просто. Вы, конечно, понимаете, что при обработке данных неизбежны коллизии, этим и объясняется возможность ложноположительного ответа. Если такое происходит, то уже поднятый бит в массиве остается поднятым. Применение фильтра Блума для ответа на вопрос о вхождении во множество сводится к вычислению по формуле:

$$Элемент\ Z\ принадлежит\ потоку = AND(h1(Z),h2(z),h3(z),h4(z)).$$

То есть мы вычисляем каждый хэш-код и проверяем, все ли элементы в соответствующих позициях массива равны 1. Если хотя бы один равен 0, то элемент заведомо раньше не встречался. Для углубленного знакомства с фильтрами Блума рекомендуем оригинальную статью Бэртона Блума «Space/Time Trade-offs in Hash Coding with Allowable Errors» (*Communications of the ACM*, 1970). Позже было опубликовано много статей, в которых обсуждаются улучшенные варианты фильтра Блума[7].

[7] Статью Блума можно найти по адресу http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.2080.

Эту структуру данных можно использовать, чтобы узнать, встречался ли элемент потока, еще до выполнения дорогостоящих вычислений, возможно, связанных с обращением к внешнему хранилищу. Допустим, вы разрабатываете приложение для мониторинга сети, которое отслеживает известные боты и (или) вредоносные хосты. Наблюдая за трафиком, вы можете опрашивать фильтр Блума и, *только* если он отвечает, что пакет пришел с вредоносного хоста, выполнять более дорогую операцию для подтверждения того, что пакет действительно следует отбросить. Даже если вы не занимаетесь созданием таких приложений, общая идея должна быть понятна. Неудивительно, что эта структура данных называется *фильтром*, ведь фильтрация – ее самое распространенное применение.

5.4. Резюме

В этой главе мы отвлеклись от архитектуры и обсудили проблемы опроса потока, вопрос о времени и четыре популярных метода обобщения. Были рассмотрены следующие темы:

- различные типы запросов;
- как интерпретировать время в потоковой системе;
- четыре популярных метода обобщения потоков, лежащих в основе многих программ потокового анализа.

Да, материал не самый простой. Но пусть это вас не беспокоит. Когда вы начнете создавать потоковые системы, многое станет на свои места. Возможно, вы захотите применить к своим задачам один из описанных методов обобщения. Архитектура – это, конечно, важно и полезно, но самое интересное начинается, когда вы на практике применяете то, о чем узнали в этой главе. Надеюсь, что вы уже готовы задавать вопросы о данных, с которыми работаете.

В следующей главе мы поговорим о том, как сохранить результаты анализа. Пора налить еще чашечку кофе.

Глава 6

Сохранение результатов сбора или анализа данных

Краткое содержание главы:

- зачем нужно сохранять данные;
- что именно сохранять;
- выбор решения для сохранения.

До сих пор мы обсуждали архитектуру и алгоритмы, применяемые в приложениях для потоковой обработки данных. Тема этой главы – что делать с обработанными данными. Нас в меньшей степени будет интересовать производительность постоянного хранилища и в большей – вопрос о том, как выбрать постоянное хранилище, если оно необходимо.

Сначала вспомним, какое место общей архитектуры мы сейчас обсуждаем. На рис. 6.1 показана уже знакомая архитектура и выделена тема данной главы.

Мы оценим различные хранилища с точки зрения потоковой обработки, отметим ключевые характеристики популярных продуктов и обсудим, на что обращать особое внимание. Для начала перечислим четыре варианта действий по завершении анализа, когда данные готовы к потреблению:

- проанализировать и отбросить данные;
- проанализировать данные и переправить их потоковой платформе;
- проанализировать данные и сохранить их для использования в режиме реального времени;
- проанализировать данные и сохранить их для доступа их пакетных программ.

Рассмотрим все их по порядку. Отбрасывание данных – вполне реалистичный сценарий. Например, можно проанализировать данные и отбросить их, если они не удовлетворяют некоему критерию. Наверное, стоит

напомнить, что если вы обрабатываете данные потоком и не имеете резервной копии, скажем из источника или вследствие обработки традиционным потоковым процессом, то отброшенные данные пропадают «с концами». Реализация этого варианта и его последствия вполне очевидны.

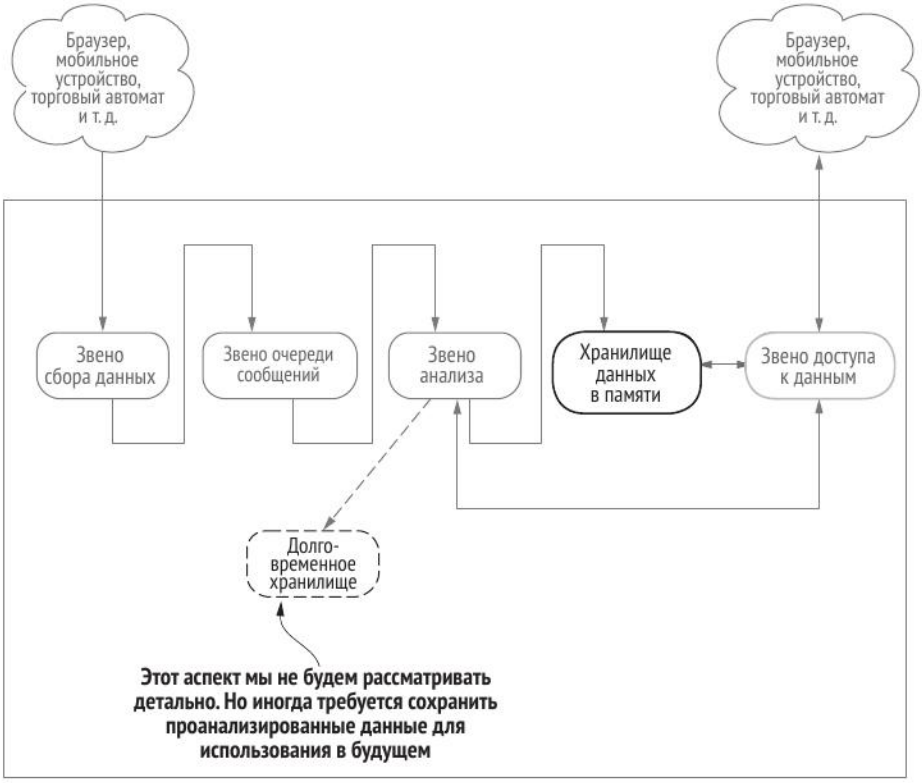


Рис. 6.1. Архитектура потоковой обработки данных – выделена тема этой главы

Еще один интересный вариант – переправка данных потоковой платформе, когда выход потокового анализа становится входом для другой системы. Такой способ часто встречается при построении цепочек нескольких технологических процессов потоковой обработки. На самом деле он положен в основу системы Apache Samza.

Осталось еще два варианта: анализ и сохранение данных для доступа в режиме реального времени или из пакетного (офлайнового) процесса. Зачем в книге, посвященной потоковым системам, тратить время на обсуждение сохранения данных для пакетного доступа? Хороший вопрос. Мы не станем подробно рассматривать выбор такого хранилища, на эту тему есть много книг и ресурсов в сети. Но мы обсудим их с точки зрения потока и соблюдения тонкого баланса между системами этих двух видов. А затем в оставшейся части главы займемся технологиями и идеями, о которых нужно знать при выборе хранилища в памяти.

6.1. Когда нужно долговременное хранилище

Бывает так, что данные, обрабатываемые в потоковой системе, нужно сохранить в хранилище, которое проектировалось для непотокового применения, например для традиционного пакетного или офлайнового доступа. С этой точки зрения, Amazon S3, HDFS, HBase и многие традиционные РСУБД можно рассматривать как непотоковые хранилища. Если возникает необходимость записывать них, то в нашем распоряжении три варианта, показанных на рис. 6.2.

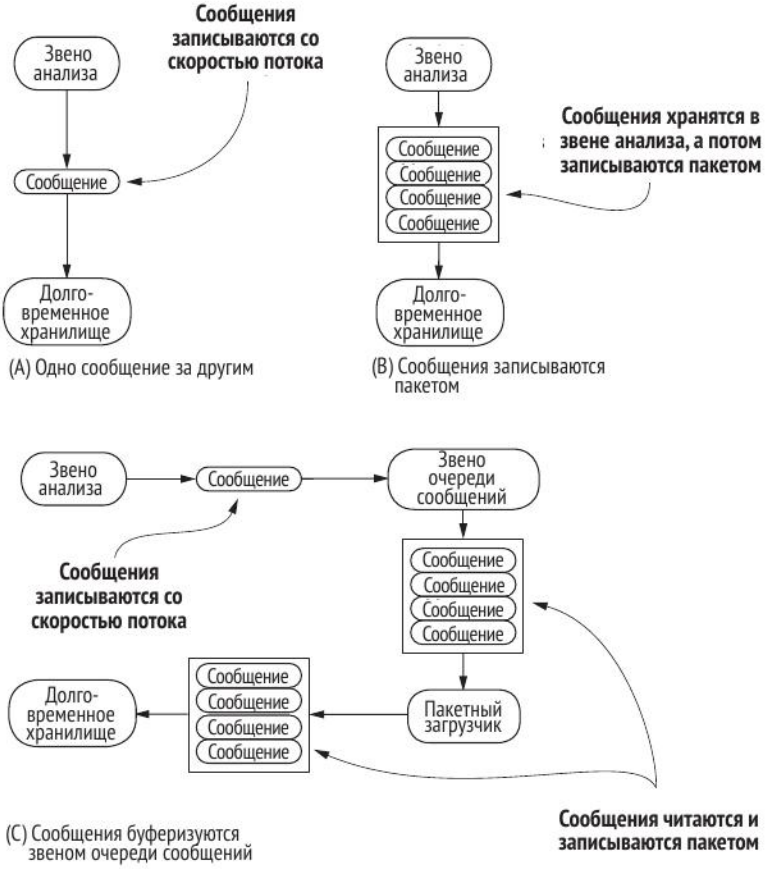


Рис. 6.2. Три способа записи данных в долговременное хранилище

Как мы увидим в последующих разделах, некоторые современные технологии и методы хранения в памяти позволяют записывать данные в долговременное хранилище иначе, чем показано на рис. 6.2.

Но пока сосредоточимся на тех возможностях, которые предлагаются для случаев, когда нам интересно записывать данные только в долговременное хранилище, а не в хранилище в памяти.

Прямая запись

Начнем с вариантов *A* и *B* на рис. 6.2. Я считаю тот и другой формой прямой записи, хотя способы, которыми звено анализа осуществляет запись в долговременное хранилище, несколько различаются. Вариант *A* – самый прямой способ поместить данные на долговременное хранение. В этом случае каждое сообщение записывается сразу после обработки – по существу, со скоростью потока. В варианте *B* сообщения накапливаются в звене анализа, а когда будет достигнут определенный порог или истечет определенное время, производится запись буфера в долговременное хранилище.

Вариантам *A* и *B* свойственны потенциальные проблемы и риски. Если хранилище неспособно справиться с темпом обработки данных – скоростью потока, то мы вообще не сможем обработать потоковые данные. При обработке данных со скоростью потока настроить пакетно-ориентированную систему хранения по мере увеличения объема и скорости поступления данных становится все труднее. И в этом нет ничего удивительного, поскольку такие системы создавались в расчете на то, что данные сначала будут загружены в пакетном режиме, а затем обработаны. Хотя вариант *B* предусматривает буферизацию сообщений в памяти перед записью, на самом деле это попытка оптимизации в ситуации медленной записи (медленной относительно скорости потока).

Непрямая запись

Вариант *C* на рис. 6.2 демонстрирует совершенно другой подход, который я называю *непрямой* записью. В этом случае мы разрываем связь между хранением данных, обрабатываемых потоком, и долговременным хранилищем. Мы записываем данные в промежуточное звено очереди сообщений. Это позволяет оградить звено анализа от проблем, связанных с низкой производительностью долговременного хранилища. Сложность немного увеличилась, но это перевешивается выгодами от переноса хранения данных в более подходящее место. Усложнение принимает вид дополнительного компонента, который на рис. 6.2 назван *пакетным загрузчиком*. Назначение пакетного загрузчика – читать пакеты сообщений из звена очереди сообщений и записывать их в долговременное хранилище.

У такого подхода два преимущества. Во-первых, как мы уже знаем, звено очереди сообщений спроектировано с учетом скоростных и объемных характеристик потока, так что можно не опасаться недостаточной скорости записи. Во-вторых, мы можем использовать компонент, предназначенный для массовой загрузки данных, позволив звену анализа сосредоточиться только на анализе потока. Мы не станем подробно рассматривать средства массовой загрузки, но важно понимать, какие у них есть возможности. От инструментов извлечения, преобразования и загрузки (ETL) обычно ожидают следующих функциональных возможностей:

- планирование задач и заданий;
- обработка ошибок;
- проверка качества данных;
- публикация данных;
- мониторинг и вычисление показателей;
- горизонтальное масштабирование;
- отказоустойчивость;
- расширяемость;
- строгая согласованность.

Хотя, глядя на этот список, легко рассуждать о том, как пригодилось бы кое-что из него для потоковой обработки, следует помнить, что зачастую нам нужно писать в долговременное хранилище типа HDFS или S3, а ни то, ни другое не приспособлены для непрерывной крупномасштабной записи, поскольку проектировались для записи данных хоть и большого объема, но в пакетном режиме. С архитектурной точки зрения, долговременное хранилище также лучше, потому что обеспечивает разделение обязанностей, когда каждый компонент отвечает за одну вещь.

Из инструментов для такого рода пакетной загрузки наиболее известны Goblin и Secor. Проект Goblin компании LinkedIn (http://gobblin.readthedocs.io/en/latest/) представляет собой универсальный каркас внесения и обработки данных, созданный LinkedIn в результате многих лет экспериментов с подобными системами. Популярен также проект Secor компании Pinterest (https://github.com/pinterest/secor). Оба предоставляют интересные возможности для ввода данных из Kafka или другого источника данных и публикации в HDFS или S3. Если в вашем проекте нужна пакетная загрузка данных в долговременное хранилище, рекомендую присмотреться к этим проектам. Внимательно изучите предлагаемый функционал и убедитесь, что он отвечает вашим требованиям.

Итак, мы обсудили два основных способа поместить результаты потокового анализа в долговременное хранилище. Теперь познакомимся с хранилищами в памяти и посмотрим, как сделать данные доступными для использования в реальном времени. Налейте себе еще кофейку и возвращайтесь.

6.2. Хранение данных в памяти

При создании системы потокового анализа наша цель – обрабатывать данные в режиме реального времени, по мере поступления. Допустим, что мы работаем в энергетической компании, развернувшей интеллектуальные измерительные приборы по всему миру. Каждый прибор передает данные раз в 30 секунд, и к одному трансформатору подключено много приборов. Данные одного прибора легко проанализировать, но что, если

бы мы могли собрать данные со всех приборов, подключенных к трансформатору, и наблюдать за развитием тенденций? Быть может, взяв окно протяженностью 30 минут, мы увидели бы признаки надвигающегося отказа трансформатора и могли бы немедленно выключить его, до того как отказ приведет к выходу других компонентов из строя. Для этого текущие данные и данные за прошлые периоды должны быть доступны в реальном времени – их нужно хранить в памяти.

Не так давно хранение большого объема данных в памяти рассматривалось скорее как средство кэширования, а не аналитики. Но времена изменились, и неизбежное произошло. В 2006 году покойный Джим Грэй выступил с докладом «Лента умерла, диск стал лентой, флэш-память стала диском, а правит бал локальность в RAM» (http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt). А недавно аналитики из компании Gartner перефразировали его: «RAM стала новым диском, а диск – новой лентой». Магнитные ленты пока еще существуют, но их исчезновение – вопрос времени. Сегодня можно приобрести серверы с 32 и даже 64 терабайтами памяти. Приблизительно за 5000 долларов в месяц можно арендовать в облаке Amazon EC2 виртуальный сервер с DRAM-памятью объемом 1 ТБ. Идея хранить весь рабочий набор данных в памяти, как бы к ней ни относиться, ныне стала реальностью, и при построении потоковой системы ее нужно рассматривать со всей серьезностью.

Вы, наверное, думаете: «У меня же есть SSD-диски – разве они недостаточно быстрые?» Так-то оно так, но одна операция поиска на SSD-диске в настоящее время занимает примерно 100 000 наносекунд (нс); а обращение к оперативной памяти – всего 100 нс, причем если данные находятся в кэше L1, то время доступа падает до 0,5 нс. Как ни крути, доступ к DRAM все еще значительно быстрее, чем даже к SSD-дискам. Может показаться, что 100 000 нс – это достаточно быстро, и в случае однократного доступа так оно и есть. Но при обработке непрерывного потока данных эти крохотные интервалы складываются, и получается заметная величина. Неудивительно, что с тех пор как появилась экономически оправданная возможность хранить весь набор данных в памяти, все производители стали наперебой предлагать решения для хранилищ в памяти – каждое лучше, чем у конкурентов. Спрашивается: как выбрать подходящее для своей задачи?

Рассмотрим, как можно классифицировать эти технологии и какие продукты имеются в каждой категории. А затем на примерах покажем, как сделать выбор.

6.2.1. Встраиваемые хранилища в памяти с оптимизацией для флэш-памяти

В эту категорию попадают продукты, предназначенные для встраивания в программу. Поэтому они рассчитаны на один узел, не предоставляют

См. раздел 9.4 главы 9

средств для доступа к данным в разных узлах и инструментов управления, которые обычно имеются в невстраиваемых системах.

Эти продукты не подходят для построения распределенных потоковых систем, о которых мы ведем речь. Напомним, что мы хотим хранить проанализированные данные, чтобы они были доступны клиентам в режиме реального времени. Используя встраиваемую базу данных, мы должны будем сами найти способ получать доступ к узлам анализа. См. рис. 6.3.

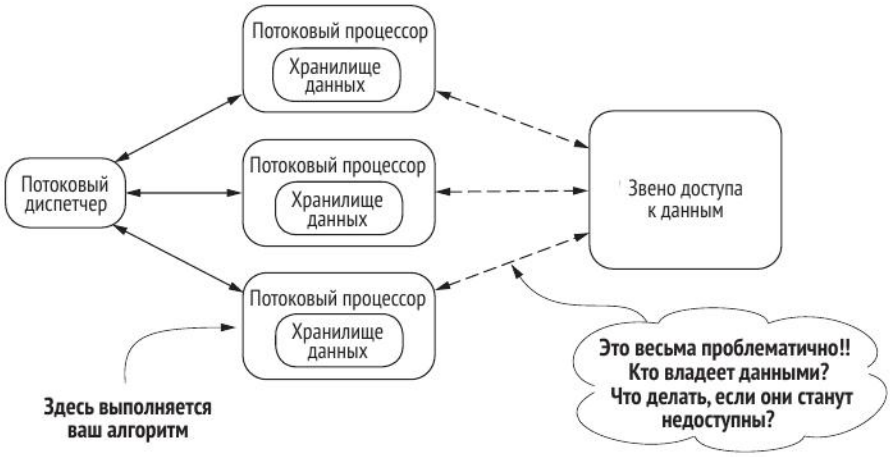


Рис. 6.3. Использование встраиваемого хранилища данных совместно с узлами анализа с точки зрения архитектуры

На рис. 6.3 показано, что встраиваемое хранилище данных размещено на одном узле с потоковым процессором. В этом случае нам предстоит решить несколько проблем, две из которых отражены на рисунке. Не нужно долго думать, чтобы понять, насколько этот дизайн далек от идеала; он хрупкий, подвержен ошибкам и расходится с рекомендуемыми паттернами правильного архитектурного проектирования ПО. На узлы потоковых процессоров возложено несколько обязанностей: потоковая обработка, локальное хранилище данных и обслуживание запросов со стороны звена доступа к данным. Поэтому я не стану тратить много времени на обсуждение этого подхода, хотя и понимаю, что в вашей потоковой системе может найтись место и для такой архитектуры. Ниже перечислены продукты этого класса, которые имеет смысл рассматривать.

- *SQLite* (www.sqlite.org) – бессерверная встраиваемая база данных, предназначенная в качестве локального хранилища для приложений и устройств. Поддерживает большую часть стандартной функциональности языка SQL; перечень того, что не поддерживается, приведен на странице www.sqlite.org/omitted.html.

- *RocksDB* (www.rocksdb.org) – встраиваемое хранилище ключей и значений, спроектированное как быстрое хранилище на базе LevelDB. Можно использовать в качестве основы для построения традиционного клиент-серверного решения.
- *LevelDB* (https://github.com/google/leveldb) – предшественник RocksDB, встраиваемое хранилище ключей и значений, предлагающее упорядоченное отображение ключей на значения.
- *LMDB* (http://symas.com/mdb/) – встраиваемое хранилище ключей и значений, разработанное для использования вместо Berkley DB. Поддерживается транзакционность, используются файлы, спроецированные на память, рассчитано на рабочую нагрузку с преобладанием операций чтения.
- *Perset* (www.mcobject.com/perst) – транзакционное объектное хранилище для Java и .NET, при проектировании которого во главу угла ставились быстродействие, простота использования и прозрачность относительно поддерживаемых языков и хранилищ данных.

Этот список ни в коем случае нельзя считать исчерпывающим. Продолжив изыскания по этой теме, вы найдете и другие продукты.

6.2.2. Система кэширования

Продукты из этой категории могут называться и по-другому: система кэширования объектов, хранилища в памяти и даже хранилище ключей и значений в памяти. Важно то, что все они предназначены для хранения данных в памяти, так что даже возможности хранить данные вне DRAM не существует, и еще – API зачастую построен вокруг концепции ключей и значений. В системах кэширования применяется много разных стратегий, к нашему обсуждению наибольшее отношение имеют стратегия постоянного сохранения кэшированных данных и стратегия актуализации кэша. Рассмотрим, как они реализуются в продуктах этого класса.

Сквозное чтение

В этом случае система кэширования читает данные из постоянного хранилища, если у нее запрашивают запись, отсутствующую в кэше, так что имеются накладные расходы на чтение при первом запросе. Клиенты кэша этого не замечают, но с точки зрения производительности необходимость читать из постоянного хранилища и записывать в него обновления, конечно, небезразлична. Эта стратегия показана на рис. 6.4.

Опережающее обновление

При такой стратегии кэш обновляет данные, к которым недавно осуществлялся доступ, до того как они будут вытеснены из-за истечения сро-

ка хранения. Идея в том, чтобы избежать накладных расходов на сквозное чтение из постоянного хранилища данных, вытесненных из кэша. Если система с опережающим обновлением настроена с учетом частоты обновления данных в постоянном хранилище, то кэш, возможно, будет возвращать клиенту актуальное значение, поддерживая синхронизацию с хранилищем. При работе с потоком данных организовать такую точную координацию часто затруднительно. Эта стратегия изображена на рис. 6.5.

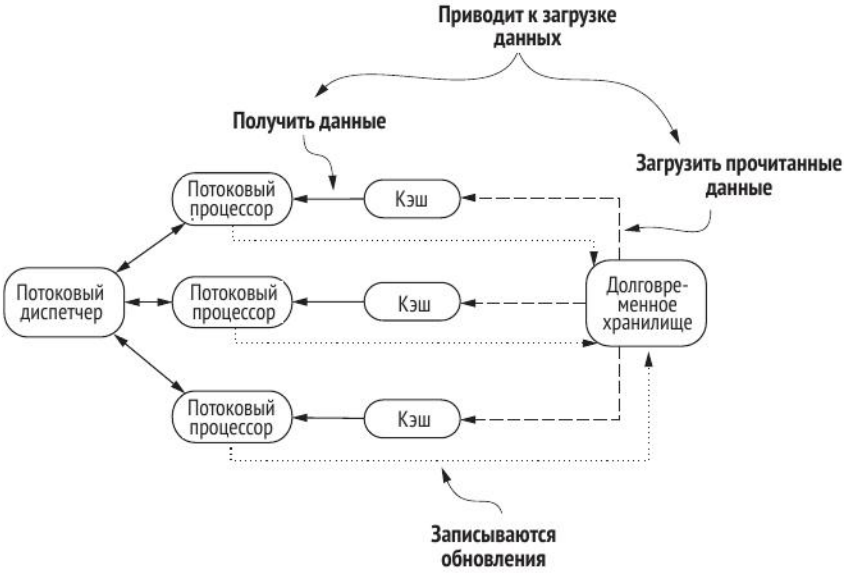


Рис. 6.4. Стратегия кэширования со сквозным чтением – данные загружаются из долговременного хранилища, если отсутствуют в кэше

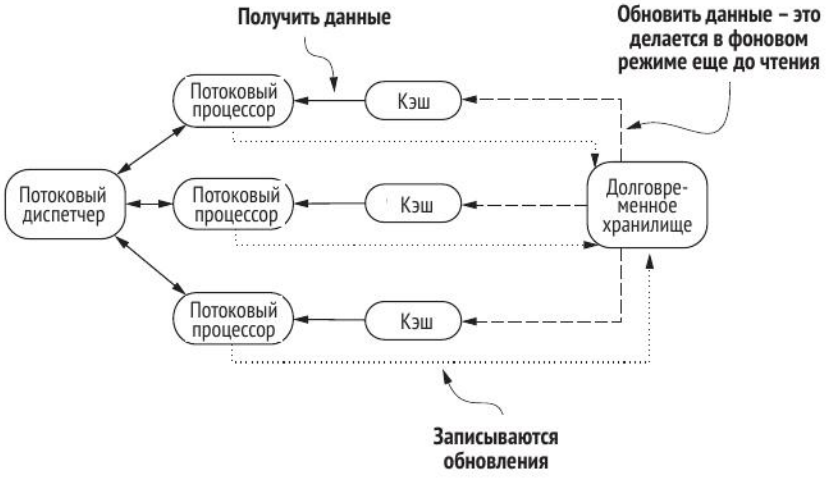


Рис. 6.5. Стратегия кэширования с опережающим обновлением – данные обновляются в соответствии с состоянием долговременного хранилища

Сквозная запись

В этом случае система кэширования сразу записывает обновленные данные в постоянное хранилище, устраняя необходимость в фоновом процессе, который сбрасывал бы данные из кэша в хранилище или загружал измененные данные в кэш. При такой стратегии система кэширования не подтверждает успешность записи, пока данные не окажутся в постоянном хранилище. Следовательно, запись сопровождается задержкой, что может рассматриваться как недостаток по сравнению с другими стратегиями. Эта стратегия изображена на рис. 6.6.

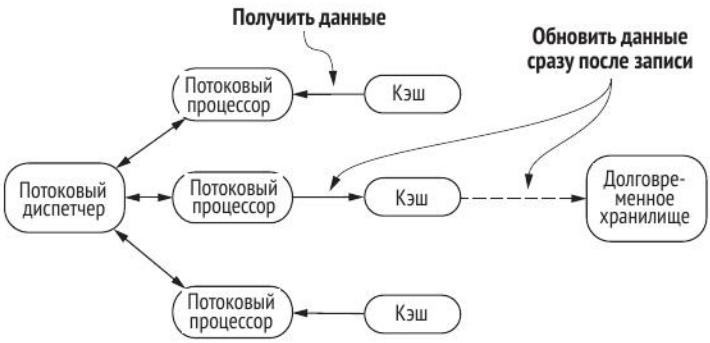


Рис. 6.6. Стратегия кэширования со сквозной записью – данные сразу записываются в долговременное хранилище

Обходная запись

Идея этой стратегии состоит в том, что процесс обновления постоянного хранилища, представленного кэшем, происходит *помимо* кэша. Здесь «помимо» означает, что обновление кэша производится в фоновом режиме, – на рис. 6.6 это показано пунктирными линиями. Зачастую для подобных операций предусмотрен отдельный путь выполнения. В таком случае система кэширования вообще не знает о том, что в постоянное хранилище записываются изменения, и полагается на то, что какой-то другой процесс обновит кэш после обновления хранилища. В этом случае система усложняется из-за необходимости обновлять два хранилища данных: постоянное и кэш. Но есть и преимущество – система кэширования вообще не должна взаимодействовать с постоянным хранилищем. Эта стратегия изображена на рис. 6.7.

Отложенная запись

При такой стратегии система кэширования записывает обновленные данные в постоянное хранилище не сразу, а в конечном итоге. В отличие от сквозной записи, когда данные записываются немедленно, в случае отложенной записи система подтверждает запись в кэш, а запись в постоянное хранилище производит позже в фоновом режиме. Преимуществом явля-

ется отсутствие накладных расходов на сквозной ввод-вывод, но в течение времени, когда данные хранятся только в оперативной памяти, существует риск их утраты. Стратегия отложенной записи показана на рис. 6.8.

