

Алекс Сюй

# System Design

Подготовка к сложному интервью



# **SYSTEM DESIGN INTERVIEW**

---

Alex Xu

Алекс Сюй

# System Design

Подготовка к сложному интервью



Санкт-Петербург · Москва · Минск

2022

ББК 32.973.2-02  
УДК 004.41  
С98

## Сью Алекс

С98 System Design. Подготовка к сложному интервью. — СПб.: Питер, 2022. — 304 с.: ил. — (Серия «Библиотека программиста»).  
ISBN 978-5-4461-1816-8

Интервью по System Design (проектированию ИТ-систем) очень популярны у работодателей, на них легко проверить ваши навыки общения и оценить умение решать реальные задачи. Пройти такое собеседование непросто, поскольку в проектировании ИТ-систем не существует единственно правильных решений. Речь идет о самых разнообразных реальных системах, обладающих множеством особенностей. Вам могут предложить выбрать общую архитектуру, а потом пройтись по всем компонентам или, наоборот, сосредоточиться на каком-то одном аспекте. Но в любом случае вы должны продемонстрировать понимание и знание системных требований, ограничений и узких мест.

Правильная стратегия и знания являются ключевыми факторами успешного прохождения интервью!

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-02  
УДК 004.41

Права на издание получены по соглашению с Byte Code LLC. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 979-8664653403 англ.  
ISBN 978-5-4461-1816-8

© Byte Code LLC  
© Перевод на русский язык ООО Издательство «Питер», 2022  
© Издание на русском языке, оформление ООО Издательство «Питер», 2022  
© Серия «Библиотека программиста», 2022

# ОГЛАВЛЕНИЕ

Об авторе . . . . .	6
Введение . . . . .	7
<b>Глава 1</b> Масштабирование от нуля до миллионов пользователей . . . . .	9
<b>Глава 2</b> Приблизительные оценки . . . . .	40
<b>Глава 3</b> Общие принципы прохождения интервью по проектированию ИТ-систем . . . . .	46
<b>Глава 4</b> Проектирование ограничителя трафика . . . . .	58
<b>Глава 5</b> Согласованное хеширование . . . . .	83
<b>Глава 6</b> Проектирование хранилища типа «ключ–значение» . . . . .	96
<b>Глава 7</b> Проектирование генератора уникальных идентификаторов в распределенных системах . . . . .	122
<b>Глава 8</b> Проектирование системы для сокращения URL-адресов . . . . .	131
<b>Глава 9</b> Проектирование поискового робота . . . . .	144
<b>Глава 10</b> Проектирование системы уведомлений . . . . .	166
<b>Глава 11</b> Проектирование ленты новостей . . . . .	183
<b>Глава 12</b> Проектирование системы мгновенного обмена сообщениями . . . . .	197
<b>Глава 13</b> Проектирование системы автозаполнения поисковых запросов . . . . .	222
<b>Глава 14</b> Проектирование YouTube . . . . .	244
<b>Глава 15</b> Проектирование Google Drive . . . . .	273
<b>Глава 16</b> Век живи — век учись . . . . .	297
Послесловие . . . . .	301

## ОБ АВТОРЕ

Алекс Сюй — опытный разработчик программного обеспечения и предприниматель. Ранее он работал в таких компаниях, как Twitter, Apple, Zynga и Oracle. Алекс получил степень магистра наук в Университете Карнеги-Меллона. Его страсть — проектирование и реализация сложных систем.

# ВВЕДЕНИЕ

Мы весьма рады, что вы решили изучить особенности интервью по проектированию ИТ-систем вместе с нами. Из всех технических интервью именно на этом задают самые сложные вопросы. Претенденту предлагается спроектировать архитектуру программной системы: новостной ленты, поиска Google, системы мгновенных сообщений и т. д. Задачи такого рода наводят ужас, ведь у них нет единственно верных решений. Они обычно отличаются масштабностью и расплывчатостью. Допускаются свободные и неясные формулировки без стандартного или правильного ответа.

Интервью по проектированию ИТ-систем широко практикуются в компаниях, так как навыки общения и решения задач, которые можно проверить на этом этапе, необходимы в повседневной работе программиста. Ответы претендента оцениваются с учетом того, как он анализирует расплывчатую задачу и какие шаги он предпринимает для ее решения. При этом во внимание принимается то, как он объясняет свои идеи, обсуждает их с другими, оценивает и оптимизирует систему.

Здесь нет единственно правильных ответов. Как и в реальной жизни, на интервью обсуждают самые разные системы, каждую со своими нюансами. Претендент должен предложить архитектуру для достижения поставленных целей. Ход обсуждения может быть разным в зависимости от того, кто проводит интервью. Кто-то может выбрать общую архитектуру, чтобы пройтись по всем компонентам, а кто-то предпочитает сосредоточиться на одной или нескольких областях. Обычно ход мыслей всех участников интервью определяется тем, насколько хорошо они понимают системные требования, ограничения и узкие места.

Цель этой книги — предоставить надежную стратегию для решения задач по проектированию систем. Правильная стратегия и знания являются ключевыми факторами успешного прохождения интервью.

В этой книге шаг за шагом описывается систематический подход к поиску ответов на вопросы о проектировании систем. Здесь вы найдете множество наглядных примеров с подробными инструкциями. Постоянно практикуясь, вы сможете хорошо подготовиться к такого рода интервью.

## **ОТ ИЗДАТЕЛЬСТВА**

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

---

# 1

---

## МАСШТАБИРОВАНИЕ ОТ НУЛЯ ДО МИЛЛИОНОВ ПОЛЬЗОВАТЕЛЕЙ

Проектирование системы с поддержкой миллионов пользователей — не простая задача. Это процесс, требующий непрерывного совершенствования и бесконечного улучшения. В этой главе мы создадим систему, которая работает в однопользовательском режиме, и постепенно расширим ее для обслуживания миллионов пользователей. Прочитав эту главу, вы освоите несколько методик, которые помогут вам справиться с вопросами на интервью по проектированию ИТ-систем.

### КОНФИГУРАЦИЯ ИЗ ОДНОГО СЕРВЕРА

Путь в тысячу ли начинается с первого шага. То же самое относится и к созданию сложной системы. Начнем с чего-то простого и разместим все на одном сервере. На рис. 1.1 показана конфигурация одного сервера, на котором запускаются все компоненты: веб-приложение, база данных, кэш и т. д.

Исследование потока запросов и источника трафика поможет разобраться в этой конфигурации. Давайте сначала взглянем на поток запросов (рис. 1.2).

1. Пользователи обращаются к веб-сайтам по их доменным именам, таким как `api.mysite.com`. Обычно система доменных имен (Domain Name System, DNS) представляет собой сторонний платный сервис, размещенный за пределами наших серверов.
2. Адрес интернет-протокола (Internet Protocol, IP) возвращается браузеру или мобильному приложению. В этом примере он имеет вид `15.125.23.214`.

3. После получения IP-адреса вашему веб-браузеру напрямую отправляются запросы протокола передачи гипертекста (Hypertext Transfer Protocol, HTTP) [1].

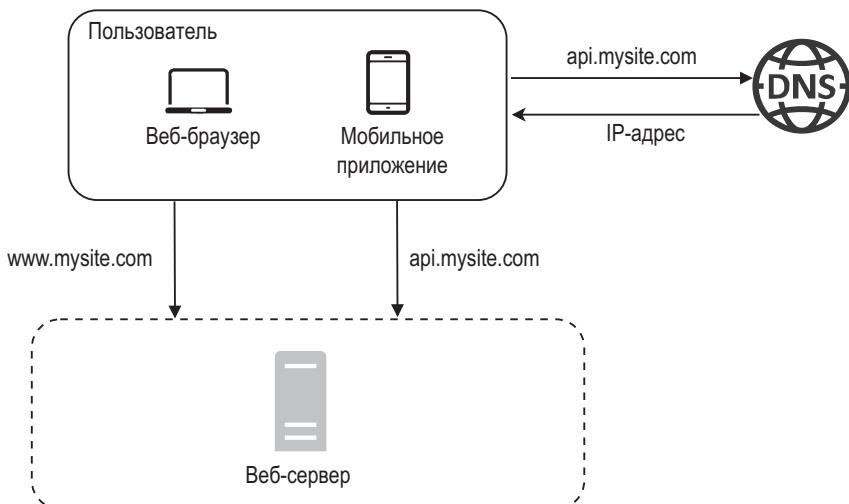


Рис. 1.1

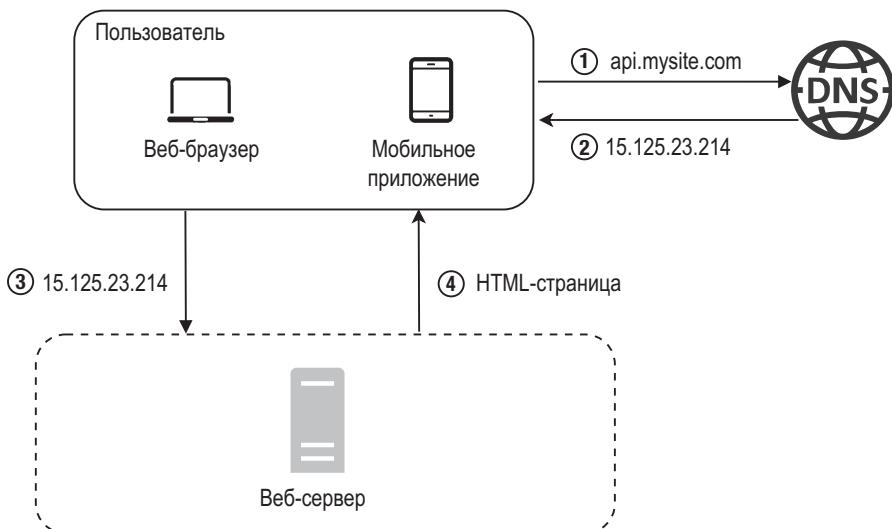


Рис. 1.2

4. Веб-сервер возвращает HTML-страницы или JSON-ответы для рендеринга.

Теперь давайте исследуем источник трафика. Трафик, который получает ваш веб-сервер, приходит из двух мест: из веб- и мобильного приложения.

- Веб-приложение использует сочетание серверных и клиентских языков. Первые (Java, Python и т. д.) предназначены для реализации бизнес-логики, системы хранения и т. п., а вторые (HTML и JavaScript) — для предоставления информации.
- Мобильное приложение использует протокол HTTP для взаимодействия с веб-сервером. Для передачи данных зачастую применяется формат API-ответов JSON (JavaScript Object Notation — «представление объектов JavaScript»), отличающийся своей простотой. Пример API-ответа в формате JSON показан ниже:

*GET /users/12 – Retrieve user object for id = 12*

```
{  
    "id": 12,  
    "firstName": "John",  
    "lastName": "Smith",  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": 10021  
    }  
    "phoneNumbers": [  
        "212 555-1234",  
        "646 555-4567"  
    ]  
}
```

## БАЗА ДАННЫХ

С увеличением количества пользователей одного сервера рано или поздно перестанет хватать, и нам понадобится несколько серверов: один для веб-/мобильного трафика, а другой — для базы данных (рис. 1.3). Разделение системы на веб-уровень и уровень данных позволяет масштабировать эти компоненты независимо друг от друга.

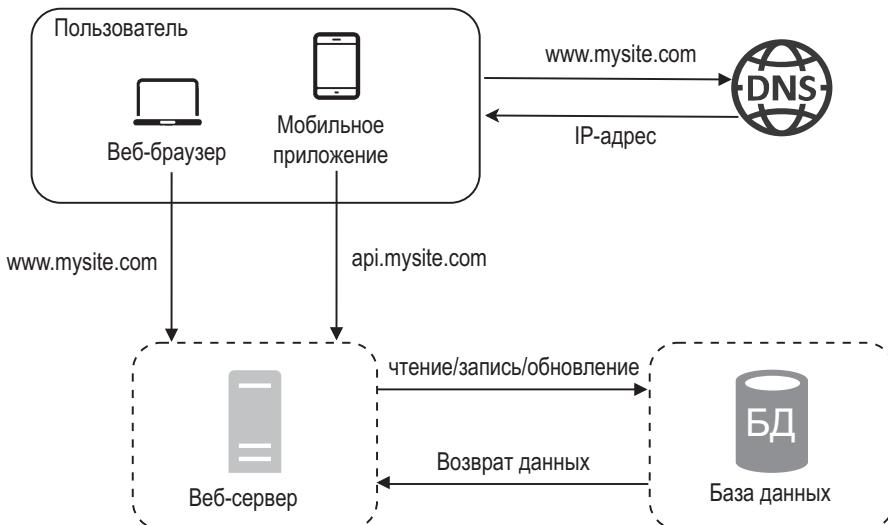


Рис. 1.3

### Какую базу данных выбрать?

Базы данных (БД) бывают как реляционными (что является традиционным решением), так и нереляционными. Давайте посмотрим, чем они отличаются.

Реляционные БД также называют системами управления реляционными базами данных (СУРБД). К самым популярным относятся MySQL, Oracle, PostgreSQL и т. д. Реляционные БД предоставляют и хранят данные в таблицах и строках. С помощью SQL можно выполнять операции объединения между различными таблицами базы данных.

Нереляционные БД также называют NoSQL. Популярностью пользуются CouchDB, Neo4j, Cassandra, HBase, Amazon DynamoDB и т. д. [2]. Эти базы данных делятся на четыре категории: хранилища «ключ–значение», графовые, столбцовые и документные. Нереляционные БД обычно не поддерживают операции соединения.

Большинство разработчиков предпочитает реляционные базы данных, поскольку они применяются уже на протяжении более 40 лет и хорошо себя зарекомендовали. Но если они не подходят для ваших конкретных

задач, стоит непременно обратить внимание на другие варианты. Нереляционные БД могут быть подходящим решением, если:

- ваше приложение нуждается в крайне низкой латентности;
- ваши данные не структурированы или не имеют никаких реляционных связей;
- вам нужно лишь сериализовать и десериализовать свои данные (JSON, XML, YAML и т. д.);
- вам нужно хранить огромные объемы данных.

## ВЕРТИКАЛЬНОЕ И ГОРИЗОНТАЛЬНОЕ МАСШТАБИРОВАНИЕ

Вертикальное масштабирование, известное как наращивание, — это процесс повышения мощности ваших серверов (процессоров, памяти и т. д.). Горизонтальное масштабирование, которое еще называют расширением, заключается в добавлении новых серверов в пул ресурсов.

Вертикальное масштабирование отлично подходит для задач с небольшим трафиком. Его главным преимуществом является простота. К сожалению, у него есть ряд серьезных ограничений.

- Вертикальное масштабирование имеет жесткий лимит. Ресурсы отдельно взятого сервера нельзя увеличивать бесконечно.
- Вертикальное масштабирование не предусматривает отказоустойчивость и резервирование избыточных ресурсов. Если один из серверов выйдет из строя, веб-сайт/приложение станет полностью недоступным.

Из-за этих ограничений для крупномасштабных приложений лучше подходит горизонтальное масштабирование.

В предыдущей конфигурации пользователи подключались к веб-серверу напрямую. Если веб-сервер выйдет из строя, они потеряют доступ к веб-сайту. Если же к веб-серверу одновременно обратится большое количество пользователей и нагрузит его до предела, в результате, как правило, ответы будут приходить медленно либо к серверу вовсе станет невозможно подключиться. Для решения этих проблем больше всего подходит балансировщик нагрузки.

## БАЛАНСИРОВЩИК НАГРУЗКИ

Балансировщик нагрузки равномерно распределяет входящий трафик между веб-серверами, которые указаны в его списке. На рис. 1.4 показано, как это работает.

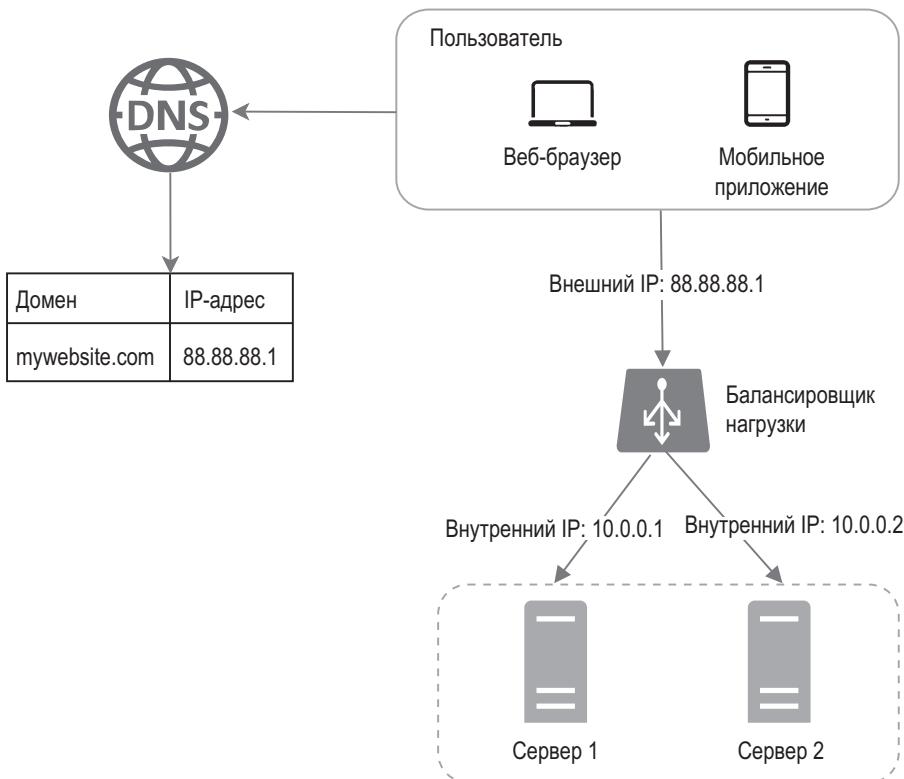


Рис. 1.4

Как видно на рис. 1.4, пользователи напрямую подключаются к внешнему IP-адресу балансировщика нагрузки. В этой конфигурации клиенты больше не имеют прямого доступа к веб-серверам. По соображениям безопасности для взаимодействия между серверами используются внутренние IP-адреса. Внутренний IP доступен только для серверов из той же сети, но не виден из интернета. Балансировщик нагрузки взаимодействует с веб-серверами с помощью внутренних IP-адресов.