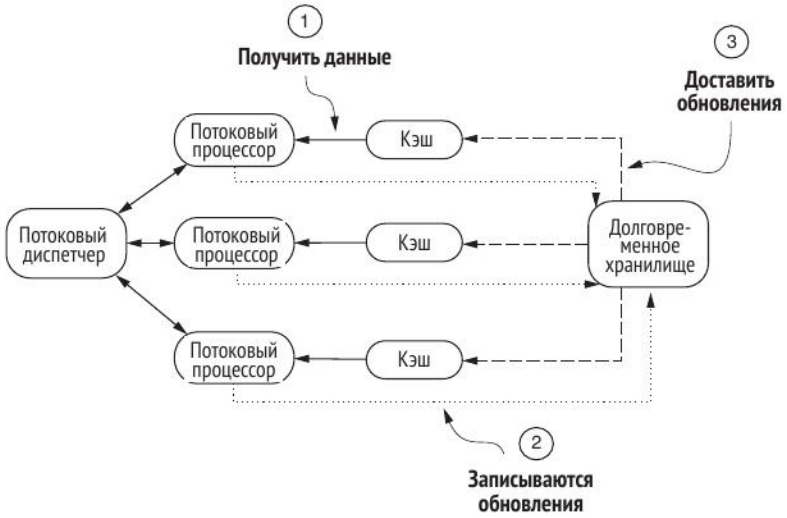


Рис. 6.7. Стратегия кэширования с обходной записью – данные записываются в долговременное хранилище, а затем должны быть доставлены в кэш

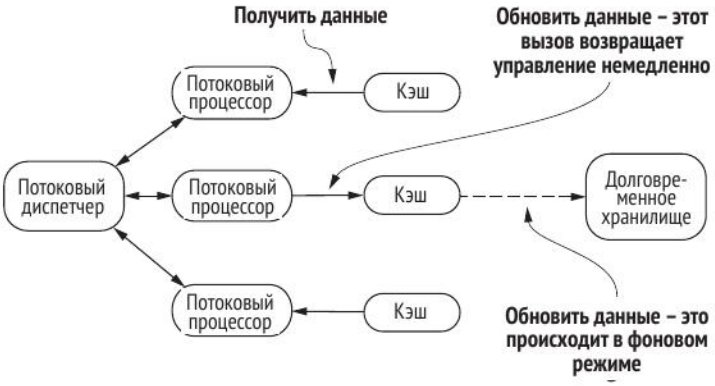


Рис. 6.8. Стратегия кэширования с отложенной записью

Таким образом, мы либо миримся с тем, что данные находятся только в памяти, либо платим за их помещение в постоянное хранилище. Часто возможность хранить данные где-либо вне оперативной памяти вообще отсутствует, но тогда есть риск потерять данные при выходе узла из строя.

Некоторые системы из этой категории, например EHCache (www.ehcache.org), предоставляют средства расширения для поддержки сквозного чтения, сквозной записи и отложенной записи. В потоковой системе, где нужно обрабатывать постоянный поток данных, такая двойная запись и распространение изменений становятся узким местом. Если результаты

промежуточные и хранить их постоянно нет необходимости, то на это узкое место можно не обращать внимания. Но в общем случае исторические данные все-таки нужны, и тогда использование системы кэширования вынуждает нас дважды записывать результаты потокового анализа, что потенциально чревато высокими издержками.

Для интересующихся перечислим несколько наиболее популярных продуктов с открытым исходным кодом в этом сегменте.

- *Memcached* (http://memcached.org) – популярная система кэширования, но для поддержания актуальности данных необходимо задействовать стратегию обходной записи.
- *EHCache* (www.ehcache.org) – развитая система кэширования, поддерживающая различные сценарии: отложенную запись, сквозную запись и сквозное чтение.
- *Hazelcast* (http://hazelcast.org) – надежный продукт, предназначенный далеко не только для кэширования. В части кэширования поддерживает стратегии сквозного чтения и сквозной записи.
- *Redis* (http://redis.io) – популярная система, хорошо зарекомендовавшая себя в роли кэша в памяти. Постоянное хранение поддерживается с помощью собственного формата файлов. Любую из рассмотренных выше стратегий следует реализовывать самостоятельно.

6.2.3. Базы данных и решетки данных в памяти

Базы данных в памяти (in-memory database, IMDB) иногда называют *системами управления базами данных в памяти*, или *решетками данных в памяти* (in-memory data grid, IMDG). В отличие от систем кэширования, при проектировании IMDB и IMDG для энергонезависимого хранения данных используется диск. Хотя продукты из этой категории используют для хранения не только DRAM-память, они конструируются так, что оперативная память использовалась в первую очередь, а диск – во вторую. На самом деле диск часто служит только для протоколирования и периодического создания списков, чтобы не потерять данных в случае сбоя. Это тонкий момент, отличающий настоящие IMDB и IMDG от традиционных баз данных, которые стали предлагать варианты с хранением в памяти. Традиционные СУБД (Microsoft SQL Server, Oracle и т. д.), равно как и современные базы данных NoSQL (например, Apache Cassandra), проектировались, имея в виду в первую очередь диск, а только во вторую – память.

Чем отличаются подходы с приоритетом диска и памяти? Принципиальные отличия показаны на рис. 6.9.

Подход с приоритетом диска – упрощенное представление дизайна традиционных базы и многих баз NoSQL. Понятно, почему продукты, которые начинали жизнь как системы на основе диска, а теперь обзавелись

возможностью хранить данные в памяти, проектировались так, а не иначе. Проблемы появляются позже: даже если продукт преобразуется в базу данных в памяти, принципы проектирования остаются неизменными; место диска занимает оперативная память, но работа с ней строится, как с устройством ввода-вывода. Я, правда, вовсе не хочу сказать, что память в этих продуктах раньше вообще не использовалась; напротив, из соображений производительности они с удовольствием использовали память для кэширования.

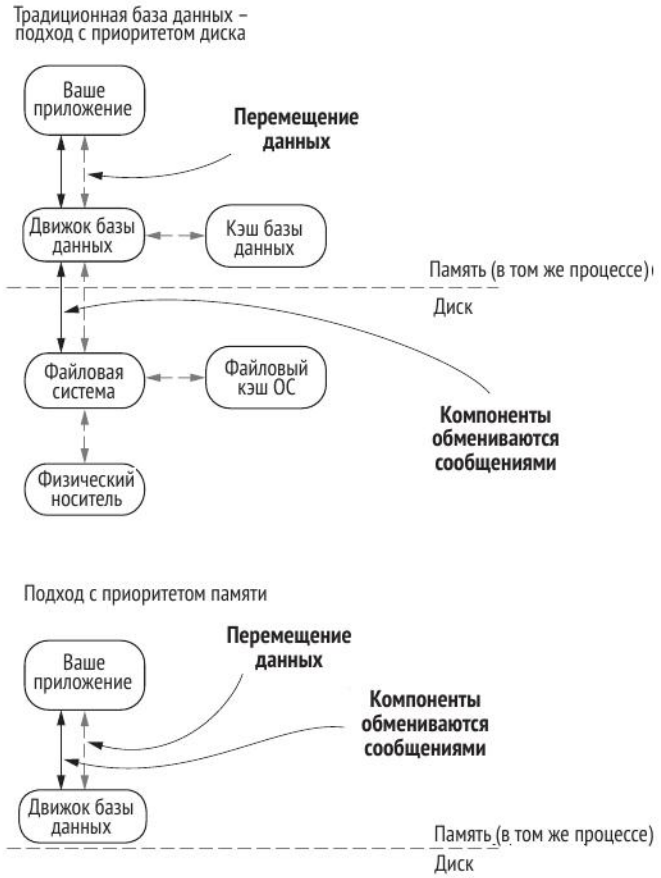


Рис. 6.9. Два разных подхода к проектированию баз данных: с приоритетом диска и с приоритетом памяти

В подходе с приоритетом памяти все поставлено с ног на голову. Программа проектируется так, чтобы использовалась только память, а диск играет вторичную роль – для повышения надежности. Это тонкое различие, но благодаря ему производительность продуктов в этом сегменте возрастает кардинально. Есть несколько продуктов, находящихся на границе с системами кэширования; что касается IMDB и IMDG, то граница между ними размыта и в настоящее время весьма подвижна.

Традиционно между IMDB и IMDG было несколько ключевых различий. В IMDG использовалась распределенная архитектура, которая позволяла производить вычисления на сервере поблизости от данных – как в хранимых процедурах из мира реляционных СУБД. IMDB часто предлагают API на основе SQL, тогда как в IMDG используется API, не основанный на SQL. Но по мере эволюции потоковых продуктов эти грани стали стираться, и теперь не редкость встретить IMDG с гораздо более развитой вычислительной моделью; например, Apache Spark вторгается на территорию вычислений в памяти. Ввиду этого процесса IMDB постепенно приближаются к IMDG, а IMDG движутся в сторону конкуренции с инструментами потокового анализа и сложными каркасами обработки событий. Перечислим несколько продуктов с открытым исходным кодом в этом сегменте.

- *MemSQL* (www.memsql.com) – первоначально система работала только в памяти, но теперь поддерживает и сохранение данных на диск. Главное достоинство – полная совместимость с MySQL, но работает она гораздо быстрее, поскольку по умолчанию все данные хранятся в памяти.
- *VoltDB* (www.voltdb.com) – высокопроизводительная совместимая с SQL база данных, предлагающая вариант хранения в памяти, а также долговечность, высокую доступность и экспорт в различных форматах.
- *Aerospike* (www.aerospike.com) – оптимизированный для флэш-памяти движок NoSQL с богатой функциональностью: от геопространственных индексов и запросов до межкластерной репликации.
- *Apache Geode* (http://geode.incubator.apache.org) – первоначально проектировался для хранения в памяти, но предлагает также хранение вне кучи и язык запросов OQL (Object Query Language), похожий на SQL.
- *Couchbase* (www.couchbase.com) – документная база данных, появившаяся как помесь Memcached и CouchDB, но далеко переросшая эти рамки. Допускает межкластерную репликацию и располагает собственным языком запросов N1QL (произносится «никель») – расширением SQL для данных в формате JSON.
- *Apache Ignite* (https://ignite.apache.org) – первоначально подавалась как матрица данных в памяти, но в итоге превратилась в IMDG с добавлением развитой структуры для grid-вычислений, поддержки SQL и интеграции с Hadoop и Spark. Поначалу продукт предлагала компания GridGain, но впоследствии он перешел под патронаж Apache Foundation.
- *Hazelcast* (https://hazelcast.org) – это IMDG, которая на момент написания книги предлагала такие функциональные возможности, как

язык запросов, распределенное агрегирование, поддержку технологии MapReduce и распределенные структуры данных.

- *Inifispan* (http://infinispan.org) – IMDG, которая также предлагает интеграцию с Apache Spark, распределенные потоки и скриптовый язык для сложных операций.

6.3. Примеры и упражнения

Мы рассмотрели различные аспекты хранилищ в памяти. Принимая во внимание многочисленные дополнительные возможности и постоянно растущий список новых участников, обсудить все продукты не представляется возможным. Но теперь мы готовы обсудить несколько конкретных примеров и разобраться, какая категория продуктов в большей степени отвечает вашим потребностям. Без сомнения, ваши требования могут в корне отличаться от наших примеров, но надеюсь, вы все-таки сможете извлечь из них полезную для себя информацию. Я знаю, что мы пока не говорили о звене доступа к данным, поэтому в последующем изложении упор делается на том, как подготовить данные для доступа, и принимаются некоторые допущения о том, как этот доступ организован.

6.3.1. Сеансовая персонализация

В этом примере будем предполагать, что строим систему потоковой обработки для популярного интернет-магазина TheOceana.com, в котором товарный каталог так же необъятен, как Мировой океан. Мы имеем поток всех действий на сайте и хотим персонализировать сайт для пользователей, находящихся в процессе совершения покупки. Эту идею часто называют сеансовой персонализацией, потому что всё происходит во время текущего сеанса пользователя. Наша цель – изменить содержимое страницы в момент запроса, исходя из характера действий пользователя в текущем сеансе. Допустим, что пользователь производил следующие действия:

1. Зашел в категорию «Солнечные очки».
2. Добавил желтые солнечные очки в корзину.
3. Зашел в категорию «Мотоциклетные шлемы».
4. Удалил солнечные очки из корзины.
5. Зашел в категорию «Куртки».
6. Зашел в категорию «Мотоциклетные перчатки».

Мы хотим следующим образом персонализировать эту последовательность действий:

- при заходе в категорию «Мотоциклетные шлемы» на шаге 3 показать шлемы, которые подошли бы к желтым солнечным очкам;

- при заходе в категорию «Мотоциклетные перчатки» на шаге 6 показан купон на солнечные очки, удаленные из корзины двумя шагами ранее.

На рис. 6.10 показана последовательность действий вместе с предложениями по части персонализации.

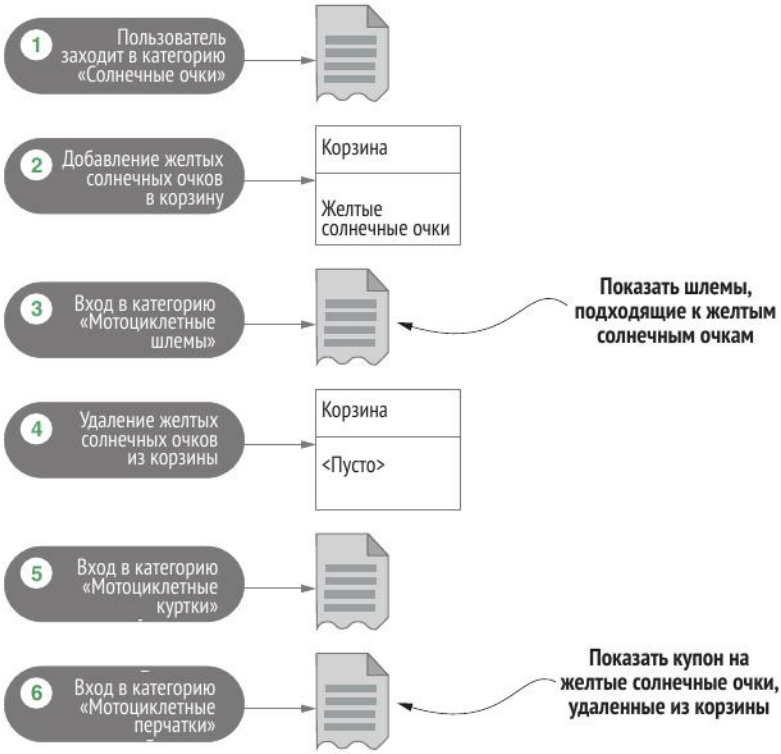


Рис. 6.10. Последовательность страниц в примере сеансовой персонализации

Я уверен, что вы можете придумать много других способов персонализировать страницу, быть может, с учетом сеанса не только этого, но и всех активных пользователей. Возможно, вы даже найдете способы объединить текущий сеанс с историей действий пользователя. Но мы сосредоточимся на двух способах персонализации контента: показе связанного контента и предложении купона на товар, удаленный из корзины. Для этого нужно отслеживать и анализировать сеансы активных посетителей. Будем считать, что сеанс посетителя продолжается 30 минут и включает все действия, произведенные пользователем на сайте, – *поток кликов*. Подумайте, как бы вы решили задачу, применяя рассмотренные выше варианты хранения данных в памяти. Сделаем это вместе.

Встраиваемое хранилище, оптимизированное для флэш-памяти

В этом случае для хранения данных в памяти узлов анализа нам нужно решить несколько проблем:

- сеансы всех посетителей должны храниться на узлах анализа;
- узел анализа должен знать, когда сеанс закончился и как вытеснить данные из памяти;
- у клиента должен быть способ опросить узел анализа, чтобы получить сеанс посетителя. Если не существует способа выполнить распределенный запрос, то клиент должен знать, на каком узле хранится сеанс конкретного посетителя;
- нужно как-то гарантировать, что если узел анализа, на котором хранится сеанс посетителя, выйдет из строя, то данные не будут потеряны.

С учетом этих требований и рисков, чтобы использовать встраиваемую базу данных, мы должны не только удовлетворить требованиям, но и смягчить риски, сопряженные с хранением данных в памяти. Таким образом, этот путь не выглядит подходящим. На самом деле если потратить время, пытаясь решить задачу с помощью любой из рассмотренных выше систем потоковой обработки – Apache Storm, Spark, Samza или Flink, – то мы быстро придем к выводу, что дизайн получается хрупким, немасштабируемым и с архитектурной точки зрения отнюдь не чистым. Я бы сказал, что этот вариант хранилища не годится.

Система кэширования

Подойдет ли в этом случае система кэширования? Чтобы ее использовать, мы должны гарантировать, по меньшей мере, выполнение следующих условий:

- что мы не потеряем данных в случае аварии узла;
- что клиенты могут запрашивать кластер кэширования по идентификатору посетителя, получая в ответ весь сеанс;
- что узлы анализа могут постоянно обновлять сеанс;
- что сеанс вытесняется из кэша по истечении 30 минут.

Можно ли обеспечить все это? Чтобы гарантировать сохранность данных в случае аварии узла, система кэширования должна хранить резервные копии данных на разных узлах. Многие продукты из этой категории не поддерживают репликацию, поэтому этот риск смягчить не всегда возможно. Возможность опрашивать кэш по идентификатору посетителя не вызывает трудностей во всех системах кэширования, потому что они предоставляют API ключей и значений и возвращают данные, ассоциированные с ключом, – в данном случае сеанс посетителя. Что касается непрерывного

обновления данных, то следует учесть, что мы не только заменяем значение, ассоциированное с ключом, но и добавляем в него новые данные. Как в таком случае мы будем обновлять сеанс посетителя, когда он бродит по сайту? Имеется только одна возможность, поскольку мы вынуждены использовать идентификатор посетителя в качестве ключа. Мы должны прочитать весь сеанс из кэша, обновить его и записать обратно в кэш. Это вызовет пробуксовку, которая несомненно плохо отразится на производительности. Быть может, найдется решение получше.

IMDB или IMDG

Как насчет IMDB или IMDG? Будет ли это удачным решением? Повторим основные требования, которые предъявлялись к системе кэширования. Следует гарантировать:

- что мы не потеряем данных в случае аварии узла;
- что клиенты могут запрашивать кластер кэширования по идентификатору посетителя, получая в ответ весь сеанс;
- что узлы анализа могут постоянно обновлять сеанс;
- что сеанс вытесняется из кэша по истечении 30 минут.

Первое требование без труда удовлетворяется многими продуктами из этого сегмента, поскольку они естественным образом поддерживают горизонтальное масштабирование и реплицируют данные на разных узлах кластера. Запрос полного сеанса по идентификатору посетителя также не вызывает трудностей у большинства IMDB и IMDG, представленных на рынке. Последнее требование – вытеснение сеанса из кэша по истечении 30 минут – легко удовлетворяется в системе Aerospike, которая работает как NoSQL-хранилище, сопоставляющее каждой записи «время жизни». В других IMDB или IMDG, конкретно в MemSQL и VoltDB, необходимо определять срок хранения сеанса вне самого хранилища и по его истечении удалять данные. В общем и целом с помощью IMDB задача решается сравнительно просто, так что эта технология подходит.

Переход на следующий уровень

Некоторые наши решения могли показаться совсем уж простыми. Давайте сделаем еще один шаг: изменится ли ваша оценка, если нужно будет поддержать два дополнительных требования:

- мы хотим принимать во внимание не сеанс единственного посетителя, а сеансы всех посетителей за текущий день;
- для принятия решения мы хотим учитывать не только текущий сеанс пользователя, но и всю его историю.

А мы пока опишем следующий пример, но теперь попросим проработать все вопросы самостоятельно.

6.3.2. Энергетическая компания следующего поколения

Допустим, что мы создаем энергетическую компанию следующего поколения, которая позволит избежать полного и частичного отключения потребителей, от которых некоторые штаты США мучатся в летние месяцы. Поставщики электроэнергии давно уже страдают от одного и того же сценария, характерного для субботнего дня: на улице жарко, и все потребители включают кондиционеры. А еще они заглядывают в холодильник, чтобы взять бутылочку холодненького, и, раз уж оказались на кухне, решают помыть посуду. Стоит такая жара, что все сидят по домам. Кому-то захочется постирать белье, кому-то – включить телевизор. Как вы понимаете, все это резко увеличивает спрос на электроэнергию, а когда спрос превышает имеющиеся мощности, начинаются проблемы.

Мы решили покончить с этим: построить интеллектуальную энергосистему и привлечь к участию потребителей. Первый шаг – установить интеллектуальные счетчики в домах потребителей. Они будут каждые пять минут сообщать о потреблении энергии и об устройствах-потребителях. На основе анализа поступающих данных мы хотим в реальном времени предлагать клиентам переменные тарифы с учетом того, когда и как они пользуются основными бытовыми приборами. Например, если на кондиционере выставляется температура чуть повыше или посудомоечная машина включается ночью, то мы предложим скидку.

Чтобы эта идея имела успех, мы должны производить потоковый анализ в двух местах: на электростанции и в домах. Сначала рассмотрим, что необходимо на электростанции:

- возможность каждые пять минут анализировать и сохранять данные, принимая во внимание погоду и исторические данные;
- возможность сохранять тариф на следующий час потребления (в предположении, что мы предлагаем почасовую тарификацию);
- возможность запросить у хранилища данных информацию о тарифе для указанного клиента;
- гарантии сохранности данных о клиентах;
- гарантии доступности данных для последующего анализа.

Рассмотрите эту задачу для каждого из обсуждавшихся хранилищ данных в памяти. Как изменятся выводы, если перенести обработку данных на счетчик? Может показаться, что все это – отдаленная перспектива, но энергетическая отрасль движется именно в этом направлении[1].

[1] Начать знакомство с этой проблематикой можно с публикации Министерства энергетики США «The SmartGrid: An Introduction», www.smartgrid.gov/files/The_Smart_Grid_Introduction_200804.pdf.

6.4. Резюме

В этой главе мы рассмотрели различные варианты хранения данных в памяти во время и после анализа. Мы не стали углубляться в изучение долговременных хранилищ на диске, поскольку они редко используются в потоковом анализе и не могут похвастаться такими же показателями производительности, как хранилища в памяти.

В этой главе мы:

- узнали о различных подходах к кэшированию;
- сравнили различные типы хранилищ данных в памяти;
- обсудили, как выбирать подходящее хранилище.

Может показаться, что мы познакомились только с половиной картины, поскольку при выборе подходящего хранилища нужно учитывать, как мы будем обращаться к данным. Глава 7 посвящена вопросу доступа к сохраненным данным. А из этой главы нужно вынести такой урок: на рынке представлено много разных вариантов хранения данных в памяти, и идея держать в памяти весь рабочий набор данных – уже не мечта; современное оборудование, программное обеспечение и непрерывное падение цен сделали ее реальностью. Как будете готовы, переворачивайте страницу – займемся доступом к сохраненным данным.

Глава 7

Получение доступа к данным

Краткое содержание главы:

- типичные паттерны взаимодействия;
- когда использовать веб-уведомления, длинные HTTP-опросы, события, посылаемые сервером, и веб-сокеты;
- пример: построение потокового API для сайта Meetup.

Мы долго изучали архитектуру и теперь готовы рассмотреть, как данные доставляются потоковому потребителю. При проектировании этого звена мы сталкиваемся с той же проблемой, что и в других случаях, – существует масса технологий и много способов организации звена. В других звеньях нам приходилось иметь дело с получением данных, их перемещением, анализом и подготовкой к использованию. Без сомнения, все они по-своему трудны, интересны и заслуживают внимания, но, на мой взгляд, звено доступа к данным сулит наибольшую награду разработчику. Именно здесь – в момент доставки данных клиенту – все наши труды окупаются. Весь смысл этого звена состоит в том, чтобы дать пользователям возможность применить наш API к построению приложений с элементами реального времени. Доступ к API такого рода сулит немало преимуществ. Первое, что приходит на ум, – быстрая отдача для бизнеса благодаря возможности видеть данные и предпринимать действия в режиме реального времени. Другое преимущество вытекает из того, что вместе со средствами для создания более привлекательных приложений разработчики получают дополнительную мотивацию к труду.

В этой главе наша цель – понять, как построить потоковый API доступа к данным, принимая во внимание различные паттерны взаимодействия, обсудить методы обработки ошибок и познакомиться с различными имеющимися протоколами. На рис. 7.1 показано то место общей архитектуры, которое мы собираемся рассматривать. Начнем с паттернов взаимодействия.

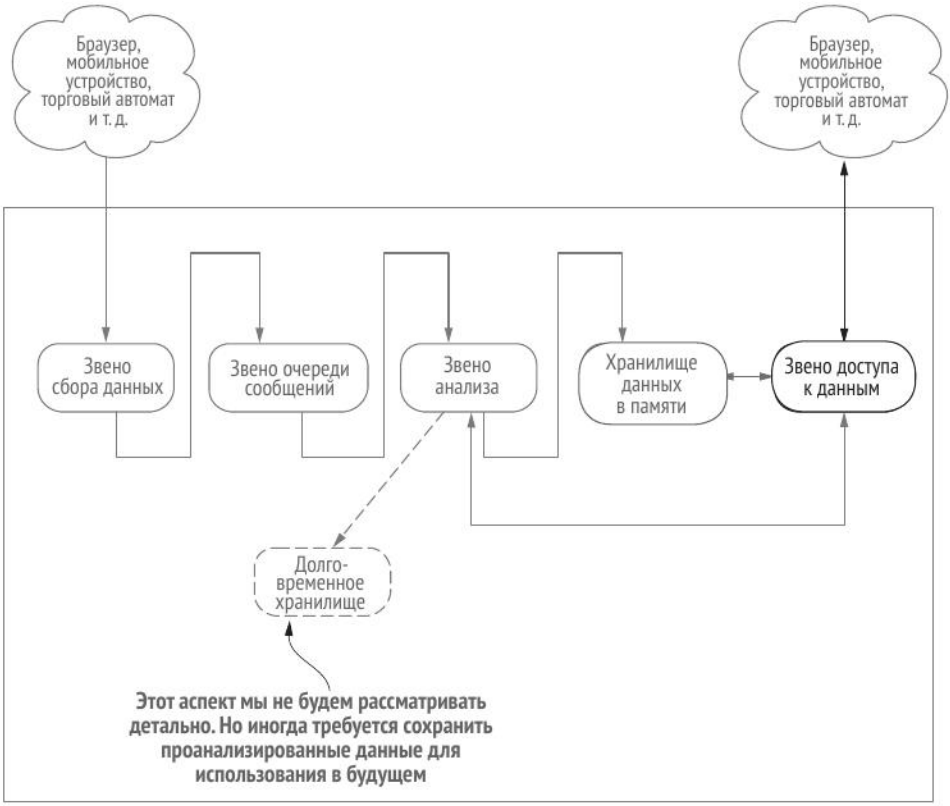


Рис. 7.1. Архитектура потоковой обработки данных – выделена тема этой главы

7.1. Паттерны взаимодействия

В потоковой системе мы обрабатываем непрерывный поток данных, который нам обычно проталкивают, но это часто перестает быть верным, когда дело доходит до паттернов взаимодействия с клиентами. Тут надо решить, какой API поддержать: проталкивание данных клиенту, вытягивание данных клиентом или то и другое. Помимо общих понятий проталкивания и вытягивания, существуют четыре распространенных паттерна: синхронизация данных (Data Sync), RMI/RPC, простой обмен сообщениями и издатель-подписчик. Они напоминают паттерны, встречавшиеся в звене очереди сообщений. Рассмотрим их по порядку.

7.1.1. Паттерн Data Sync

Может показаться, что любое взаимодействие между нашим API и клиентом включает синхронизацию данных, но паттерн Data Sync подразумевает синхронизацию базы данных или хранилища между API и клиентом. На рис. 7.2 показано, как устроен этот паттерн.

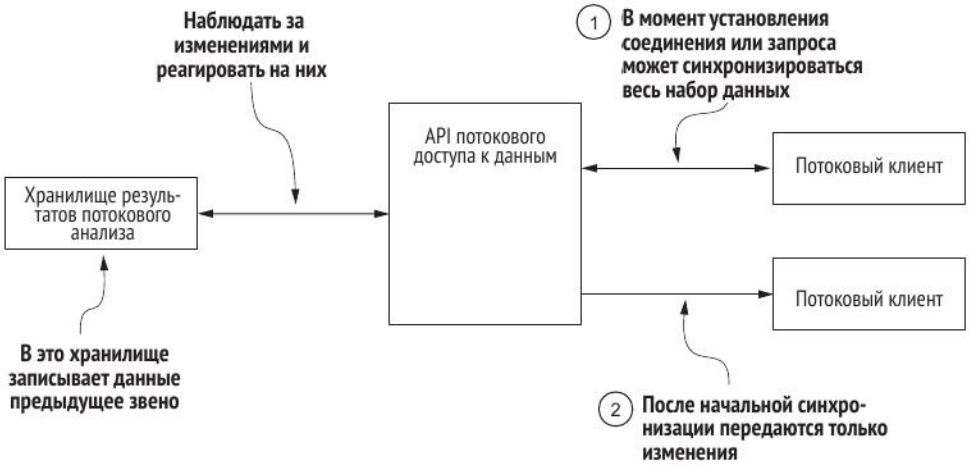


Рис. 7.2. Паттерн взаимодействия Data Sync – показаны начальная и последующая синхронизации

Общая идея этого паттерна заключается в том, что потоковый API наблюдает за изменениями хранилища, в которое производит запись звено анализа (может существовать механизм уведомлений), и посылает обновления потоковому клиенту. Как показано на рис. 7.2, обычно этот паттерн включает два шага. Во-первых, в момент подключения – например, на этапе установки мобильного приложения, – клиент запрашивает текущий набор данных. Впоследствии либо изменения набора данных проталкиваются клиенту, либо клиент вытягивает их по собственной инициативе. Этот паттерн кажется простым, однако рассмотрим его плюсы и минусы.