На рис. 1.4 показано, как за счет добавления балансировщика нагрузки и второго сервера нам удалось решить проблему с отсутствием отказоустойчивости и улучшить доступность веб-уровня. Подробности объясняются ниже.

- Если сервер 1 выходит из строя, весь трафик перенаправляется к серверу 2. Благодаря этому веб-сайт остается доступным. Чтобы сбалансировать нагрузку, мы добавим в пул серверов новый исправный веб-сервер.
- Если посещаемость веб-сайта стремительно растет и для обслуживания трафика не хватает двух серверов, балансировщик нагрузки может изящно справиться с этой проблемой. Для этого достаточно расширить пул серверов, и балансировщик начнет автоматически передавать запросы новым веб-серверам.

Веб-уровень выглядит хорошо, но что насчет уровня данных? Текущая конфигурация предусматривает лишь одну БД, что исключает поддержку отказоустойчивости и резервирования. Для решения этих проблем обычно применяют репликацию. Давайте посмотрим, что это такое.

## РЕПЛИКАЦИЯ БАЗЫ ДАННЫХ

Цитата из английской Википедии: «Репликация баз данных может использоваться во многих СУБД, обычно в режиме “ведущий–ведомый”, где роль ведущего сервера играет оригинал (master), а его копии являются ведомыми (slave)» [3].

Ведущая база данных обычно поддерживает только операции записи. Ведомые БД получают от ведущей копии ее содержимого и поддерживают только операции чтения. Все команды для модификации данных, такие как вставка, удаление или обновление, должны направляться ведущей базе данных. В большинстве приложений чтение происходит намного чаще, чем запись, поэтому ведомых БД обычно больше, чем ведущих. На рис. 1.5 показана ведущая база данных с несколькими ведомыми.

Преимущества репликации базы данных:

- Повышенная производительность. В модели «ведущий–ведомый» все операции записи и обновления происходят на ведущих узлах, а операции чтения распределяются между ведомыми. Это улучшает

производительность, увеличивая количество запросов, которые можно обрабатывать параллельно.

- Надежность. Если один из ваших серверов с базой данных сломается из-за стихийного бедствия, такого как тайфун или землетрясение, данные не будут потеряны. Вам не нужно беспокоиться о потере данных, так как они реплицируются по разным местам.
- Высокая доступность. За счет репликации данных по разным местам ваш веб-сайт будет продолжать работать, даже если одна из БД выйдет из строя, поскольку у вас по-прежнему будет доступ к данным, размещенным на другом сервере.

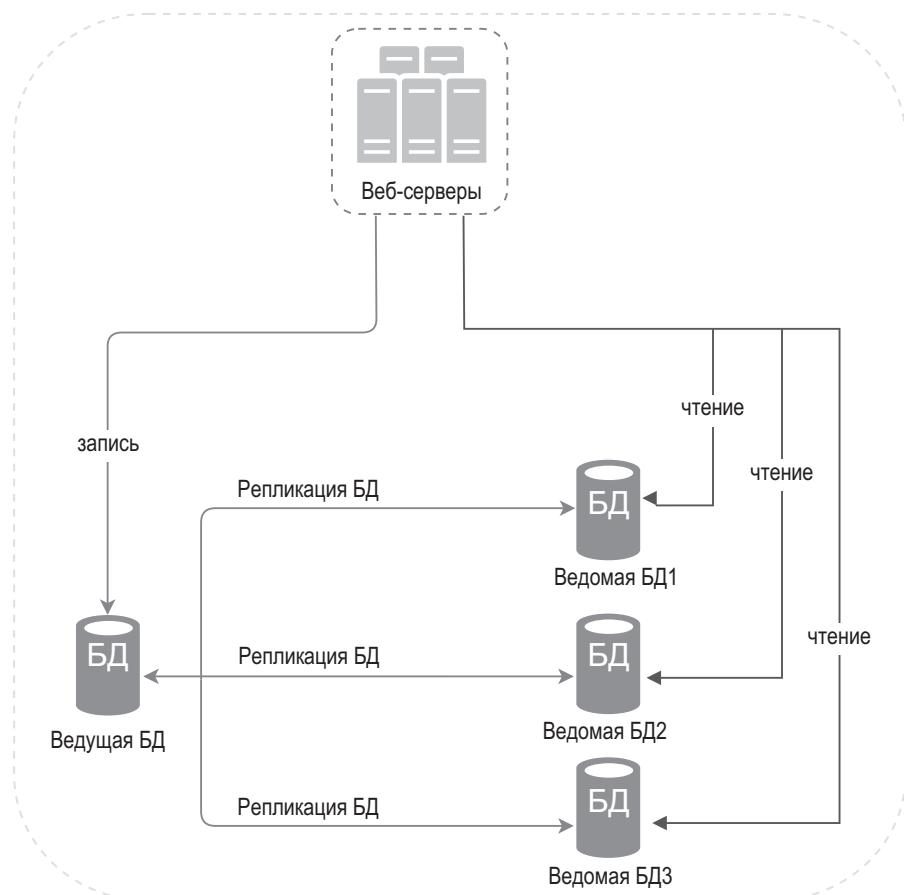


Рис. 1.5

В предыдущем разделе мы обсудили то, как балансировщик нагрузки улучшает доступность системы. Зададим здесь тот же вопрос: что, если одна из БД перестанет работать? Архитектурная конфигурация, представленная на рис. 1.5, может справиться с этой ситуацией:

- Если имеется лишь одна ведомая база данных и она выходит из строя, операции чтения будут временно перенаправлены к ведущей БД. Сразу после выявления проблемы новая ведомая БД заменит старую. Если ведомых БД несколько, операции чтения перенаправляются к другим исправным экземплярам. Новый сервер базы данных заменит старый.
- Если ведущая база данных выйдет из строя, ее место займет одна из ведомых. Все операции будут временно выполняться на сервере новой ведущей БД. Новая ведомая БД, предназначенная для репликации данных, немедленно заменит старую. В промышленных системах переквалификация ведомой БД в ведущую требует дополнительных усилий, так как ее содержимое может быть неактуальным. Недостающие данные придется обновить с помощью скриптов восстановления. В качестве решения можно использовать и другие методы, включая конфигурации с несколькими ведущими узлами и циклическую репликацию, но они более сложные, поэтому мы не станем рассматривать их в этой книге. Если вам интересна эта тема, обратитесь к справочным материалам [4] [5].

На рис. 1.6 показана архитектура системы после добавления балансировщика нагрузки и репликации базы данных.

Рассмотрим эту конфигурацию.

- Пользователь получает из DNS IP-адрес балансировщика нагрузки.
- Пользователь подключается к балансировщику нагрузки с помощью этого IP-адреса.
- HTTP-запрос направляется либо к серверу 1, либо к серверу 2.
- Веб-сервер считывает пользовательские данные из ведомой БД.
- Веб-сервер направляет любые операции по изменению данных ведущей БД, включая чтение, обновление и удаление.

Итак, мы как следует разобрались в веб-уровне и уровне данных. Теперь пришло время улучшить время загрузки/ответа. Для этого можно до-

бавить слой кэша и разместить статические ресурсы (JavaScript/CSS/изображения/видеофайлы) в сети доставки содержимого (content delivery network, CDN).

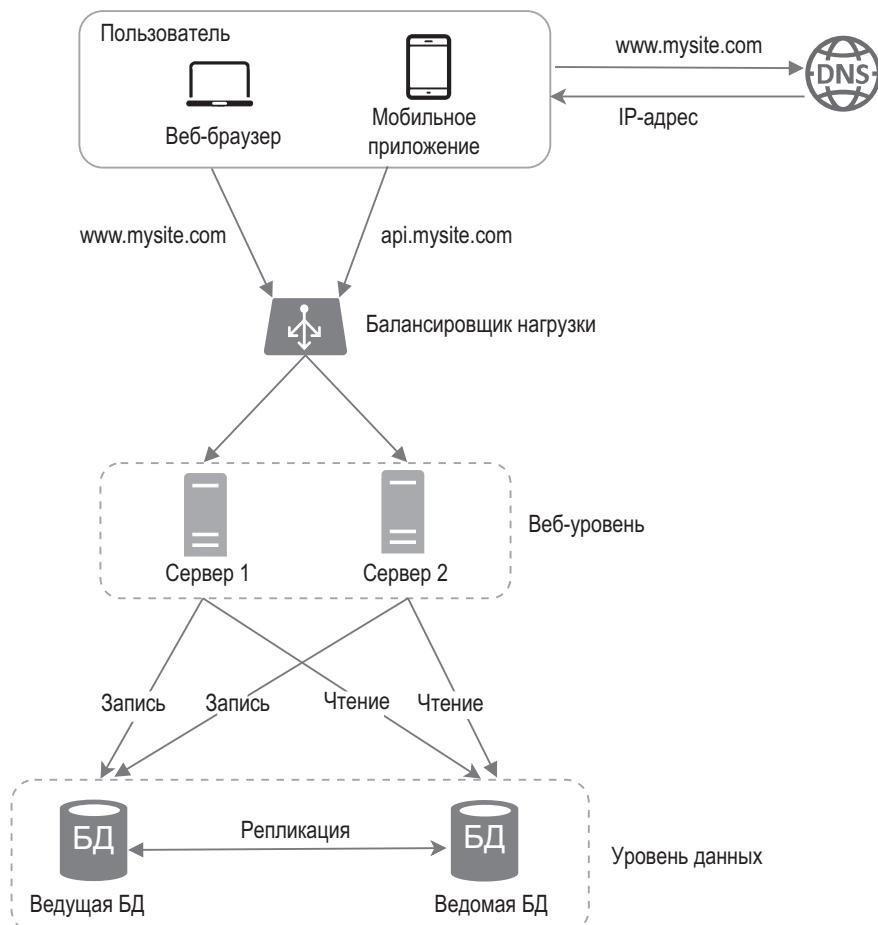


Рис. 1.6

## КЭШ

Кэш — это участок памяти, в который временно записываются результаты ресурсоемких ответов или данных, к которым часто обращаются. Это позволяет ускорить обслуживание последующих запросов. Как про-

иллюстрировано на рис. 1.6, при каждой загрузке новой веб-страницы выполняется один или несколько запросов к БД для извлечения данных. Многократное обращение к базе данных существенно влияет на производительность. Кэш может смягчить эту проблему.

### Уровень кэша

Уровень кэша — это слой временного хранилища данных, который по своей скорости работы намного опережает БД. К преимуществам отдельного уровня кэша можно отнести улучшение производительности системы, возможность снизить нагрузку на базу данных и масштабировать этот уровень независимо от других. На рис. 1.7 показан пример конфигурации сервера кэширования.

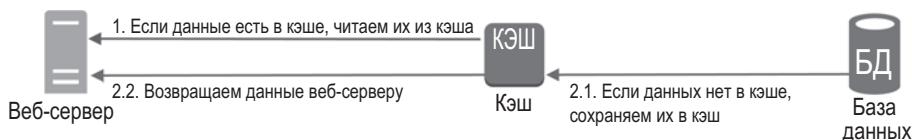


Рис. 1.7

Получив запрос, веб-сервер сначала проверяет наличие ответа в кэше. Если ответ есть, данные возвращаются клиенту. Если нет, то веб-сервер обращается к базе данных, сохраняет ответ в кэше и пересыпает его обратно клиенту. Эта стратегия называется кэшем сквозного чтения. В зависимости от типа и размера данных, а также от того, как к ним обычно обращаются, можно использовать и другие подходы. В предыдущем исследовании объясняется принцип работы разных стратегий кэширования [6].

Взаимодействовать с серверами кэширования просто, так как большинство из них предоставляют API-интерфейсы для распространенных языков программирования. В следующем фрагменте кода показан типичный пример использования API-интерфейса Memcached:

```

SECONDS=1
cache.set('myKey', 'hi there', 3600 * SECONDS)
cache.get('myKey')

```

## Некоторые аспекты использования кэша

Вот несколько соображений касательно использования систем кеширования.

- Определитесь с тем, когда будет использоваться кэш. Это лучше делать в ситуациях, когда чтение данных происходит часто, а изменение — редко. Поскольку кэшированные данные хранятся в энергозависимой памяти, сервер кеширования не подходит для постоянного хранения. Например, если он перезапустится, все данные, хранившиеся в памяти, будут утрачены. В связи с этим данные необходимо записывать в постоянные хранилища.
- Выбор срока действия. Рекомендуется реализовать механизм, ограничивающий срок действия кэша. Просроченные данные немедленно удаляются. Если такого механизма нет, данные будут храниться в памяти постоянно. Срок действия лучше не делать слишком коротким, иначе система будет слишком часто обновлять данные, загружая их из БД. С другой стороны, из-за слишком длинного срока действия данные могут оказаться неактуальными.
- Согласованность. Это подразумевает синхронизацию данных в хранилище и кэше. Несогласованность может возникнуть из-за того, что операции изменения данных в хранилище и кэше выполняются не за одну транзакцию. При масштабировании системы в пределах нескольких регионов может быть непросто поддерживать согласованность. Подробнее об этом можно почитать в документе *Scaling Memcache at Facebook*, опубликованном Facebook [7].
- Предотвращение сбоев. Наличие лишь одного сервера кеширования может оказаться потенциальной единой точкой отказа (single point of failure, SPOF), которая, согласно английской Википедии, имеет следующее определение: «Единая точка отказа — это компонент, выход из строя которого приводит к прекращению работы всей системы» [8]. В связи с этим, чтобы избежать SPOF, рекомендуется использовать несколько серверов кеширования, размещенных в разных центрах обработки данных (ЦОД). А еще можно выделить какой-нибудь дополнительный объем памяти: это создаст буфер на случай, если память начнет использоваться более активно.
- Политика вытеснения. Когда кэш полностью заполнен, любой запрос на добавление новых элементов может привести к удале-

нию существующих. Это называют вытеснением кэша. Самой популярной политикой считается вытеснение давно неиспользуемых данных (least-recently-used, LRU). Для разных ситуаций могут также подойти вытеснение наименее часто используемых данных (least-frequently-used, LFU) или метод «первым пришел, первым ушел» (FIFO, first-in-first-out).

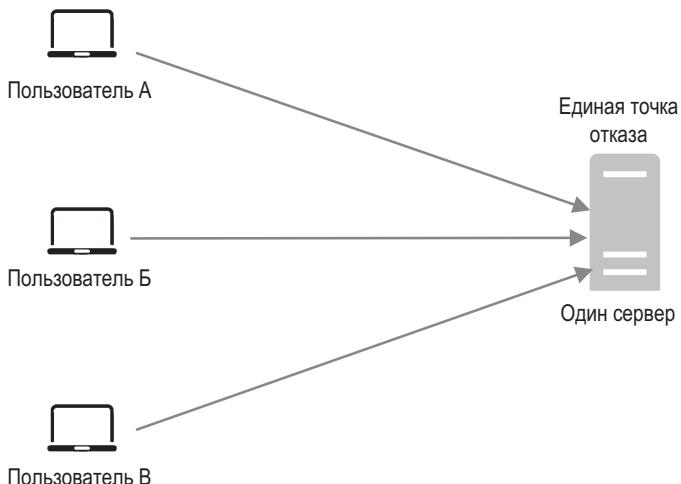


Рис. 1.8

## СЕТЬ ДОСТАВКИ СОДЕРЖИМОГО (CDN)

CDN — это сеть географически распределенных серверов, которая используется для доставки статического содержимого. Серверы CDN кэшируют такие статические файлы, как изображения, видео, CSS, JavaScript и т. д.

Кэширование динамического содержимого — идея относительно новая. Здесь мы не будем углубляться в детали. Ограничимся лишь следующим: такой способ позволяет записывать в кэш HTML-страницы в зависимости от пути, параметров, cookie-файлов и заголовков запроса. Подробнее об этом можно почитать в статье из списка дополнительной литературы [9]. В этой книге мы сосредоточимся на использовании CDN для кэширования статического содержимого.

Вот общий принцип работы CDN: когда пользователь посещает веб-сайт, ближайший к нему сервер CDN доставляет статическое содержимое.

Очевидно, что чем дальше от серверов CDN находятся пользователи, тем медленнее загружается веб-сайт. Например, если серверы CDN расположены в Сан-Франциско, пользователь из Лос-Анджелеса получит содержимое быстрее, чем пользователь из Европы. На рис. 1.9 проиллюстрировано, как CDN может уменьшать время загрузки.

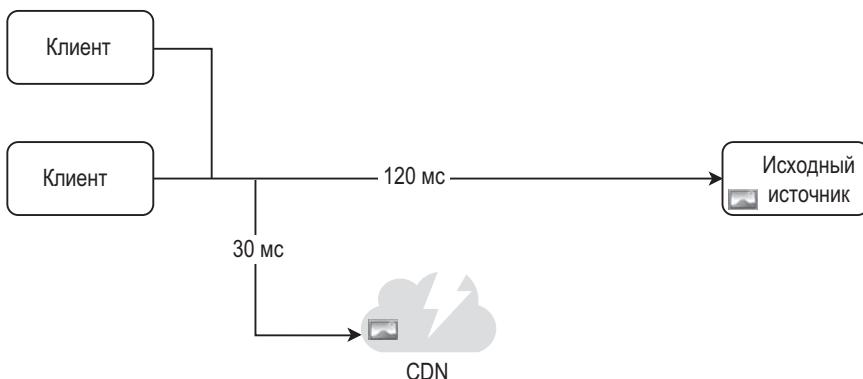


Рис. 1.9

Принцип работы CDN продемонстрирован на рис. 1.10.

- Пользователь А пытается получить image.png с помощью URL-адреса изображения. Домен этого URL-адреса предоставляет провайдером CDN. Ниже показано, как могут выглядеть URL-адреса изображений, на примере CDN от Amazon и Akami:
  - ◆ <https://mysite.cloudfront.net/logo.jpg>
  - ◆ <https://mysite.akamai.com/image-manager/img/logo.jpg>
- Если в кэше сервера CDN нет image.png, он запрашивает этот файл из оригинального источника, например веб-сервера или онлайн-хранилища вроде Amazon S3.
- Источник возвращает серверу CDN файл image.png вместе с дополнительным HTTP-заголовком TTL (Time-to-Live — «время жизни»), который определяет, как долго изображение будет находиться в кэше.
- CDN кэширует изображение и возвращает его пользователю А. Оно остается в кэше CDN, пока не истечет срок TTL.