Достоинства

- Простой протокол.
- У клиента имеется полный набор данных.
- API легко разработать, потому что нужно поддержать только начальную синхронизацию и дельту, основанную либо на времени, либо на версии локальных данных.
- У клиента всегда имеется согласованное представление самых актуальных данных.

Недостатки

- Набор данных может быть велик, для его передачи требуется широкополосная сеть.
- Данные могут не поместиться на конечном устройстве.
- Необходимо разрешать конфликты из-за разных версий данных.
- Необходимо определить политику объединения, описывающую, как изменения данных, произведенные клиентом, увязывать с изменениями, произведенными сервером.

Но даже несмотря на указанные недостатки, во многих приложениях этот паттерн взаимодействия может оказаться вполне приемлемым. Например, в многопользовательской мобильной игре – когда необходимо, чтобы на каждом устройстве было представлено одно и то же состояние игры и не обязательно рассматривать каждый ход как отдельную транзакцию, – передача текущего набора может иметь смысл.

7.1.2. Удаленный вызов метода и удаленный вызов процедуры

В случае паттерна взаимодействия RMI (удаленный вызов метода) или RPC (удаленный вызов процедуры) API устроен так, что сервер вызывает метод подключившегося клиента, когда поступают новые данные или возникает событие, представляющее интерес для клиента. Общая схема показана на рис. 7.3.

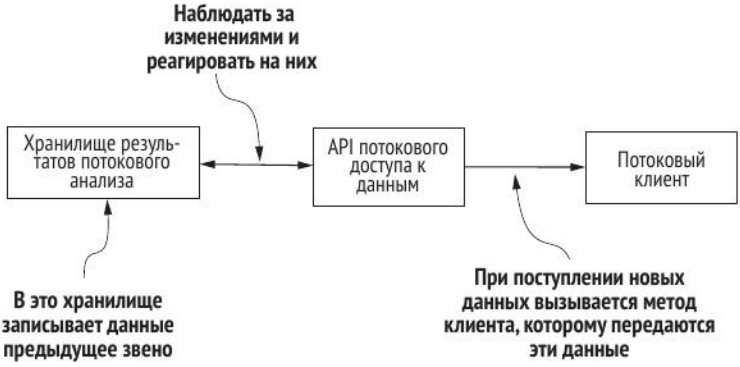


Рис. 7.3. Схема паттерна взаимодействия RMP/RPC

Как видим, паттерн RMI/RPC довольно прост. В общем случае наш API следит за изменениями в хранилище данных, в которое пишет звено анализа, и уведомляет о них клиента посредством вызова методов. Существу.т две разновидности паттерна: либо данные (например, самое последнее значение) передаются в виде аргументов метода, либо посредством вызова метода клиент уведомляется о наступлении некоторого условия.

Достоинства

- Простой протокол.
- Клиент может заниматься другими делами и реагировать в тот момент, когда вызывается обработчик.
- API легко разработать – нужно лишь, чтобы клиент как-то зарегистрировал оконечную точку.

Недостатки

- Трудно обнаружить ошибки – что, если клиент недоступен? Как клиент может узнать о том, что сервер не получил новых данных?

- Частота обновлений может быть настолько велика, что клиент захлебнется.
- Как должен быть устроен API обработки ошибок на стороне клиента?

7.1.3. Простой обмен сообщениями

В этом случае клиент отправляет запрос потоковому API, а API в ответ передает последние данные. Если не включить в запрос метаданные, говорящие серверу «возвращать только те данные, которые новее *X*», или не предусмотреть в API возможность сообщить клиенту, что данные не будут изменяться в течение времени *X*, то этот паттерн неэффективен, поскольку клиент будет запрашивать данные, не изменившиеся с момента последнего запроса. Постоянное запрашивание одних и тех же данных – это пустая трата ресурсов. В вышеупомянутых вариантах добавляется информация в запрос или в ответ, чтобы клиент не запрашивал у потокового API неизменившиеся данные. На рис. 7.4 показан принцип работы этого паттерна.

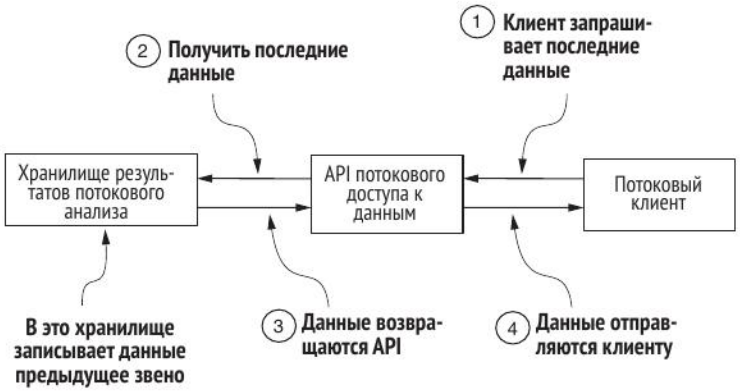


Рис. 7.4. Схема паттерна взаимодействия «простой обмен сообщениями», при котором клиент запрашивает последние данные

Как видим, потоковый клиент запрашивает последние данные, API делает запрос к хранилищу результатов потокового анализа, данные возвращаются API потокового доступа, а затем передаются клиенту. У этой схемы могут быть вариации. Например, иногда у клиента есть возможность запросить изменения, начиная с некоторого момента. Тогда API либо возвращает изменения, произошедшие после этого момента, либо вообще не возвращает никаких данных.

Достоинства

- Простой протокол и простой с точки зрения потребителя вызов API.
- Клиенту отправляются только последние данные.
- Для получения последних данных клиент должен запоминать небольшой объем метаданных.

- API не должен хранить состояние клиента.

Недостатки

- В протоколе возможен избыточный диалог, поскольку клиент может раз за разом отправлять постоянные запросы новых данных.
- Не существует механизма, позволяющего уведомить клиента о появлении новых данных.
- Если клиент долгое время был отключен или темп поступления данных очень высок, то объем новых данных может оказаться большим.

7.1.4. Издатель-подписчик

В этом случае клиент подписывается на некоторый канал, а API рассылает сообщения всем подписавшимся на канал клиентам в момент изменения данных. Термин *канал* означает, что данные могут группироваться в категории или темы. Общая схема этого паттерна показана на рис. 7.5.

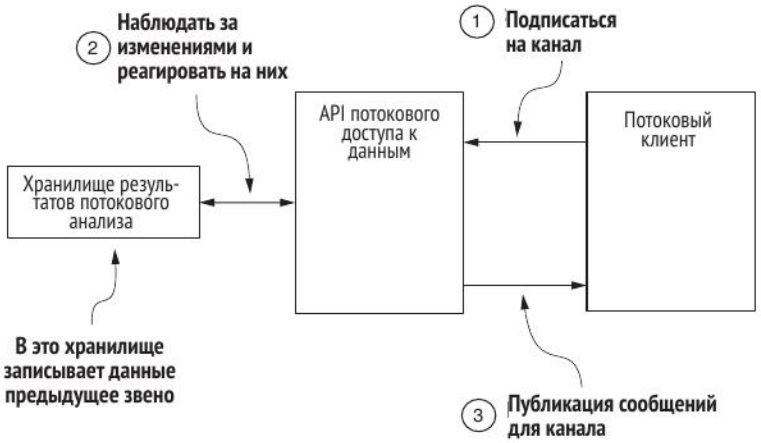


Рис. 7.5. Паттерн взаимодействия издатель-подписчик

Читатели, знакомые с паттерном проектирования наблюдатель, наверняка узнали поток данных, изображенный на рис. 7.5. Существует объект, который хранит список зависимых от него объектов и состояние. Когда состояние изменяется, объект уведомляет все зависимые объекты. В паттерне издатель-подписчик потоковый API хранит список потоковых клиентов и, когда изменяются данные в хранилище результатов потокового анализа, уведомляет их всех.

У этого паттерна много преимуществ, по сравнению с другими. И, пожалуй, самое важное – тот факт, что потоковый API хранит всех клиентов, подписавшихся на канал, и публикует для них сообщения при появлении новых данных. Это снимает бремя забот и с клиентов, и с самого потокового API. Клиентам не нужно постоянно запрашивать новые данные или

отслеживать метаданные, информирующие о возможной доступности новых данных. А потоковый API освобождается от необходимости отвечать на запросы клиентов о данных, которые не изменялись. На первый взгляд, никакого затруднения для потокового API в этом нет, но представьте себе миллионы клиентов, запрашивающих неизменившиеся данные; отвечать на такие запросы – значит впустую растрачивать сетевые ресурсы и подвергать API тяжкому испытанию.

Этот паттерн получает все более широкое распространение по мере того, как мы движемся в сторону реактивного программирования и потоковых систем. Если вы начнете отслеживать какой-нибудь хэштег в Твиттере, то увидите этот паттерн в действии. Он используется также в мессенджере Slack. Для тех, кто не знает, что такое Slack, скажу, что он позволяет пользователям присоединиться к каналу, после чего рассылает всем подписавшимся обновления в случае, когда кто-нибудь отправляет в канал новое сообщение. Вы наверняка вспомните еще о многих примерах применения этого паттерна.

Достоинства

- Клиент может заниматься другими делами и реагировать на данные по мере их поступления.
- Клиенту не нужно хранить метаданные, описывающие текущие данные.
- API может оптимизировать отправку данных нескольким клиентам.

Недостатки

- Более сложный протокол.
- API должен хранить метаданные обо всех клиентах и иметь возможность распределять их между серверами на случай сбоя.

7.2. Протоколы отправки данных клиентам

Понимать общие паттерны взаимодействия уже неплохо – для начала. Но чтобы построить свой потоковый API, мы должны познакомиться с распространенными протоколами и решить, какие из них можно использовать в API. Для каждого протокола мы будем обращать внимание на следующие факторы:

- частота сообщений;
- направление взаимодействия;
- задержка сообщений;
- эффективность;
- отказоустойчивость и надежность.

7.2.1. Веб-уведомления

Веб-уведомления (webhook) появились примерно в 2007 году. Не будучи официальным стандартом W3C, они получили широкое признание в качестве способа, позволяющего клиенту определить оконечную точку HTTP, которая должна вызываться при поступлении новых данных или выполнении некоторого условия. Концептуально это аналог *обратных вызовов*, применяемых во многих языках программирования. Но есть и очевидные различия. Во-первых, обратные вызовы в этом случае выполняются с помощью POST-запросов по протоколу HTTP. Во-вторых, клиент часто реализуется сторонним разработчиком. Общая схема работы показана на рис. 7.6.

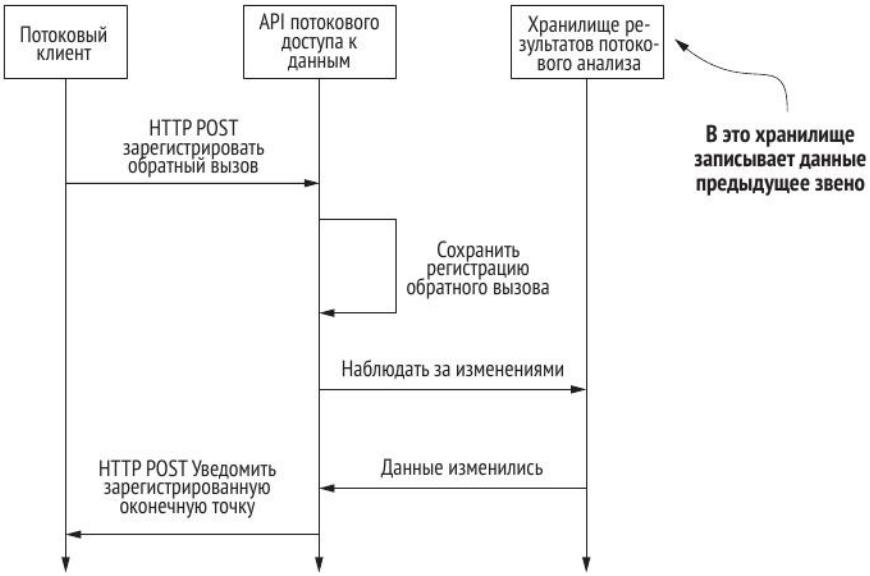


Рис. 7.6. Регистрация и обратный вызов веб-уведомления

На первом шаге клиент отправляет запрос о регистрации обратного вызова методом HTTP POST. Иногда этот шаг выполняется вручную пользователем, который заполняет форму на сайте. Информация об обратном вызове сохраняется. После этого потоковый API вызывает все зарегистрированные обратные вызовы при поступлении новых данных или наступлении некоторого события.

Рассмотрим вышеупомянутые факторы.

- *Частота сообщений.* Учитывая, что все обновления отправляются методом HTTP POST, будет справедливо считать, что частота отправки обновлений мала. В протоколе нет ничего такого, что помешало бы потоковому API вызывать его с высокой частотой, но, принимая во внимание текстовую природу протокола и внутренне присущие

ему накладные расходы, применять его в этой ситуации было бы опрометчиво.

- *Направление взаимодействия.* Взаимодействие всегда направлено в сторону от сервера к клиенту. Если необходимо что-то изменить, клиент должен заново зарегистрироваться. Процесс регистрации никак не стандартизирован, он полностью отдан на усмотрение разработчика потокового API.
- *Задержка сообщений.* Задержка умеренная. Во многих системах стек HTTP прекрасно оптимизирован и, хотя протокол текстовый, есть возможность воспользоваться встроенными в него механизмами сжатия и разбиения на порции, если объем обновления велик.
- *Эффективность.* С точки зрения потокового API, этот протокол эффективен. Помимо списка оконечных точек обратных вызовов, никакого состояния хранить не нужно. Для отправки обновления (данных или события) сервер может отправить асинхронный POST-запрос каждой из зарегистрированных оконечных точек.
- *Отказоустойчивость и надежность.* Сам протокол не дает никаких гарантий – вся ответственность возлагается на разработчика. Мы должны решить, что делать в случае ошибки при выполнении POST-запроса. А поскольку взаимодействие одностороннее, клиент никак не может подтвердить получения данных. Протокол HTTP, в принципе, позволяет отправить отдельный GET-запрос, чтобы узнать, был ли POST-запрос успешным, но это только усложняет задачу, потому что теперь нужно обрабатывать еще и возможную ошибку при выполнении этого запроса. Выбрав этот протокол для реализации потокового API, мы должны будем ответить, по меньшей мере, на следующие вопросы:
  - что делать с сообщениями, при отправке которых клиенту POST-запрос завершился ошибкой? Следует ли повторно отправлять сообщение?
  - может ли клиент как-то получить сообщения, не доставленные из-за ошибки при выполнении POST-запроса?

Веб-уведомления – довольно простой протокол. Но из обсуждения факторов видно, что он не поддерживает всего того, что мы хотим от потокового API. Обычно веб-уведомления применяются в системах, где интенсивность потока сообщений мала и в случае пропуска некоторых сообщений ничего страшного не случится.

7.2.2. Длинный HTTP-опрос

Длинный HTTP-опрос подразумевает, что клиент устанавливает соединение с сервером (в данном случае – сервером потокового API), это со-

единение остается открытым, и сервер посылает по нему данные клиенту, как только они появляются. Общий поток управления показан на рис. 7.7.

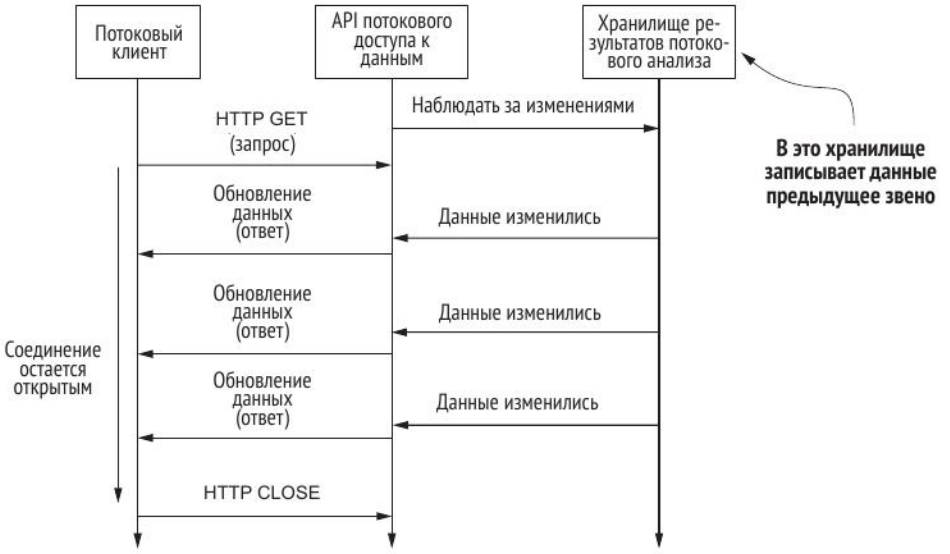


Рис. 7.7. Поток управления в длинном HTTP-опросе

Клиент отправляет запрос, а сервер не закрывает соединение, пока имеются обновления, и посылает их клиенту. Затем клиент открывает новое соединение, и цикл повторяется. Таким образом, клиент управляет опросом наличия изменений, но за это приходится расплачиваться тем, что потоковый API вынужден держать открытыми соединения со всеми клиентами.

Рассмотрим вышеупомянутые факторы, чтобы понять, о чем следует помнить, если мы решим строить потоковый API на основе этого протокола.

- *Частота сообщений.* Как и в других протоколах на основе HTTP, ничто не мешает использовать длинный HTTP-опрос, если темп обновлений высок. Но, учитывая текстовую природу протокола и неизбежные накладные расходы на доставку сообщения клиенту, закрытие соединений и необходимость повторно устанавливать их, при высокой частоте обновлений издержки могут оказаться чрезмерными.
- *Направление взаимодействия.* Клиент устанавливает соединение и может указать, в каких обновлениях заинтересован, а сервер отвечает.
- *Задержка сообщений.* Как и в случае веб-уведомлений, задержка при передаче по HTTP умеренная.
- *Эффективность.* Хотя сам протокол HTTP, который используется для передачи данных, может быть реализован эффективно, протокол длинного HTTP-опроса эффективным не назовешь. Наш сервер по-

токового API должен держать соединения открытыми, пока клиенты ожидают данных а, как только клиент получает сообщение, он тут же открывает соединение повторно. Возможности сервера API ограничены количеством одновременно открытых соединений. Кроме того, нужно как-то гарантировать, что в случае отключения клиента соединение закрывается. Например, будет ли соединение переустановлено автоматически, если мобильное устройство быстро переключается между сетью Wi-Fi и сотовой сетью или теряет соединение, а затем получает новый IP-адрес?

- *Отказоустойчивость и надежность.* Как и веб-уведомления, этот протокол не дает никаких гарантий, все должны делать мы сами. Выбрав этот протокол для реализации потокового API, мы должны будем решить, по меньшей мере, следующие проблемы:
  - гарантировать, что новые сообщения, поступившие в промежуток времени, пока клиент переподключался, не будут потеряны;
  - чтобы единственный сервер потокового API не захлебнулся, нужно придумать схему балансирования нагрузки;
  - чтобы клиент не терял и не пропускал сообщения, нужно обрабатывать отказы серверов потокового API.

Длинный HTTP-опрос гораздо ближе к тому, что мы хотим от потокового API. Этот протокол приобрел популярность в связи с распространением асинхронного программирования на стороне клиента, чему способствовало использование технологии AJAX (асинхронный JavaScript и XML). В первых веб-чатах длинный HTTP-опрос применялся для взаимодействия в режиме реального времени. Он также используется и в клиентах, не относящихся к вебу, поскольку в основе лежит протокол HTTP. Поддерживается большинством, если не всеми языками программирования, и доступен на любых устройствах – от серверов до небольших компьютеров на плате Raspberry Pi.

7.2.3. События, посылаемые сервером

Протокол событий, посылаемых сервером (server-sent events, SSE), был разработан в 2015 году как усовершенствование длинного HTTP-опроса. Вскоре была выпущена и рекомендация W3C. Во-первых, он устраняет неэффективность, связанную с тем, что клиент постоянно закрывает и открывает соединения для каждого сообщения. Во-вторых, для устройств с ограниченными ресурсами, например мобильных, он поддерживает прокси-сервер push-уведомлений, что позволяет устройству переходить в режим ожидания, когда оно не используется, и получать сообщения от прокси. Это дает значительную экономию энергии по сравнению с поддержанием открытых соединений в режиме бездействия. На рис. 7.8 и 7.9 показаны два способа использования SSE.

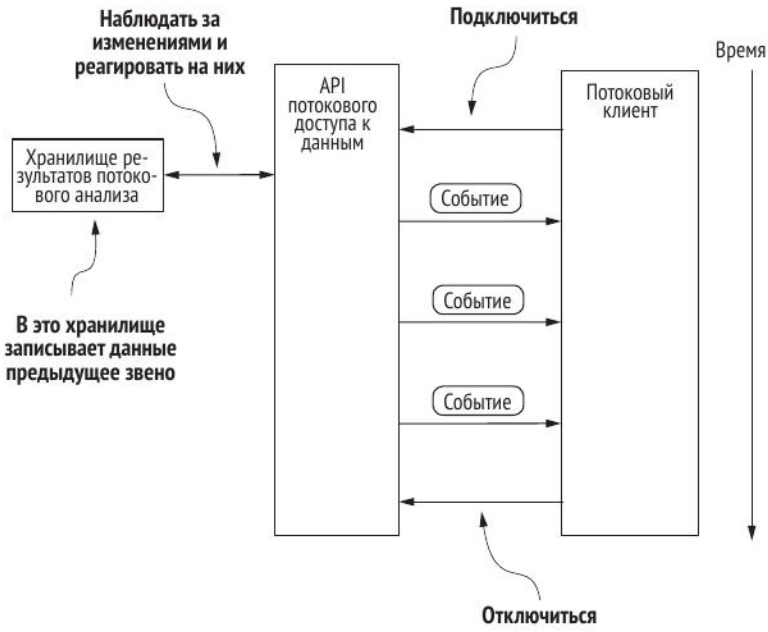


Рис. 7.8. Поток управления в SSE для подключенного клиента

На рис. 7.8 показано, что потоковый клиент сначала устанавливает соединение с сервером потокового API, по которому сервер передает сообщения по мере появления новых данных. В отличие от длинного опроса, когда соединение закрывается и заново открывается для каждого сообщения, здесь все сообщения передаются по одному и тому же соединению. В результате сеть используется более эффективно, а у клиента появляется возможность заниматься другими делами в ожидании событий. Но при этом клиент все-таки должен держать соединение открытым. Однако SSE поддерживает также *режим уведомлений без организации соединения*, показанный на рис. 7.9.

Поток управления на рис. 7.9 несколько сложнее, чем на рис. 7.8. Рассмотрим его подробнее.

1. Клиент, в данном случае браузер в мобильном устройстве, подключается к потоковому API.
2. После получения одного события проходит некоторое время (сколько именно, определяет разработчик), и устройство решает перейти в режим ожидания для экономии энергии.
3. Перед тем как перейти в режим ожидания, мобильное устройство посылает сообщение прокси-серверу уведомлений с просьбой взять на себя управление соединением. В сообщение может быть включен идентификатор последнего полученного события, чтобы прокси-сервер знал, с какого места продолжать.

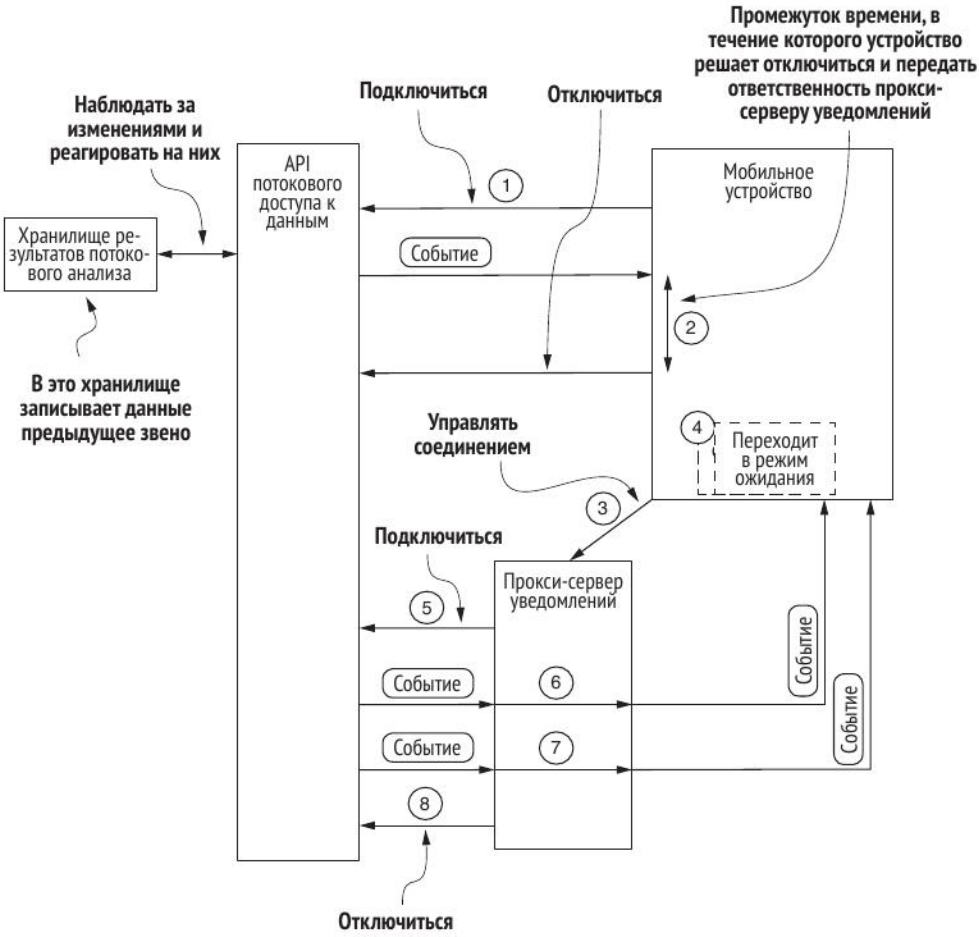


Рис. 7.9. Поток управления в SSE без организации соединения

4. Мобильное устройство переходит в режим ожидания.
5. Прокси-сервер устанавливает соединение с потоковым API.
6. Получив событие, прокси-сервер уведомлений отправляет его мобильному устройству, применяя специфичную для него push-технологию. Устройство просыпается, обрабатывает сообщение и возвращается в режим ожидания.
7. То же, что шаг 6.
8. В какой-то момент прокси-сервер разрывает соединение с потоковым API, и на этом все заканчивается.

Описанная схема с участием прокси-сервера позволяет мобильному устройству экономить энергию и уменьшать потребление данных. То и другое – побочные продукты того факта, что соединение с сервером API поддерживает прокси-сервер уведомлений, а не само мобильное устройство. Устройство же не платит высокую цену за поддержание TCP-соединения

в активном состоянии. Когда появляются новые данные, прокси-сервер с помощью push-технологии пробуждает устройство, чтобы оно могло получить и обработать сообщение, а затем снова уснуть. В результате пользователям становится удобнее работать с нашим потоковым API.

Учитывая, что оба варианта похожи, при рассмотрении интересующих нас факторов я их объединю, но там, где необходимо, буду отмечать различия.

- *Частота сообщений.* Хотя в качестве транспортного протокола используется HTTP, частота обновлений может быть гораздо выше, чем в случае длинного HTTP-опроса. Связано это с тем, что в этом протоколе имеется лишь одно постоянное соединение. Однако беспечный разработчик может столкнуться с проблемами, если клиент не способен читать данные в темпе поступления сообщений из сети.
- *Направление взаимодействия.* Если не считать начального установления соединения, то протокол является однонаправленным – сервер отправляет события клиенту.
- *Задержка сообщений.* Как уже отмечалось выше, задержка сообщений во всех протоколах на основе HTTP умеренная.
- *Эффективность.* Протокол настолько эффективен, насколько возможно при использовании HTTP в качестве транспортного механизма. Для каждого клиента существует только одно соединение, по которому передаются все данные, так что платить за постоянное открытие и закрытие соединений не надо. На стороне сервера повышение эффективности зависит от того, насколько быстро данные записываются в сокет, и следует позаботиться о том, чтобы клиент мог читать данные с такой же скоростью. В протоколе не предусмотрена возможность согласования и управления темпом передачи сообщений. Поэтому если клиенты не справляются, то следует поискать другой протокол.
- *Отказоустойчивость и надежность.* К сожалению, этот протокол, как и другие протоколы на основе HTTP, не дает никаких гарантий. Поскольку протокол однонаправленный – от сервера к клиенту, – его нельзя сделать стопроцентно отказоустойчивым и надежным с точки зрения сохранности сообщений. Тем не менее кое-какие возможности у нас есть.
  - Мы можем устроить так, чтобы вновь поступающие сообщения не передавались клиенту, если сетевой буфер заполнен. Это показывает, что клиент не успевает читать сообщения. Сам протокол не позволяет клиенту сообщить об этом серверу API; мы вынуждены полагаться на косвенную информацию от сетевой подсистемы ОС.