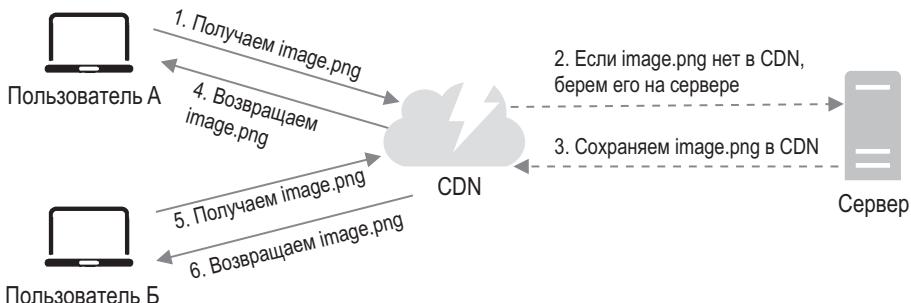


Рис. 1.10

5. Пользователь Б отправляет запрос на получение того же файла.
6. Если срок TTL еще не истек, изображение возвращается из кэша.

### Нюансы использования CDN

- Стоимость. Серверы CDN предоставляются сторонними компаниями, а перемещение данных в CDN и из CDN стоит денег. Кэширование нечасто используемых ресурсов не даст существенных преимуществ, поэтому из CDN их лучше убрать.
- Подбор подходящего срока годности кэша. Для содержимого, которое зависит от времени, необходимо предусмотреть срок годности кэша. Он должен быть не слишком длинным, но и не слишком коротким. В первом случае содержимое может потерять свою актуальность, а во втором — привести к повторной перезагрузке содержимого с исходных серверов в CDN.
- Возможность сбоев. Вы должны подумать о том, как ваши веб-сайты/приложения будут справляться с недоступностью CDN. Если CDN временно выходит из строя, у клиента должна быть возможность обнаружить эту проблему и запросить ресурсы из исходного источника.
- Аннулирование файлов. Файлы можно удалять из CDN до истечения их срока годности одним из следующих способов:
  - ♦ аннулировать объект CDN с помощью API-интерфейсов, предоставляемых поставщиками CDN;

- ♦ использовать версионирование, чтобы возвращать разные версии объектов. Для этого к URL-адресу можно добавить параметр с номером версии. Например, версия 2 может быть представлена строкой запроса: `image.png?v=2`.

На рис. 1.11 показана конфигурация после добавления CDN и кэша.

1. Статические ресурсы (JS, CSS, изображения и т. д.) больше не раздаются веб-серверами. Для повышения производительности они извлекаются из CDN.
2. Нагрузка на базу данных снижается за счет кэширования.

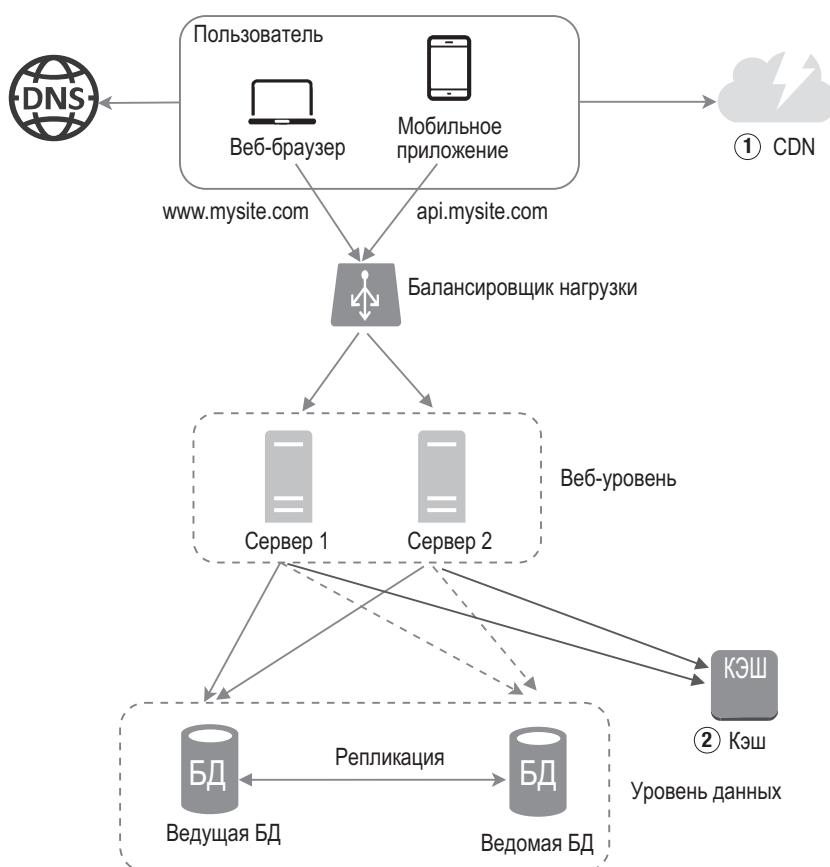


Рис. 1.11

## ВЕБ-УРОВЕНЬ БЕЗ СОХРАНЕНИЯ СОСТОЯНИЯ

Пришло время поговорить о горизонтальном масштабировании веб-уровня. Для этого нужно вынести из него состояние (например, информацию о пользовательских сеансах). Данные сеансов рекомендуется записывать в постоянные хранилища, такие как реляционные БД или NoSQL. Каждый веб-сервер в кластере может запросить состояние из базы данных. Таким образом получается веб-уровень без сохранения состояния.

### Архитектура с сохранением состояния

От того, хранит сервер состояние или нет, зависит, будет ли он «помнить» данные клиента (состояние) между разными запросами.

На рис. 1.12 показан пример архитектуры с сохранением состояния.

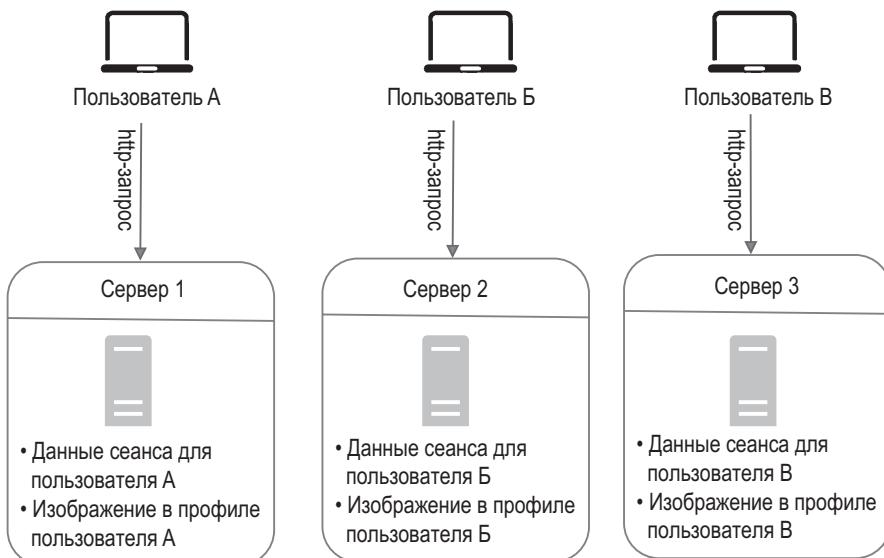


Рис. 1.12

На рис. 1.12 данные сеанса и изображение в профиле пользователя А хранятся на сервере 1. Чтобы аутентифицировать пользователя А, HTTP-

запрос должен быть направлен к этому серверу. Если отправить этот запрос, к примеру, серверу 2, аутентификация не пройдет, так как на втором сервере нет данных соответствующего сеанса. Точно так же все HTTP-запросы пользователя Б должны направляться к серверу 2, а запросы пользователя В — к серверу 3.

Проблема в том, что каждый запрос с отдельно взятого клиента необходимо отправлять на соответствующий сервер. В большинстве балансировщиков нагрузки для этого предусмотрены липкие сеансы [10], но такой подход увеличивает накладные расходы. Из-за него добавление и удаление серверов дается с трудом. Также возникают проблемы, если сервер выходит из строя.

### Архитектура без сохранения состояния

На рис. 1.13 показана архитектура без сохранения состояния.

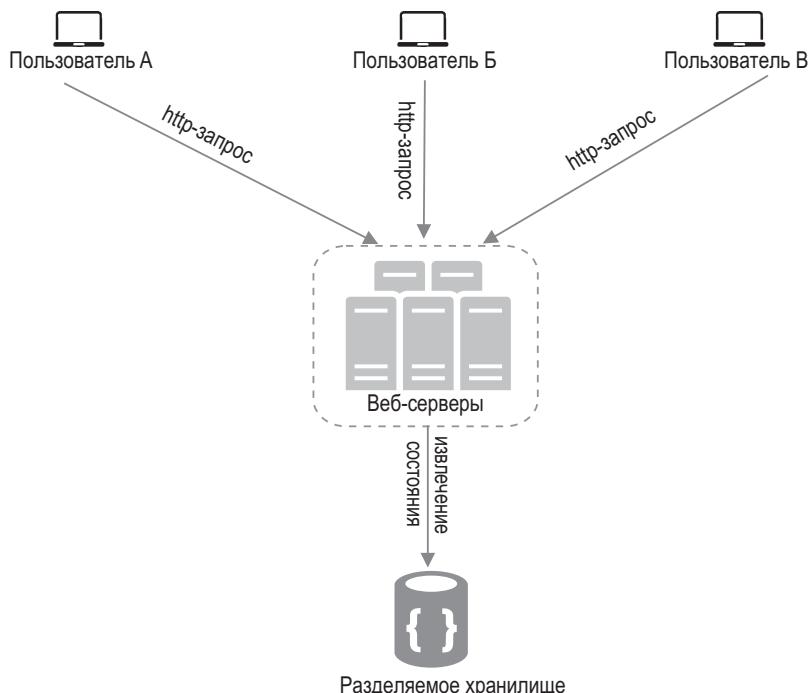


Рис. 1.13

В этой не хранящей состояния архитектуре пользовательские HTTP-запросы могут быть направлены любым веб-серверам, которые извлекают данные о состоянии из общего хранилища. Хранилище отделено от веб-серверов. Отсутствие состояния делает систему более простой, надежной и масштабируемой.

На рис. 1.14 показана обновленная конфигурация с веб-уровнем, не хранящим состояние.

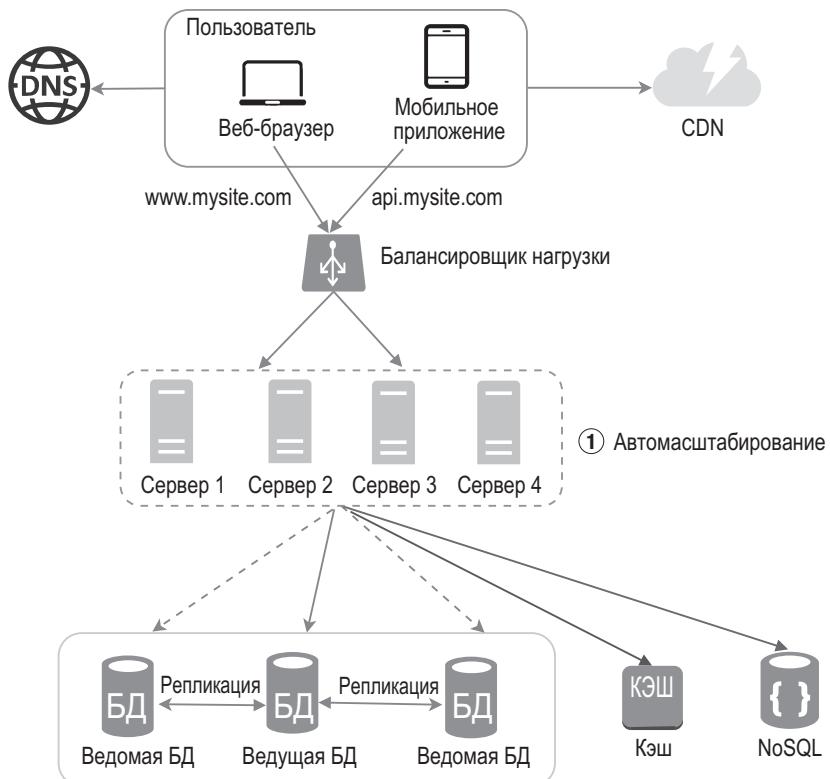


Рис. 1.14

На рис. 1.14 данные сеанса вынесены из веб-уровня и теперь находятся в постоянном хранилище, роль которого могут играть реляционные базы данных: Memcached/Redis, NoSQL и т. д. Здесь хранилище NoSQL выбрано в связи с простотой его масштабирования. Автомасштабирование означает, что добавление и удаление веб-серверов происходит автоматически в за-

в зависимости от объемов трафика. После того как данные о состоянии вынесены в отдельное хранилище, автомасштабирование веб-уровня легко достигается за счет добавления и удаления серверов с учетом нагрузки.

Ваш веб-сайт стремительно развивается, привлекая множество пользователей со всего мира. Для улучшения доступности и UX в различных регионах крайне необходима поддержка нескольких центров обработки данных.

## ЦЕНТРЫ ОБРАБОТКИ ДАННЫХ

На рис. 1.15 показана демонстрационная конфигурация с двумя центрами обработки данных (ЦОД). В нормальных условиях пользователи, скажем,

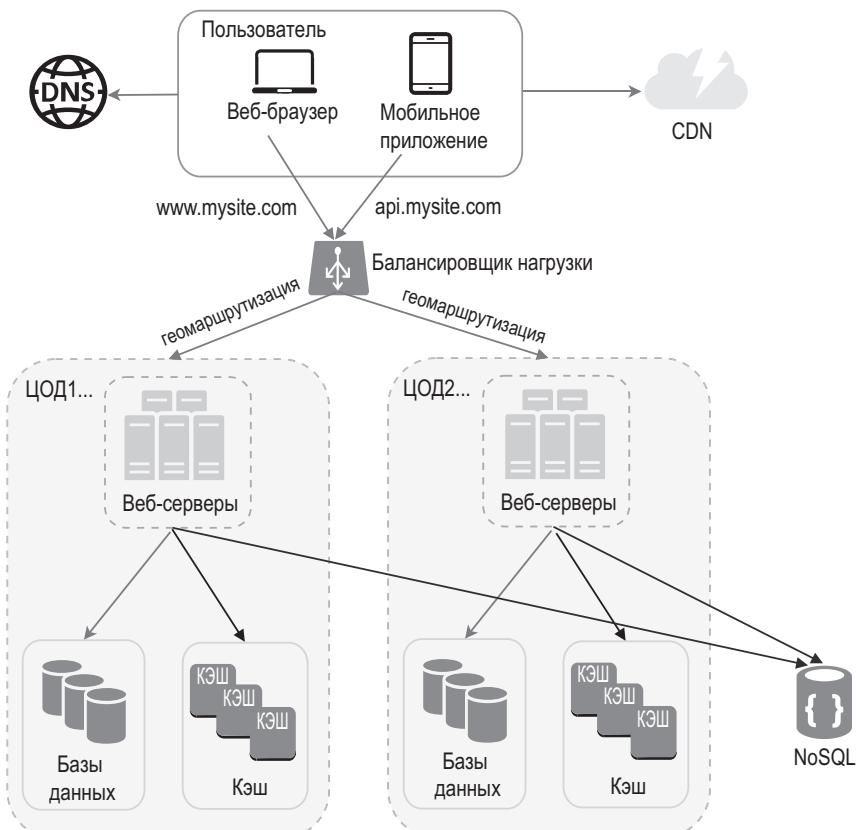


Рис. 1.15

из США с помощью geoDNS направляются к ближайшему центру обработки данных с разделением трафика между регионами US-East и US-West в пропорции  $x\%$  к  $(100 - x)\%$ . Это называется географической маршрутизацией. geoDNS – это сервис, который сопоставляет доменные имена с IP-адресами в зависимости от местонахождения пользователя.

В случае любого серьезного нарушения работы одного из центров обработки данных мы перенаправляем весь трафик к исправному ЦОД. На рис. 1.16 ЦОД2 (US-West) недоступен, поэтому 100 % трафика направляется к ЦОД1 (US-East).

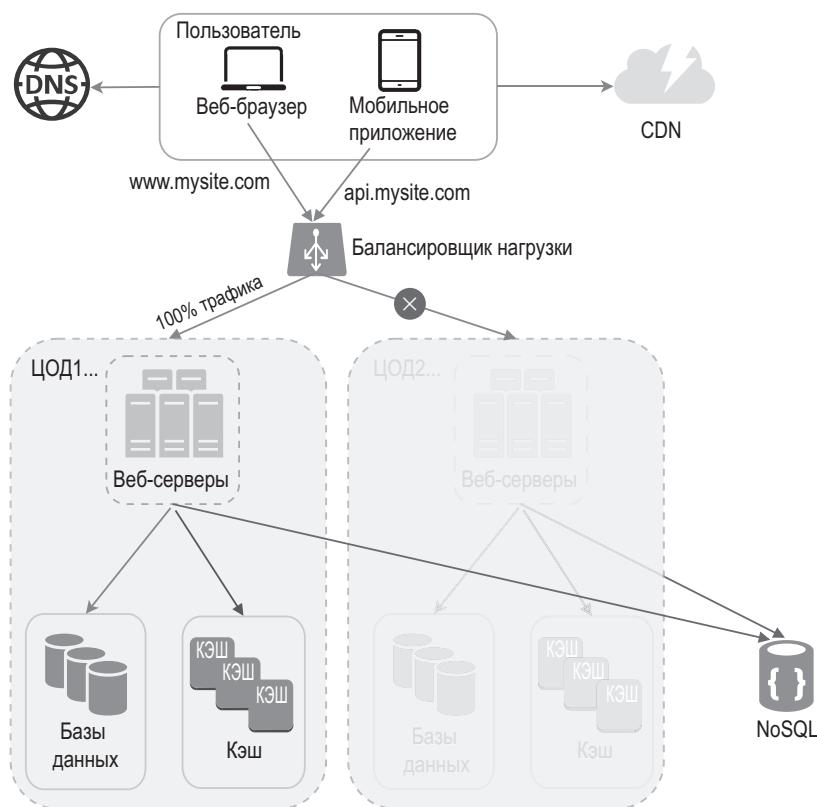


Рис. 1.16

Для реализации архитектуры с несколькими центрами обработки данных необходимо решить несколько технических вопросов.

- Перенаправление трафика. Необходимы эффективные инструменты для направления трафика к подходящему ЦОД. GeoDNS позволяет выбирать центр обработки данных, который находится ближе всего к пользователю.
- Синхронизация данных. Пользователи могут работать с разными локальными базами данных и кэшами в зависимости от региона. В случае сбоя трафик может быть перенаправлен к ЦОД, в котором нет запрашиваемых данных. Распространенным решением является репликация данных между несколькими ЦОД. В одном из исследований показано, как Netflix реализует асинхронную репликацию между разными центрами обработки данных [11].
- Тесты и развертывание. В конфигурации с несколькими ЦОД тестирование веб-сайта/приложения необходимо проводить в разных местах. Автоматические средства развертывания незаменимы в поддержании согласованности всех ЦОД [11].

Чтобы еще сильнее улучшить масштабируемость нашей системы, мы должны разделить ее компоненты; это позволит масштабировать их независимо друг от друга. Во многих реальных распределенных системах для решения этой задачи используют очереди сообщений.

## ОЧЕРЕДЬ СООБЩЕНИЙ

Очередь сообщений — это устойчивый компонент, который загружается в память и поддерживает асинхронное взаимодействие. Он служит буфером и распределяет асинхронные запросы. Очередь сообщений имеет простую базовую архитектуру. Сервисы ввода, так называемые производители/издатели, создают сообщения и публикуют их в очереди. Другие сервисы или серверы, которые называют потребителями/подписчиками, подключаются к очереди и выполняют действия, определенные в сообщениях. Эта модель показана на рис. 1.17.

Благодаря разделению очередь сообщений является предпочтительной архитектурой для создания масштабируемых и надежных приложений. Производитель может публиковать сообщения в очереди, даже если потребитель не в состоянии их обработать. И наоборот — потребитель может считывать сообщения из очереди, даже если производитель недоступен.



Рис. 1.17

Рассмотрим следующий сценарий: ваше приложение поддерживает функции редактирования фотографий, такие как обрезка, повышение четкости, размытие и т. д. Для выполнения этих операций нужно какое-то время. На рис. 1.18 показано, как веб-серверы публикуют в очереди сообщений задания по обработке фотографий. Рабочие узлы достают задания из очереди и выполняют асинхронную обработку. Производитель и потребитель могут масштабироваться независимо друг от друга. Когда очередь становится слишком большой, для сокращения времени обработки добавляются новые рабочие узлы. Если же очередь в основном пустует, количество рабочих узлов можно уменьшить.

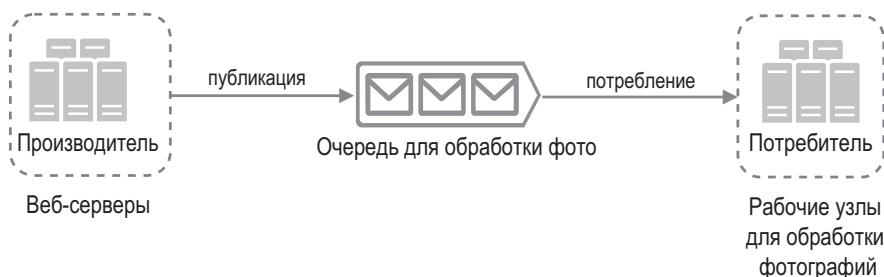


Рис. 1.18

## ЛОГИРОВАНИЕ, МЕТРИКИ, АВТОМАТИЗАЦИЯ

Если вы имеете дело с небольшим веб-сайтом на базе нескольких серверов, то без поддержки логирования, метрик и автоматизации в принципе можно обойтись. Если же ваш сайт дорос до обслуживания крупной компании, эти инструменты являются незаменимыми.

**Логирование.** Мониторинг логов играет важную роль, помогая выявлять в системе ошибки и проблемы. Логи можно отслеживать на каждом от-

дельном сервере, но есть также инструменты, позволяющие собирать их в централизованном сервисе ради удобства поиска и просмотра.

**Метрики.** Сбор разного рода метрик помогает лучше понять предметную область и оценить работоспособность системы. Вам может пригодиться что-то из следующего:

- метрики уровня сервера: процессор, память, дисковый ввод/вывод и т. д.;
- агрегированные метрики: производительность всего уровня базы данных, уровня кэша и т. д.;
- ключевые бизнес-метрики: суточное количество активных пользователей, удержание, доход и т. д.

**Автоматизация.** Когда система становится большой и сложной, для повышения продуктивности необходимо разработать новые или использовать уже готовые средства автоматизации. Рекомендуется применять непрерывную интеграцию — это когда каждая фиксация кода автоматически проверяется, что помогает обнаруживать проблемы на ранних этапах. Кроме того, автоматизация процессов сборки, тестирования, развертывания и других может существенно повысить продуктивность разработчиков.

## Добавление очередей сообщений и других инструментов

На рис. 1.19 показана обновленная архитектура. Ради экономии бумаги на этой схеме изображен лишь один центр обработки данных.

1. Архитектура включает в себя очередь сообщений, которая помогает сделать систему менее связанной и более устойчивой к отказам.
2. Также добавлены средства логирования, автоматизации и сбора метрик.

Объемы данных растут ежедневно, что увеличивает нагрузку на БД. Пришло время масштабировать уровень данных.

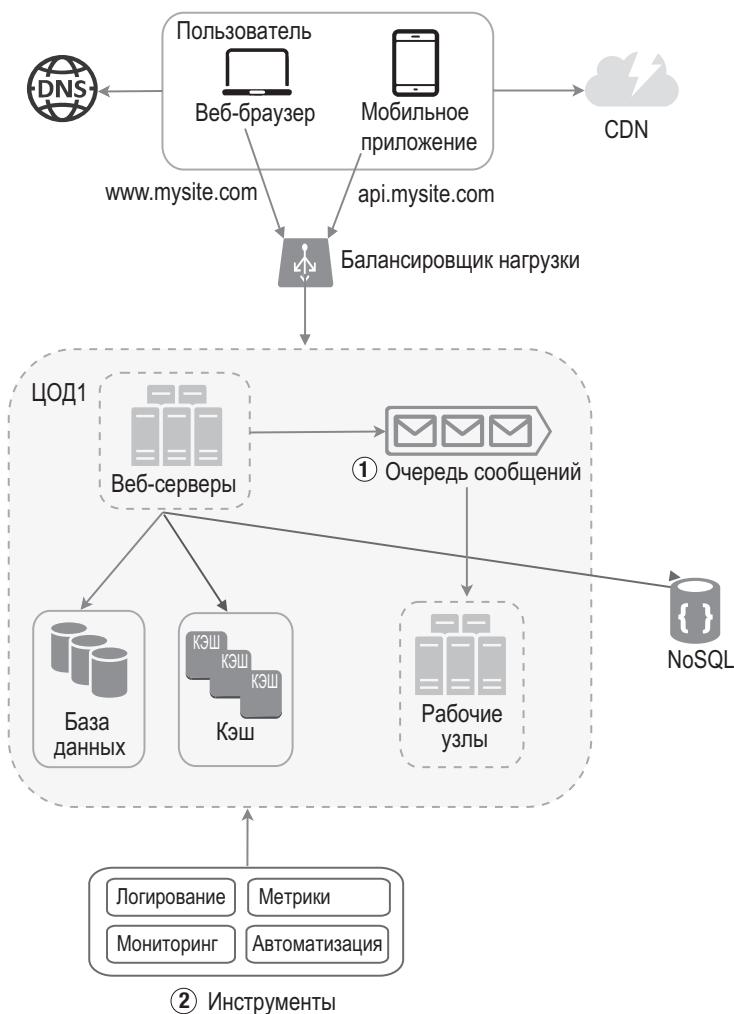


Рис. 1.19

## МАСШТАБИРОВАНИЕ БАЗЫ ДАННЫХ

Есть два общих подхода к масштабированию баз данных: вертикальный и горизонтальный.

## Вертикальное масштабирование

Вертикальное масштабирование (или наращивание) подразумевает повышение производительности существующего компьютера за счет добавления ресурсов процессора, памяти, диска и т. д. Серверы баз данных бывают довольно мощными. Сервис Amazon RDS (Relational Database Service — «сервис реляционных баз данных») [12] предлагает серверы БД с 24 Тб оперативной памяти. Они позволяют хранить и обрабатывать множество информации. В 2013 году на сайт [stackoverflow.com](http://stackoverflow.com) ежедневно заходило больше 10 миллионов уникальных пользователей, но в то время у него была всего одна ведущая база данных [13]. При этом у вертикального масштабирования есть серьезные недостатки.

- Вы можете добавлять к своему серверу дополнительные ресурсы процессора, памяти и т. д., но аппаратные ограничения игнорировать не получится. Если у вас много пользователей, одного сервера будет недостаточно.
- Повышенный риск возникновения единой точки отказа.
- Вертикальное масштабирование имеет высокую общую стоимость. Мощные серверы очень дорогие.

## Горизонтальное масштабирование

Горизонтальное масштабирование (или расширение) заключается в добавлении новых серверов. Сравнение вертикального и горизонтального масштабирования представлено на рис. 1.20.

Шардинг позволяет разделить крупные наборы данных на более мелкие и простые в использовании части, которые называют шардами. Все шарды имеют одну и ту же схему, но каждый из них хранит уникальные данные.

Пример сегментированных баз данных показан на рис. 1.21. Сервер БД для сохранения пользовательской информации выбирается на основе ID пользователя. При каждом обращении к данным используется функция хеширования, которая находит подходящий шard. В нашем примере функция хеширования имеет вид `user_id % 4`. Если результат равен 0, для хранения и извлечения данных используется сегмент 0. Если результат равен 1, выбирается сегмент 1. Та же логика распространяется и на остальные сегменты.

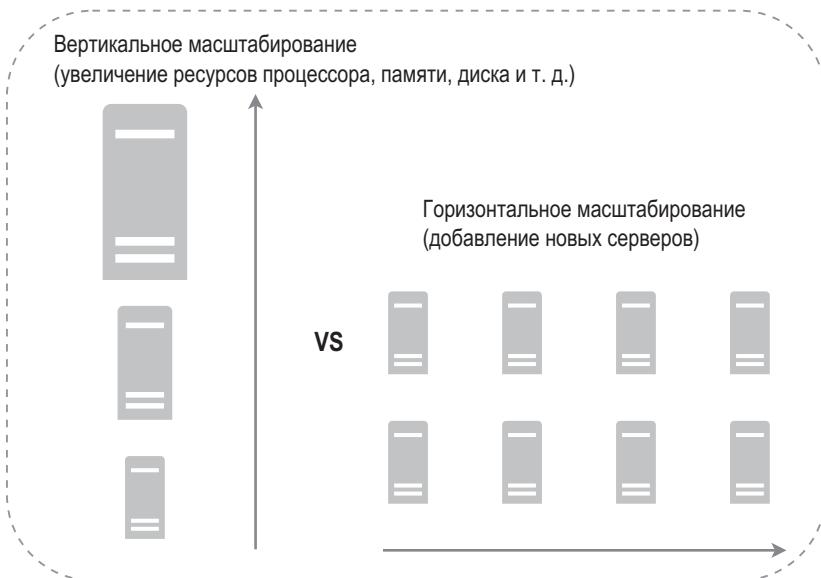


Рис. 1.20

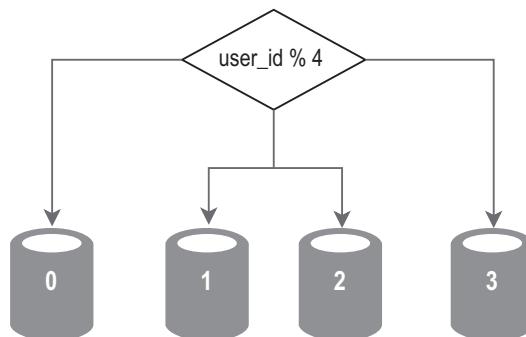


Рис. 1.21

На рис. 1.22 показана таблица с пользовательской информацией, хранящаяся в сегментированных базах данных.

При реализации стратегии сегментирования самый важный фактор — это выбор ключа. Ключ шардинга (или ключ раздела) состоит из одного или нескольких столбцов, на основе которых происходит распределение данных. Как видно на рис. 1.22, ключом выступает `user_id`. Ключ по-

заполняет эффективно извлекать и изменять данные, направляя запросы к подходящей БД. При выборе ключа шардинга один из важнейших критериев — возможность равномерного распределения данных.

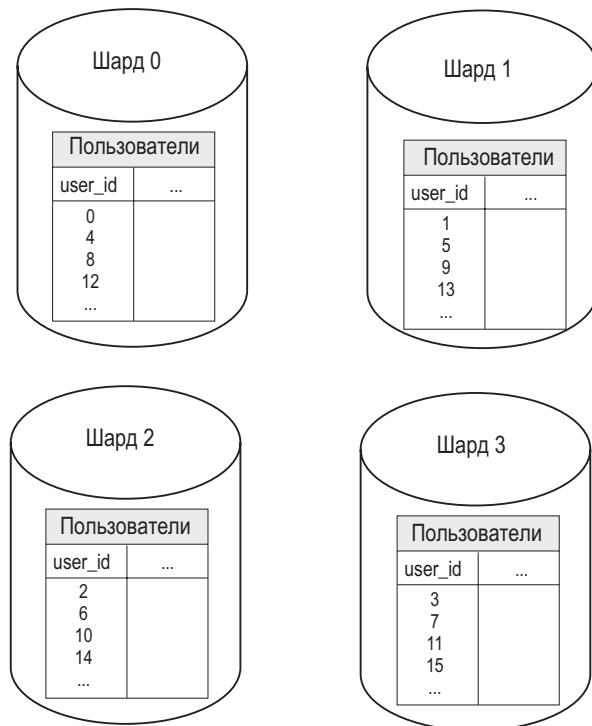


Рис. 1.22

Шардинг отлично подходит для масштабирования баз данных, но это далеко не идеальное решение. Оно усложняет систему и создает дополнительные трудности.

- **Повторное сегментирование данных.** Это может понадобиться, когда 1) отдельный шард полностью заполняется из-за стремительного развития системы или 2) некоторые шарды заполняются быстрее других из-за неравномерного распределения данных. В такой ситуации необходимо обновить функцию сегментирования и переместить имеющиеся данные. Для решения этой проблемы зачастую применяют согласованное хеширование, которое описано в главе 5.

- **Проблема знаменитостей.** Слишком частое обращение к определенному шарду может вызвать перегрузку сервера. Представьте, что информация о Кэтти Перри, Джастине Бибере и Леди Гаге очутилась в одном и том же сегменте. Если речь идет о социальных приложениях, этот сегмент будет перегружен операциями чтения. Для решения этой проблемы, возможно, придется выделить по отдельному шарду для каждой знаменитости. Может случиться так, что каждый сегмент потребует дальнейшего разделения.
- **Соединение и денормализация.** После сегментирования базы данных между несколькими серверами становится сложно выполнять операции соединения, охватывающие несколько шардов. Распространенное решение состоит в денормализации базы данных таким образом, чтобы запросы могли выполняться в рамках одной таблицы.

На рис. 1.23 база данных сегментирована, чтобы справиться с растущими объемами трафика. Вместе с тем некоторые нереляционные функции перенесены в хранилище NoSQL, чтобы снизить нагрузки на БД. Вы можете ознакомиться со статьей, в которой описано множество примеров применения NoSQL [14].

## МИЛЛИОНЫ ПОЛЬЗОВАТЕЛЕЙ

Масштабирование системы — это пошаговый процесс. Вещи, описанные в этой главе, могут здорово вам помочь. Но если ваша аудитория пользователей куда больше нескольких миллионов, вам могут понадобиться тонкие оптимизации и новые стратегии. Например, вам, возможно, придется оптимизировать свою систему и разбить ее на еще более мелкие сервисы.