- Поскольку между клиентами и потоковым API существует постоянное соединение, важно следить за тем, чтобы единственный сервер не захлебнулся. Поэтому следует придумать, как балансировать нагрузку.
- Чтобы потоковый API не пропускал сообщения при отправке клиенту, нужно гарантировать, что при отказе одного сервера API другой сервер сможет продолжить отправку сообщений с того места, где произошел сбой. У этого решения есть две стороны: во-первых, потоковый API должен организовать распределенный механизм отслеживания идентификаторов последних сообщений, отправленных каждому клиенту. В случае отказа исправный сервер подхватывает работу с того места, где вышедший из строя остановился. Во-вторых, клиент может оказать посильную помощь, посылая идентификатор последнего полученного им сообщения в момент переподключения. Сервер API сможет использовать эту информацию для отправки последующих сообщений.

В целом этот протокол демонстрирует более высокую производительность и позволяет создать более эластичный API. Постепенно он идет на смену длинному HTTP-опросу, так что имеет смысл присмотреться к нему, если характеристики отказоустойчивости и надежности вас устраивают.

Наконец, мы подошли к протоколу веб-сокетов, обладающему большей гибкостью, чем все рассмотренные выше.

7.2.4. Веб-сокеты

Протокол WebSocket существует с 2011 года. Это полнодуплексный протокол, в котором транспортным механизмом является TCP. Его поддерживают все основные браузеры для настольных и мобильных устройств. Хотя обычно он используется для организации диалога между веб-клиентами и веб-серверами, для всех популярных языков программирования имеются библиотеки, открывающие любопытные дополнительные возможности. WebSocket – интересный протокол в том смысле, что в нем на этапе начального согласования используется HTTP, после чего отправляется запрос перехода на другой протокол и происходит переключение на TCP. На рис. 7.10 показан поток управления в WebSocket.

Сделаем несколько замечаний по поводу этого рисунка.

1. Этот начальный обмен данными между клиентом и сервером – согласование – осуществляется по протоколу HTTP. Клиент инициирует процедуру согласования, посылая запрос Upgrade. Сервер отвечает, завершая согласование, после чего производится переход с HTTP на TCP.

2. Все возникающие события сервер отправляет клиенту по TCP.
3. Запрос «притормози» присутствует, потому что протокол двунаправленный, и клиент в любой момент может послать сообщение серверу. Содержимое и семантика сообщения определяются разработчиком. В данном случае мы можем предположить, что клиент не успевает читать данные и просит сервер уменьшить частоту отправки сообщений.
4. В какой-то момент клиент или сервер захочет закрыть соединение. В принципе, возможно неожиданное закрытие, но правильная процедура подразумевает согласование закрытия. На рис. 7.10 закрытие инициирует клиент, но это может делать и сервер. Вне зависимости от того, какая сторона выступает инициатором, порядок один и тот же: запрос согласования закрытия, на который следует ответ согласования закрытия. После этого транспортное TCP-соединение закрывается.

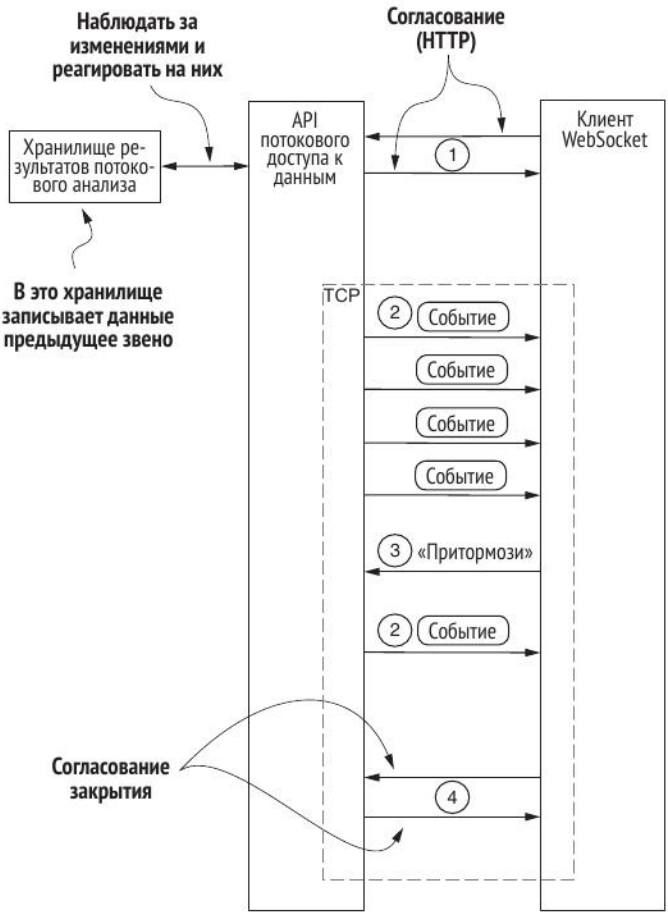


Рис. 7.10. Поток управления в WebSocket – клиент отправляет сообщение

См. код в листинге 9.3.1

Как нетрудно видеть, у этого протокола много аспектов. Посмотрим, что можно сказать об интересующих нас факторах.

- *Частота сообщений.* Учитывая, что все данные передаются по протоколу TCP, частота сообщений в этом протоколе может быть намного выше, чем во всех рассмотренных выше. Как и в случае SSE, увеличение отчасти связано с наличием постоянного состояния, но важную роль играет и сам протокол. И риски те же, что в SSE: необходимо, чтобы клиент успевал справляться с темпом отправки сообщений. Искусственное замедление темпа передачи (дросселирование) в протоколе не предусмотрено, но на рис. 7.10 показано, что протокол позволяет реализовать такую семантику.
- *Направление взаимодействия.* В отличие от предыдущих протоколов, этот является двунаправленным от начала до конца, что позволяет создавать интересные подпротоколы.
- *Задержка сообщений.* Все взаимодействие производится по протоколу TCP, так что задержка мала по сравнению с протоколами на основе HTTP.
- *Эффективность.* Все данные передаются по TCP, и существует всего одно постоянное соединение между клиентом и сервером, поэтому протокол очень эффективен.
- *Отказоустойчивость и надежность.* К сожалению, этот протокол, как и все остальные, не дает никаких гарантий. Но, в отличие от других, он двунаправленный, что позволяет надстроить над ним семантику отказоустойчивости и надежности. Для этого нужно сделать следующее.
  - Если мы получаем события и пытаемся отправлять их клиенту быстрее, чем клиент способен их читать, то следует организовать буферизацию, а не писать сразу. Благодаря двунаправленности протокола клиент может играть активную роль и посылать серверу сообщение (что-то типа «притормози») с просьбой уменьшить темп передачи.
  - Как и в случае SSE, чтобы исключить потерю сообщений, мы должны следить за тем, какие сообщения были отправлены каждому клиенту. Тогда в случае отказа сервера или переподключения клиента следующим будет отправлено очередное сообщение, и разрыва в последовательности сообщений не возникнет. В протоколе нет встроенных средств поддержки такого поведения, но их нетрудно добавить. Преимуществом веб-сокетов над SSE в этом отношении является тот факт, что, приложив небольшие усилия, мы сможем сделать так, что клиент будет посылать потверждение того, что не только получил,

но и обработал сообщение. Это позволяет построить систему, обладающую высокой отказоустойчивостью и надежностью.

После данного обсуждения не вызывает удивления тот факт, что этот протокол более эффективен, гибок и функционален, чем все остальные, и вот почему:

- на протяжении всего времени, пока клиент желает потреблять данные, между ним и сервером открыто единственное TCP-соединение;
- взаимодействие между клиентом и сервером производится не по протоколу HTTP, что снижает накладные расходы;
- взаимодействие двунаправленное, т. е. клиент и сервер могут обмениваться сообщениями по одному соединению, не открывая дополнительных;
- мы можем реализовать отказоустойчивость и гарантировать доставку сообщений;
- благодаря низкому уровню протокола потоковый API и клиенты могут обеспечить более высокую пропускную способность (измеряемую количеством сообщений в секунду);
- протокол дает возможность разрабатывать специализированные подпротоколы, в которых стороны обмениваются сообщениями после установления соединения.

В потоковых системах, где клиенты могут работать по протоколу HTTP, WebSocket быстро становится одним из самых распространенных протоколов. Именно его следует выбирать, если вы хотите предложить отказоустойчивый и надежный API. Как уже отмечалось, двунаправленный протокол позволяет управлять обеими взаимодействующими сторонами: клиентом и сервером. Если же вы создаете систему, в которой клиенты с ограниченными ресурсами не поддерживают HTTP, то стоит обратить внимание на два других протокола: Data Distribution Service (DDS) и MQ Telemetry Transport (MQTT). Оба проектировались скорее для модели издатель-подписчик, но их можно использовать и для работы с потоком данных.

Мы довольно подробно рассмотрели все протоколы. Итог обсуждению подведен в табл. 7.1, которая может играть роль руководства, если вы захотите сравнить протоколы.

Таблица 7.1. Краткое сравнение различных протоколов

Протокол	Частота сообщений	Направление взаимодействия	Задержка сообщений	Эффективность	Отказоустойчивость и надежность
Веб-уведомления	Низкая	Однонаправленный (от сервера к клиенту)	Средняя	Низкая	Нет

Протокол	Частота сообщений	Направление взаимодействия	Задержка сообщений	Эффективность	Отказоустойчивость и надежность
Длинный HTTP-опрос	Средняя	Двунаправленный	Средняя	Средняя	Нет
События, отправляемые сервером	Высокая	Однонаправленный	Низкая	Высокая	По умолчанию нет. Существует возможность частичной реализации
WebSocket	Высокая	Двунаправленный	Низкая	Высокая	По умолчанию нет. Существует возможность полной реализации

7.3. Фильтрация потока

Прежде чем переходить к обсуждению различных аспектов и нюансов фильтрации, я хотел бы определить, что понимаю под *фильтрацией*. В контексте потокового API фильтрация – это способность ограничивать множество порождаемых событий и свойства этих событий только теми, что представляют интерес для потокового клиента. Это не совсем то, что понимается под фильтрацией, или редукцией, о которой мы говорили при обсуждении потоковых алгоритмов. При рассмотрении фильтрации потока необходимо понимать ряд вещей.

- *Где производится фильтрация* – в звене анализа, потокового API или клиента (звено клиента – тема следующей главы).
- *Тип фильтрации* – статическая или динамическая.

7.3.1. Где производится фильтрация

Принимая решение о том, где производить фильтрацию, следует учитывать несколько факторов. Если к потоку применяется агрегирование или другие потоковые алгоритмы, то вся фильтрация должна быть сосредоточена в звене анализа. Допустим, нас интересует часовое энергопотребление, а также потребление накопительным итогом для всех домов в Чикаго с выдачей раз в 30 секунд. С учетом всего того, что мы уже знаем, выполнять такое вычисление удобно в звене анализа, и это самое подходящее место для фильтрации. В некоторых случаях, когда число событий в секунду мало, отдельное звено анализа оказывается излишним, и тогда фильтрацию можно производить в звене потокового API. Но помните, что по мере роста трафика или усложнения вычислений может понадобиться добавить звено анализа. Пример ситуации, когда фильтрация в звене потокового API имеет смысл, возникает, когда порождаемые сообщения совпадают с исходными или лишь немного отличаются от них.

Допустим, имеется поток координат GPS и метаданных о транспортных средствах для всех автомобилей на всех автострадах США. Потоковый клиент мог бы отфильтровать из этого потока только события, имевшие место на определенной территории или характеризующиеся какими-то общими метаданными – скажем, маркой и моделью. Фильтрация такого типа уместна в звене потокового API. По мере поступления данных мы будем отбрасывать все события, не удовлетворяющие критерию.

В этой связи отметим важный момент: пользователь (потоковый клиент) может быть заинтересован как в фильтрации событий целиком, так и в фильтрации отдельных свойств событий. Если это напоминает вам SQL, то вы абсолютно правы: так это и следует интерпретировать. Команда SQL `select` позволяет с помощью фразы `where` отобрать только нужные строки и оставить лишь представляющие интерес столбцы. Это именно то, что хочет пользователь; концептуально реляционная таблица становится потоком, а столбцы – свойствами событий.

В следующем разделе мы рассмотрим два типа фильтрации.

7.3.2. Статическая и динамическая фильтрация

Фильтрация может быть статической и динамической. В случае *статической* фильтрации решение о том, что должен содержать поток, принимается заранее. Можно считать, что речь идет о фиксированном потоке. Что он собой представляет, решает разработчик или архитектор платформы, а клиент не может изменить правила фильтрации. Можно провести аналогию с представлениями в реляционной базе данных: представление определяется проектировщиком, пользователь не может его изменить.

Динамическая фильтрация определяется на этапе выполнения, ей управляет потоковый клиент. Это можно сравнить с произвольным запросом к таблице реляционной базы данных. Только в мире потоков «таблицей» будет объединение всех схем событий в потоке, что дает потоковому клиенту весьма широкие возможности. Но еще нужно определить, как реализовать эти подходы протоколе, выбранном для потокового API.

В табл. 7.2 приведены соображения по поводу интеграции различных типов фильтрации с рассмотренными выше протоколами.

Таблица 7.2. Интеграция фильтрации с различными протоколами

Протокол	Динамическая фильтрация	Статическая фильтрация
Веб-уведомления	При регистрации оконечной точки необходимо указывать используемый запрос	При регистрации оконечной точки необходимо указывать используемый запрос
Длинный HTTP-опрос	Необходимо каким-то образом закодировать запрос в URL; наверное, самое простое – представить фильтр самим URL	Необходимо каким-то образом закодировать запрос в URL, быть может, с помощью параметров запроса

Протокол	Динамическая фильтрация	Статическая фильтрация
События, отправляемые сервером	Необходимо каким-то образом закодировать способ фильтрации в URL при вызове конструктора EventSource, быть может, так же, как в длинном HTTP-опросе	Так же, как в длинном HTTP-опросе, у пользователя должен быть способ задать запрос в параметрах URL при конструировании EventSource
WebSocket	Этот протокол обеспечивает максимальную гибкость. Например, после подключения клиент может послать сообщение, описывающее фильтрацию; в статическом случае это может быть просто имя	После подключения клиент может послать сообщение, описывающее запрос для фильтрации

Как видим, в каждом протоколе найдется способ сообщить о том, как фильтровать, – разница только в степени гибкости. Проектируя потоковый API, не забывайте, что многие разработчики и пользователи бизнес-приложений знакомы с языком SQL. Поэтому подумайте, как можно воспользоваться этими знаниями и предложить SQL-подобный синтаксис для задания фильтрации. Существуют разные способы добавить SQL-подобные средства в API. Если конвейер создается из продуктов, написанных на JVM-совместимых языках, то можно применить Apache Calcite (https://calcite.apache.org). Даже если используются другие языки, этот проект может подсказать кое-какие идеи для разбора SQL на выбранной платформе.

7.4. Пример: построение потокового API для сайта Meetup

Давайте-ка выйдем на свежий воздух – применим все, чему научились, к конкретному примеру. Допустим, что мы строим потоковую систему, в которой источником данных будут приглашения сайта Meetup.com. Этот источник можно увидеть в действии по адресу http://stream.meetup.com/2/rsvps – поступает непрерывный поток данных в формате JSON, записи создаются, когда кто-то вводит приглашение посетить встречу.

Мы хотим, чтобы пользователь мог подключиться к нашему потоковому API и отфильтровать данные по названию темы. Поскольку предмет этой главы – потоковый API, сосредоточимся на том, как сделать данные доступными пользователям.

Во второй части книги, где мы будем разрабатывать реальную систему, придется подумать и о том, как потреблять данные из хранилища результатов анализа. Но пока нас интересует только выбор способа взаимодействия и протокола. На рис. 7.11 показано, чего мы хотим добиться.

Рис. 7.11. Диаграмма последовательности для потового API сайта Meetup с фильтрацией при подключении

Происходящее на рис. 7.11 начинается, когда клиент подключается к нашему потоковому API и передает фильтр. Наш API наблюдает за появлением новых данных в хранилище результатов анализа и отправляет новые приглашения (RSVP-события) клиенту. Мы должны ответить на два вопроса: на каком паттерне взаимодействия остановиться и какой протокол использовать? В табл. 7.3 и 7.4 перечислены варианты того и другого с анализом их применимости в данном случае.

Таблица 7.3. Какой паттерн взаимодействия выбрать?

Паттерн взаимодействия	Подходит?	Комментарии
Синхронизация данных	Нет	В этом случае мы не передаем полный набор данных, а данные поступают непрерывно. Поэтому паттерн синхронизации данных не годится
RMI/RPC	Нет	Применение RMI/RPC было бы неудачным решением, потому что мы хотим передать поступающие данные потребителю, а не вызывать метод потребителя при каждом новом событии
Простой обмен сообщениями	Может быть	Этот паттерн – не лучшая идея, потому что мы хотим не просто вернуть последние данные, а еще и применить к ним фильтр. Наверное, это можно было бы сделать, но с непомерными накладными расходами на стороне сервера API

Паттерн взаимодействия	Подходит?	Комментарии
Издатель-подписчик	Да	А этот паттерн вполне подходит. Мы еще и выиграем в эффективности благодаря протоколам, которые с ним совместимы, да и модель точно соответствует характеру движения данных

Таким образом, на роль кандидата в этом примере претендуют два паттерна взаимодействия. Теперь посмотрим на протоколы.

Таблица 7.4. Какой протокол выбрать?

Протокол	Подходит?	Комментарии
Веб-уведомление	Нет	Накладные расходы будут слишком велики, поскольку потоковому API нужно будет отправлять POST-запрос зарегистрированному обратному вызову при каждом новом событии. Не пойдет
Длинный HTTP-опрос	Да	Этот протокол будет неплохо работать, поскольку фильтр задается на этапе подключения клиента. Но нужно будет дать клиенту возможность указать в параметрах URL желаемый способ фильтрации результатов
События, отправляемые сервером	Да	Этот протокол будет работать с той же оговоркой, что в случае длинного HTTP-опроса: передавать фильтр в параметрах запроса при подключении
Веб-сокеты	Да	И этот протокол – хороший выбор. Как и в других случаях, у клиента должен быть способ передать фильтр. Отличие в том, что этот протокол допускает задание фильтра как в параметрах запроса, так и в сообщении, которое клиент посылает, когда инициирует соединение

Итак, существует несколько возможностей построить потоковый API, отвечающий нашим требованиям. Решая, на каком паттерне взаимодействия и протоколе остановиться, принимайте во внимание несколько факторов. Во-первых, подумайте о том, какие клиенты вы собираетесь поддерживать и как должна производиться фильтрация. Чтобы охватить максимально широкий круг клиентов, можно выбрать длинный опрос, события, отправляемые сервером и WebSocket. Но если важнее, чтобы критерий фильтрации можно было задавать уже после установления соединения, тогда надо использовать более гибкий протокол, например WebSocket.

7.5. Резюме

В этой главе мы рассмотрели паттерны взаимодействия и протоколы, применяемые для передачи данных потоковому клиенту. Мы также обсудили фильтрацию потока данных. Мы не стали заострять внимание на реализа-

ции отказоустойчивости, потому что в этом отношении все обстоит так же, как описано в предыдущих главах.

Вот какие уроки следует запомнить:

- важно внимательно изучать требования и выбирать соответствующие им паттерны взаимодействия и протоколы. Не существует одного рецепта на все случаи жизни;
- выбирая паттерн взаимодействия и протокол, обращайте особое внимание на требования к отказоустойчивости и надежности, поскольку от них в значительной мере зависит выбор;
- не забывайте о поддержке статической и (или) динамической фильтрации. Поначалу это можно упустить из виду, но фильтрация – вещь, которая очень скоро понадобится вашим клиентам.

Некоторые протоколы могут не в полной мере отвечать стоящей перед вами задаче, но надеюсь, что, прочитав эту главу, вы не встретите затруднений при изучении других протоколов и будете понимать, как оценивать новый протокол.

Глава 8

Возможности конечных устройств и ограничения доступа к данным

Краткое содержание главы:

- принципы построения потокового клиента;
- введение в веб-клиенты;
- трудности опроса потока.

Хотите верьте, хотите нет, но мы почти закончили путешествие, начавшееся получением данных в звене сбора и продолжившееся их перемещением, анализом и подготовкой к потреблению. И теперь, наконец, надо с этими данными что-то сделать. На рис. 8.1 показана общая архитектура с акцентом на теме этой главы.

Для многих технологии, описанные в этой главе, со всеми их идеями и бессчетными решениями, и есть то место, где потоковая система оживает. Раньше мы говорили о технологиях «заднего плана» – что там происходит, конечные пользователи и заказчики, скорее всего, никогда не увидят. А сейчас мы подобрались к «последней миле»: передаче данных от потокового API клиенту. Мы будем говорить о том, как предпринять действия на основе потока данных; это может быть мобильное приложение, веб-браузер, торговый автомат, уборочный комбайн, какая-то другая система на предприятии – да вообще любое подключенное к сети устройство.

Возможности, открывающиеся на этом уровне, многообразны и увлекательны – почти все, что только можно себе представить. Вы можете сконструировать информационную панель, на которой показывается все, что происходит в бизнесе, давать в реальном времени рекомендации клиентам по выбору товаров, критически анализировать только что выполненный

удар в гольфе или корректировать маршрут и скорость движения локомотива с целью экономии топлива. Идеи и технологии, благодаря которым все это возможно, и составляют содержание настоящей главы. Во введении к главе 7 я говорил, что рассматриваемое в ней звено – мое любимое. С точки зрения профессионала в области потоковой обработки данных, так оно и есть. Но именно продукты, применяемые при разработке данного звена, и результаты их работы ярко демонстрируют мощь потоковой системы. Благодаря им можно наконец вкусить плоды тяжелого труда.

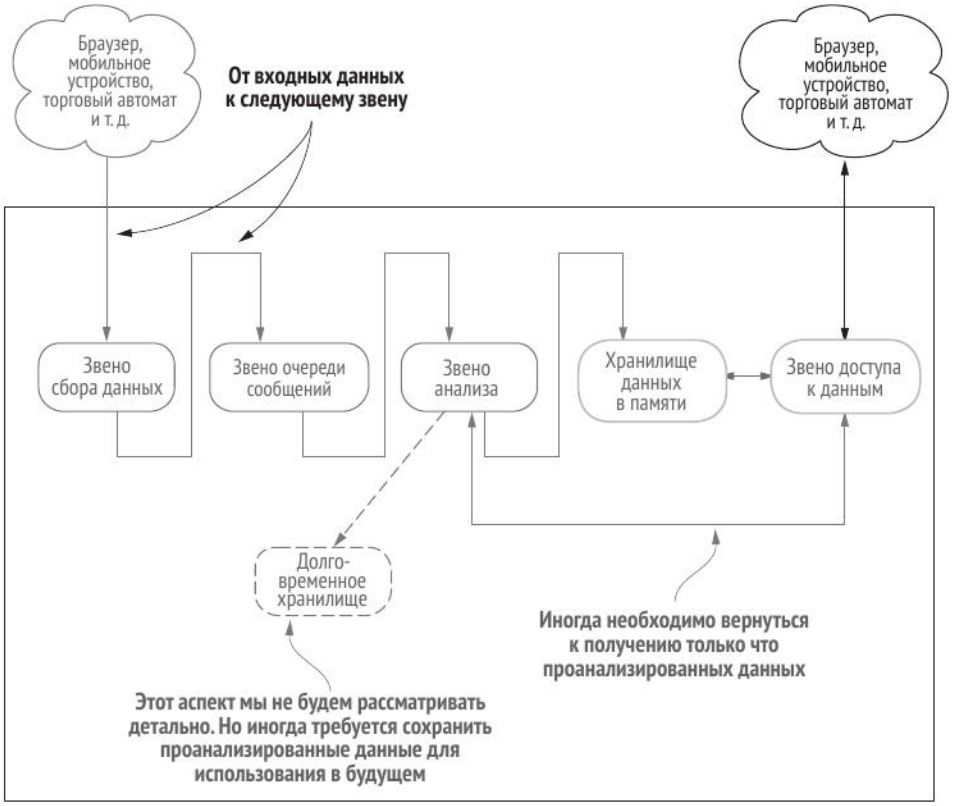


Рис. 8.1. Архитектура потоковой обработки данных – выделена тема этой главы

Возникает искушение рассмотреть все имеющиеся на сегодняшний день клиенты. Но ландшафт меняется так быстро, что пытаться охватить все разнообразие бессмысленно. Гораздо важнее понять основные современные концепции, которые останутся актуальными даже с учетом сногсшибательного темпа эволюции клиентских устройств. Мы сосредоточимся на концепциях и не станем обсуждать технические характеристики представленных на рынке устройств. А затем рассмотрим несколько примеров клиентов, воплощающих эти концепции.

Но хватит слов, наливайте себе кофе – и приступим.

8.1. Основные концепции

Все бесконечное разнообразие продуктов в этом звене можно отнести к одной из трех категорий, показанных на рис. 8.2.

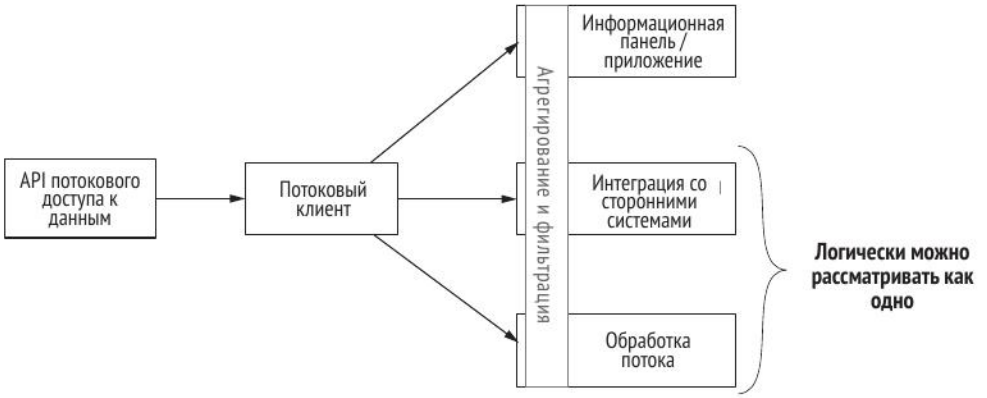


Рис. 8.2. Три категории клиентских приложений

На рис. 8.2 не указаны конкретные протоколы или паттерны взаимодействия между API потокового доступа к данным и потоковым клиентом. Протокол будет важен, когда мы перейдем к детальному обсуждению концепций, а пока о нем можно не думать. Как видим, все категории приложений пересекают агрегирование и фильтрация, потому что независимо от того, какого рода клиентское приложение мы разрабатываем, может возникнуть необходимость в агрегировании и фильтрации потока. Иногда это делается на стороне клиента, но чаще в самой потоковой системе. К этому вопросу мы еще вернемся.

Прежде чем двигаться дальше, я кратко опишу категории на рис. 8.2.

Конечное приложение или пользовательский интерфейс

Смысл этой категории (обозначенной на рисунке «Информационная панель / приложение») в том, что имеется приложение – а точнее, компонент приложения, – частью работы которого является потребление потока. Приложения этого класса чрезвычайно разнообразны: тут и простые информационные панели в окне браузера (ладно, признаю, что они могут быть весьма сложными), и социальные чаты или мобильные мессенджеры, и даже многопользовательские игры. В общем случае это приложения, которые напрямую подключаются к потоку и все манипуляции с данными производят на стороне клиента.

Интеграция со сторонними приложениями и потоковыми процессорами

Логически интеграцию со сторонними приложениями и потоковыми процессорами можно объединить в одну категорию. Все такие приложе-

ния устроены единообразно: потребляют поток, реализуют какую-то бизнес-логику и затем взаимодействуют с другой системой. Эта система может быть внешней по отношению к организации или частью более крупного приложения, полностью контролируемого вами. Например, ваша система может потреблять поток данных о банковских транзакциях и передавать их внешней системе обнаружения мошенничества для анализа. Частично вы можете обрабатывать данные самостоятельно, но, вообще говоря, потребляете поток и отправляете данные системе, которую не контролируете. Другой пример взаимодействия со сторонней системой – разработка системы, которая принимает поток данных о поведении пользователей, – например, открытии писем с маркетинговыми предложениями или показах рекламных объявлений, – а затем вычисляет оценку расположенности посетителя к покупке, обновляемую далее на SalesForce.com.

Нередко вам придется создавать потоковые приложения, взаимодействующие только с другими системами внутри организации. Например, это может быть распределенная система, состоящая из небольших взаимодействующих между собой служб. Один из способов построить ее – сделать все службы производителями и потребителями потока. Каждая будет «прослушивать» поток, получая из него данные для работы, что-то делать с ними и отправлять результаты обратно в поток, откуда их подхватит другая служба.

Мы упомянули эти две категории не просто для того, чтобы вам было куда отнести свое приложение, а чтобы иметь систему ориентиров при рассмотрении различных ключевых концепций. Вот и займемся ими. Для каждой концепции поговорим о стратегиях, применяемых при потреблении стороннего API и в том случае, когда это ваш собственный API. В последнем случае обсуждение стратегии может привести к внутренним дискуссиям о том, как изменить потоковый API в соответствии с изложенным в главе 7, и это совершенно нормально. Вполне естественно, что существует некоторая напряженность между двумя звеньями архитектуры, требующая обсуждения, в котором участвовали бы разрабатывающие их группы и вырабатывалось общее понимание.

8.1.1. Достаточная скорость чтения

Скорость чтения, – возможно, не первое, о чем вы думаете при построении потокового клиента, но зачастую это один из важнейших вопросов. Задача обеспечения быстрого чтения клиентом имеет две стороны: потоковый API и потоковый клиент. Поскольку эта глава посвящена потоковым клиентам, то начнем с рассмотрения данного аспекта.

Почему быстрое чтение так важно для API? Все дело в потере данных (и это самое главное) и в эффективном использовании ресурсов сервера. Тут многое зависит от технологии доставки потока от API. Мы подробно

остановимся на двух популярных методах, обсуждавшихся в главе 7: события, отправляемые сервером, и веб-сокеты. Они похожи на обеих сторонах: клиентской и серверной. На рис. 8.3 показана характерная для обоих подходов ситуация, когда клиент читает недостаточно быстро.

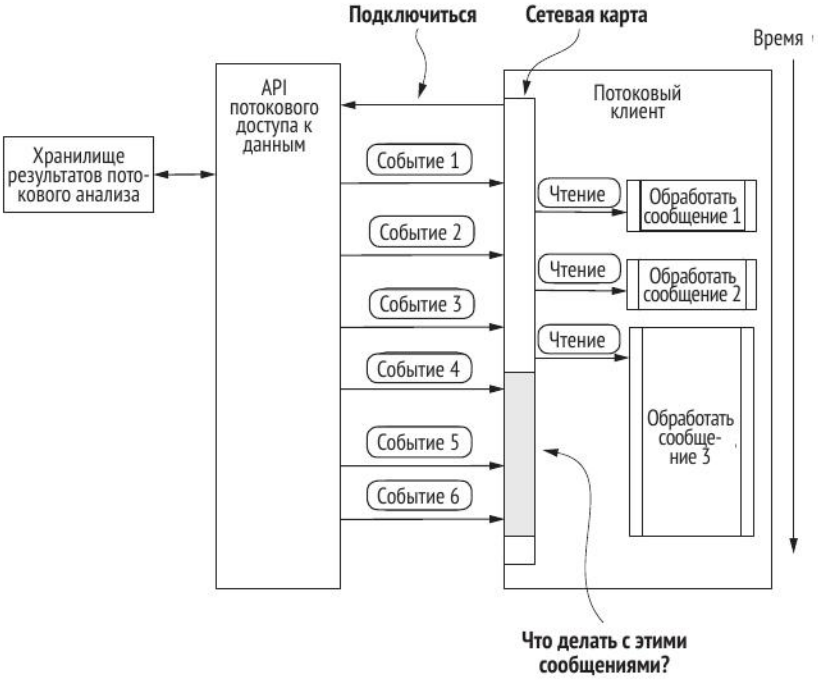


Рис. 8.3. Пример медленного клиента для двух паттернов взаимодействия – события, отправляемые сервером, и веб-сокеты

Как видим, клиент смог обработать первые два сообщения до отправки следующего. Но обработка третьего сообщения заняла слишком много времени, поэтому потоковому API пришлось решать, что делать с сообщениями 4–6. Придержать их в памяти или слепо отправить? В данном примере хранение в памяти вполне допустимо – ведь речь идет всего о трех сообщениях. Слепая отправка тоже может быть выходом, хотя мы должны хорошо понимать технологию, применяемую при организации потокового API. Что, если сообщения отбрасываются, когда сетевые буферы заполнены? В таком случае данные могут быть потеряны. Примем, что потоковый API хранит сообщения в памяти.

Когда сообщений всего два, может показаться, что проблема не стоит выеденного яйца. Ну а если скорость потока составляет 1000 сообщений в секунду и клиент не успевает? Разработчик API поставлен перед дилеммой: допустить потенциальное отбрасывание сообщений, чтобы не исчерпать ресурсы сервера, или применить противодавление (back-

pressure) к предшествующим системам. Чтобы разрешить эту ситуацию без ущерба для системы, разработчик API может уведомить клиента о том, что тот не справляется. К сожалению, не все сторонние API предоставляют такую возможность, но если вы сами разрабатываете потоковый API, то настоятельно рекомендую включить ее и предлагать своим клиентам.

Обратимся теперь к стороне клиента. Здесь нужно ответить на три важных вопроса:

- откуда мне знать, что я читаю недостаточно быстро?
- что произойдет, если я об этом не узнаю?
- как мне масштабировать клиента, чтобы он успевал за потоком?

Ответы на эти вопросы зависят от того, какой API потребляется: сторонний или разработанный внутри организации.

Сторонний потоковый API

Разные API могут по-разному уведомлять о том, что вы не поспеваете за потоком, и последствия также могут быть различны. Например, на момент написания этой книги API Твиттера отключал клиентов, которые отстали слишком далеко. Твиттер не объясняет, что значит «слишком далеко», но посылает в этом случае *предупреждение о приближении к срыву*. Если клиент не поспевает, то Twitter API посылает такое предупреждение раз в пять минут. Сколько раз оно будет послано до отключения клиента, в документации не сказано.

Что можно сделать, если потребляемый API не посылает предупреждений и не предлагает никакого другого механизма, позволяющего узнать, что вы не поспеваете? Одна из возможных стратегий – читать временные метки полученных сообщений и сравнивать их с текущим временем. Если расхождение начинает расти, значит, можно предположить, что клиент отстает. Но для этого служба-производитель должна снабжать все сообщения временными метками. Если это не так, то единственное, что остается, – попробовать определить ожидаемую скорость потока. Некоторые потоки по природе своей эпизодические, в таких случаях можно выявить некоторую закономерность и на ее основе строить гипотезы о том, поспевает ли клиент за потоком.

И тут мы подходим ко второму вопросу: что произойдет, если мы не поспеваем? Вопрос интересный, но ответ на него, к сожалению, зависит от потребляемого потокового API. Так, API Твиттера ясно говорит, что соединение будет закрыто, если потребитель не поспевает за потоком. Другие API могут не закрывать соединение, а отбрасывать данные, которые клиент не успевает прочитать. Разрабатывая клиента, потребляющего данные от стороннего потокового API, не забудьте задать этот вопрос.

Собственный потоковый API

Если вы полностью контролируете все части системы, то позаботьтесь о том, чтобы потоковый API недвусмысленно давал знать, что произойдет с данными и (или) соединением, если потребитель не поспевает. Можете включить в поток сообщения о состоянии и протоколировать отправку таких сообщений в журнале, обязательно описав все это в документации. Сообщение о состоянии должно содержать информацию о том, на сколько отстал клиент, и предупреждение о закрытии соединения, если клиент не догонит поток. Все мы знаем, что документацию читают не всегда, так что включение необходимых данных прямо в реализацию позволит клиентам вовремя реагировать на изменения и сделает ваш потоковый API более удобным для работы. Но имейте в виду, что это возлагает дополнительную нагрузку на клиента, который и так уже испытывает трудности, поскольку вы заставляете его обрабатывать еще один тип сообщения. Отправку таких сообщений обязательно следует протоколировать в журнале, чтобы впоследствии проанализировать и использовать для отладки медленных потребителей или проблем в вашем же потоковом API.

8.1.2. Запоминание состояния

Обработка состояния в клиентской части потокового приложения может принимать разные формы. Во-первых, клиент может запомнить, где остановился, сохранив метаданные о последнем обработанном сообщении, чтобы, когда пользователь снова запустит приложение, можно было начать со следующего за ним. Можно также сохранять актуальные результаты потокового вычисления на стороне клиента. Это даст возможность регулярно обновлять пользовательский интерфейс, если новые данные обрабатываются в фоновом режиме. Это основательные причины хранить состояние работы на стороне клиента, они относятся к разным ситуациям и вряд ли окажут заметное влияние на клиента.

Но не следует заходить в этом отношении слишком далеко, заставляя клиента производить потоковые вычисления с получаемыми данными. Конечно, без каких-то вычислений не обойтись. Я бы оставил по клиентскую сторону от черты все вычисления, связанные с окном. И хотя эта черта проведена на песке, она все же представляется полезным ориентиром в случае, когда объем и темп поступления данных невелики. Но нет сомнений, что со временем поток данных будет расти, – иначе зачем бы вкладывать средства в разработку потоковой системы. Возможно, вы подумали о потоках, рост которых нежелателен. Например, никто не хочет, чтобы объем или скорость потока данных о нештатных ситуациях на производственном предприятии возрастали. Но не говорите опрометчиво: «Да ладно, уж для этого-то потока можно производить вычисления на стороне клиента». Один раз ступив на этот путь, вы получите хрупкую, пло-

хо согласованную кодовую базу, и хуже того – клиент начнет сбоить, если поток все-таки возрастет. Увы, сказать «баста, никакого агрегирования и вычислений на стороне клиента» проще, чем сделать. В реальности иногда приходится и состояние запоминать, и вычисления производить в клиентской части потокового приложения. Поэтому поговорим о том, как к этому подойти правильно.

Учитывая невероятное изобилие способов реализации клиентского ПО, мы покажем только, как сохранять состояние в клиентах на основе браузера. И хотя при этом за кадром остаются различные клиентские устройства, общие принципы остаются в силе. Какая бы технология ни применялась при разработке клиента, в случае объемного скоростного потока нужно обращать особое внимание на структуру данных и потребление памяти. Мне встречались реализованные внутри браузера клиенты, которые со временем исчерпывали всю память браузера, что приводило к его краху. А все дело в том, что они выполняли вычисления, которые следовало бы оставить платформе потокового анализа.

При построении потокового клиента внутри браузера для сохранения состояния есть две возможности: веб-хранилище и база IndexedDB. Веб-хранилище делится на две части: локальное и сеансовое. Оно предназначено для хранения пар ключ-значение способом, интуитивно более понятным, чем куки. В локальном хранилище данные остаются даже после закрытия браузера. Сеансовое хранилище разделено в соответствии с доменами, для которых хранятся данные, и уничтожается после закрытия сеанса.

У веб-хранилища есть ограничения. Во-первых, порядок хранения ключей определяется пользовательским агентом. Поэтому нельзя рассчитывать на то, что доступ к сохраненным данным из потока будет производиться в том же порядке, в каком они были получены. Но если вы сохранили 10 лучших событий, то можете прочитать их и отсортировать в памяти. Во-вторых, каждому домену в веб-хранилище отведено всего 5 МБ. Это не жесткое правило, которому следуют все браузеры, а рекомендованное ограничение. Подробнее о нем можно прочитать в спецификации W3C по адресу http://www.w3.org/TR/webstorage/.

Второй вариант хранения, база IndexedDB, продолжает с того места, на котором веб-хранилище остановилось. Она предназначена для чтения ключей по порядку, эффективного поиска значений, хранения нескольких значений для одного ключа и упорядоченного обхода большого числа записей. Подробнее см. спецификацию по адресу http://www.w3.org/TR/IndexedDB/.

Для хранения состояния в браузере подойдет любая технология, поскольку состояние должно быть невелико по размеру. Но если вы окажетесь в ситуации, когда на стороне клиента необходимо выполнять вычисления или агрегирование, то в качестве механизма хранения я рекомендую использовать IndexedDB. Однако повторю еще раз: по возможности пере-

носите вычисления в потоковый API. Если потоковый API разрабатываете вы сами, подумайте о том, как производить в нем вычисления в интересах клиентов и как снять с них заботы об управлении состоянием. Ваша цель – по возможности освободить клиента от всякого состояния. Возможно, вы думаете, что собираетесь использовать на стороне клиента Java или Node.js, так что беспокоиться не о чем. Быть может, вам и не нужно «сражаться» с браузерными технологиями, но в любом случае следует помнить, что при потреблении высокоскоростного потока клиент не должен хранить слишком объемное состояние или пытаться выполнить трудоемкие вычисления. При определенной скорости потока вы все равно исчерпаете ресурсы клиентской среды. Вот почему я всегда – вне зависимости от клиентской технологии – ратую за то, чтобы перенести вычисления в звено потокового API и стремиться к созданию клиентов без состояния.

8.1.3. Смягчение последствий потери данных

На рис. 8.4 показано, где возможна потеря данных.

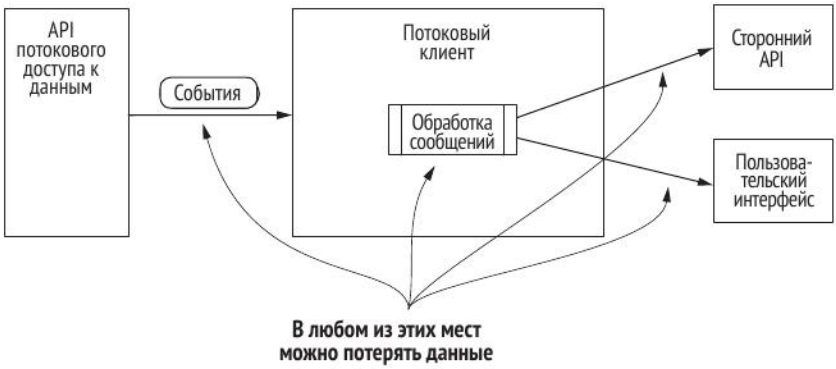


Рис. 8.4. Потоковый клиент может терять данные при получении, обработке и отправке

Мы видим четыре места, в которых возможна потеря данных. Во-первых, при отправке потоковым API, во-вторых, на стадии обработки и, наконец, когда мы сами отправляем данные пользовательскому интерфейсу или стороннему API. Уменьшить шансы потери данных в каждом месте можно двумя способами. Мы сосредоточимся на получении и отправке данных, а не на обработке, поскольку потеря данных на стадии обработки – почти наверняка результат программной ошибки. И пользовательский интерфейс мы тоже исключим из рассмотрения, поскольку он находится целиком под нашим контролем. Таким образом, остаются только получение данных и отправка после обработки.

Нас будут интересовать клиенты, реализованные вне браузера. Представим, что мы потребляем поток заказов на поставку и должны отправить их системе ввода заказов. Интеграция такого типа часто осуществляется

с помощью специальных клиентских программ, а не браузера. В данном случае мы хотим гарантировать, что данные не будут потеряны на пути от потокового API и что сторонняя система получит новое сообщение. Для этого воспользуемся идеями, о которых говорили в главе 2: протоколирование сообщений на стороне получателя (RBML) и протоколирование сообщений на стороне отправителя (SBML). В случае совместного использования RBML и SBML обычно реализуются с помощью гибридного протоколирования сообщений (HML), рассмотренного в разделе 2.3.3. Применим эти идеи к нашей задаче и посмотрим, удастся ли уменьшить вероятность потери данных. На рис. 8.5 показано, куда следует поместить части HML, связанные с получателем и отправителем.

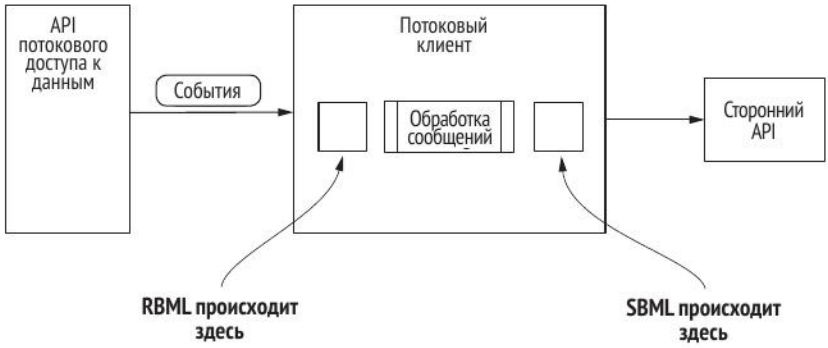


Рис. 8.5. Применение HML для уменьшения вероятности потери данных

Принцип работы следующий: сначала мы реализуем получательскую часть HML – сразу после получения сообщения от потокового API. Затем сообщение обрабатывается, и до отправки стороннему API реализуется отправительская часть HML. Все довольно просто, только вот как узнать, когда нужно удалять сообщения из обоих журналов? На стороне получателя это можно сделать, после того как сообщение успешно обработано и записано в журнал отправителя. А как удалить сообщение из журнала отправителя? Для этого в нашем распоряжении имеются два метода: подтверждение и паттерн записать-и-прочитать. Если сторонний API возвращает нам код успешного получения данных, то это можно расценивать как подтверждение и удалить запись из журнала. Это лучшее, на что можно надеяться, – архитектурно чистый и естественный способ программирования клиента.

К сожалению, не все сторонние API поддерживают такие ответы. Тогда остается еще паттерн записать-и-прочитать, когда мы отправляем сообщение, а затем пытаемся сделать запрос к системе и посмотреть, можно ли извлечь только что отправленные данные. Это не самая удачная идея, но она хотя бы позволяет реализовать обходной путь в случае плохо спроектированной программы. Если нужно интегрировать несколько API, то

система становится запутанной и хрупкой, но иногда это лучшее, что можно сделать. Однако бывает и так, что ни одна из этих стратегий не годится. В таком случае приходится искать нестандартные решения, гарантирующие получение данных, после чего запись из журнала можно удалять. Можно ли сказать, что реализации получательской части HML в том месте, где мы получаем данные от потокового API, достаточно, чтобы считать, что мы смягчили риск потери данных? Я склонен ответить «да».

8.1.4. Обработка ровно один раз

Полно случаев, когда очень важно, чтобы каждое сообщение обрабатывалось ровно один раз. Так, в примере из предыдущего раздела мы получаем данные из потока заказов и должны обновить стороннюю систему обработки заказов. Поскольку сторонний API писали не мы, нельзя делать никаких предположений о его идемпотентности. Следовательно, мы должны гарантировать, что не отправляем сообщений-дубликатов. В идеале для этого нужно подтверждение от потокового API. Этот паттерн взаимодействия – подтверждение от стороннего API и подтверждение обработки сообщения от потокового клиента – показан на рис. 8.6. Еще раз отметим, что в силу природы задачи использование браузера в качестве клиента практически невозможно.

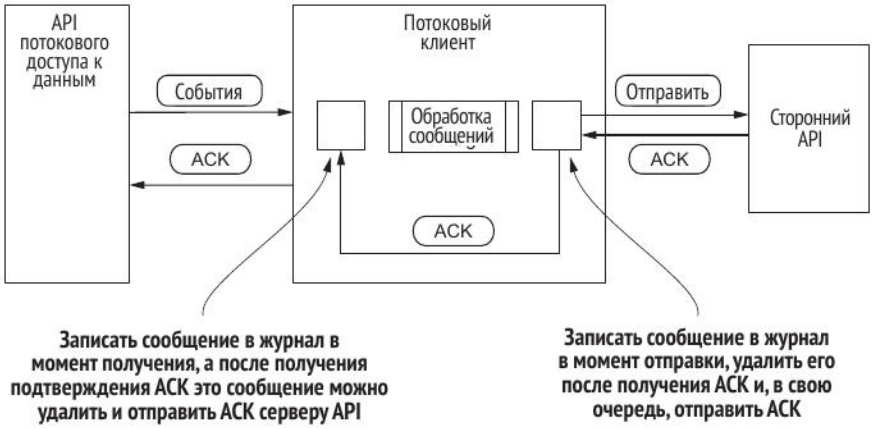


Рис. 8.6. HML с полным подтверждением

Тут у нас целый ряд задач. Во-первых, нужно реализовать подтверждение сообщений в стороннем API, как обсуждалось в главе 7. Затем то же самое нужно сделать для потокового API. К сожалению, во многих случаях мы не контролируем потоковый API и должны смириться с невозможностью отправить ему подтверждение. Как же гарантировать, что сообщение обрабатывается ровно один раз? Придется хранить историю всех сообщений, полученных за некоторый период. Если у каждого сообщения существует гарантированно уникальный идентификатор, то его можно

записывать в «хранилище обработанных сообщений». В противном случае надо вычислять хэш сообщения или использовать еще какой-то механизм цифровых отпечатков пальцев. Сохранение информации о всех виденных сообщениях увеличивает сложность клиента и предъявляет дополнительные требования к хранилищу, но попутно мы защищаемся от аварий потокового API. Соответствующая архитектура показана на рис. 8.7.

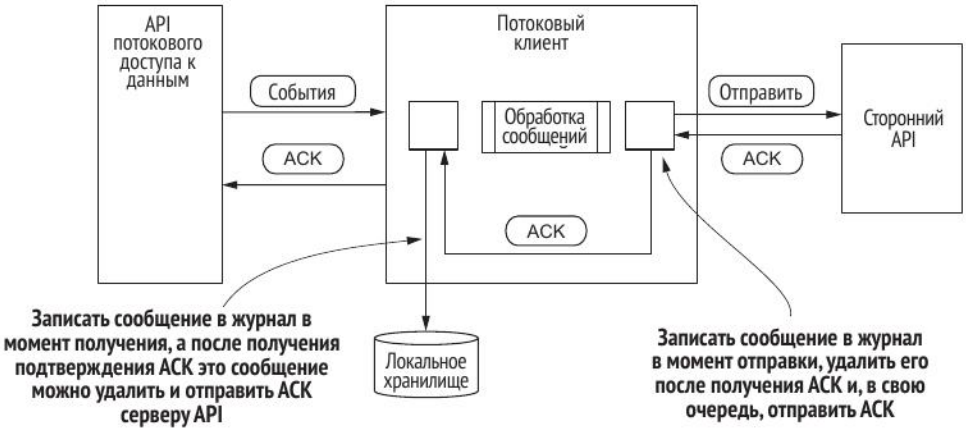


Рис. 8.7. HML с частичным подтверждением и локальным хранилищем

На первый взгляд, все достаточно просто, но что, если серверов потокового API несколько? В какой-то момент один из них обязательно откажет или будет выведен из эксплуатации для технического обслуживания. В таком случае наш потоковый клиент подключится к другому серверу API и начнет потреблять порождаемый им поток. Как гарантировать, что ранее обработанное сообщение не будет обработано снова? Напомним: мы не можем предполагать, что потоковый API помнит, какие сообщения были нам доставлены. Быть может, это и так, если сервер был остановлен штатно, но что, если он «грохнется», не успев запомнить, какое сообщение отправил последним, и после восстановления отправит его еще раз? Чтобы корректно обработать такую ситуацию, нам нужно распределенное хранилище для запоминания ранее виденных сообщений. Это показано на рис. 8.8, где потоковых API и потоковых клиентов несколько.

Для простоты на рис. 8.8 показан только один потоковый клиент, но думаю, что общая картина ясна. В какой-то момент времени сервер потокового API 01 пропадает, а сервер потокового API 02 отправляет нашему потоковому клиенту сообщение, которое тот уже видел. И в этом случае, и во всех остальных мы должны проверить, обрабатывалось ли уже это сообщение, и если да, то отбросить его. Но в этом дизайне есть одна тонкость. Для хранения ранее обработанных сообщений мы используем распределенное хранилище. Тем самым мы – о, чудо – защитились от двойной обработки сообщений. Но если сообщения поступают в высоком темпе, то

запросы, отправляемые службе по сети, ставят под сомнение нашу способность выполнить соглашение об уровне обслуживания. Помните об этом, измеряйте производительность и, если заметите опасность, подумайте об использовании локального хранилища, из которого сообщения переносятся в распределенное. Такая схема работы показана на рис. 8.9.

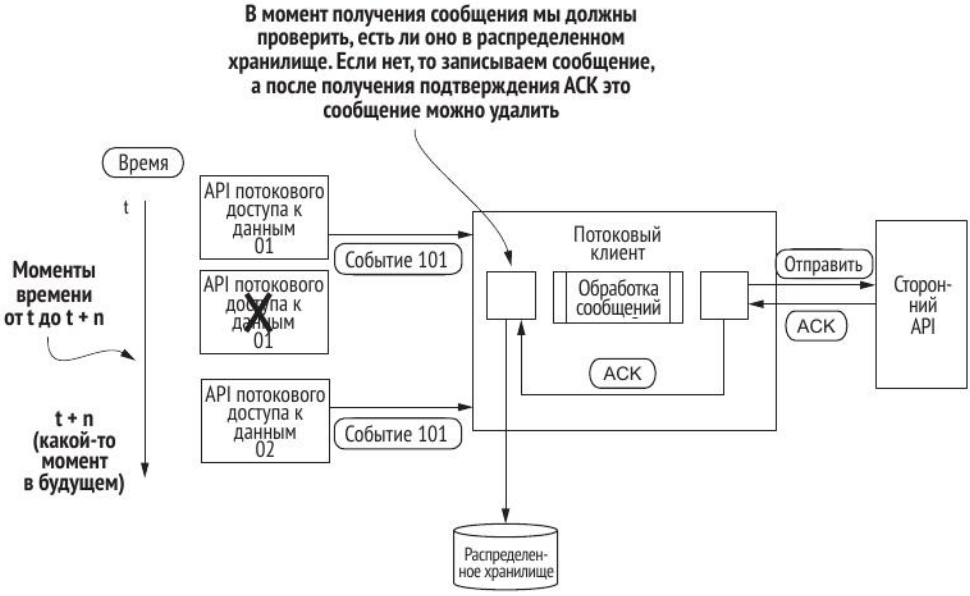


Рис. 8.8. HML с частичным подтверждением и распределенным хранилищем обработанных сообщений

Когда потоковый клиент обнаруживает, что соединение с сервером потокового API пропало, он должен синхронизировать локальное хранилище с распределенным. При условии, что этот шаг выполняется до получения следующего сообщения, мы сможем гарантировать, что ни одно сообщение не будет обработано дважды.

8.2. Все по-настоящему: компания SuperMediaMarkets

Уф, однако же немало мы изучили! По большей части это низкоуровневые детали, которые надо иметь в виду и иногда пускать в дело при создании потокового клиента. Спрашивается, как, когда и зачем все это следует использовать? Хорошие вопросы, особенно если вы подумываете об использовании браузера в качестве клиента или встречали пример веб-клиента, потребляющего поток данных. Понимаю – и в следующем разделе разберу случай веб-клиента. Но прежде я хочу рассмотреть конкретный пример, который вам предстоит обдумать на предмет применения описанных выше концепций.

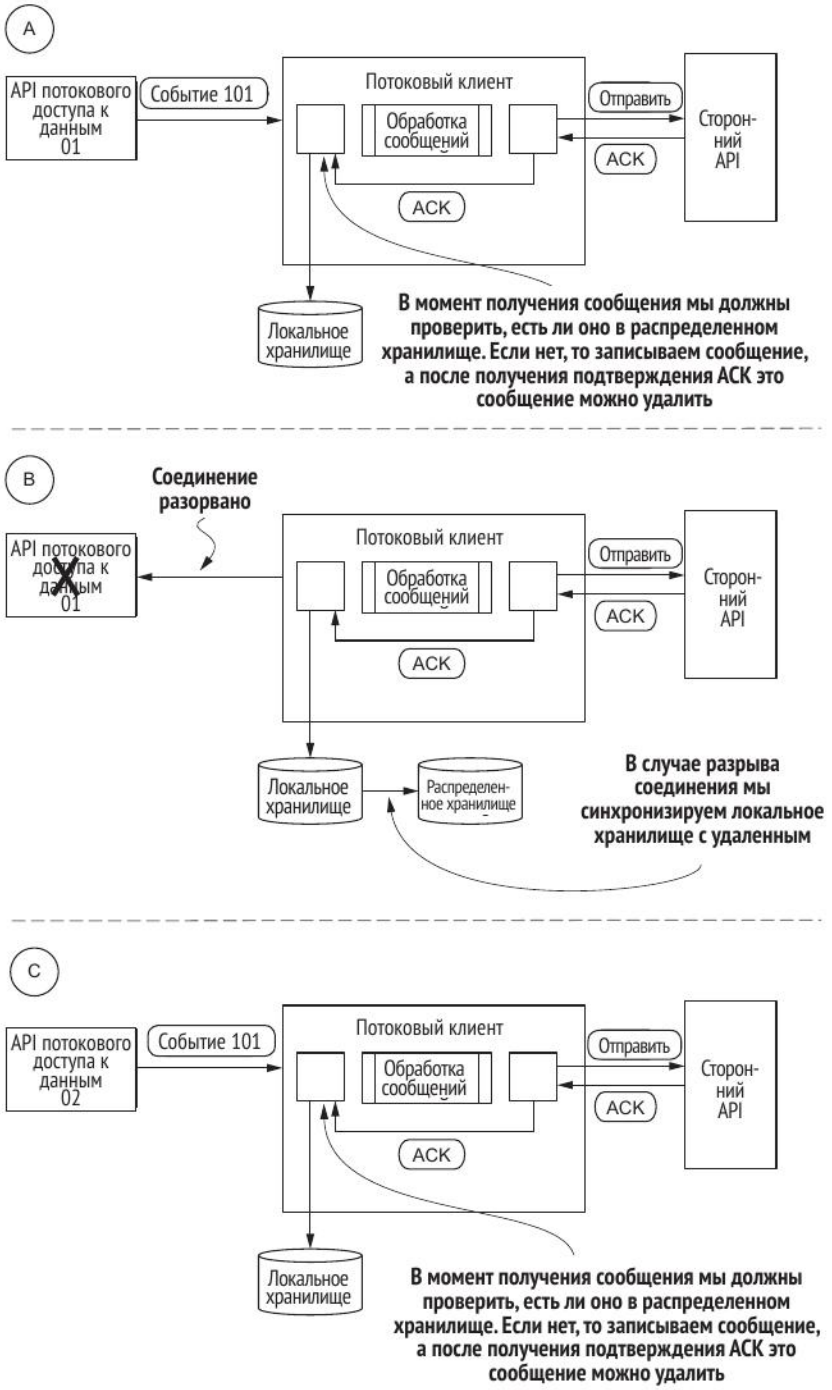


Рис. 8.9. Время обработки сообщений от сервера потокового API изменилось, но сообщения по-прежнему обрабатываются ровно один раз

Допустим, что мы работаем в компании SuperMediaMarkets, которая представляет онлайновую торговую площадку с медийными активами, играя роль посредника между их производителями (фотографами, изготовителями видео, музыкантами, художниками и т. д.) и потребителями (маркетологами, веб-дизайнерами, блоггерами и т. д.). Производитель загружает актив, мы продаем его потребителю и за свой тяжелый труд получаем комиссию. Можно считать, что это своего рода eBay для цифровых медиа. Наша услуга популярна, поэтому покупки и продажи на сайте осуществляются круглые сутки. Недавно команда разработчиков создала потоковый конвейер, который регистрирует все, что делается на сайте, и предоставляет другим разработчикам для создания потоковых приложений. Имея такой поток, мы хотели бы построить клиент, который позволит нам обрабатывать продажи, вычислять и, возможно, выплачивать отчисления авторам в режиме реального времени. Чтобы этот клиент был полезен, должны выполняться следующие требования:

- каждое отчисление следует выплачивать ровно один раз;
- отчисления не должны теряться.