Все методики, изложенные в этой главе, должны послужить хорошей основой в борьбе с новыми трудностями. В завершение перечислим шаги, которые предпринимаются в ходе масштабирования системы для поддержки миллионов пользователей:

- веб-уровень не должен хранить состояния;
- резервирование должно быть предусмотрено на каждом уровне;
- кэширование данных следует проводить как можно более активно;

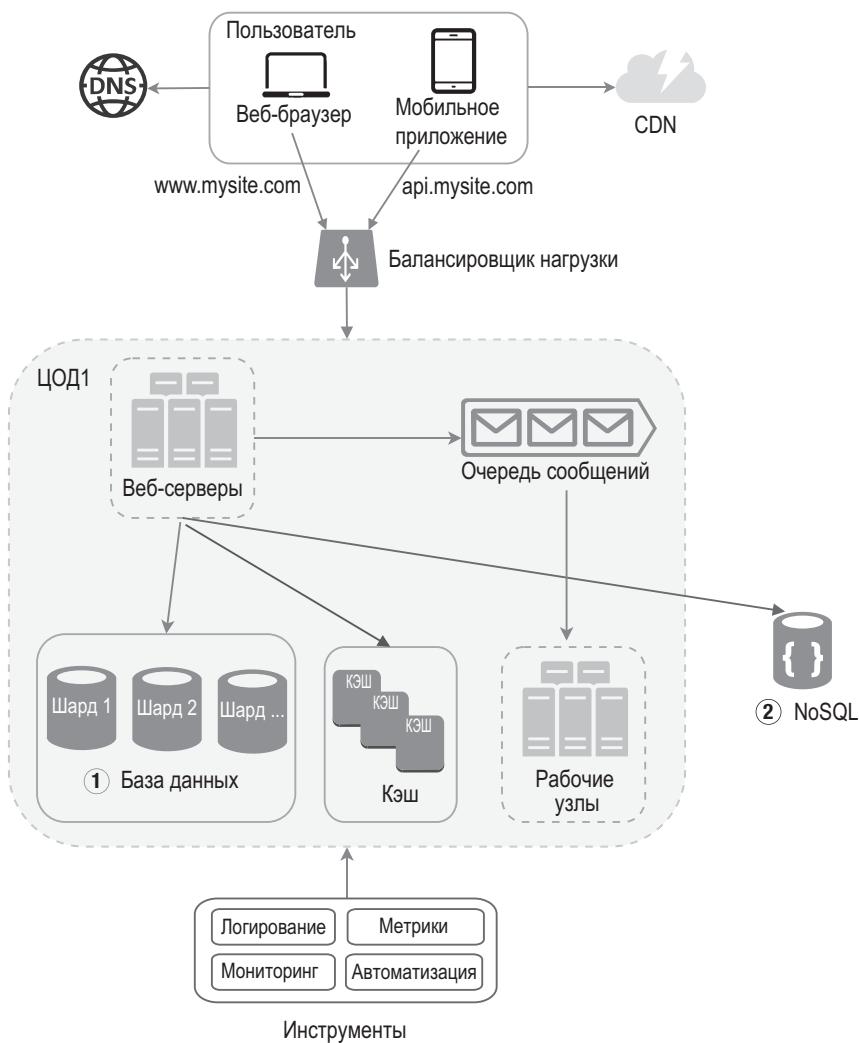


Рис. 1.23

- система должна поддерживать больше одного центра обработки данных;
- статические ресурсы нужно хранить в CDN;
- для масштабирования данных следует применять шардинг;
- уровни должны быть разделены на отдельные сервисы;

- необходимо выполнять мониторинг системы и использовать средства автоматизации.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Протокол передачи гипертекста: <https://ru.wikipedia.org/wiki/HTTP>
- [2] Should you go Beyond Relational Databases?: <https://blog.teamtreehouse.com/should-you-go-beyond-relational-databases>
- [3] Репликация: [https://ru.wikipedia.org/wiki/Репликация\\_\(вычислительная\\_техника\)](https://ru.wikipedia.org/wiki/Репликация_(вычислительная_техника))
- [4] Репликация с несколькими ведущими серверами: [https://en.wikipedia.org/wiki/Multi-master\\_replication](https://en.wikipedia.org/wiki/Multi-master_replication)
- [5] NDB Cluster Replication: Multi-Master and Circular Replication: <https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-replicationmulti-master.html>
- [6] Caching Strategies and How to Choose the Right One: <https://codeahoy.com/2017/08/11/caching-strategies-and-how-tochoose-the-right-one/>
- [7] R. Nishtala, «Facebook, Scaling Memcache at», 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13).
- [8] Единая точка отказа (англ.): [https://en.wikipedia.org/wiki/Single\\_point\\_of\\_failure](https://en.wikipedia.org/wiki/Single_point_of_failure)
- [9] Доставка динамического контента с Amazon CloudFront: <https://aws.amazon.com/ru/cloudfront/dynamic-content/>
- [10] Configure Sticky Sessions for Your Classic Load Balancer: <https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elbsticky-sessions.html>
- [11] Active-Active for Multi-Regional Resiliency: <https://netflixtechblog.com/active-active-for-multi-regional-resiliencyc47719f6685b>
- [12] Инстансы Amazon EC2 High Memory: <https://aws.amazon.com/ru/ec2/instance-types/high-memory/>
- [13] What it takes to run Stack Overflow: <http://nickcraver.com/blog/2013/11/22/what-it-takes-to-runstack-overflow>
- [14] What The Heck Are You Actually Using NoSQL For: <http://highscalability.com/blog/2010/12/6/what-the-heck-are-youactually-using-nosql-for.html>

---

# 2

---

## ПРИБЛИЗИТЕЛЬНЫЕ ОЦЕНКИ

В ходе интервью по проектированию ИТ-систем претендента иногда просят «на коленке» оценить емкость или требования к производительности системы. Согласно Джону Дину, старшему сотруднику Google, «наколеночные вычисления — это оценки, основанные на мысленных экспериментах и типичных показателях производительности, которые дают хорошее представление о том, какие архитектуры соответствуют вашим требованиям» [1].

Для эффективного выведения приблизительных оценок нужно хорошо разбираться в основах масштабирования. Вы должны уверенно владеть следующими концепциями: степень двойки [2], показатели латентности, которые должен знать любой программист, и показатели доступности.

### СТЕПЕНЬ ДВОЙКИ

В распределенных системах данные могут достигать огромных размеров, но все вычисления сводятся к элементарным свойствам. Чтобы получить правильный результат, нужно обязательно знать объем данных, используя вторую степень. Байт — это последовательность из 8 бит. Символ ASCII занимает в памяти один байт (8 бит). В табл. 2.1 перечислены единицы измерения данных.

Таблица 2.1

Степень	Примерное значение	Полное название	Сокращенное обозначение
10	1 тысяча	1 килобайт	1 Кб
20	1 миллион	1 мегабайт	1 Мб
30	1 миллиард	1 гигабайт	1 Гб
40	1 триллион	1 терабайт	1 Тб
50	1 квадрильон	1 петабайт	1 Пб

## ПОКАЗАТЕЛИ ЛАТЕНТНОСТИ, КОТОРЫЕ ДОЛЖЕН ЗНАТЬ ЛЮБОЙ ПРОГРАММИСТ

В 2010 году доктор Дин из Google поделился данными о продолжительности типичных компьютерных операций [1]. Некоторые из этих показателей потеряли актуальность в связи с повышением производительности компьютеров. Но они по-прежнему должны давать хорошее представление о том, насколько быстрыми или медленными являются те или иные операции.

**Таблица 2.2**

Название операции	Время
Обращение к кэшу L1	0,5 нс
Ошибканое предсказание перехода	5 нс
Обращение к кэшу L2	7 нс
Блокирование/разблокирование мьютекса	100 нс
Обращение к основной памяти	100 нс
Сжатие 1 Кб с помощью Zippy	10 000 нс = 10 мкс
Отправка 2 Кб по сети 1 Гбит/с	20 000 нс = 20 мкс
Последовательное чтение из памяти 1 Мб	250 000 нс = 250 мкс
Перемещение пакета туда и обратно внутри одного ЦОД	500 000 нс = 500 мкс
Время поиска по диску	10 000 000 нс = 10 мс
Последовательное чтение 1 Мб из сети	10 000 000 нс = 10 мс
Последовательное чтение 1 Мб с диска	30 000 000 нс = 30 мс
Передача пакета из Калифорнии в Нидерланды и обратно	150 000 000 нс = 150 мс

Обозначения:

нс = наносекунда, мкс = микросекунда, мс = миллисекунда

1 нс =  $10^{-9}$  секунд

1 мкс =  $10^{-6}$  секунд = 1000 нс

1 мс =  $10^{-3}$  секунд = 1000 мкс = 1 000 000 нс

Один разработчик ПО из Google написал утилиту для визуализации показателей, опубликованных доктором Дином. Помимо прочего, она делает поправку на современные реалии. На рис. 2.1 даны визуализированные показатели латентности по состоянию на 2020 год (источник данных: справочный материал [3]).

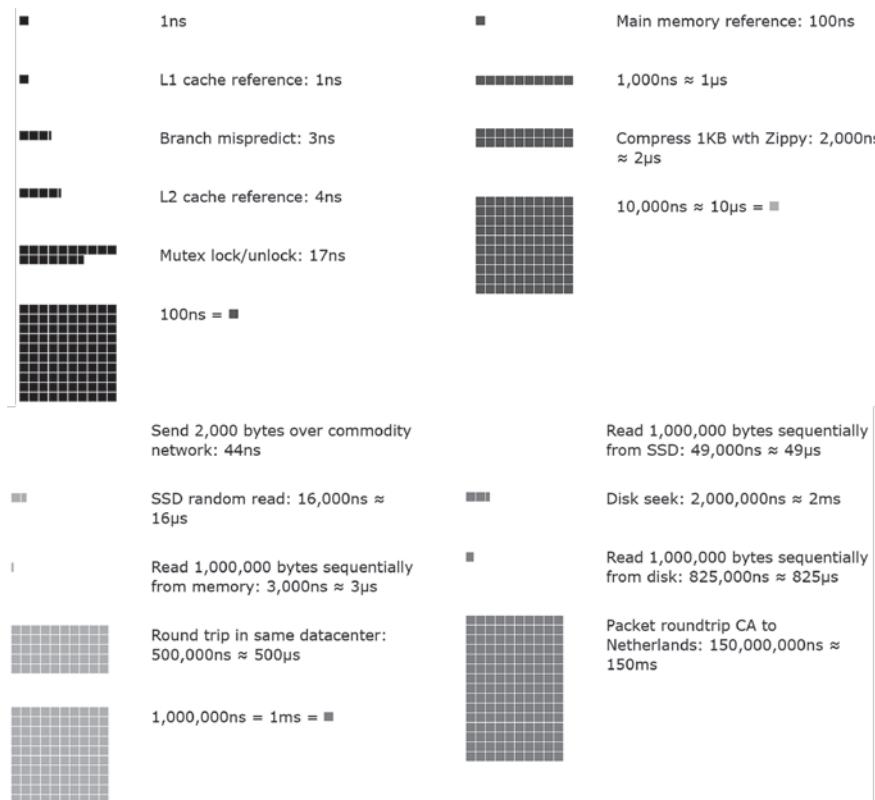


Рис. 2.1

Проанализировав цифры, представленные на рис. 2.1, мы можем сделать следующие выводы:

- память быстрая, а диск медленный;
- по возможности следует избегать поиска по диску;
- простые алгоритмы сжатия отличаются высокой скоростью;

- прежде чем отправлять данные по интернету, их по возможности нужно сжимать;
- центры обработки данных обычно находятся в разных регионах, и передача информации между ними занимает время.

## ПОКАЗАТЕЛИ ДОСТУПНОСТИ

Высокая доступность — это способность системы долго и непрерывно находиться в рабочем состоянии. Она измеряется в процентах. 100 % означает, что сервис никогда не пристаивает. У большинства сервисов доступность варьируется от 99 % и 100 %.

Поставщики сервисов часто используют такой термин, как соглашение об уровне услуг (service level agreement, SLA). Это соглашение между вами (поставщиком) и вашим клиентом, которое официально определяет уровень беспрерывной работы вашего сервиса. Облачные провайдеры Amazon [4], Google [5] и Microsoft [6] предлагают SLA от 99,9 % и выше. Время беспрерывной работы обычно измеряется в девятках. Чем больше девяток, тем лучше. Как видно из табл. 2.3, количество девяток соответствует ожидаемому времени простоя системы.

**Таблица 2.3**

Доступность	Суточный простой	Недельный простой	Месячный простой	Годичный простой
99 %	14,40 минуты	1,68 часа	7,31 часа	3,65 дня
99,9 %	1,44 минуты	10,08 минуты	43,83 минуты	8,77 часа
99,99 %	8,64 секунды	1,01 минуты	4,38 минуты	52,60 минуты
99,999 %	864,00 миллисекунды	6,05 секунды	26,30 секунды	5,26 минуты
99,9999 %	86,40 миллисекунды	604,80 миллисекунды	2,63 секунды	31,56 секунды

## ПРИМЕР: ОЦЕНКА ТРЕБОВАНИЙ К QPS И ХРАНИЛИЩУ ДЛЯ TWITTER

Пожалуйста, имейте в виду, что следующие цифры относятся лишь к нашему упражнению и не имеют ничего общего с реальными показателями Twitter.

Предположения:

- 300 миллионов активных пользователей в месяц;
- 50 % из них пользуются Twitter ежедневно;
- пользователи в среднем публикуют по 2 твита в день;
- 100 % твитов содержат медиаданные;
- данные хранятся на протяжении 5 лет.

Оценки:

- ежедневные активные пользователи (daily active users, DAU) = 300 миллионов \* 50 % = 150 миллионов;
- запросов в секунду (queries per second, QPS) = 150 миллионов \* 2 твита / 24 часа / 3600 секунд = ~3500;
- пиковый показатель QPS = 2 \* QPS = ~7000.

Здесь мы оценим лишь место, необходимое для хранения данных:

- средний размер твита:
  - ◆ tweet\_id — 64 байта;
  - ◆ текст — 140 байтов;
  - ◆ медиаданные — 1 Мб;
- объем данных: 150 миллионов \* 2 \* 10% \* 1 Мб = 30 Тб в день;
- объем данных за 5 лет: 30 Тб \* 365 \* 5 = ~55 Пб.

## СОВЕТЫ

Главное в наколеночных оценках — процесс. То, как вы решаете задачу, важнее результатов. Интервьюеры могут проверить ваши навыки решения задач. Вот несколько советов.

- Округление и приближение. Во время интервью сложно проводить серьезные математические расчеты. Например, сколько будет  $99987/9,1$ ? Не нужно тратить ценнное время на сложную арифметику. От вас не ждут высокой точности. Для удобства пользуйтесь

округленными и приблизительными числами. Вопрос с делением можно упростить до  $100\ 000/10$ .

- Записывайте свои предположения, чтобы позже на них можно было сослаться.
- Не забывайте о единицах измерения. Когда вы записываете 5, это означает 5 Кб или 5 Мб? Вы можете запутаться. Указывайте единицы измерения, поскольку это помогает избавиться от неопределенности.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] J. Dean, Google Pro Tip: Use Back-Of-The-Envelope-Calculations To Choose The Best Design: <http://highscalability.com/blog/2011/1/26/google-pro-tip-use-back-of-the-envelope-calculations-to-choose.html>
- [2] System design primer: <https://github.com/donnemartin/systemdesign-primer>
- [3] Latency Numbers Every Programmer Should Know: [https://colin-scott.github.io/personal\\_website/research/interactive\\_latency.html](https://colin-scott.github.io/personal_website/research/interactive_latency.html)
- [4] Amazon Compute Service Level Agreement: <https://aws.amazon.com/compute/sla/>
- [5] Compute Engine Service Level Agreement (SLA): <https://cloud.google.com/compute/sla>
- [6] SLA summary for Azure services: <https://azure.microsoft.com/en-us/support/legal/sla/summary/>

---

# 3

---

## ОБЩИЕ ПРИНЦИПЫ ПРОХОЖДЕНИЯ ИНТЕРВЬЮ ПО ПРОЕКТИРОВАНИЮ ИТ-СИСТЕМ

Вас только что пригласили на такое желанное интервью в компании вашей мечты. Координатор по найму прислал вам расписание на намеченный день. На первый взгляд все довольно неплохо, но тут вам на глаза попадается пункт об интервью по проектированию ИТ-систем.

Многие боятся такого рода интервью. Вопросы на них могут быть расплывчатыми и слишком уж общими, например: «Спроектируйте общеизвестный продукт X». Отсутствие энтузиазма у претендента вполне объяснимо. В конце концов, кому под силу в течение часа спроектировать популярный продукт, над которым работали сотни, если не тысячи инженеров?

Хорошая новость в том, что от вас этого никто не ожидает. Реальные системы имеют чрезвычайно сложную архитектуру. Например, поиск Google только с виду выглядит просто; количество технологий, стоящих за этой простотой, поистине потрясающее. Но если никто не ждет, что вы за час успеете спроектировать настоящую систему, то какая польза от этого интервью?

Интервью по проектированию ИТ-систем имитирует реальный процесс, в ходе которого пара коллег совместно работают над какой-то общей не совсем ясной задачей и находят подходящее решение. Задача носит общий характер, и идеального решения не существует. Важна не столько архитектура, которую вы предложите, сколько сам процесс проектирования. Это позволяет вам продемонстрировать свои навыки инженера, обосновать выбор тех или иных архитектурных решений и конструктивно ответить на возникшие вопросы.

Давайте представим себя на месте интервьюера — человека, который задает вопросы, и попробуем понять, о чем он думает, когда заходит

в конференц-зал и встречает вас. Его основная задача — правильно оценить ваши способности. Худшим исходом для него будет плохо проведенное интервью или недостаток сведений, которые не позволят дать вам окончательную оценку. Что пытается узнать интервьюер во время собеседования по проектированию ИТ-систем?

Многие считают, что этот вид интервью сводится к техническим навыкам проектирования претендента. Но это лишь малая его часть. Эффективное интервью четко сигнализирует о способности претендента работать совместно, выдерживать давление и конструктивноправляться с расплывчатыми задачами. Еще одним ключевым навыком, на который особенно обращают внимание многие интервьюеры, является способность задавать подходящие вопросы.

Хороший интервьюер также ищет отрицательные признаки. Чрезмерно усложненные архитектурные решения являются настоящей болезнью многих инженеров, которые восхищаются чистотой результата и не желают идти на компромиссы. Они зачастую не догадываются о тех расходах, которые влекут за собой переусложненные системы, и многим компаниям это неведение дорого обходится. Эту склонность явно не стоит демонстрировать. Среди других нежелательных качеств можно выделить узкий кругозор, упрямство и т. д.

В этой главе мы разберем некоторые полезные советы и предложим простой и эффективный подход к решению задач на интервью по проектированию ИТ-систем.

## УДАЧНОЕ ИНТЕРВЬЮ ПО ПРОЕКТИРОВАНИЮ ИТ-СИСТЕМ ЗА 4 ШАГА

Каждое интервью по проектированию ИТ-систем уникально. Хорошая задача должна носить общий характер и не иметь какого-то одного универсального решения. Но у всех интервью есть общие этапы.

### Шаг 1: понять задачу и определить масштаб решения

— Почему тигр зарычал?

В заднем ряду кто-то с готовностью поднял руку.

- Джимми? — кивнул учитель.
- Потому что он ПРОГОЛОДАЛСЯ.
- Отлично, Джимми.

В своем классе Джимми всегда первым отвечал на вопросы. Всякий раз, когда учитель что-то спрашивает, в классе всегда найдется ребенок, который с радостью рискнет поднять руку, даже если у него нет ответа. В нашем случае это Джимми.

Джимми — отличник. Он гордится своим умением быстро отвечать на все вопросы. На экзаменах он обычно раньше всех сдает свою работу. Он первый в списке тех, кого учитель посыпает на олимпиады.

НЕ БУДЬТЕ такими, как Джимми.

На интервью по проектированию ИТ-систем быстрый ответ без обдумывания не даст вам дополнительных баллов. Решение задачи без глубокого понимания ее требований является очень плохим знаком, так как такое интервью — это не викторина. Правильного ответа попросту нет.

Поэтому не спешите предлагать решение. Повремените. Хорошенько подумайте и задайте вопросы, чтобы уточнить требования и допущения. Это чрезвычайно важно.

Мы, инженеры, любим сложные задачи и спешим выдать на-гора готовые решения, но такой подход, скорее всего, не приведет к проектированию нужных систем. Один из самых важных навыков любого инженера состоит в умении задавать правильные вопросы, делать подходящие предположения и собирать всю информацию, необходимую для создания системы. Поэтому не бойтесь спрашивать.

Услышав вопрос, интервьюер либо прямо ответит, либо попросит вас высказать собственные предположения. Во втором случае свои мысли лучше записать на доске или бумаге. Позже они вам пригодятся.

О чем следует спрашивать? Попытайтесь уточнить требования к архитектуре. Вот список вопросов, с которых можно начать:

- Какие именно возможности мы будем реализовывать?
- Сколько пользователей у нашего продукта?

- Как скоро ожидается наращивание мощностей? Какой масштаб планируется через 3 месяца, полгода, год?
- Как выглядит технологический стек компании? Какие существующие сервисы можно применить для упрощения архитектуры?

## Пример

Если вас попросят спроектировать ленту новостей, вам следует прояснить требования, которые к ней предъявляются. Ваш разговор с интервьюером может выглядеть так:

**Кандидат:** «Это мобильное или веб-приложение? Или и то и другое?»

**Интервьюер:** «И то и другое».

**Кандидат:** «Какие самые важные возможности должны быть у этого продукта?»

**Интервьюер:** «Возможность публиковать статьи и видеть новостные ленты своих друзей».

**Кандидат:** «Новостная лента сортируется хронологически или в каком-то особом порядке? Особый порядок означает, что каждой статье назначается определенный вес. Например, статьи ваших близких друзей важнее статей, опубликованных в группе».

**Интервьюер:** «Чтобы не усложнять, предположим, что лента сортируется в обратном хронологическом порядке».

**Кандидат:** «Сколько друзей может быть у пользователя?»

**Интервьюер:** «5000».

**Кандидат:** «Какой объем трафика?»

**Интервьюер:** «10 миллионов активных пользователей в день (DAU)».

**Кандидат:** «Может ли лента содержать изображения и видеофайлы наряду с текстом?»

**Интервьюер:** «В ленте могут быть медиафайлы, включая изображения и видео».

Выше приведены некоторые из вопросов, которые вы можете задать своему интервьюеру. Важно определиться с требованиями и прояснить неоднозначные моменты.

## Шаг 2: предложить общее решение и получить согласие

На этом этапе мы пытаемся выработать общее решение и согласовать его с интервьюером. Очень желательно в ходе этого процесса наладить совместную работу.

- Предложите начальный план архитектуры. Поинтересуйтесь мнением интервьюера. Относитесь к нему как к члену своей команды, с которым вы вместе работаете. Хорошие интервьюеры зачастую любят поговорить и поучаствовать в решении задачи.
- Нарисуйте на доске или бумаге блок-схемы с ключевыми компонентами, такими как клиенты (мобильные/браузерные), API-интерфейсы, веб-серверы, хранилища данных, кэши, CDN, очереди сообщений и т. д.
- Выполните приблизительные расчеты, чтобы понять, соответствует ли ваше решение масштабу задачи. Рассуждайте вслух. Прежде чем что-то считать, пообщайтесь с интервьюером.

По возможности пройдитесь по нескольким конкретным сценариям использования. Это поможет вам сформировать общую архитектуру и, скорее всего, обнаружить крайние случаи, о которых вы еще не думали.

Следует ли на этом этапе обозначать конечные точки API-интерфейса и схему базы данных? Это зависит от задачи. Если вас просят спроектировать что-то масштабное, такое как поиск Google, то лучше не углубляться настолько сильно. Если же речь идет о серверной части многопользовательской игры в покер, эти аспекты вполне можно указать. Общайтесь с интервьюером.

### Пример

Давайте посмотрим, как происходит разработка общей архитектуры на примере новостной ленты. Вам не нужно понимать, как на самом деле работает эта система. Все детали будут описаны в главе 11.

На высоком уровне архитектура делится на два потока: публикацию статей и составление новостной ленты.

- Посты в ленте. Когда пользователь публикует пост, соответствующая информация записывается в кэш или базу данных, а сам пост появляется в ленте новостей его друзей.

- Составление ленты новостей. Новостная лента формируется путем группировки постов ваших друзей в обратном хронологическом порядке.

На рис. 3.1 и 3.2 показан общий принцип публикации постов и, соответственно, составления новостной ленты.



Рис. 3.1

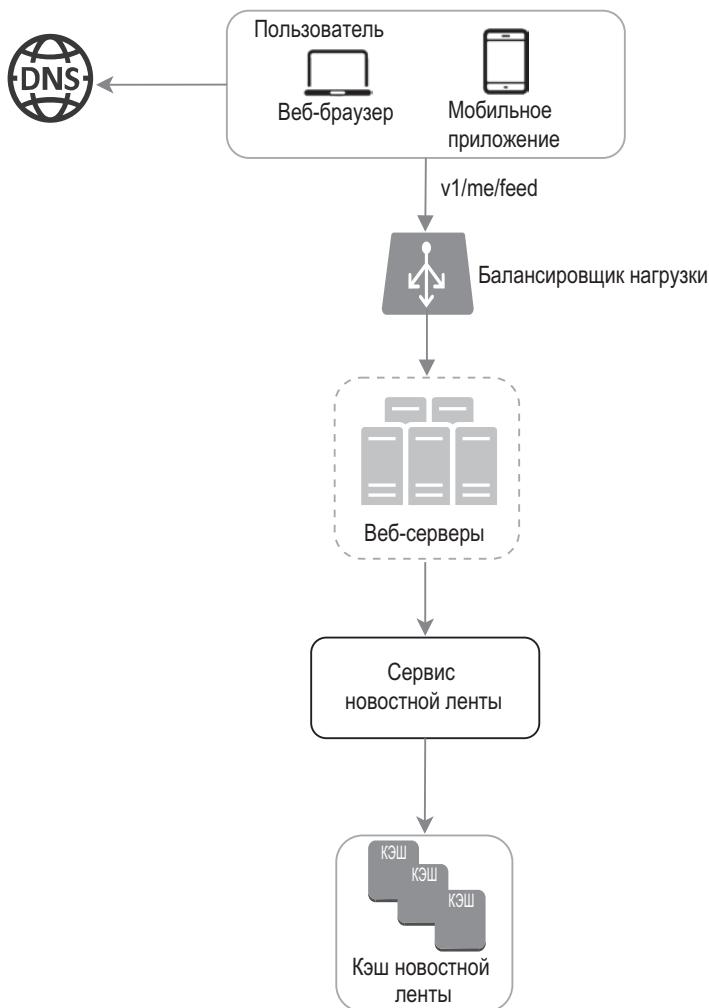


Рис. 3.2

### Шаг 3: глубокое погружение в проектирование

На этом этапе вы и интервьюер достигли следующих целей:

- согласовали общие требования и будущий масштаб;
- начертили примерную схему архитектуры;

- узнали мнение интервьюера о вашем общем решении;
- получили какое-то базовое представление о том, на каких областях нужно сосредоточиться при подробном проектировании (исходя из того, что ответил интервьюер).

Работая совместно с интервьюером, вы должны определить компоненты архитектуры и назначить им тот или иной приоритет. Стоит подчеркнуть, что все интервью проходят по-разному. Иногда вам могут дать понять, что желательно сосредоточиться на общей архитектуре. Иногда, если кандидат претендует на серьезную должность, обсуждение может коснуться характеристик производительности системы с вероятным упором на узкие места и оценку необходимых ресурсов. В большинстве случаев кандидата просят углубиться в детали каких-то компонентов системы. Если вы проектируете сервис для сокращения URL-адресов, особый интерес будет представлять функция хеширования, превращающая длинный адрес в короткий. В системе обмена сообщениями двумя интересными аспектами являются снижение латентности и поддержка онлайн/оффлайн-статусов.

Ключевую роль играет рациональное использование времени. Очень легко зациклиться на несущественных деталях, которые не позволяют оценить ваши способности. Вы должны показать интервьюеру то, что он хочет о вас узнать. Страйтесь не углубляться в ненужные подробности. Например, лучше не скатываться в подробное обсуждение алгоритма EdgeRank, с помощью которого Facebook взвешивает публикации в своей ленте, так как это отнимает ценнее время и не демонстрирует вашу способность проектировать масштабируемые системы.

## Пример

Итак, мы обсудили общую архитектуру ленты новостей, и интервьюер одобрил ваше решение. Теперь мы исследуем два самых важных сценария использования.

1. Публикация постов.
2. Выдача новостной ленты.

На рис. 3.3 и 3.4 в деталях показан принцип работы этих двух функций. Подробнее об этом речь пойдет в главе 11.

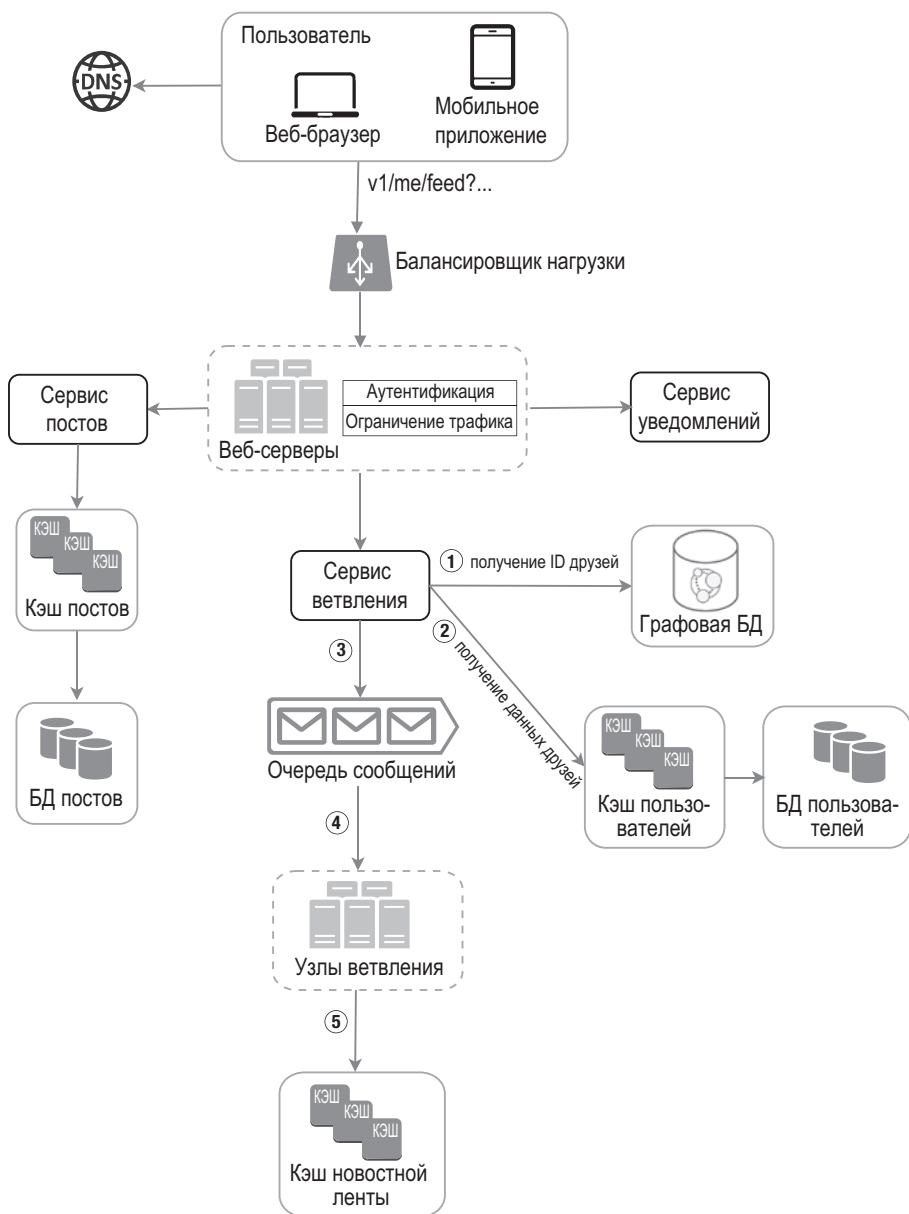


Рис. 3.3

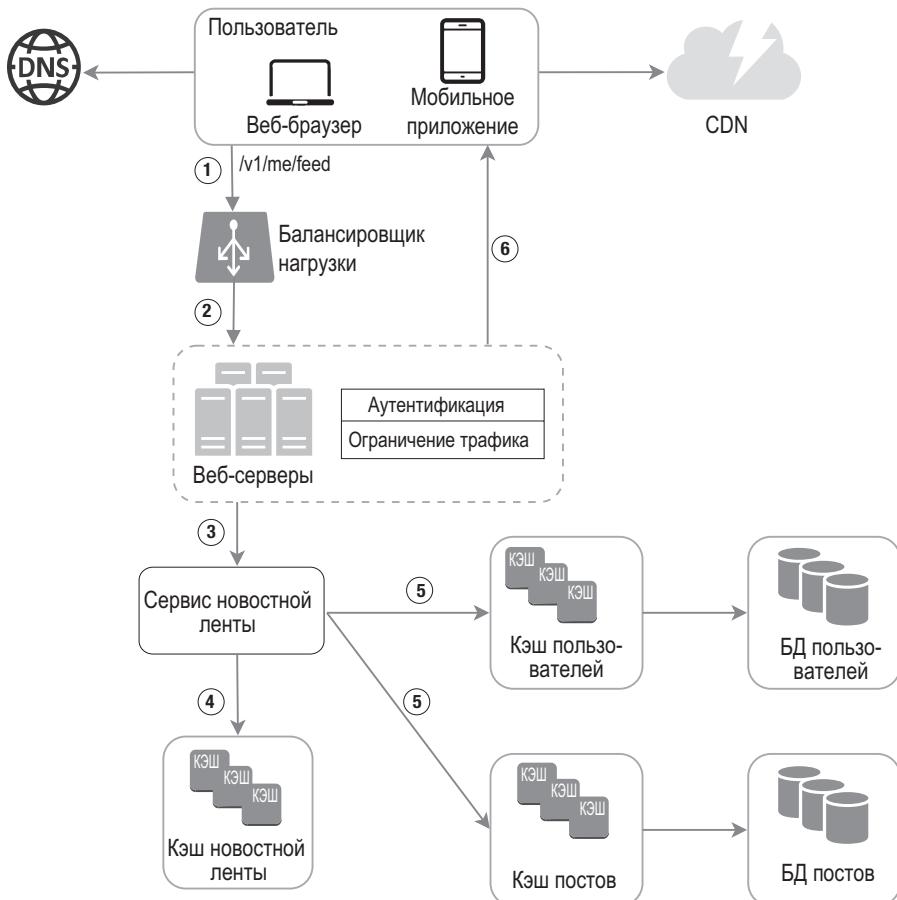


Рис. 3.4

### Шаг 4: подведение итогов

На заключительном этапе вам могут задать несколько уточняющих вопросов или предложить обсуждение других аспектов на ваш выбор. Вот несколько советов, которыми можно руководствоваться.

- Интервьюер может попросить вас обозначить узкие места системы и обсудить потенциальные улучшения. Никогда не утверждайте,

что ваше решение идеальное. Всегда можно что-то улучшить. Это отличная возможность продемонстрировать свое критическое мышление и оставить о себе хорошее впечатление.

- Возможно, стоит провести краткий обзор вашей архитектуры, особенно если вы предложили несколько решений. Это позволит интервьюеру освежить память, что может пригодиться после длинного собеседования.
- Будет интересно поговорить о внештатных ситуациях, таких как поломка серверов, разрыв сети и т. д.
- Стоит затронуть эксплуатационные вопросы. Как вы отслеживаете метрики и журналы ошибок? Как выкатывается система?
- Проведение следующего этапа масштабирования — тоже весьма интересная тема. Например, если ваша текущая архитектура поддерживает 1 миллион пользователей, какие изменения нужно внести, чтобы увеличить эту цифру до 10 миллионов?
- Если еще остается время, предложите дальнейшие улучшения.

В заключение составим краткий список того, что следует и не следует делать.

### Рекомендуется

- Всегда уточняйте. Не надейтесь на то, что ваши предположения окажутся верными.
- Определитесь с требованиями.
- Не существует правильного или лучшего ответа. У молодого стартапа и известной компании с миллионами пользователей разные архитектуры, которые решают разные задачи. Убедитесь в том, что вы понимаете требования.
- Делитесь мыслями с интервьюером. Общайтесь с ним.
- По возможности предложите разные подходы.
- Согласовав с интервьюером общую архитектуру, подробно обсудите каждый компонент. Начинайте проектирование с самых важных компонентов.

- Предлагайте разные идеи. Хороший интервьюер работает с кандидатом, как с членом своей команды.
- Никогда не сдавайтесь.

### **Не рекомендуется**

- Не приходите на интервью, не подготовившись к типичным вопросам.
- Не начинайте работать над решением, пока не уточните требования и собственные предположения.
- Не уделяйте сразу слишком много времени какому-то одному компоненту. Предложите общую архитектуру, а затем уже погружайтесь в детали.
- Если испытываете затруднения, не стесняйтесь обратиться за подсказкой.
- Опять же, не замыкайтесь в себе. Не рассуждайте молча.
- Не думайте, что интервью заканчивается, как только вы изложили свое решение. Интервьюер сам определяет, продолжать или нет. Постоянно интересуйтесь его мнением.

### **Время, выделяемое на каждый этап**

Вопросы, задаваемые на интервью по проектированию ИТ-систем, обычно носят крайне общий характер, и для обсуждения всей архитектуры целиком недостаточно 45 минут или часа. Очень важно рационально использовать свое время. Сколько минут следует выделить на каждый этап? Ниже приводятся очень приблизительные рекомендации о том, как разбить 45-минутное интервью. Пожалуйста, имейте в виду, что это лишь примерные расчеты: распределение времени зависит от масштабов задачи и требований, которые озвучил интервьюер.

Шаг 1. Понять задачу и определить масштаб: 3–10 минут.

Шаг 2. Предложить общее решение и получить одобрение: 10–15 минут.

Шаг 3. Подробное проектирование: 10–25 минут.