Наша команда предоставила API на основе веб-сокетов, поэтому для реализации потокового клиента мы решили использовать Node.js. Не паникуйте – мы не собираемся забираться в дебри изучения Node.js. Но если Node.js – ваш конёк, то вы сумеете воспользоваться следующим далее обсуждением для реализации надежного потокового клиента. А теперь раскроем требования – поговорим о том, что вообще нужно сделать и что для этого требуется от нашего потокового API.

Как мы собираемся удовлетворить первому требованию: выплачивать отчисление ровно один раз? До сих пор мы говорили о потоковом конвейере, который целиком находится под нашим контролем и поддерживает семантику «ровно один раз». Но наш клиент будет расположен между потоковым API и сторонней системой процессинга. Чтобы удовлетворить требованию, мы должны (a) посылать процессору запрос об уплате ровно один раз и (b) гарантировать, что мы получаем сообщение от нашего потокового API ровно один раз. На рис. 8.10 показано, как это можно сделать.

Обсудим шаги, показанные на рис. 8.10.

1. *Событие уплаты отчислений*. На этом шаге мы получаем событие от потокового API.
2. *Протоколировать событие*. Это часть RBML алгоритма HML. Получив сообщение, мы сохраняем его.
3. *Сохранить событие*. Мы сохраняем событие в базе RocksDB, которая прекрасно подходит на роль локального хранилища, способного быстро вставлять и удалять записи.

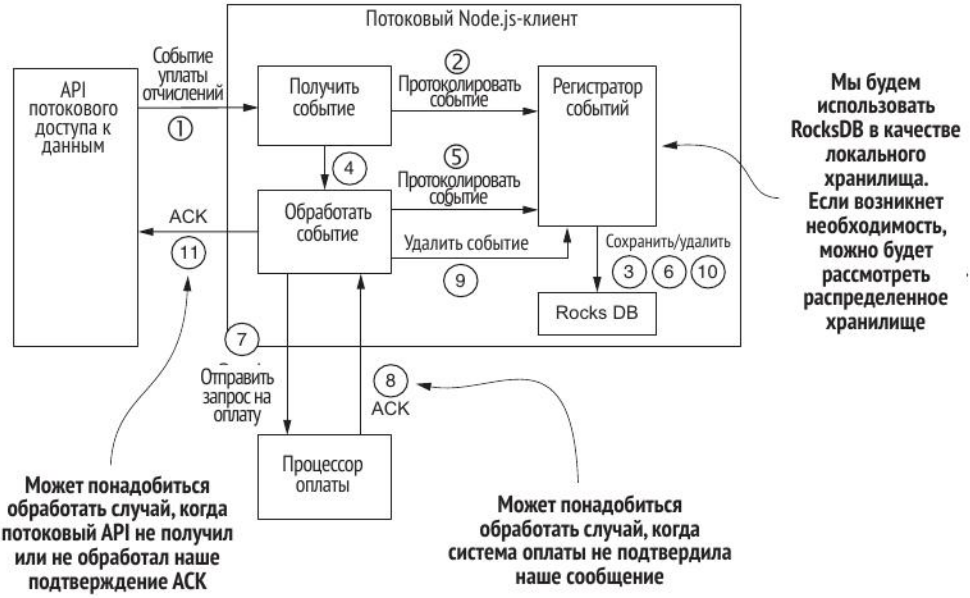


Рис. 8.10. Node.js-клиент – показаны шаги, необходимые для однократной обработки

4. *Обработать событие.* В этой точке исходное полученное событие сохранено, и мы можем безопасно отправить его.
5. *Протоколировать событие*. Мы записываем сообщение в журнал перед отправкой – это часть SBML алгоритма HML; сообщение сохраняется на случай, если при отправке возникнут проблемы.
6. *Сохранить событие*. То же, что на шаге 3.
7. *Отправить запрос на оплату.* Мы готовы послать запрос процессору оплаты.
8. *Получить подтверждение.* Ждать подтверждения успешной обработки запроса об оплате. Возможно, здесь придется сделать что-то другое, если процессор оплаты не поддерживает подтверждений. В таком случае стоит посмотреть, нет ли какого-нибудь способа опросить систему, чтобы получить подтверждение успешной обработки запроса. Иногда это невозможно, и тогда придется искать другое решение, быть может, более асинхронное, с большим временем ожидания. Или запустить отдельный процесс, который будет проверять успешность обработки платежей.
9. *Удалить событие.* В этой точке мы знаем, что система процессинга обработала запрос на оплату, поэтому событие уплаты отчислений, полученное на шаге 1, можно безопасно удалить из хранилища.
10. *Удалить.* Здесь запись – в данном случае событие уплаты отчислений – удаляется из RocksDB.

11. *Подтверждение.* В этой точке мы закончили обработку события и должны отправить подтверждение потоковому API. Если потоковый API не поддерживает подтверждений, то придется найти другие средства, которые дадут знать, что сообщение обработано и посылать его еще раз не нужно. Есть шанс, что потоковый API не получит подтверждения из-за сетевых проблем или не обработает его. В таком случае невозможно гарантировать, что сообщение не будет получено повторно. Поэтому на шаге 7 перед отправкой запроса на оплату процессору мы должны дополнительно проверить, что не пытаемся повторно отправить уже обработанный запрос.

С первым требованием мы успешно разобрались. Как вы думаете, удастся ли удовлетворить второе требование – отчисления не должны теряться? Вообще говоря, для удовлетворения этого требования не нужно вносить никаких изменений. Следует позаботиться о том, чтобы не делать слишком много в блоке, относящемся к получению данных. Мы хотим записать данные в постоянное хранилище как можно раньше – именно в этом промежутке времени риск потерять данные максимален. Записывать в постоянное хранилище нужно также после того, как внесены изменения в данные.

Но все-таки риск потерять данные остается. Что случится, если диск с базой данных RocksDB выйдет из строя? Мы потеряем данные. Воспрепятствовать этому можно несколькими способами, но я упомяну только два. Во-первых, можно использовать RAID-массив в конфигурации, обеспечивающей резервирование. Во-вторых, можно взять распределенное хранилище. В последнем случае нужно гарантировать, что данные получены и сохранены, что усложняет систему. Некоторых дополнительных шагов можно было бы избежать, если бы созданный другой командой потоковый конвейер позволял запрашивать события, имевшие место в прошлом. В таком случае вам предстоит решить, готовы ли вы отдать судьбу данных – и, возможно, их потерю – в чужие руки.

Надеюсь, что теперь вы лучше понимаете, как применять изложенные в этой главе концепции. Далее мы рассмотрим веб-клиент и построение информационной панели для конечного пользователя.

8.3. Введение в веб-клиент

Давайте-ка отвлечемся от низкоуровневых деталей и поговорим об использовании веб-клиента в качестве потокового. В этом случае о многих деталях думать не нужно, потому что веб-клиент не применяется для взаимодействия со сторонним API, который может быть транзакционным и неидемпотентным. А применяется он главным образом для отображения результатов обработки потока. Нам часто приходится сохранять со-

стояние и, возможно, выполнять некоторые вычисления. Хотя в отрасли уже сейчас наблюдается ажиотаж вокруг создания клиентов, обрабатывающих данные, и в будущем он будет только нарастать, использование браузера для отображения результатов потокового приложения сейчас и в будущем останется полезным инструментом, поскольку помогает пользователям лучше понять свой бизнес. Некоторые ставят под вопрос применение браузера для отображения потока, называя это просто «бантиками». Временами я склонен согласиться с ними, хотя недавно работал с двумя заказчиками, которые использовали информационную панель в браузере по следующим причинам:

- им впервые удалось понять, как люди пользуются их сайтом и как функционирует их бизнес. Вы, наверное, думаете, что все это можно узнать из типичного аналитического отчета, но, когда видишь живой поток данных, картина предстает совершенно в другом свете;
- крупное новостное издательство использовало информационную панель в браузере в операционном зале, чтобы редакторы в реальном времени видели, как воспринимаются статьи. С помощью этого инструмента они принимали решения о том, как повысить популярность статьи и извлечь больший доход из сопутствующей рекламы. Это эффективный подход, который трудно автоматизировать.

Если поток высокоскоростной, то клиент, реализованный в браузере, без сомнения, не будет успевать за ним. Звучит ужасно, но, на мой взгляд, это то же самое, что выбор HD-телевизора. Сравнивая картинку на экране дорогого и дешевого телевизора, мы можем заметить тонкие различия в насыщенности цветов или резкости, но сама-то картинка не меняется. Так же обстоит дело с информационной панелью в браузере, не поспевающей за потоком. Какие-то данные будут пропущены, но общую картину и тенденции уловить можно. Я работал с сотнями заказчиков, и эта мысль неизменно оказывалась верной. Все говорят, что нужна стопроцентная точность, но когда доходит до дела, выясняется, что правильной общей картины и тенденций более чем достаточно.

Это не значит, что заботиться вообще не о чем. Безусловно, мы должны сохранять состояние и, возможно, выполнять некоторые простые вычисления. Выше я уже говорил о хранении состояния в веб-хранилище и в базе IndexedDB. Не опускаясь на уровень кода, посмотрим, как мы могли бы применить эти технологии для хранения состояния и вычислений. На рис. 8.11 показан веб-клиент, который рисует штабельную диаграмму объема продаж каждым торговым представителем. По мере поступления данных от потокового API программа на JavaScript обновляет локальное хранилище в браузере. Есть два варианта сохранения текущего состояния диаграммы: последние значения, полученные от потокового API, и теку-

щее состояние диаграммы в целом. Хранить полное состояние диаграммы удобно, потому что если пользователь закроет страницу и вернется к ней позже, у нас будет точка отсчета, при этом состояние, хранящееся в локальном, а не в сеансовом хранилище, сохранится и после закрытия браузера.

Но следует иметь в виду две вещи. Во-первых, прежде чем принимать решение о сохранении данных в локальном хранилище, то есть оставлять их на диске после закрытия браузера, проконсультируйтесь со специалистами по безопасности. Во-вторых, бывает так, что потоковый API отправляет не накопительные итоги, а только инкрементные изменения. В этом случае вы должны запоминать изменения и вычислять итоги самостоятельно. Если для этого приходится сохранять много данных, подумайте, не стоит ли использовать IndexedDB вместо локального хранилища.

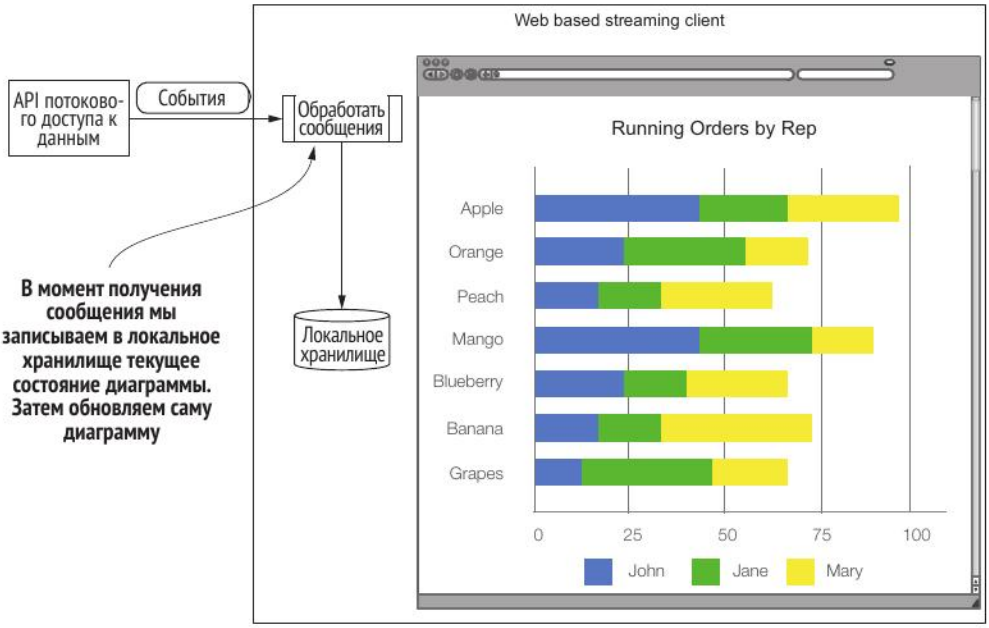


Рис. 8.11. Веб-клиент, использующий локальное хранилище для запоминания состояния диаграммы

8.3.1. Интеграция со службой потокового API

При интеграции браузерного клиента с потоковым API нет недостатка в гибкости. Без особого труда можно осуществить интеграцию с API, поставляющим поток с помощью длинного HTTP-опроса, веб-сокетов или событий, отправляемых сервером. В листингах 8.1–8.3 показано, как делается во всех трех случаях.

Листинг 8.1. События, отправляемые сервером

```
var source = new EventSource("<SSE Service URL>");

source.onmessage = function(event) {
    document.getElementById("event").innerHTML += event.data + <br>";
};
```

Подключиться к серверу событий

Функция, вызываемая при получении нового сообщения

Листинг 8.2. Длинный HTTP-опрос

```
(function pollServer() {
  setTimeout(function() {
    $.ajax({ url: "<Server To Talk To>", success: function(data) {
      document.getElementById("event").innerHTML += data.value + <br>";
    }, dataType: "json", complete: pollServer });
  }, 30000);
})();
```

Функция, вызываемая при каждом срабатывании таймера

AJAX-функция, которая передает URL-адрес, от которого мы получаем данные

Переустановка таймера с указанием вызываемой функции

Тело функции, вызываемой в случае успеха; получает HTML-элемент «event» и обновляет его

Листинг 8.3. Веб-сокеты

```
function webSocket() {
  if ("WebSocket" in window) {
    // открыть веб-сокет
    var ws = new WebSocket("ws://<url to service>");
    ws.onmessage = function (event) {
      document.getElementById("event").innerHTML += event.data + <br>";
    };
  }
}
```

Устанавливает соединение с сервером веб-сокета

Функция, вызываемая при получении нового сообщения

Обновляет элемент «event», записывая в него новые данные

Из этих листингов можно составить представление о принципах интеграции. В случае событий, отправляемых сервером, и веб-сокетов мы обрабатываем сообщения по одному, так что применить к ним бизнес-логику легко и просто. В случае длинного опроса нужно помнить, что запрос инициирует браузер всякий раз, как срабатывает таймер.

Какая бы технология ни применялась при построении потокового API, мы сможем интегрировать ее с браузером, написав код на JavaScript. В главе мы рассмотрим этот вопрос более детально, когда будем разрабатывать клиентское приложение.

8.4. На пути к языку запросов

См. рис. 9.6 и 9.7 в главе 9

Не успела заработать интеграция с потоковым клиентом – не важно, реализованным внутри браузера или написанным на вашем любимом языке программирования, – как пользователи уже просят добавить возможность фильтрации и применения других критериев к потоку данных. Им нужна возможность предъявлять запросы, к которой они привыкли, работая с традиционными хранилищами данных. Возможно, это произойдет не сразу, поскольку поначалу они целиком увлечены созерцанием протекающих данных и новыми открытиями о работе своего бизнеса. Однако новизна быстро приедается, и вот тогда-то наступает черед запросов. Но вы попали в хорошую компанию, поскольку сейчас в отрасли ведутся активные работы в этом направлении. К сожалению, пока многое придется делать на стороне клиента; хотя во все движки потоковой обработки SQL-подобные средства уже добавлены или будут добавлены в ближайшем будущем, поддержка SQL в сторонних потоковых API разительно отличается.

Включить SQL в потоковый клиент интересно. Это одна из самых распространенных моделей доступа к данным. Если используемый вами потоковый API поддерживает SQL или какой-нибудь другой язык запросов, то вам, возможно, останется только написать тонкий обертывающий слой. По существу, вы будете предлагать ту же самую поддержку, передавая запрос API. Но если потоковый API не поддерживает никакого языка запросов и это бремя ложится на ваши плечи, то в клиенте удастся сделать не слишком много. Чтобы предоставить надлежащую поддержку запросов, нужно будет либо подключить команду, которая писала потоковый API, либо, если это сторонний продукт, разработать для него прокси. Но прежде чем вдаваться в детали, взгляните на архитектуру, изображенную на рис. 8.12.

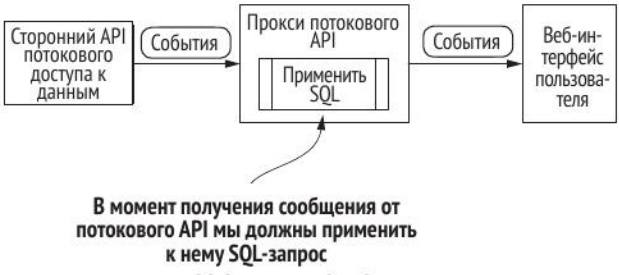


Рис. 8.12. Верхнеуровневая архитектура прокси потокового API

При такой архитектуре мы можем включить развитые средства запросов в приложение, работающее в браузере. Нужно только взаимодействовать с прокси потокового API так, как будто это настоящий

потоковый API. Тогда мы можем принять от пользователя SQL-запрос и передать его прокси, который будет выполнять запрос для поступающих потоков данных.

Если вы окажетесь в такой ситуации и должны будете реализовать прокси потокового API, предоставляющий пользователям средства SQL-запросов, то присмотритесь к проекту Apache Calcite (https://calcite.apache.org). Эта библиотека специально спроектирована для применения SQL к потоковым или статическим данным. Возможно, для вашего источника данных существует адаптер Calcite, а если нет, тоже ничего страшного – библиотека расширяемая, так что адаптировать ее к своим потребностям не сложно.

8.5. Резюме

В этой главе мы рассмотрели концепции, о которых нужно помнить при разработке потокового клиента. Материала было много, и, возможно, он показался вам сложнее, чем можно было бы ожидать от главы, посвященной клиентской стороне потока.

Были рассмотрены следующие вопросы:

- базовые концепции, существенные на стадии доставки потока данных потребителю;
- проблемы, возникающие при опросе потока.

Главный урок этой главы заключается в том, что проблема доставки потока пользователям гораздо глубже, чем просто навороченная информационная панель. Рано или поздно пользователи захотят предъявлять запросы, и зачастую поток необходимо доставлять стороннему приложению. И тогда искусство доставки потока клиенту начинает жить собственной жизнью.

В главе 9 мы воплотим все, что узнали, на практике – построим законченную потоковую систему.

Часть II

Потоки в реальном мире

В этой части книги мы применим на практике все, что узнали в главах 1–8. В главе 9 – единственной в этой части – мы построим законченную систему потоковой обработки данных. В качестве набора данных будем использовать RSVP API сайта Meetup.com. Начнем с обсуждения набора данных и цели приложения, а затем разработаем все звенья, одно за другим. В конце главы у нас будет работающая потоковая система и четкое представление о следующих шагах на пути к его внедрению.

Глава 9

Анализ приглашений Meetup.com в режиме реального времени

Краткое содержание главы:

- построение законченного конвейера потоковой обработки данных;
- планирование внедрения.

Примите поздравления! Вы добрались до главы, где мы приложим к делу весь изученный материал. Мы построим законченный конвейер потоковой обработки данных и потребляющее его приложение. Вместо того чтобы использовать фиктивный набор данных (и оставить вас в сомнениях, как же все это работает в действительности), мы возьмем настоящий поток: потоковый API приглашений (RSVP) сайта Meetup (www.meetup.com/meetup_api/docs/stream/2/rsvps/#websockets). А веб-приложение позволит получить представление о том, что же такое этот поток. Для отладки, а также на случай, если поток вдруг прекратит существование, в придачу к коду включен файл, позволяющий смоделировать поток данных. По завершении главы у вас будут полнофункциональный потоковый конвейер и веб-приложение. И вы будете прекрасно оснащены, чтобы перейти на следующий уровень, взяв тот же самый или совершенно другой поток.

Но прежде чем отправиться в путь, я хочу отметить два момента. Было бы совершенно невозможно осветить все стороны каждого принимаемого технического решения. И немыслимо реализовать в одной главе все то, что обсуждалось в предыдущих восьми.

В процессе реализации каждого звена я буду отмечать, какие варианты возможны. А несколько принятых решений перечислю прямо сейчас.

- Используются только технологии с открытым исходным кодом, по возможности проекты Apache.
- Для сборки всех проектов используется Apache Maven 3.3.9.
- В качестве языка программирования выбран Java 1.8.
- Потоковое приложение работает в браузере и написано на JavaScript.
- Код, приведенный в тексте главы, нередко сокращен, чтобы привлечь внимание к наиболее важным частям. Но доступен также полный код для платформ Linux, OS X и Windows.
- На всех рисунках показаны окна терминала в OS X, но для Windows имеются аналогичные команды.

Весь код, рассматриваемый в этой главе, имеется в репозитории GitHub по адресу https://github.com/apsaltis/StreamingData-Book-Examples.

На рис. 9.1 показана уже хорошо знакомая нам архитектурная диаграмма с одним небольшим добавлением: на этот раз указаны конкретные используемые технологии.

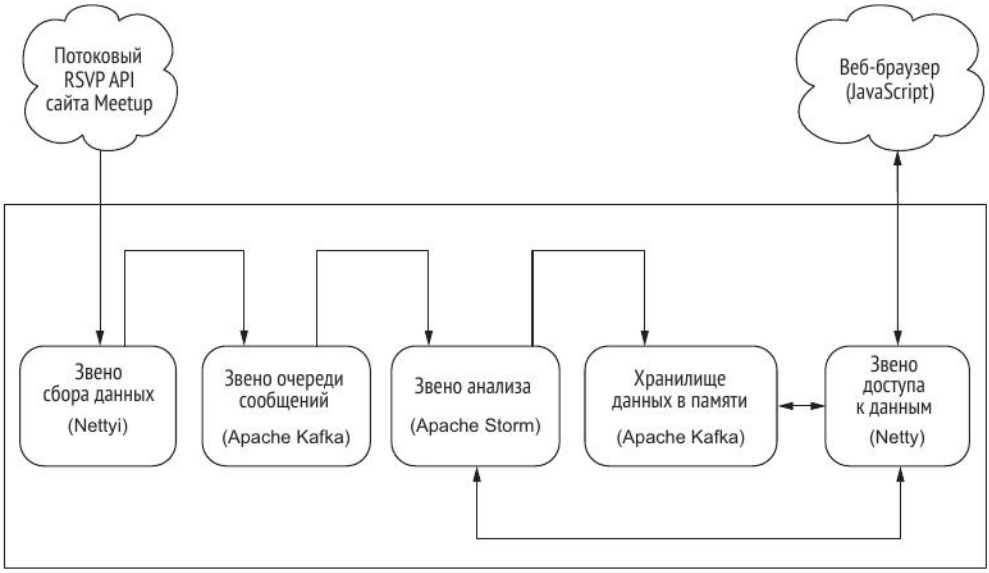


Рис. 9.1. Архитектурная диаграмма с указанием технологий

Для каждой технологии я расскажу, что вам нужно сделать, и объясню ее место в общей картине, причины, по которым выбрал именно ее, и на что обращать внимание в случае ее применения в собственных проектах. Из предыдущих глав вы уже знаете о возможных вариантах для некоторых звеньев, и тогда я буду просто отсылать к соответствующим местам текста.

Ну вот, все готово, и мы можем приступить к построению конвейера! Самое время наполнить чашечку кофе и продолжить обсуждение, начав со звена сбора данных.

9.1. Звено сбора данных

Приступая к построению звена сбора данных, мы должны решить, как собираемся вносить данные в систему. Речь идет о таких аспектах, как используемые протоколы, формат данных и будут ли данные проталкиваться службой или вытягиваться из нее. Потоковый RSVP API сайта Meetup реализован поверх протокола HTTP с разбиением на порции (chunking), а также в виде WebSocket-службы. В обоих случаях данные возвращаются в формате JSON. Для своей службы сбора данных мы будем использовать WebSocket API, а для клиента – библиотеку Netty (http://netty.io).

WebSocket API и Netty я выбрал по двум причинам: как уже отмечалось, протокол WebSocket эффективен, и мы будем использовать WebSocket и Netty позже, при разработке API доступа к данным. Это позволит упростить решение – к чему всегда следует стремиться при построении реального потокового конвейера. Почему Netty, а не какая-то другая технология? Netty демонстрирует фантастическую производительность и предлагает низкоуровневые механизмы управления взаимодействием с клиентом. То и другое очень важно и подробно обсуждается в разделе, посвященном API доступа к данным.

9.1.1. Диаграмма последовательности службы сбора данных

При построении службы сбора данных следует принимать во внимание следующие характеристики:

- управление подключением к Meetup API;
- гарантии сохранности данных;
- интеграция со звеном очереди сообщений.

На рис. 9.2 показано, как все будет работать в связке.

Как видим, служба сбора данных состоит из нескольких классов. Она интегрирована с системой Apache Kafka, которую мы будем использовать в звене очереди сообщений. Чтобы данные не терялись, мы реализуем алгоритм HML, описанный в главе 2.

Рассмотрим внимательнее основные методы, показанные на рис. 9.2, начав с метода `initialize` класса `HybridMessageLogger`. Этот довольно простой метод приведен в листинге 9.1.

Листинг 9.1. Инициализация HybridMessageLogger

См. раздел 2.3.3 главы 2

```
public class CollectionServiceWebSocketClient {
    public static void main(String[] args) throws Exception {
        try{
```

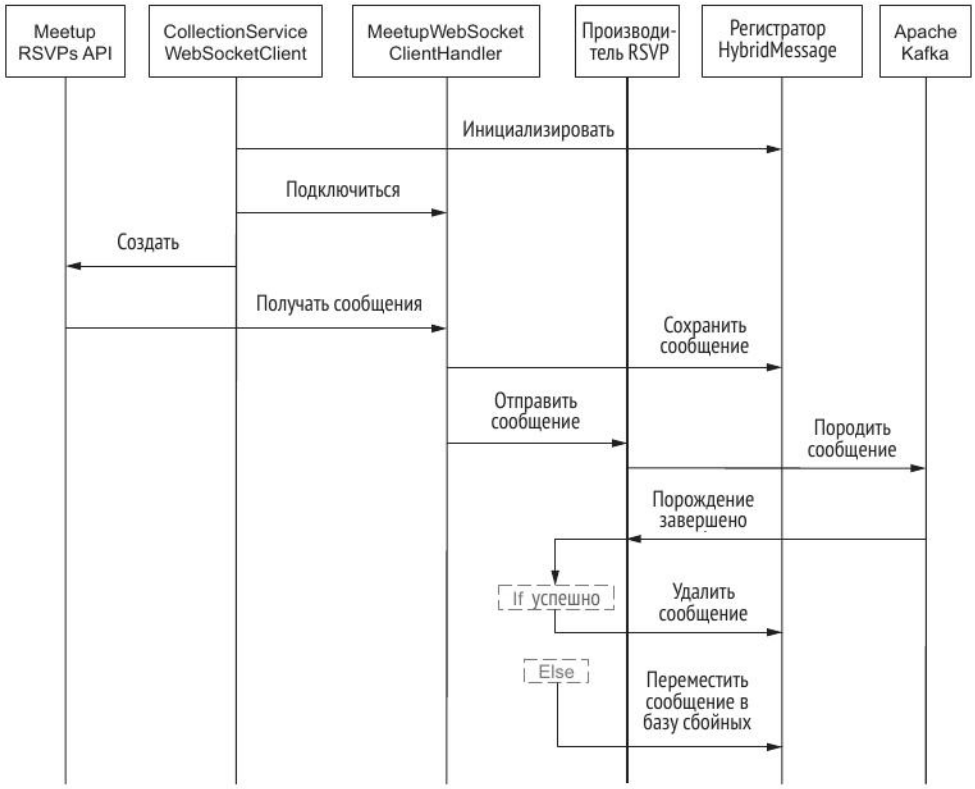


Рис. 9.2. Диаграмма последовательности для службы сбора данных

```
            HybridMessageLogger.initialize();
        }catch (Exception exception){
            System.err.println("Could not initialize HybridMessageLogger!");
            exception.printStackTrace();
            System.exit(-1);
        }
    }
}

final class HybridMessageLogger {

    private static RocksDB transientStateDB = null;
    private static RocksDB failedStateDB = null;
    private static Options options = null;
    private static Path transientPath = new File("state/transient").toPath();
    private static Path failedPath = new File("state/failed").toPath();

    static void initialize() throws Exception {
        RocksDB.loadLibrary();
```

Инициализировать HybridMessageLogger. В случае ошибки выйти

Загрузить RocksDB

```
        options = new Options().setCreateIfMissing(true);

        try {
            ensureDirectories();
            transientStateDB = RocksDB.open(options,
            transientPath.toString());
            failedStateDB = RocksDB.open(options, failedPath.toString());
        } catch (RocksDBException | IOException e) {
            e.printStackTrace();
            throw new Exception(e);
        }
    }

    private static void ensureDirectories() throws IOException {

        if(Files.notExists(transientPath)){
            Files.createDirectories(transientPath);
        }
        if(Files.notExists(failedPath)){
            Files.createDirectories(failedPath);
        }
    }
}
```

Создать в файловой системе каталоги, если их еще нет

Когда класс `CollectionServiceWebSocketClient` вызывает метод `initialize` класса `HybridMessageLogger`, производится несколько важных действий. Первым делом загружается библиотека RocksDB (http://rocksdb.org) – быстрое встраиваемое хранилище ключей и значений, идеально подходящее для использования совместно с HML. Вы, конечно, можете рассмотреть и альтернативы, но имейте в виду следующие минимальные требования к хранилищу:

- *оно должно быть быстрым* – читать и писать придется много; для каждого сообщения, полученного от Meetup API, будут, по меньшей мере, две операции чтения и две операции записи;
- *данные должны храниться локально* – если мы будем передавать данные по сети, то можем их потерять, и это придется учитывать.