Шаг 4. Подведение итогов: 3–5 минут.

---

# 4

---

## ПРОЕКТИРОВАНИЕ ОГРАНИЧИТЕЛЯ ТРАФИКА

Ограничитель трафика используется в сетевых системах для управления скоростью передачи данных от клиента или сервиса. В мире HTTP он ограничивает количество запросов, которые пользователь может отправить за определенный промежуток времени. Если исчерпано максимальное число API-запросов, заданное ограничителем трафика, все последующие вызовы блокируются. Вот несколько примеров:

- пользователь может создать не больше двух сообщений в секунду;
- с одного IP-адреса можно создать не больше 10 учетных записей в день;
- на одном устройстве можно получить не больше 5 наград в неделю.

В этой главе вам будет предложено разработать ограничитель трафика. Но сначала рассмотрим преимущества использования этого компонента в API.

- Предотвращение нехватки ресурсов, вызванной DoS-атакой (Denial of Service — «отказ в обслуживании») [1]. Почти все API-интерфейсы, предоставляемые крупными технологическими компаниями, накладывают какие-то ограничения на трафик. Например, Twitter ограничивает количество твитов до 300 за 3 часа [2]. API Google Docs имеет по умолчанию следующее ограничение: 300 запросов на чтение в минуту для каждого пользователя [3]. Ограничитель трафика предотвращает как спланированные, так и непредумышленные DoS-атаки, блокируя избыточные вызовы.
- Экономия бюджета. Ограничение избыточных запросов позволяет освободить часть серверов и выделить больше ресурсов для высокопроизводительных запросов.

приоритетных API. Это чрезвычайно важно для компаний, которые используют платные сторонние API. Например, вам может стоить денег каждое обращение к внешним API, которые позволяют проверить кредитные средства, отправить платеж, получить записи медкарты и т. д. Ограничение количества вызовов играет важную роль в снижении расходов.

- Предотвращение перегрузки серверов. Чтобы снизить нагрузку на серверы, ограничитель трафика фильтрует лишние запросы, исходящие от ботов или недобросовестных пользователей.

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Ограничение трафика можно реализовать с помощью разных алгоритмов, каждый из которых имеет как преимущества, так и недостатки. Общение с интервьюером помогает прояснить, какого рода ограничитель нужно спроектировать.

**Кандидат:** «Какого рода ограничитель трафика мы будем проектировать: клиентский или серверный на стокроне API?»

**Интервьюер:** «Отличный вопрос. Мы сосредоточимся на серверном ограничителе трафика API».

**Кандидат:** «Каким образом этот ограничитель трафика будет блокировать API-запросы: по IP, ID пользователей или другим свойствам?»

**Интервьюер:** «Ограничитель трафика должен быть достаточно гибким для того, чтобы поддерживать разные наборы правил блокировки».

**Кандидат:** «Каков масштаб системы? Она предназначена для стартапа или для крупной компании с множеством пользователей?»

**Интервьюер:** «Система должна быть способна справиться с большим количеством запросов».

**Кандидат:** «Будет ли система работать в распределенном окружении?»

**Интервьюер:** «Да».

**Кандидат:** «Ограничитель трафика реализован в коде приложения или в виде отдельного сервиса?»

**Интервьюер:** «Это остается на ваше усмотрение».

**Кандидат:** «Нужно ли уведомлять пользователей, запросы которых блокируются?»

**Интервьюер:** «Да».

## Требования

Ниже приведен краткий список требований к системе.

- Точная фильтрация избыточных запросов.
- Низкая латентность. Ограничитель трафика не должен увеличивать время возвращения HTTP-ответов.
- Минимально возможное потребление памяти.
- Распределенное ограничение трафика. Ограничитель должен быть доступен разным серверам или процессам.
- Обработка исключений. Когда запрос пользователя блокируется, возвращайте понятное сообщение об ошибке.
- Высокая отказоустойчивость. Если с ограничителем трафика возникнут какие-то проблемы (например, станет недоступным сервер кэширования), это не должно повлиять на систему в целом.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Чтобы не усложнять, будем использовать стандартную модель клиент-серверного взаимодействия.

### Где разместить ограничитель трафика?

Очевидно, что ограничитель трафика можно реализовать либо на клиентской, либо на серверной стороне.

- Клиентская реализация. В целом, клиент — это не самое надежное место для ограничения трафика, так как клиентские запросы могут быть легко подделаны злоумышленниками. Более того, реализацией клиента может заниматься кто-то другой.

- Серверная реализация. На рис. 4.1 показан ограничитель трафика, размещенный на стороне сервера.

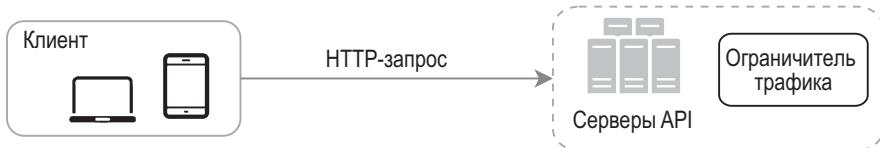


Рис. 4.1

Помимо клиентской и серверной реализаций существует и третий путь. Вместо того чтобы размещать ограничитель трафика на серверах, его можно оформить в виде промежуточного слоя, который фильтрует запросы к вашему API, как показано на рис. 4.2.

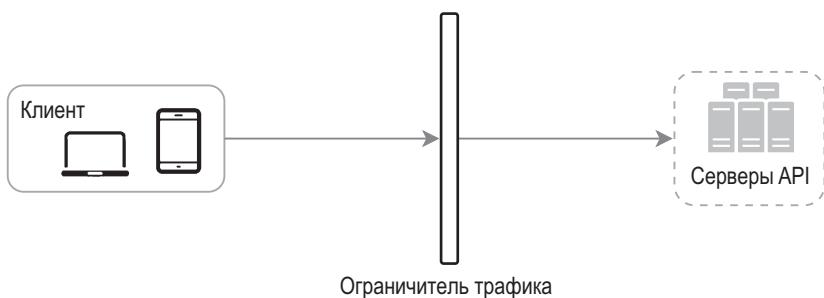


Рис. 4.2

Воспользуемся примером на рис. 4.3, чтобы продемонстрировать, как работает ограничение трафика в этой конфигурации. Предположим, наш API допускает два запроса в секунду, а клиент успел послать серверу сразу три. Первые два запроса направляются к API-интерфейсу. Однако ограничитель трафика блокирует третий запрос, возвращая код состояния HTTP 429. Такой ответ говорит о том, что пользователь послал слишком много запросов.

Облачные микросервисы [4] пользуются широкой популярностью, и ограничение трафика в них обычно реализовано внутри компонента под названием «шлюз API». Шлюз API — это полностью управляемый сервис

с поддержкой ограничения трафика, завершения SSL-запросов, аутентификации, ведения белых списков IP-адресов, обслуживания статического содержимого и т. д. Пока что нам достаточно знать, что шлюз API — это промежуточный слой, который поддерживает ограничение трафика.

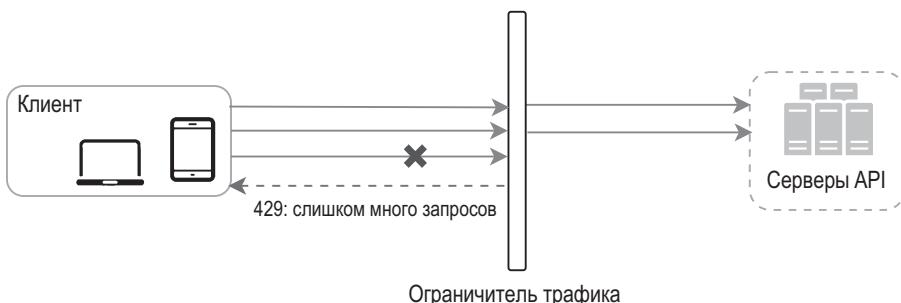


Рис. 4.3

При проектировании ограничителя трафика необходимо задаться следующим вопросом: где он должен быть реализован — на стороне сервера или в шлюзе? Универсального ответа не существует. Все зависит от текущего технологического стека, инженерных ресурсов, приоритетов и целей вашей компании. Вот несколько общих рекомендаций.

- Проанализируйте ваш текущий технологический стек, включая язык программирования, сервис кэширования и т. д. Убедитесь в том, что язык, который вы используете, позволит вам эффективно реализовать ограничитель трафика на стороне сервера.
- Выберите алгоритм ограничения трафика, который соответствует вашим бизнес-требованиям. Когда все реализуется на стороне сервера, алгоритм находится под вашим полным контролем. Но если вы используете сторонний шлюз, выбор может быть ограничен.
- Если вы уже используете микросервисы и ваша архитектура предусматривает шлюз API для выполнения аутентификации, ведения белых списков IP-адресов и т. д., ограничитель трафика можно добавить в этот шлюз.
- Разработка собственного сервиса для ограничения трафика занимает время. Если ваших инженерных ресурсов для этого недостаточно, лучше выбрать коммерческий шлюз API.

## Алгоритмы ограничения трафика

Ограничение трафика может быть реализовано с помощью разных алгоритмов, у каждого из которых есть определенные преимущества и недостатки. Эта глава посвящена другим темам, но чтобы выбрать алгоритм или сочетание алгоритмов, которые подходят для ваших задач, не помешает иметь о них общее представление. Вот список популярных вариантов:

- алгоритм маркерной корзины (token bucket);
- алгоритм дырявого ведра (leaking bucket);
- счетчик фиксированных интервалов (fixed window counter);
- журнал скользящих интервалов (sliding window log);
- счетчик скользящих интервалов (sliding window counter).

### Алгоритм маркерной корзины

Алгоритм маркерной корзины довольно популярен. Он простой и понятный, и многие интернет-компании, такие как Amazon [5] и Stripe [6], используют его для фильтрации API-запросов.

Алгоритм маркерной корзины работает следующим образом.

- Маркерная корзина — это контейнер с заранее определенной емкостью. В нее регулярно помещают маркеры. Когда она окончательно заполняется, маркеры больше не добавляются. На рис. 4.4 показана маркерная корзина с емкостью 4. Наполнитель ежесекундно помещает в корзину 2 маркера. Когда корзина заполняется, последующие маркеры отбрасываются.
- Каждый запрос потребляет один маркер. При поступлении запроса мы проверяем, достаточно ли маркеров в корзине. На рис. 4.5 показано, как это работает:
  - ◆ если маркеров достаточно, мы удаляем по одному из них для каждого запроса, и запрос проходит дальше;
  - ◆ если маркеров недостаточно, запрос отклоняется.

На рис. 4.6 проиллюстрирована логика траты маркеров, заправки корзины и ограничения трафика. В этом примере корзина вмещает 4 маркера, а скорость заправки равна 2 маркера в минуту.



Рис. 4.4

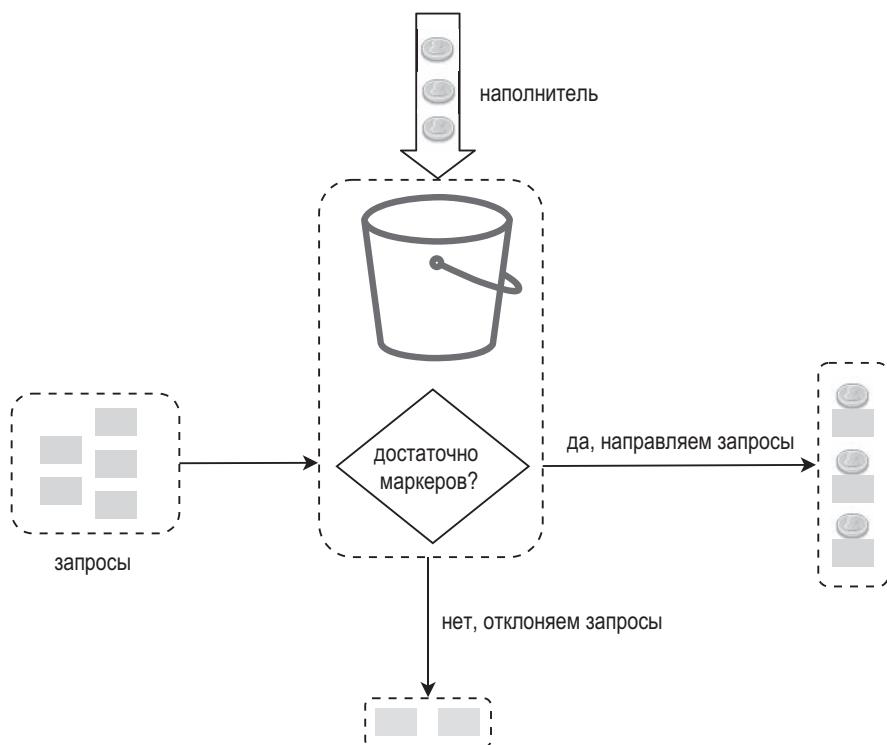


Рис. 4.5

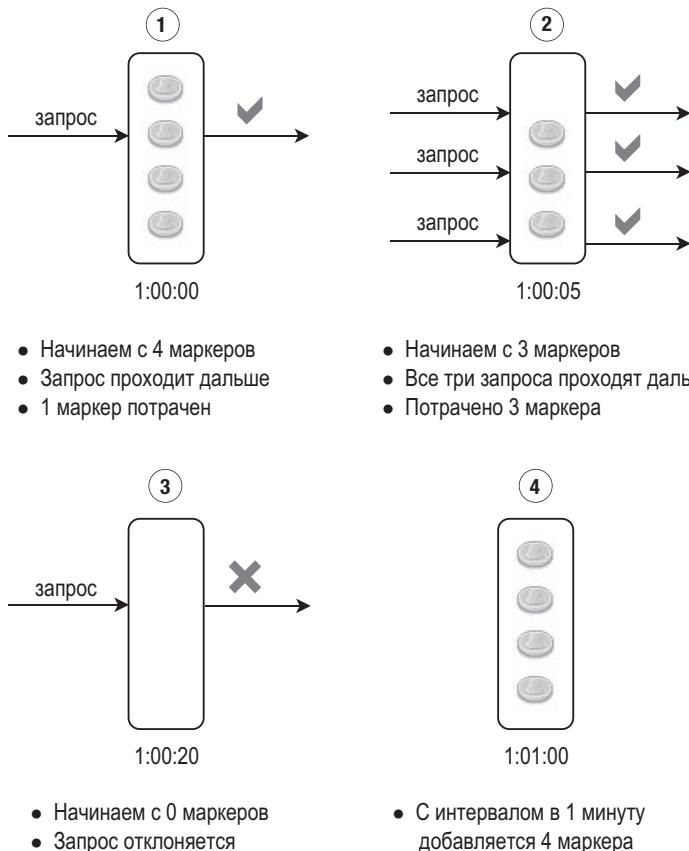


Рис. 4.6

Алгоритм маркерной корзины принимает два параметра:

- размер корзины: максимальное количество маркеров, которое может в ней находиться;
- частота пополнения: количество маркеров, ежесекундно добавляемых в корзину.

Сколько корзин нам нужно? Это зависит от правил ограничения трафика. Вот несколько примеров.

- Для разных конечных точек API обычно нужны разные корзины. Например, если пользователь публикует 1 сообщение в секунду,

добавляет 150 друзей в день и «лайкает» 5 сообщений в секунду, каждому пользователю нужно выделить 3 корзины.

- Если нам нужно фильтровать запросы в зависимости от IP-адресов, каждому IP-адресу требуется корзина.
- Если система допускает не больше 10 000 запросов в секунду, логично предусмотреть глобальную корзину для всех запросов.

Преимущества:

- легкая реализация;
- эффективное потребление памяти;
- маркерная корзина может справиться с короткими всплесками трафика; пока в корзине остаются маркеры, запрос обрабатывается.

Недостатки:

- несмотря на то что алгоритм принимает лишь два параметра (размер корзины и частота пополнения), подобрать подходящие значения может быть непросто.

### **Алгоритм дырявого ведра**

Алгоритмы дырявого ведра и маркерной корзины очень похожи, только первый обрабатывает запросы с фиксированной скоростью. Обычно его реализуют с использованием очереди типа FIFO. Этот алгоритм работает так:

- при поступлении запроса система проверяет, заполнена ли очередь. Запрос добавляется в очередь при наличии места;
- в противном случае запрос отклоняется;
- запросы извлекаются из очереди и обрабатываются через равные промежутки времени.

Принцип работы алгоритма проиллюстрирован на рис. 4.7.

Алгоритм дырявого ведра принимает два параметра:

- размер ведра: равен размеру очереди; очередь хранит запросы, которые обрабатываются с постоянной скоростью;
- скорость утечки: определяет, сколько запросов можно обработать за определенный промежуток времени (обычно за 1 секунду).

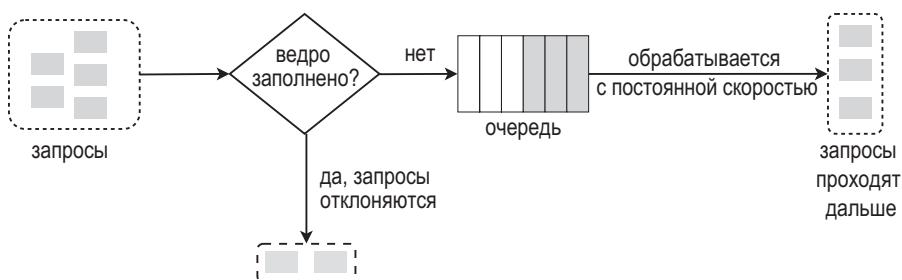


Рис. 4.7

Дырявые ведра для ограничения трафика использует платформа электронной коммерции Shopify [7].

Преимущества:

- эффективное потребление памяти при ограниченном размере очереди;
- запросы обрабатываются с постоянной скоростью, поэтому этот алгоритм подходит для задач, которые требуют стабильной скорости обработки.

Недостатки:

- всплеск трафика наполняет очередь старыми запросами, и, если их вовремя не обработать, новые запросы будут отклоняться;
- несмотря на то что алгоритм принимает лишь два параметра, подобрать подходящие значения может быть непросто.

### Счетчик фиксированных интервалов

Счетчик фиксированных интервалов работает так.

- Алгоритм делит заданный период времени на одинаковые интервалы и назначает каждому из них счетчик.
- Каждый запрос инкрементирует счетчик на 1.
- Как только счетчик достигнет заранее заданного лимита, новые запросы начинают отклоняться, пока не начнется следующий интервал.