После загрузки RocksDB мы создаем каталоги, в которых будем хранить данные (если они еще не существуют). И наконец, создаем две базы данных, `transientStateDB` и `failedStateDB`. В базе `transientStateDB` сообщения будут храниться до получения подтверждения об успешном сохранении от Kafka. Если в процессе отправки сообщения Kafka произойдет ошибка, то сообщение перемещается в базу `failedStateDB`. Мы рассмотрим этот вопрос чуть ниже.

Теперь можно создать объект `MeetupWebSocketClientHandler` и подключиться к потоковому RSVP API сайта Meetup. Как видно из следующего листинга, это по большей части трафаретный код Netty.

См. раздел 2.1.5 главы 2

Листинг 9.2. Создание обработчика протокола WebSocket и подключение к Meetup API

```
public class CollectionServiceWebSocketClient {
    public static void main(String[] args) throws Exception {

    ... инициализация HybridMessageLogger .......

        final String URL = 0 < args.length? args[0] :
    "ws://stream.meetup.com/2/rsvps";
        URI uri = new URI(URL);
        final int port = 80;

        try {
            final MeetupWebSocketClientHandler handler =
              new MeetupWebSocketClientHandler(
                WebSocketClientHandshakerFactory.newHandshaker(
                  uri, WebSocketVersion.V13, null, false,
                  new DefaultHttpHeaders()));

            Bootstrap b = new Bootstrap();
            b.group(group)
              .channel(NioSocketChannel.class)
              .handler(new ChannelInitializer<SocketChannel>() {
                  @Override
                  protected void initChannel(SocketChannel ch) {
                    ChannelPipeline p = ch.pipeline();
                    p.addLast(
                      new HttpClientCodec(),
                      new HttpObjectAggregator(8192),
                      WebSocketClientCompressionHandler.INSTANCE,
                      handler);
                  }
              });

              Channel ch = b.connect(uri.getHost(), port).sync().channel();
              handler.handshakeFuture().sync();

              .... обработать сообщения ping и bye протокола  WebSocket .....
        } finally {
            group.shutdownGracefully();
        }
    }
}
```

❶ Определить URL-адрес, к которому мы будем подключаться

❷ Создать обработчик, который будет получать все сообщения

Добавить наш обработчик в конвейер Netty

Подключиться к Meetup API и ждать выхода из обработчика

После инициализации `HybridMessageLogger` можно настраивать обработчик протокола WebSocket. В Netty мы создаем обработчик, который отвечает за все взаимодействие со второй стороной соединения, и добавляем его в конвейер. Но сначала нужно задать URL-адрес, к которому мы подключаемся. Для упрощения отладки и перехода на другой WebSocket-сервер мы в точке ❶ проверяем, были ли переданы аргументы в командной строке. Если да, то предполагается, что первый аргумент и есть URL-адрес. Мы не проверяем корректность URL, но в реальной программе это следовало бы сделать. Если аргументов нет, то по умолчанию берется текущий адрес потокового RSVP API сайта Meetup. Теперь мы готовы настроить объект `MeetupWebSocketClientHandler` ❷, передав его конструктору `WebSocketClientHandshaker`, – стандартный объект Netty, берущий на себя низкоуровневые детали протокола, чтобы мы могли сосредоточиться на обработке сообщений. Внутри конструктора `MeetupWebSocketClientHandler` конструируется объект `RSVPProducer`.

Листинг 9.3. Конструирование `MeetupWebSocketClientHandler` и `RSVPProducer`

```
class MeetupWebSocketClientHandler extends SimpleChannelInboundHandler<Object>
{
    private final static RSVPProducer rsvpProducer = new RSVPProducer();
}
```

Конструирование RSVPProducer

А внутри конструктора `RSVPProducer` конструируется объект `KafkaProducer`, как показано ниже.

Листинг 9.4. Конструирование `RSVPProducer`

```
final class RSVPProducer {
    private static KafkaProducer<byte[], byte[]> kafkaProducer;

    RSVPProducer(){
        Properties producerProperties = new Properties();
        producerProperties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092");
        producerProperties.put(ProducerConfig.CLIENT_ID_CONFIG,
            "meetup-collection-service-kafka");
        producerProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.ByteArraySerializer");
        producerProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.ByteArraySerializer");
        kafkaProducer = new KafkaProducer<>(producerProperties);
    }
}
```

Задать необходимые свойства производителя

Сконструировать KafkaProducer

Сконструировав `MeetupWebSocketClientHandler`, `RSVPProducer` и `KafkaProducer`, мы можем добавить обработчик `MeetupWebSocketClientHandler` в конвейер Netty.

Листинг 9.5. Добавление обработчика в конвейер Netty

```
public static void main(String[] args) throws Exception {

    ... инициализация HybridMessageLogger .......
    ... задание URL-адреса ...
    ... конструирование обработчика ...

    Bootstrap bootStrap = .....
    bootStrap.channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<SocketChannel>() {
          @Override
          protected void initChannel(SocketChannel ch) {
              ChannelPipeline channelPipeline = ch.pipeline();
              channelPipeline.addLast(new HttpClientCodec());
              channelPipeline.addLast(new HttpObjectAggregator(8192));
              channelPipeline.addLast(
                  WebSocketClientCompressionHandler.INSTANCE);
                  channelPipeline.addLast(handler);
              }
        });
    }
}
```

❶ Построить канальный конвейер

❷ Задать обработчик

Создание канального конвейера ❶ и задание обработчика ❷ гарантируют, что это последний обработчик сообщений, читаемых Netty из сети. В этот момент все необходимые объекты сконфигурированы и конвейер готов к работе – осталось только подключить его к потоковому API сайта Meetup, как показано ниже.

Листинг 9.6. Подключение к Meetup API

```
public class CollectionServiceWebSocketClient {
    public static void main(String[] args) throws Exception {

        ... инициализация HybridMessageLogger .......
        ... задание URL-адреса ...
        ... конструирование обработчика ...
        ... конфигурирование конвейера

        Channel ch = b.connect(uri.getHost(), port).sync().channel();
        handler.handshakeFuture().sync();
```

Здесь программа блокируется в ожидании остановки службы сбора данных

После выполнения этой строки служба сбора данных будет работать в фоновом режиме, подключенная к Meetup API и готовая принимать сообщения

```
        .... обработать сообщения ping и bye протокола  WebSocket .....
    }
}
```

Обработчик `MeetupWebSocketClientHandler` будет получать новое сообщение для каждого приглашения RSVP, полученного сайтом Meetup.com. В листинге ниже показано, как эти сообщения обрабатываются методом `channelRead0` класса `MeetupWebSocketClientHandler`.

Листинг 9.7. Обработка сообщений в классе `MeetupWebSocketClientHandler`

```
class MeetupWebSocketClientHandler extends SimpleChannelInboundHandler<Object>
{
    private final WebSocketClientHandshaker handshaker;
    private ChannelPromise handshakeFuture;
    private final static RSVPProducer rsvpProducer = new RSVPProducer();

    @Override
    public void channelRead0(ChannelHandlerContext ctx, Object msg)
            throws Exception {
        Channel channel = ctx.channel();
        if (!handshaker.isHandshakeComplete()) {
            // соединение установлено
            handshaker.finishHandshake(channel, (FullHttpResponse) msg);
            handshakeFuture.setSuccess();
            return;
        }

        if (msg instanceof FullHttpResponse) {
          // если мы получили такое сообщение, значит, произошла ошибка
          // и следует возбудить исключение
          FullHttpResponse response = (FullHttpResponse) msg;
          throw new IllegalStateException(
            "Unexpected FullHttpResponse (getStatus=" + response.status()
            + ", content=" +
            response.content().toString(CharsetUtil.UTF_8) + ')');
        }

        WebSocketFrame frame = (WebSocketFrame) msg;
        if (frame instanceof TextWebSocketFrame) {
            TextWebSocketFrame textFrame = (TextWebSocketFrame) frame;

            final String messageKey =
                new UUID(random.nextLong(), random.nextLong()).toString();

            // Это как раз то, что нам нужно, поэтому выделяем данные и
            // отправляем их следующему звену.
```

❶ Члены класса, которые мы будем использовать

❷ Объявление метода channelRed0

❸ Убедиться, что процедура согласования протокола WebSocket завершилась

❹ Проверяем, что по ошибке не получили сообщения типа FullHttpResponse

❺ Случайное значение в качестве ключа сообщения

```
            // Сначала нужно прочитать байты из ByteBuf
            final byte[] messagePayload =
                new byte[textFrame.content().readableBytes()];
            textFrame.content().readBytes(messagePayload);
            HybridMessageLogger.addEvent(messageKey,messagePayload);
            rsvpProducer.sendMessage(messageKey,messagePayload);
        } else if (frame instanceof CloseWebSocketFrame) {
            // Нас просят закрыть соединение
            channel.close();
            rsvpProducer.close();
        }
    }
}
```

❻ Читаем все данные из ByteBuf

❼ Получательская сторона протокола HML

Отправить сообщение звену очереди

При необходимости закрыть соединение

В точке ❶ мы должны настроить несколько объектов, необходимых обработчику: уже рассмотренный `RSVPProducer` и ряд других. Настоящая работа начинается в точке ❷ – это метод, который Netty вызывает при получении каждого сообщения из сети. Однако тот факт, что сообщение получено, еще не означает, что мы готовы его обрабатывать. Сначала в точке ❸ проверяется, что процедура согласования протокола WebSocket завершена. Если это так, то мы должны выполнить еще одну, последнюю, проверку в точке ❹, прежде чем приступить к обработке данных. Приход сообщения по протоколу HTTP был бы ошибкой, так что в этом случае мы возбуждаем исключение. Наконец-то мы дошли до точки ❺, где делается нечто полезное. Сначала мы создаем ключ сообщения, который нужен как для протоколирования на стороне получателя, так и для Kafka. Мы используем UUID, поскольку он гарантированно уникален и позволит Kafka равномерно распределить данные. Имея ключ, мы готовы прочитать полученные данные. Код в точке ❻ – самый безопасный способ прочитать полученные данные. В данном случае это приглашение RSVP в формате JSON. Зная ключ и сообщение, мы можем вызвать `HybridMessageLogger` в точке ❼.

Метод `addEvent` класса `HybridMessageLogger` показан ниже – никаких хитростей в нем нет.

См. раздел 2.3.3 главы 2

Листинг 9.8. Метод `addEvent` класса `HybridMessageLogger`

```
static void addEvent(final String eventKey, final byte[] eventData) throws Exception
{
    try {
        final byte[] keyBytes = eventKey.getBytes(StandardCharsets.UTF_8);

        byte[] value = transientStateDB.get(keyBytes);

        if (value == null) {
```

Получить байты ключа в кодировке UTF-8 ❶

❷ Проверить, существует ли такой ключ в базе

```
            transientStateDB.put(keyBytes, eventData);
        }
    } catch (RocksDBException ex) {
        // надо бы запротоколировать эту ошибку
        throw new Exception(ex.getMessage());
    }
}
```

❸ Если не существует, добавить

Мы преобразуем ключ в массив байтов ❶ и проверяем, существует ли в базе значение, ассоциированное с таким ключом ❷. Это нужно для того, чтобы для каждого ключа хранилось не более одного значения. Если ключ не найден, то в точке ❸ мы добавляем в базу пару ключ-значение. Напомним, что этот метод вызывается до отправки данных звену очереди сообщений. Это часть RBML алгоритма HML из главы 2. Риск потерять данные все же остается – если какая-то ошибка произойдет в службе сбора данных между точками ❺ и ❼ в листинге 9.7. Существуют способы устранить и этот риск, но они выходят за рамки этой главы. Оставляю вам эту задачу в качестве упражнения. Теперь, когда данные находятся в безопасности (точка ❻ в листинге 9.7), мы наконец можем передать сообщение объекту `rsvpProducer`, который отправит его Kafka.

Листинг 9.9. Метод `send` класса `RSVPProducer`

```
final class RSVPProducer {
    private static KafkaProducer<byte[], byte[]> kafkaProducer;
    private static final String messageTopic = "meetup-raw-rsvps";

    ... Конструктор ...
    void sendMessage(final String messageKey, final byte[] message) {
        ProducerRecord<byte[],byte[]> producerRecord =
            new ProducerRecord<>(messageTopic,
              messageKey.getBytes(StandardCharsets.UTF_8),
              message);
        kafkaProducer.send(producerRecord,
              new TopicCallbackHandler(messageKey));
    }
}
```

Имя темы для Kafka

❶ Создать объект ProducerRecord, в котором хранятся тема, ключ и значение

❷ Отправить данные Kafka

Мы задаем имя темы Kafka, в которую будем отправлять данные. Саму тему мы создадим в следующем разделе, когда будем настраивать Kafka. При выполнении кода тема должна быть уже создана. Прежде чем отправлять данные Kafka, необходимо создать объект `ProducerRecord` (точка ❶), а уже потом отправить его ❷. Обратите внимание, что мы передаем Kafka объект обратного вызова. Его конструктору передается параметр `eventKey`,

что позволит нам определить, прошли ли отправка и получение данных успешно. Сам класс обратного вызова показан в следующем листинге, он не очень интересен.

Листинг 9.10. Класс `TopicCallbackHandler`

```
private final class TopicCallbackHandler implements Callback {
    final String eventKey;

    TopicCallbackHandler(final String eventKey){
        this.eventKey = eventKey;
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (null == metadata){
            // пометить запись как ошибочную
            HybridMessageLogger.moveToFailed(eventKey);
            // запротоколировать исключение или как-то обработать его
        }else{
            // удалить данные из локального хранилища
            try {
                HybridMessageLogger.removeEvent(eventKey);
            } catch (Exception e) {
                // нужно запротоколировать ...
            }
        }
    }
}
```

❶ Конструктор принимает аргумент eventKey

❷ Если вместо метаданных передан null, значит, произошла ошибка

❸ Данные успешно обработаны, удалить запись из базы

В точке ❶ конструктору передается аргумент `eventKey`, чтобы впоследствии его можно было использовать в методе `onCompletion` для удаления сохраненного сообщения из локального хранилища. Если при отправке данных или их обработке в Kafka произошла ошибка, то аргумент `RecordMetadata` в точке ❷ будет `null`, и это надо как-то обработать. Если данные успешно сохранены в Kafka, то в точке ❸ мы их удаляем – это отправительская часть алгоритма HML из главы 2.

Понимаю, что всё это кажется громоздким, но на самом деле кода не так много. Зато мы теперь имеем полнофункциональную службу, которую можно подключить к потоковому RSVP API сайта Meetup. Она гарантирует, что данные не теряются, и передает их звену очереди сообщений. Но прежде чем запускать эту службу, нам надо проработать следующий раздел, посвященный Apache Kafka.

9.2. Звено очереди сообщений

В разделе 3.1 мы обсуждали различные реализации звена очереди сообщений. Для построения конвейера, связанного с сайтом Meetup, я остановился на Apache Kafka. Это масштабируемая высокопроизводительная система, которая гарантирует сохранность данных и легко интегрируется с другими продуктами. В этом разделе мы не будем детально вникать во все аспекты Kafka, а ограничимся сведениями, которые позволят довести до конца начатый пример. Дополнительную информацию можно найти на странице проекта по адресу http://kafka.apache.org или в книгах, например Neha Narkhede «Kafka: The Definitive Guide» (O'Reilly, 2016).

См. раздел 2.1.5 главы 2

9.2.1. Установка и настройка Kafka

Скачать Kafka можно по адресу http://kafka.apache.org/downloads.html. В этой главе используется версия 0.10.0.1, но более поздние тоже должны работать. Для скачивания и установки выполните в окне терминала показанные ниже команды.

Листинг 9.11. Скачивание и установка Kafka

```
$> wget http://www-us.apache.org/dist/kafka/0.10.0.1/kafka_2.11-0.10.0.1.tgz
$> tar -xvf kafka_2.11-0.10.0.1.tgz
```

При работе с Kafka важно знать о четырех концепциях:

- производители;
- темы;
- брокеры;
- потребители.

Производители, брокеры и потребители в общих чертах рассматривались в главе 3. Темы в Kafka – это логические группы сообщений, для которых можно выполнять операции записи и чтения. Производитель отправляет сообщения брокеру, указывая тему, а потребитель запрашивает сообщения из конкретной темы. В точке ❶ листинга 9.9 мы отправляем данные в тему `meetup-raw-events`. Поэтому нам следует запустить Kafka и создать эту тему, выполнив показанные ниже команды.

Листинг 9.12. Запуск Kafka, Apache ZooKeeper и создание темы

```
$> cd kafka_2.11-0.10.0.1
```

❶ Перейти в установочный каталог Kafka

```
$> bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
```

Запустить Apache ZooKeeper ❷

```
$> bin/kafka-server-start.sh -daemon config/server.properties   ❸ Запустить Kafka

$> bin/kafka-topics.sh --zookeeper localhost:2181 --create \    ❹ Создать тему
--topic meetup-raw-rsvps --partitions 1 --replication-factor 1

$> Created topic "meetup-raw-rsvps".   ❺ Ожидаемое сообщение
```

Прежде всего мы переходим в установочный каталог Kafka ❶. Для хранения некоторых метаданных о темах и брокерах Kafka полагается на Apache ZooKeeper, поэтому мы запускаем ZooKeeper ❷, а уже потом – один брокер Kafka ❸. В этот момент ZooKeeper и Kafka работают, и мы можем создать нужную тему ❹. Если тема успешно создана, то мы увидим сообщение ❺. Чтобы проверить, создана ли тема, выполните команду, показанную ниже.

Листинг 9.13. Команда вывода списка тем Kafka

```
$> bin/kafka-topics.sh --zookeeper localhost:2181 --list   Команда вывода списка тем

    meetup-raw-rsvps   ← Ожидаемое сообщение
$>
```

Теперь Kafka готова получать сообщения от нашей службы сбора данных.

9.2.2. Интеграция службы сбора данных с Kafka

Написав службу сбора данных и запустив Kafka, мы можем интегрировать обе системы и проверить, все ли работает. Для тестирования интеграции понадобится открыть два окна консоли. В первом будет работать консольный потребитель Kafka. Для этого выполните следующую команду.

Листинг 9.14. Запуск консольного потребителя Kafka

```
$> bin/kafka-console-consumer.sh --zookeeper localhost:2181\
    --topic meetup-raw-rsvps
```

Поскольку в эту тему еще не отправлено ни одного сообщения, после выполнения данной команды ничего не произойдет. Это нормально – потребитель просто ждет поступления сообщений.

Следующий шаг – запустить нашу службу. Для этого выполните показанные ниже команды.

Листинг 9.15. Сборка и запуск службы сбора данных

```
$> cd $EXAMPLE_CODE_HOME/Chapter9/collection-service   ❶ Перейти в каталог с исходным кодом службы сбора данных
$> mvn clean package   ← ❷ Собрать проект с помощью Maven
```

```
$> java -jar target/collection-service-0.0.1.jar
    WebSocket Client connected and ready to consume RSVPs!
```

❸ Выполнить результирующий jar-файл

Перейдя в каталог с исходным кодом службы сбора данных ❶, мы собираем проект с помощью Apache Maven ❷. Если на вашем компьютере Maven не установлена, скачайте ее с сайта https://maven.apache.org и установите в соответствии с находящимися там же инструкциями. Теперь можно запустить службу ❸. После появления сообщения о том, что служба сбора данных подключена и готова к работе, в окне консольного потребителя Kafka начнут появляться сообщения. Пример RSVP-сообщения приведен на рис. 9.3. В любой момент вы можете снять службу, нажав **Ctrl+C** в окне консоли.


```
{
  "venue": {
    "venue_name": "mount work regional park, munn road parking lot",
    "lon": -123.464233,
    "lat": 48.500511,
    "venue_id": 12469832
  },
  "visibility": "public",
  "response": "yes",
  "guests": 2,
  "member": {
    "member_id": 215609630,
    "photo": "http://photos3.meetupstatic.com/photos/member/a/6/6/4/
    thumb_261342596.jpeg",
    "member_name": "Amanda Dunn"
  },
  "rsvp_id": 1638108083,
  "mtime": 1478449512424,
  "event": {
    "event_name": "Sunday Funday Mt. Work from Munn Road",
    "event_id": "235366173",
    "time": 1478455200000,
    "event_url": "http://www.meetup.com/Random-Activities-Victoria/events/235366173/"
  },
  "group": {
    "group_topics": [{
      "urlkey": "bike",
      "topic_name": "Bicycling"
    },
    {
      "urlkey": "local-activities",
      "topic_name": "Local Activities"
    }
  ],
  "group_city": "Victoria",
  "group_country": "ca",
  "group_id": 20896502,
  "group_name": "Random Activity Tribe",
  "group_lon": -123.29,
  "group_urlname": "Random-Activities-Victoria",
  "group_state": "BC",
  "group_lat": 48.46
  }
}
```


Рис. 9.3. Пример RSVP-сообщения сайта Meetup в формате JSON

9.3. Звено анализа

См. раздел 4.2 главы 4

Для реализации звена анализа существует немало проектов – при последней проверке я насчитал более 25 продуктов, претендующих на эту роль. Мы будем использовать Apache Storm; это зрелый, простой в установке продукт, который предлагает потоковый анализ индивидуальных кортежей – как раз то, что нужно для обработки RSVP-сообщений сайта Meetup. Архитектура, которую мы рассматривали на протяжении всей книги и воплощаем в жизнь в этой главе, позволяет сменить используемую в этом звене технологию, причем во многих случаях это даже не повлияет на другие звенья. Чтобы глубже разобраться в движках распределенной потоковой обработки, я предлагаю вам после прочтения этой главы попробовать заменить Storm другим продуктом. После установки и настройки Storm мы построим топологию, которая вычисляет *n* самых популярных тем в поступивших приглашениях. В Meetup *темой* называются ключевые слова, выбранные организатором для идентификации группы. В следующем листинге приведен фрагмент RSVP-сообщения в формате JSON.

Листинг 9.16. RSVP-сообщение в формате JSON

```
{
  ...                                          ❶ Другие части сообщения
  "group": {
    "group_topics": [
      ...                                      ❷ Темы группы, для краткости многие опущены
      {
        "urlkey": "exploring-the-city",
        "topic_name": "Exploring the City"
      },
      {
        "urlkey": "charity-work",
        "topic_name": "Charity Work"
      },
      {
        "urlkey": "local-businesses",
        "topic_name": "Local Businesses"
      },
      {
        "urlkey": "cultural-activities",
        "topic_name": "Cultural Activities"
      }
    ],                                                          Метаданные о группе
    ...
    "group_name":"AMS Connected - Where Expats & Amsterdam Truly Unite"
```

```
    ...
  }
}
```

В полном объекте JSON данных гораздо больше, чем в этом фрагменте. Имеются дополнительные метаданные ❶ о мероприятии и месте его проведения – другие элементы JSON, которые вы, возможно, захотите подвергнуть анализу. Например, включена географическая привязка; имея эту информацию, вы могли бы нанести мероприятия на карту и показать, в каких регионах активность максимальна. Или, быть может, проанализировать популярные темы по месту проведения мероприятия. В части ❷ находятся темы, ассоциированные с группой; в настоящее время Meetup позволяет организатору группы указать не более 15 тем в описании своей группы. Для анализа в этом звене мы будем использовать все темы группы, полученные в JSON-сообщении.

9.3.1. Установка Storm и подготовка Kafka

В этой главе мы установим Storm только для разработки и локального выполнения. Дополнительные сведения об установке Storm в производственной среде см. в книге Sean T. Allen «Storm Applied» (Manning, 2015). Выполните показанные ниже команды.

Листинг 9.17. Установка Apache Storm

```
Скачать Storm по адресу: http://storm.apache.org/downloads.html
Распаковать дистрибутив

Или выполнить следующие команды:

$> wget http://www-us.apache.org/dist/storm/apache-storm-1.0.2/apache-storm-
     1.0.2.tar.gz
$> tar -xvf apache-storm-1.0.2.tar.gz
```

Скачать с этого сайта и распаковать

Команда загрузки и установки

После скачивания и распаковки Storm мы можем создать тему Kafka, которую будем использовать для хранения результатов аналитики. Ниже описаны соответствующие шаги.

Листинг 9.18. Создание в Kafka темы для результатов анализа

```
$> cd $KAFKA_INSTALL_DIR
$> bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic meetuptopn-rsvps
    --partitions 1 --replication-factor 1
$> Created topic "meetup-topn-rsvps"
```

Перейти в установочный каталог Kafka

Создать тему, в которую будут записываться результаты анализа

Сначала перейдите в каталог, куда была установлена Kafka, а затем создайте тему «meetup-topn-rsvps». Теперь, когда Storm готов к работе, а тема создана, мы можем изучить код анализа и его интеграцию с другими частями конвейера.

9.3.2. Построение топологии Storm для нахождения *n* самых популярных тем

Сначала познакомимся с терминологией Storm. *Топологией* в Storm называется ориентированный ациклический граф (DAG), состоящий из *кранов* (spout) *и болтов* (bolt). Через краны данные вливаются из источника, а каждый элемент данных представляется в топологии кортежем. *Болтами* называются элементы бизнес-логики, осуществляющие вычисления с данными: агрегирование, фильтрация, пополнение или даже запись в другое место назначения. Желающим более подробно познакомиться с *Storm* повторно рекомендую книгу Sean T. Allen «Storm Applied» (Manning, 2015).

Построить топологию Storm для нахождения *n* самых популярных тем в приглашениях Meetup можно разными способами. Можно было бы применить традиционный многоболтовый подход, показанный на рис. 9.4.

См. раздел 5.3.3 главы 5

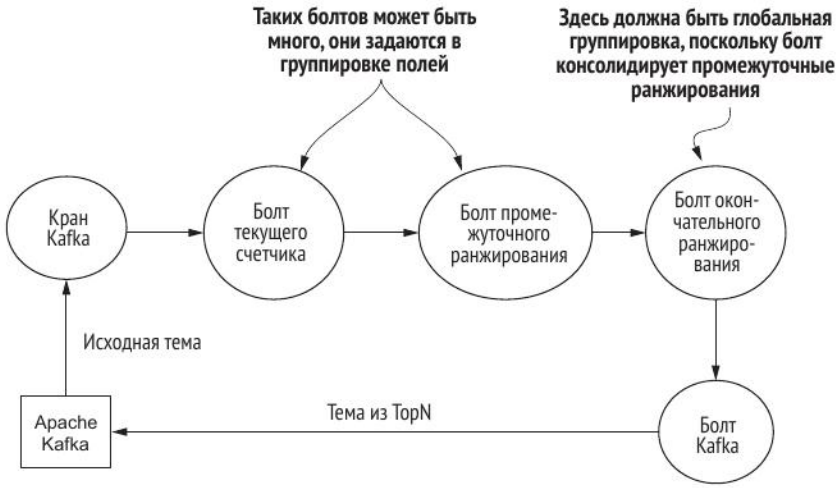


Рис. 9.4. Многоболтовый подход к нахождению *n* самых популярных тем

Топология, показанная на рис. 9.4, позволяет иметь много болтов текущих счетчиков и промежуточного ранжирования, каждый из которых порождает фрагмент конечного результата. Если таких болтов много, то при использовании группировки полей данные раздаются узлам кластера. Для болта окончательного ранжирования ситуация иная, по-

скольку его задача – породить полное ранжирование из промежуточных. Поэтому такой болт может быть только один, и группировка является глобальной – чтобы данные от всех промежуточных болтов стекались в болт окончательного ранжирования, который определит *n* самых популярных тем по всем данным. Чтобы увидеть эту топологию во плоти, ознакомьтесь с кодом проекта Apache Storm Starter по адресу https://github.com/apache/storm/tree/master/examples/storm-starter, где имеется хороший пример.

Но я предпочел более простой подход на основе топологии с использованием библиотеки обобщения потока stream-lib, которую можно найти по адресу https://github.com/addthis/stream-lib. Эта топология, изображенная на рис. 9.5, выглядит несколько иначе.

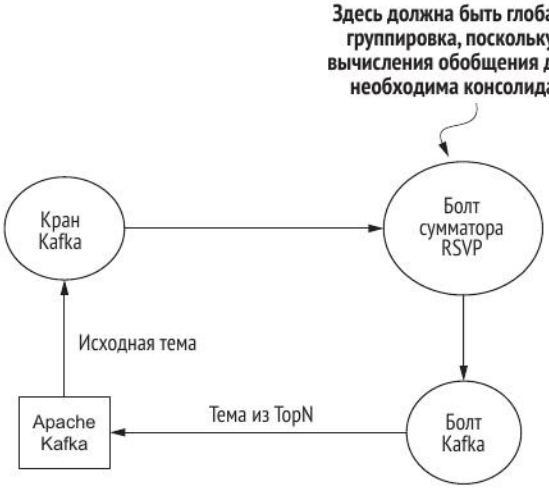


Рис. 9.5. Топология с использованием обобщения потока

Благодаря этой архитектуре мы устраняем несколько болтов и получаем возможность познакомиться с интеграцией библиотеки stream-lib, которая предлагает развитые средства анализа потока данных. В нашей топологии, как и в многоболтовой, имеется единственный болт, через который проходят все данные. Разница в том, что теперь мы используем библиотеку stream-lib, а из нее – класс `StreamSummary`. Этот класс предоставляет алгоритм нахождения первых *n* элементов, основанный на алгоритме экономии памяти и структуре данных для обобщения потока, описанных в работе Metwally, Agrawal, Abbadi «Efficient Computation of Frequent and Top-k Elements in Data Streams» (http://dl.acm.org/citation.cfm?id=2131596). В библиотеке есть и другие алгоритмы, специально предназначенные для операций с данными в потоке. Рекомендую (в качестве упражнения) заменить класс `StreamSummary` алгоритмом Count-Min-Sketch, который также имеется в stream-lib (класс `CountMinSketch`), чтобы наглядно увидеть, как рассмотренный в главе 5 алгоритм работает с реальными данными. Ниже показано, как конфигурируется наша топология.

Листинг 9.19. Конфигурирование топологии

```
private StormTopology buildTopolgy() {
    final TopologyBuilder tb = new TopologyBuilder();

    tb.setSpout("kafka_spout", new KafkaSpout<>(getKafkaSpoutConfig()), 1);

    tb.setBolt("rsvpSummarizer", new MeetupRSVPSummaryBolt(),1)
                .globalGrouping("kafka_spout", STREAM_NAME);

    tb.setBolt("summarySerializer", new
        KafkaBolt(),1).shuffleGrouping("rsvpSummarizer");

    return topologyBuilder.createTopology();
}
```

Создать TopologyBuilder ❶

Добавить в топологию кран ❷

❸ Добавить в топологию SummaryBolt

❹ Добавить в топологию KafkaBolt;

❺ Вернуть созданную топологию

Сначала ❶ мы создаем объект `ToplogyBuilder`, с помощью которого будем добавлять краны и болты. Затем ❷ настраиваем кран – объект встроенного класса `KafkaSpout`, – который будем использовать (чуть ниже мы рассмотрим код метода `getKafkaSpoutConfig`). Следующий шаг ❸ – настройка болта, который вычисляет *n* самых популярных тем средствами stream-lib. В точке ❹ мы добавляем в топологию болт `KafkaBolt`. И напоследок ❺ возвращаем топологию. Как мы увидим ниже, возвращенную топологию можно либо выполнить в локальном режиме, либо отправить в кластер. Метод `getKafkaSpoutConfig` содержит код инициализации.

Листинг 9.20. Метод `getKafkaSpoutConfig`

```
private KafkaSpoutConfig<String, String> getKafkaSpoutConfig() {
    return new KafkaSpoutConfig.Builder<>(
        getKafkaConsumerProps(),
        getKafkaSpoutStreams(),
        getTuplesBuilder(),
        getRetryService())
    .build();
}
```

Сконструировать объект-построитель KafkaSpoutConfig

Задать потоки для крана

Задать свойства потребителя

Создать построитель кортежей

Создать службу повтора

Построить SpoutConfig

В классе `KafkaSpoutConfig` несколько конфигурационных свойств, каждое из которых задается отдельным методом. Ниже показан метод `getKafkaConsumerProps`.

Листинг 9.21. Метод `getKafkaConsumerProps`

```
private Map<String, Object> getKafkaConsumerProps() {
    Map<String, Object> props = new HashMap<>();
```

❶ Создать отображение, содержащее все свойства

```
    props.put(KafkaSpoutConfig.Consumer.ENABLE_AUTO_COMMIT, "true");
    props.put(KafkaSpoutConfig.Consumer.BOOTSTRAP_SERVERS, "127.0.0.1:9092");
    props.put(KafkaSpoutConfig.Consumer.GROUP_ID, TOPIC_NAME + "-group");
    props.put(KafkaSpoutConfig.Consumer.KEY_DESERIALIZER,
      "org.apache.kafka.common.serialization.StringDeserializer");
    props.put(KafkaSpoutConfig.Consumer.VALUE_DESERIALIZER,
      "org.apache.kafka.common.serialization.StringDeserializer");
    return props;
}
```

❷ Задать десериализатор ключей

Задать десериализатор значений ❸

В отображении ❶ хранятся все свойства. Первое свойство, `ENABLE_AUTO_COMMIT`, означает, что потребитель Kafka – объект `Consumer` – должен фиксировать смещения, потому что мы обрабатываем сообщения автоматически. Второе свойство касается конфигурирования серверов Kafka – в данном случае сервер будет работать на локальной машине. Далее мы задаем свойство `GROUP_ID`, которое используется классом Kafka `Consumer`, чтобы уникально идентифицировать потребителей в группе. Интерес представляют также свойства ❷ и ❸, определяющие десериализаторы. Тем самым мы сообщаем классу `Consumer`, что хотим преобразовать байтовые массивы ключей и значений в строки. Далее мы рассмотрим метод `getKafkaSpoutStreams`.

Листинг 9.22. Метод `getKafkaSpoutStreams`

```
private KafkaSpoutStreams getKafkaSpoutStreams() {
    final Fields outputFields = new Fields( "topic",
                                            "partition",
                                            "offset",
                                            "key",
                                            "value");
    return new KafkaSpoutStreamsNamedTopics.Builder(
            outputFields,
            STREAM_NAME,
            new String[]{TOPIC_NAME}).build();
}
```

❶ Объявить, какие поля будет порождать кран

❷ Создать поток с этими полями

Все поля ❶, порождаемые краном, будут присутствовать в кортежах, которые получает болт. С каждым полем будет ассоциировано некоторое значение. Мы связываем ❷ с выходными полями имя потока и возвращаем объект `KafkaSpoutStream`. Далее показан следующий вызываемый построителем метод – `getTuplesBuilder`.

Листинг 9.23. Метод `getTuplesBuilder`

```
private KafkaSpoutTuplesBuilder<String, String> getTuplesBuilder() {
    return new KafkaSpoutTuplesBuilderNamedTopics.Builder<>(
```

```
        new TopicTupleBuilder(TOPIC_NAME)).build();
}
```

Сконструировать построитель кортежей

Теперь рассмотрим класс `TopicTupleBuilder`.

Листинг 9.24. Класс `TopicTupleBuilder`

```
public class TopicTupleBuilder<K,V> extends KafkaSpoutTupleBuilder<K,V> {
    public TopicTupleBuilder(String... topics) {
        super(topics);
    }

    @Override
    public List<Object> buildTuple(ConsumerRecord<K, V> consumerRecord) {
        return new Values(consumerRecord.topic(),
            consumerRecord.partition(),
            consumerRecord.offset(),
            consumerRecord.key(),
            consumerRecord.value());
    }
}
```

Принимает произвольное число тем и вызывает конструктор базового класса

Определяет, как конструировать кортеж из объекта ConsumerRecord ❶

Точка ❶ – то, ради чего существует построитель. Это довольно простой метод, который создает кортеж из объекта класса `ConsumerRecord`, являющегося частью Kafka. Для заполнения кортежа информацией, полученной из `ConsumerRecord`, можно было бы сделать и что-нибудь более сложное, например поискать дополнительные метаданные и включить их в кортеж. Мы рассмотрели почти все конфигурационные свойства, хранящиеся в `KafkaSpoutConfig`. Остался только метод `getRetryService`.

Листинг 9.25. Метод `getRetryService`

```
private KafkaSpoutRetryService getRetryService() {
    return new KafkaSpoutRetryExponentialBackoff(
        TimeInterval.microSeconds(500),
        TimeInterval.milliSeconds(2),
        Integer.MAX_VALUE,
        TimeInterval.seconds(10));
}
```

Создает и конфигурирует службу ExponentialBackoff

Этот метод создает и конфигурирует объект встроенного класса `ExponentialBackoff`. Подробнее о том, как устроена эта реализация повтора, см. в документации класса по адресу http://mng.bz/hdrh.

Таким образом, с точкой ❷ в листинге 9.19 мы разобрались. Теперь рассмотрим класс `MeetupRSVPSummaryBolt`, объект которого создается в точке ❸ того же листинга. Именно в этом классе вычисляются первые *n* объектов.

Листинг 9.26. Класс `MeetupRSVPSummaryBolt`

```
public class MeetupRSVPSummaryBolt extends BaseRichBolt {
    private static StreamSummary<String> streamSummary =
        new StreamSummary<>(100000);
    private static final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void execute(Tuple tuple) {
        String jsonString = tuple.getString(4);

        JsonNode root = objectMapper.readTree(jsonString);
        JsonNode groupTopics = root.get("group").get("group_topics");

        Iterator<JsonNode> groupTopicsItr = groupTopics.iterator();
        while(groupTopicsItr.hasNext()){
          JsonNode groupTopic = groupTopicsItr.next();
          final String topicName = groupTopic.get("topic_name").asText();
          streamSummary.offer(topicName);
        }

        collector.emit(new Values(UUID.randomUUID().toString(),
            objectMapper.writeValueAsString(streamSummary.topK(10))
        ));
    }
}
```

❶ Сконструировать объект StreamSummary

❷ Сконструировать JSON-сериализатор

❸ Получить объект RSVP в формате JSON

❹ Преобразовать его в JsonNode

❺ Получить узел group_topics

❻ Получить название групповой темы

❼ Добавить его в сводку

❽ Получить первые 10 тем

В точках ❶ и ❷ мы настраиваем объекты, которые понадобятся соответственно для формирования сводки потока и манипулирования данными в формате JSON. В точке ❸ мы получаем из кортежа данные приглашения RSVP в формате JSON. В листинге 9.21 ❶ мы объявили, что данные RSVP будут находиться в позиции 4 кортежа. Получив представление в формате JSON, мы можем найти узел `group_topics`, обойти его, получить ❻ название темы и добавить ❼ его в формируемую сводку. Обойдя таким образом все темы, мы можем породить выходные данные. Это делается в точке ❽, где мы порождаем UUID и JSON-представление первых 10 результатов из сводки потока. Далее приведен фрагмент этой сводки в формате JSON.

Листинг 9.27. Фрагмент `StreamSummary` в формате JSON

```
[
{
  "item": "Social",
  "count": 59,
  "error": 0
},
```

```
  {
    "item": "Social Networking",
    «count»: 45,
    «error»: 0
  },
  ... (еще 8 элементов) ...
]
```

Тут нет ничего сложного: групповая тема, представленная элементом `item`, сколько раз она встречалась и счетчик ошибок. Мы еще вернемся к этой структуре, когда будем строить пользовательский интерфейс.

На последнем шаге конфигурирования топологии мы используем болт для записи в Kafka. В состав Storm входит встроенный болт для Kafka, и еще один включен в пример кода для этого звена. Поскольку мы разобрали код порождения данных для Kafka при обсуждении службы сбора данных, этот листинг мы опустим, поскольку ничего нового в нем нет. В коде к этой главе он имеется, так что можете скачать и посмотреть, что он делает.

Теперь мы наконец готовы отправить построенную топологию кластеру Storm. Сделать это можно двумя способами. В листинге ниже показано, как это делается для локального и удаленного кластеров.

Листинг 9.28. Отправка топологии кластеру Storm

```
private void runLocally() throws InterruptedException {
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology(this.TOPOLOGY_NAME, getConfig(),buildTopolgy());
    stopWaitingForInput();
}

private void runRemotely() throws AlreadyAliveException,
                                  InvalidTopologyException,
                                  AuthorizationException {
    StormSubmitter.submitTopology(TOPOLOGY_NAME,
        getConfig(),
        buildTopolgy());
}
```

❶ Создать локальный кластер

❷ Отправить удаленному кластеру

Локальный кластер ❶ хорош для разработки и тестирования. Нам только и нужно, что создать кластер и передать ему задачу. Метод `stopWaitingForInput` ожидает, пока пользователь нажмет любую клавишу, после чего завершает работу. В точке ❷ мы передаем ту же задачу удаленному кластеру. Этот кластер может быть запущен на той же машине, но все равно является настоящим кластером Storm. В этом упражнении мы будем использовать только `LocalCluster`, но я все же настоятельно рекомендую собрать реальный кластер Storm и развернуть на нем нашу топологию. Ну а мы наконец готовы интегрировать звено анализа с ранее разработанными звеньями.

9.3.3. Интеграция звена анализа в конвейер

В конце этого раздела у нас будет служба сбора данных, которая передает данные в Kafka, и звено анализа, которое читает эти данные, находит первые *n* тем и записывает результат снова в Kafka. Для начала нужно собрать звено анализа, написанное в предыдущем разделе, для чего выполните следующие команды.

Листинг 9.29. Сборка и запуск звена анализа

```
$> cd $EXAMPLE_CODE_HOME/Chapter9/analysis-tier
$> mvn clean package
$> $STORM_INSTALL_DIR/bin/storm jar target/analytics-0.0.1.jar
com.streamingdata.analysis.TopMeetupTopicsTopology
```

Перейти в каталог с исходным кодом звена анализа

Собрать проект с помощью Maven

Развернуть с помощью Storm

Соберите проект и отправьте топологию локальному кластеру – напомним, что он используется для разработки и тестирования. Итак, топология работает. Если служба сбора данных в данный момент не запущена, запустите ее. Спустя некоторое время вы увидите в окне консоли, в котором запущена топология, сообщение о том, что кортежи обрабатываются. Помимо окон, в которых запущены служба сбора данных, Kafka и звено анализа, откроем еще одно окно, в котором можно будет наблюдать данные, передаваемые Kafka звеном анализа. Для этого воспользуемся консольным потребителем Kafka, его можно запустить следующей командой.

Листинг 9.30. Потребление *n* самых популярных тем

```
$> $KAFKA_HOME/bin/kafka-console-consumer.sh --zookeeper localhost:2181
    --topic meetup-topn-rsvps
```

Теперь в этом окне начнут появляться сообщения. Поздравляю! В данный момент работают служба сбора данных, Kafka, ZooKeeper и звено анализа. Мы потребляем поток приглашений от Meetup, анализируем его и подготавливаем к потреблению. Первые пять глав книги уже воплощены на практике! В следующем разделе мы займемся построением API доступа к данным и еще на один шаг приблизимся к созданию законченного конвейера.

9.4. Хранилище данных в памяти

См. раздел 6.2 главы 6

В этом звене мы будем использовать Apache Kafka, хотя это не первая технология, которая приходит на ум, когда думаешь о хранилище данных в памяти. Я выбрал ее по следующим причинам:

- она уже используется в звене очереди сообщений, поэтому архитектура останется простой и чистой;
- хотя Kafka пишет данные на диск, скорость чтения и записи достаточно высока для того приложения, которое мы разрабатываем;
- на стороне потребителя, по существу, производится потоковый доступ. А интерпретация данных, к которым обращается звено доступа, как потока упрощает построение конвейера.

Код этого звена сводится к единственной строке: `tb.setBolt("summary-Serializer", new KafkaBolt(),1).shuffleGrouping("rsvpSummarizer")` в точке ❹ листинга 9.19. Эта строка связывает встроенный в Storm болт `KafkaBolt` с выходом нашего объекта `"rsvpSummarizer"`. Когда код в точке ❽ листинга 9.25 порождает первые 10 элементов в формате JSON, болт `KafkaBolt` получает их и записывает в Kafka. Таким образом, для записи данных в хранилище в памяти понадобилась всего одна строчка – неплохое достижение. Конечно, количество строк кода – не решающий фактор, но что-то он все-таки значит. После запуска звена анализа, как описано в конце раздела 9.3, мы можем переходить к разделу 9.5, где воспользуемся полученными данными.

9.5. Звено доступа к данным

Это последний шаг, после которого система оживет. Все уже на мази. Мы запитываем систему данными, передаем их звену очереди сообщений, анализируем и подготавливаем к потреблению. А сейчас сделаем их доступными пользовательскому интерфейсу (UI). В этом разделе обсуждается реализация API доступа к данным и простого UI, который потребляет данные, пользуясь этим API. В конце раздела мы будем иметь готовую систему и сможем наглядно увидеть работу сквозного конвейера потоковой обработки данных.

Для реализации API в этом звене я использовал Netty, чему есть две причины. Во-первых, это высокопроизводительный сетевой каркас с надежной поддержкой протокола WebSocket. Во-вторых, Netty предоставляет низкоуровневый доступ к соединению. Если мы собираемся использовать систему для производственных целей, то такой доступ понадобится как минимум для того, чтобы понять, подключен ли еще клиент и можно ли писать в сокет. Тем, кто хочет больше узнать о Netty, и особенно тем, кто собирается использовать его в производственной среде, я настоятельно рекомендую прочитать книгу Norman Maurer, Allen Wolfthal «Netty In Action» (Manning, 2015).

Итак, перейдем к коду. Все начинается с кода запуска Netty, который показан ниже.

Листинг 9.31. StreamingDataService.java

См. раздел 7.2.4 главы 7

```
public final class StreamingDataService {
    private static final int PORT = 8080;                          // Прослушиваемый порт
    private static final String bindAddress = "127.0.0.1";         // IP-адрес для привязки

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();      // ❶ Циклы обработки событий для Netty

        try {
          ServerBootstrap bootstrap = new ServerBootstrap();
          bootstrap.group(bossGroup, workerGroup)
            .channel(NioServerSocketChannel.class)
            .handler(new LoggingHandler(LogLevel.DEBUG))
            .childHandler(new ChannelInitializer<SocketChannel>() {   // Конвейер, который будет обрабатывать запрос ❷

              @Override
              public void initChannel(SocketChannel ch){
                ChannelPipeline pipeline = ch.pipeline();
                pipeline.addLast(new HttpServerCodec());
                pipeline.addLast(new HttpObjectAggregator(65536));
                pipeline.addLast(new WebSocketServerCompressionHandler());
                pipeline.addLast(new WebSocketServerProtocolHandler(
                    "/streaming", null, true));
                pipeline.addLast(new MeetupTopNSocketServerHandler());   // Наш обработчик протокола WebSocket ❸
              }
            });

            Channel channel = bootstrap                            // Привязать сокет
              .bind(bindAddress, PORT)
              .sync()
              .channel();

            System.out.println("Open your web browser to http://" +   // Дать знать пользователю
                bindAddress + ":" + PORT + '/');
            channel.closeFuture().sync();                          // Ждать завершения
          }finally {
            bossGroup.shutdownGracefully();                        // Остановить Netty после завершения работы
            workerGroup.shutdownGracefully();
          }
    }
}
```

Этот код может показаться сложным, но большая его часть – трафаретный код Netty. В точке ❶ объявляются группы событий Netty, а в точке ❷ они конфигурируются. В точке ❸ мы добавляем обработчик `Meetup-`

`TopNSocketServerHandler` в конец конвейера. На верхнем уровне работу Netty можно описать как выполнение собранного нами конвейера, поэтому мы хотим, чтобы наш обработчик находился в его конце.

Далее мы должны рассмотреть класс `MeetupTopNSocketServerHandler`. Если вы скачаете код для этой главы и заглянете в этот класс, то увидите в нем много трафаретного кода, поскольку он занимается низкоуровневыми сетевыми деталями, которые раскрывает Netty. Поэтому мы покажем только части, относящиеся к интересующему нас предмету, и первая из них – метод `handleWebSocketRequest`.

Листинг 9.32. Метод `handleWebSocketRequest`

```
class MeetupTopNSocketServerHandler extends SimpleChannelInboundHandler<Object> {
    ConcurrentHashMap<ChannelHandlerContext,StreamMessageConsumer>
        channelToConsumer = new ConcurrentHashMap<>();                    ❶ Отобразить контекст на потребителя

    private void handleWebSocketRequest(ChannelHandlerContext ctx,
        WebSocketFrame frame) {
        // Игнорировать входные данные.....                               Пример доступа к входным данным ❷
        // Доступ к ним осуществляется следующим образом:
        // final String jsonRequest = ((TextWebSocketFrame) frame).text();
        if (!channelToConsumer.containsKey(ctx)) {                        ❸ Определить, подключен ли клиент
          StreamMessageConsumer streamMessageConsumer = new
              StreamMessageConsumer("meetup-topn-rsvps",
                                    UUID.randomUUID().toString(), ctx);
          channelToConsumer.put(ctx, streamMessageConsumer);
          streamMessageConsumer.process();                                ❹ Начать обработку данных
        }

        TextWebSocketFrame returnframe = new
            TextWebSocketFrame(mapper.writeValueAsString("{response:success}"));
        ctx.channel().write(returnframe);                                 Вернуть код успешного завершения ❺
    }
}
```

В точке ❶ мы определяем отображение, которое позволит связать подключенный канал с тем, что окажется потребителем Kafka. Если бы мы хотели, чтобы клиент передавал информацию нашему обработчику в момент подключения, то точка ❷ – то место, где эту информацию следовало бы получить. В точке ❸ происходит самое интересное – мы проверяем, видели ли уже этот канал, и если нет, то создаем объект `StreamMessageConsumer`, что (как мы скоро увидим) приводит к созданию потребителя Kafka. Сконструировав объект и поместив его в свой кэш, мы в точке ❹ говорим, чтобы он приступил к обработке. Все подготовив, мы в точке ❺ возвращаем клиенту код успешного завершения.

Теперь рассмотрим относящиеся к делу части класса `StreamMessageConsumer`, рабочей лошадки этой службы. Два самых важных метода, `process` и `run`, показаны в следующем листинге.

Листинг 9.33. Методы `process` и `run`

```
class StreamMessageConsumer {

    void process() throws Exception {
        messageProcessingThread.start();
    }

    final class MessageProcessor extends Thread {
        @Override
        public synchronized void run(){
          while(!done){
            ConsumerRecords<String, String> records = consumer.poll(100);
            for(ConsumerRecord<String, String> record : records) {
              if (!channelHandlerContext.channel().isOpen()) {
                close();
              }else if(channelHandlerContext.channel().isWritable()) {
                channelHandlerContext.channel().writeAndFlush(new
                    TextWebSocketFrame(record.value()));
              }
            }
          }
        }
    }
}
```

❶ Вызывается из MeetupTopNSocketServerHandler

Пытаемся потребить данные от Kafka ❷

❸ Если канал закрыт, чистим

❹ Записать сообщение клиенту

Метод ❶ вызывается из метода `handleWebSocketRequest` класса `MeetupTopNSocketServerHandler`, показанного в листинге 9.32. Он запускает внутренний поток, который обрабатывает сообщения от Kafka. В точке ❷ мы опрашиваем Kafka на предмет появления новых сообщений, а затем обходим записи. Если WebSocket-клиент, подключившийся к службе, отключится, то мы обнаружим это в точке ❸ и, в свою очередь, остановим потребителя Kafka. В противном случае мы в точке ❹ записываем в сокет сообщение, полученное от Kafka.

Теперь потоковый API доступа к данным готов к работе. Чтобы собрать и запустить его, выполните в новом окне терминала приведенные ниже команды.

Листинг 9.34. Сборка и запуск службы потокового доступа к данным

```
$> cd $EXAMPLE_CODE_HOME/Chapter9/streaming-api
$> mvn clean package
$> java -jar target/streaming-api-0.0.1.jar
```

Теперь потоковый API работает. Следующий шаг – запустить службу сбора данных и развернуть топологию `TopMeetupTopicsTopology`. После того как они запустятся, нам останется только открыть пользовательский интерфейс и наблюдать за тем, как текут данные. Для этого перейдите в браузере по адресу http://127.0.0.1:8080 – вы увидите UI, показанный на рис. 9.6.

См. раздел 8.3.1 главы 8

Рис. 9.6. Пользовательский интерфейс без данных

И еще один шаг. Нажмите кнопку **Open** – если все работает правильно, то по экрану побегут данные, как показано на рис. 9.7.

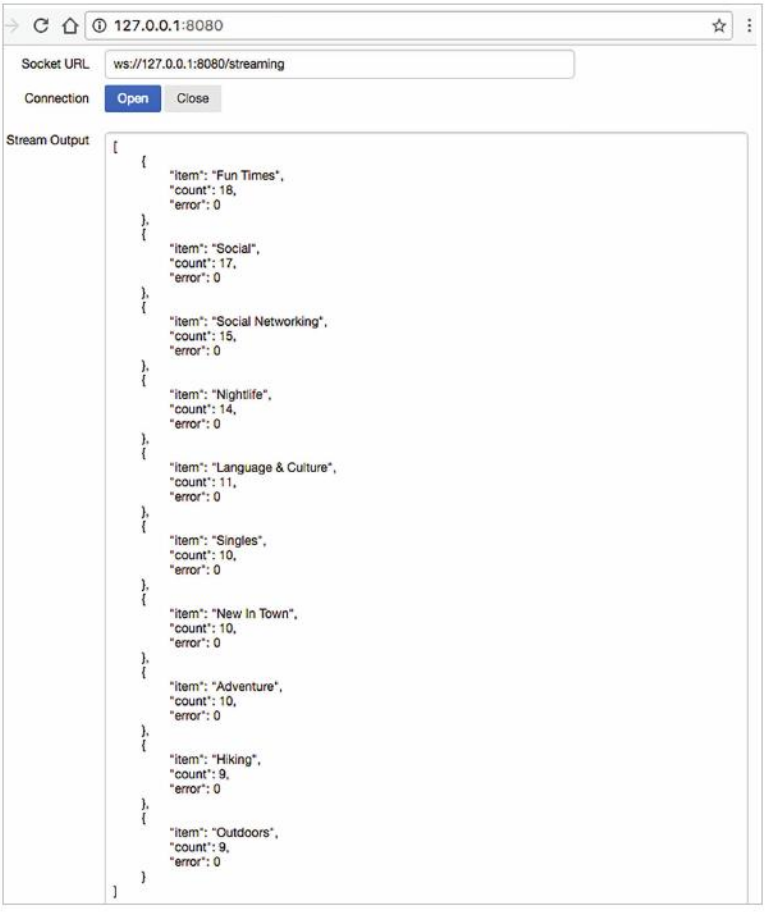


Рис. 9.7. Пользовательский интерфейс, отображающий поток данных

Мои поздравления! Если вы видите бегущие данные, значит, у вас есть законченный работающий конвейер потоковой обработки данных!

9.5.1. На пути к производственному режиму

Итак, у нас есть полный потоковый конвейер и потоковый API, для создания которых пришлось написать сравнительно немного кода. Отметим несколько моментов, которые следует продумать и добавить к потоковому API, прежде чем развертывать его в производственном режиме.

- Бывает так, что WebSocket-клиент не подключен к потоковому API или не может обработать данные, которые вы ему отправляете. Чтобы предотвратить потерю данных, подумайте о том, чтобы реализовать алгоритм HML по образцу того, что мы сделали в службе сбора данных.
- Если к службе подключено много клиентов, то, наверное, стоит более правильно контролировать группы, используемые при подключении к Kafka. В примере кода к этой главе отображение между WebSocket-клиентами и группами Kafka было взаимно-однозначным. Но было бы разумно организовать связь *n*:1 – тогда может существовать несколько WebSocket-клиентов, потребляющих одни и те же данные из Kafka.
- Ваши пользователи захотят предъявлять запросы к данным, поступающим из Kafka, и вы должны подумать, как добавить такой механизм. Один из возможных способов – позволить клиенту передавать запрос на стадии подключения, а затем сохранить его и применять к потоку данных.

9.6. Резюме

В этой главе мы от начала до конца построили конвейер потоковой обработки данных, попутно отметив моменты, которые следует иметь в виду при разработке производственного конвейера. Вооруженные всеми полученными знаниями, вы теперь вполне можете создавать такие конвейеры самостоятельно.

Вот чему мы научились:

- как реализуется каждое звено;
- как звенья интегрируются между собой;
- соображения о производственной реализации различных звеньев.

Надеюсь, вы наслаждались, читая эту главу, так же, как наслаждался я, когда ее писал: смотреть, как система оживает, очень поучительно. Самое время налить кофейку и полюбоваться потоком данных в пользовательском интерфейсе.

Предметный указатель