Давайте посмотрим, как это работает, на конкретном примере. На рис. 4.8 в качестве интервала выбрана 1 секунда, а система допускает не больше 3 запросов в секунду. На каждом секундном интервале система проверяет количество поступивших запросов и отклоняет лишние.

Основная проблема этого алгоритма в том, что всплески трафика на границе временных интервалов могут привести к тому, что система может превысить квоту и принять больше запросов. Рассмотрим следующую ситуацию:

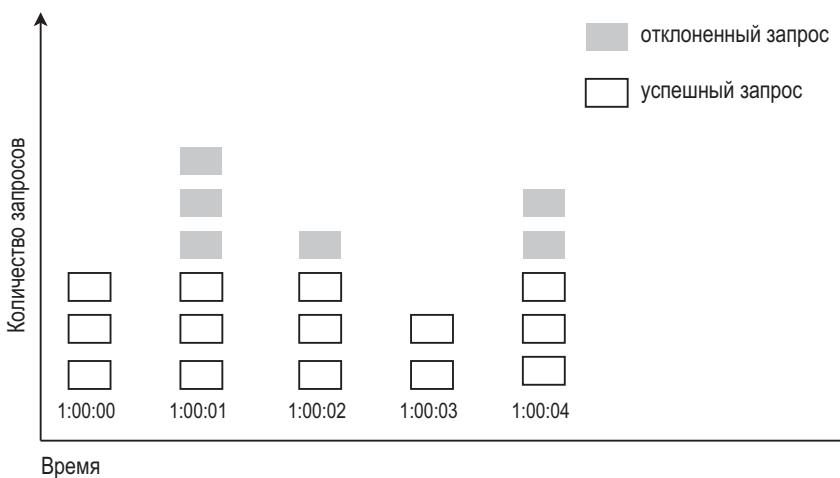


Рис. 4.8

На рис. 4.9 система допускает не больше 5 запросов в минуту, а квота ради удобства сбрасывается ежеминутно. Как видите, на интервале между 2:00:00 и 2:01:00 имеется пять запросов и еще пять — между 2:01:00 и 2:02:00. Таким образом, за одну минуту, между 2:00:30 и 2:01:30, система принимает 10 запросов — в два раза больше позволенного.

Преимущества:

- эффективное потребление памяти;
- понятность;
- сброс квоты доступных запросов в конце временного интервала подходит для ряда задач.

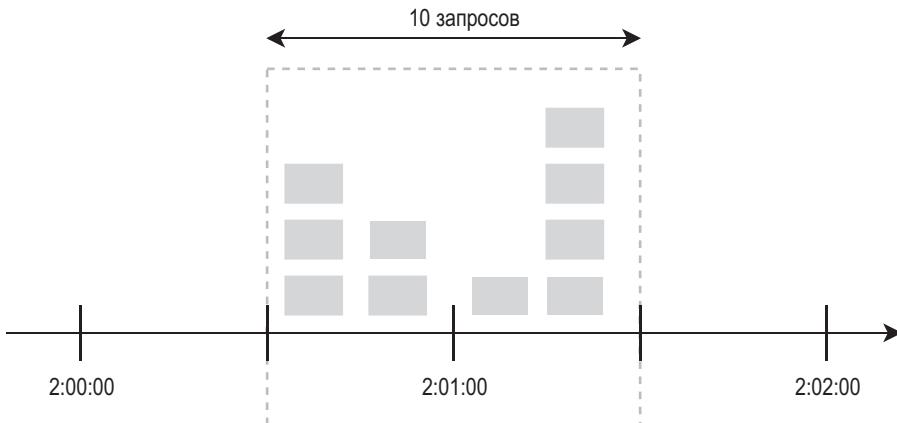


Рис. 4.9

Недостатки:

- всплески трафика на границе временных интервалов могут привести к приему запросов, количество которых превышает квоту.

### Журнал скользящих интервалов

Как уже упоминалось ранее, у счетчика фиксированных интервалов есть серьезная проблема: на границах интервала может быть принято больше запросов. С этим помогает справиться журнал скользящих интервалов. Вот как он работает.

- Алгоритм следит за временными метками запросов. Временные метки обычно хранятся в кэше, например, в упорядоченных множествах Redis [8].
- Когда поступает новый запрос, все просроченные запросы отбрасываются. Просроченными считают запросы раньше начала текущего временного интервала.
- Временные метки новых запросов заносятся в журнал.
- Если количество записей в журнале не превышает допустимое, запрос принимается, а если нет — отклоняется.

Рассмотрим этот алгоритм на примере, представленном на рис. 4.10.

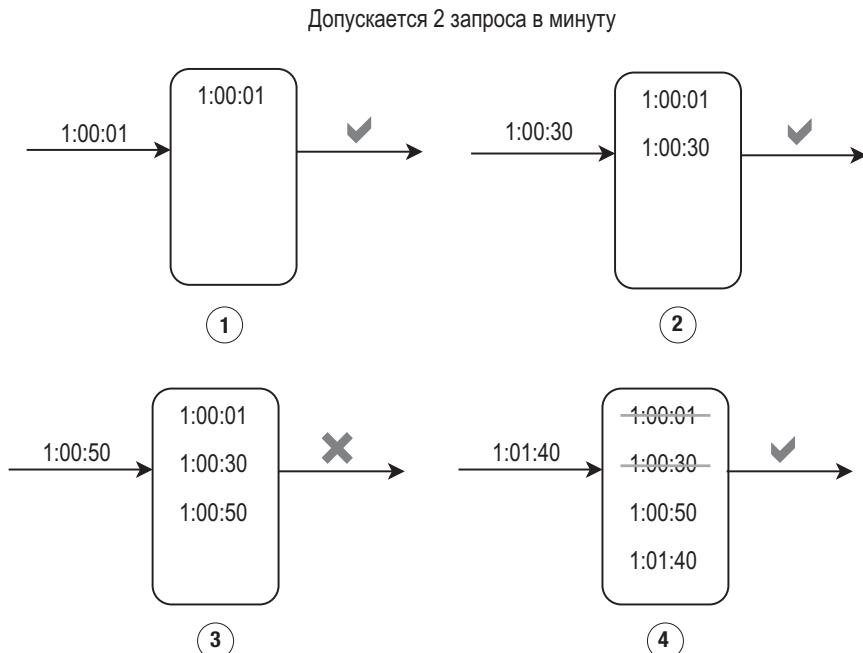


Рис. 4.10

В этом примере ограничитель трафика допускает не больше 2 запросов в минуту. Обычно в журнал записываются временные метки Linux. Но в нашем примере для удобочитаемости используется формат времени, понятный человеку.

- Когда в 1:00:01 приходит новый запрос, журнал пуст, поэтому запрос принимается.
- В 1:00:30 приходит новый запрос, и временная метка 1:00:30 записывается в журнал. После этого размер журнала равен 2, что не превышает допустимое значение. Запрос принимается.
- В 1:00:50 приходит новый запрос, и временная метка 1:00:50 записывается в журнал. После этого размер журнала равен 3, что превышает допустимое значение. Запрос отклоняется, хотя временная метка остается в журнале.
- В 1:01:40 приходит новый запрос. Запросы в диапазоне [1:00:40, 1:01:40) находятся на текущем временном интервале,

а запросы, отправленные до 1:00:40, являются устаревшими. Две просроченные временные метки, 1:00:01 и 1:00:30, удаляются из журнала. После этого размер журнала равен 2, поэтому запрос принимается.

Преимущества:

- ограничение трафика, реализованное с помощью этого алгоритма, получается очень точным; на любом скользящем интервале запросы не превышают заданный лимит.

Недостатки:

- этот алгоритм потребляет много памяти, потому что даже в случае отклонения запроса соответствующая временная метка записывается в журнал.

### Счетчик скользящих интервалов

Счетчик скользящих интервалов — это гибридный подход, сочетающий в себе два предыдущих алгоритма. Его можно реализовать двумя разными способами. Одну из реализаций мы рассмотрим далее, а ссылка на описание другой будет дана в конце этого раздела. Принцип работы этого алгоритма показан на рис. 4.11.

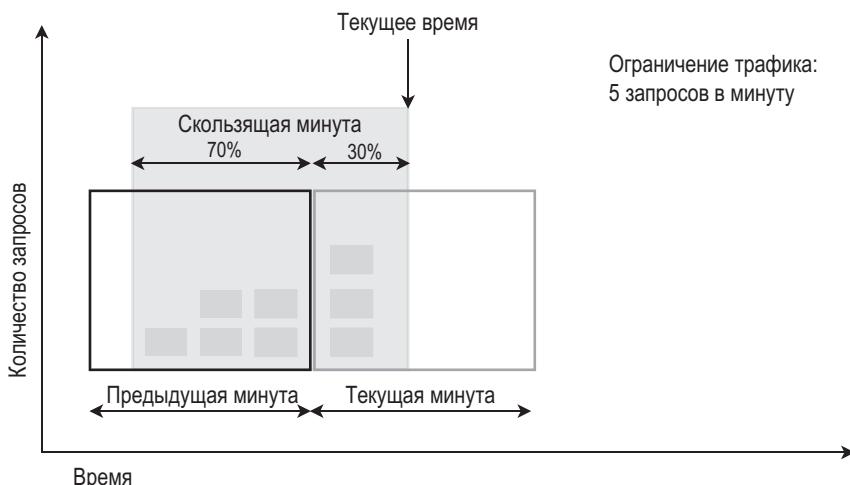


Рис. 4.11

Предположим, ограничитель трафика допускает не больше 7 запросов в минуту. У нас 5 запросов за предыдущий интервал и 3 в текущем. На отметке в 30 % от текущего интервала количество запросов на скользящем интервале вычисляется по следующей формуле:

- запросы на текущем интервале + запросы на предыдущем интервале \* процент предыдущего интервала, который занимает скользящий интервал;
- используя эту формулу, мы получаем  $3 + 5 * 0,7\% = 6,5$  запроса. В зависимости от ситуации это число можно округлить к большему или меньшему значению. В нашем примере оно округляется до 6.

Так как ограничитель трафика допускает не больше 7 запросов в минуту, текущий запрос может быть принят. Но после получения еще одного запроса лимит будет исчерпан.

Мы не хотим сделать эту книгу еще толще, так что не станем рассматривать вторую реализацию. Заинтересовавшиеся читатели могут обратиться к справочному материалу [9]. Этот алгоритм не идеален. Он имеет как преимущества, так и недостатки.

Преимущества:

- сглаживание всплесков трафика: текущая частота запросов зависит от той, которая использовалась на предыдущем интервале;
- экономия памяти.

Недостатки:

- работает только для нежестких ретроспективных интервалов; частота получается приблизительной, так как подразумевается, что запросы на предыдущем интервале распределены равномерно. Но это может быть не настолько серьезной проблемой, как кажется на первый взгляд: согласно экспериментам, проведенным компанией Cloudflare [10], из 400 миллионов запросов только 0,003 % были ошибочно отклонены или приняты сверх квоты.

## Общая архитектура

Алгоритмы ограничения трафика имеют простой принцип работы. В целом, нам нужен счетчик, чтобы знать, сколько запросов отправлено

одним пользователем, с одного IP-адреса и т. д. Если счетчик превышает лимит, запрос отклоняется.

Где следует хранить счетчики? Базу данных лучше не использовать ввиду медленного доступа к диску. Мы выбрали резидентный кэш, так как он быстрый и поддерживает стратегию удаления записей в зависимости от времени их создания. Одним из популярных решений для ограничения трафика является Redis [11]. Это резидентное хранилище предлагает две команды: `INCR` и `EXPIRE`.

- `INCR` увеличивает хранимый счетчик на 1.
- `EXPIRE` устанавливает срок хранения счетчика, по истечении которого тот автоматически удаляется.

На рис. 4.12 показана общая архитектура ограничения трафика, которая работает следующим образом:



Рис. 4.12

- Клиент шлет запрос промежуточному ограничителю трафика.
- Промежуточный ограничитель трафика извлекает счетчик из соответствующего бакета Redis и проверяет, достигнут ли лимит.
  - ◆ Если лимит достигнут, запрос отклоняется.
  - ◆ Если лимит не достигнут, запрос направляется серверам API. Тем временем система инкрементирует счетчик и сохраняет его обратно в Redis.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

Общая архитектура, показанная на рис. 4.12, оставляет без ответа следующие вопросы:

- Как создаются правила ограничения трафика? Где эти правила хранятся?
- Что делать с отклоненными запросами?

В этом разделе мы сначала ответим на вопросы, касающиеся правил ограничения трафика, а затем перейдем к стратегиям обработки отклоненных запросов. В конце будут рассмотрены такие аспекты, как ограничение трафика в распределенных окружениях, подробная архитектура, а также оптимизация и мониторинг производительности.

### Правила ограничения трафика

Компания Lyft открыла исходный код своего компонента для ограничения трафика [12]. Давайте заглянем внутрь и посмотрим, какие правила там используются:

```
domain: messaging
descriptors:
  - key: message_type
    Value: marketing
    rate_limit:
      unit: day
      requests_per_unit: 5
```

В этом примере система сконфигурирована для приема не более пяти рекламных сообщений в день. Вот еще один пример:

```
domain: auth
descriptors:
  - key: auth_type
    Value: login
    rate_limit:
      unit: minute
      requests_per_unit: 5
```

В этом случае клиентам нельзя входить в систему чаще 5 раз в минуту. Правила обычно записываются в конфигурационные файлы и сохраняются на диске.

## Превышение ограничений трафика

Когда запрос отклоняется, API возвращает клиенту HTTP-ответ с кодом 429 («слишком много запросов»). В зависимости от ситуации отклоненные запросы могут быть записаны в очередь, чтобы позже их можно было обработать. Например, если некоторые заказы отклоняются из-за перегруженности системы, мы можем отложить их на потом.

### Заголовки ограничителя трафика

Как клиент узнает, что его трафик ограничивается? И откуда он может узнать количество запросов, которые он может выполнить, прежде чем вступят в силу ограничения? Ответ заключается в заголовках HTTP-ответов. Ограничитель трафика возвращает клиентам следующие HTTP-заголовки:

- **X-Ratelimit-Remaining**. Количество допустимых запросов, которое остается в текущем интервале.
- **X-Ratelimit-Limit**. Количество вызовов, доступных клиенту в каждом временном интервале.
- **X-Ratelimit-Retry-After**. Количество секунд, которое должно пройти, прежде чем ваши запросы престанут отклоняться.

Если пользователь отправит слишком много запросов, клиенту будет возвращен код ошибки 429 и заголовок X-Ratelimit-Retry-After.

## Подробная архитектура

На рис. 4.13 представлена подробная архитектура системы.

- Правила хранятся на диске. Рабочие узлы регулярно считывают их с диска и сохраняют в кэш.
- Когда клиент обращается к серверу, его запрос сначала проходит через промежуточный ограничитель трафика.

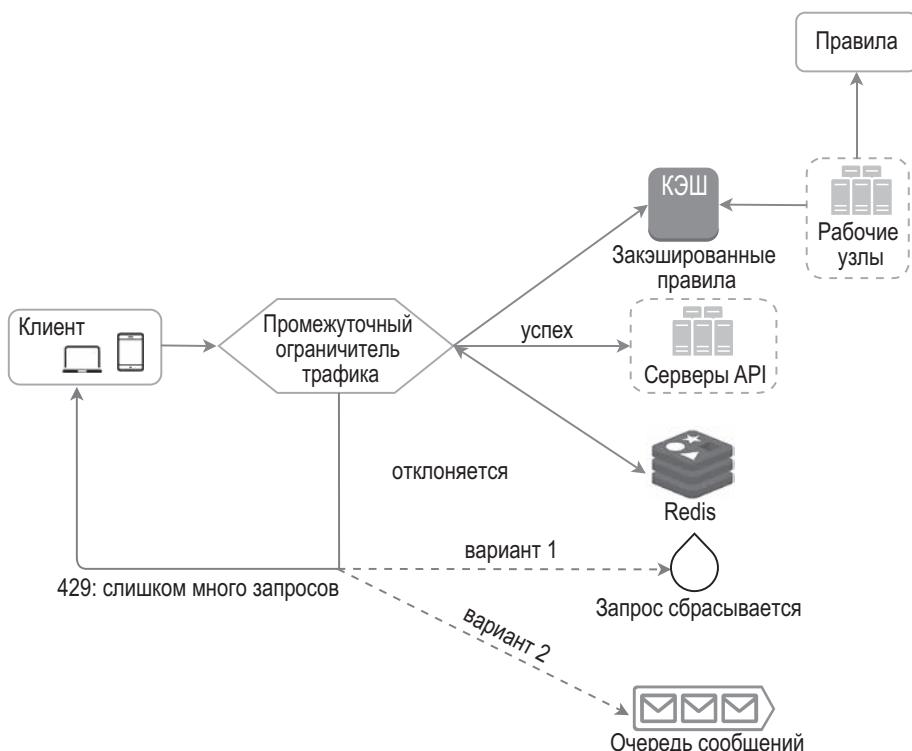


Рис. 4.13

- Промежуточный ограничитель трафика загружает правила из кэша. Он извлекает из кэша Redis счетчики и временную метку последнего запроса. Ограничитель трафика принимает решение в зависимости от полученной информации:
  - ◆ если запрос не отклоняется, он направляется к серверам API;
  - ◆ если запрос отклоняется, ограничитель трафика возвращает клиенту код ошибки 429. Тем временем запрос либо сбрасывается, либо направляется в очередь.

### Ограничитель трафика в распределенном окружении

Спроектировать ограничитель трафика, который работает в рамках одного сервера, несложно. А вот масштабирование системы для поддержки

множества серверов и конкурентных потоков выполнения — совсем другое дело. Существует две проблемы:

- состояние гонки;
- сложность синхронизации.

### Состояние гонки

Как уже отмечалось ранее, ограничитель трафика в целом работает так:

- считывает значение *счетчика* из Redis;
- проверяет, превышает ли (*счетчик + 1*) установленный лимит;
- если нет, инкрементирует значение счетчика в Redis на 1.

Как показано на рис. 4.14, состояние гонки может возникнуть в высококонкурентном окружении.

Начальное значение счетчика: 3

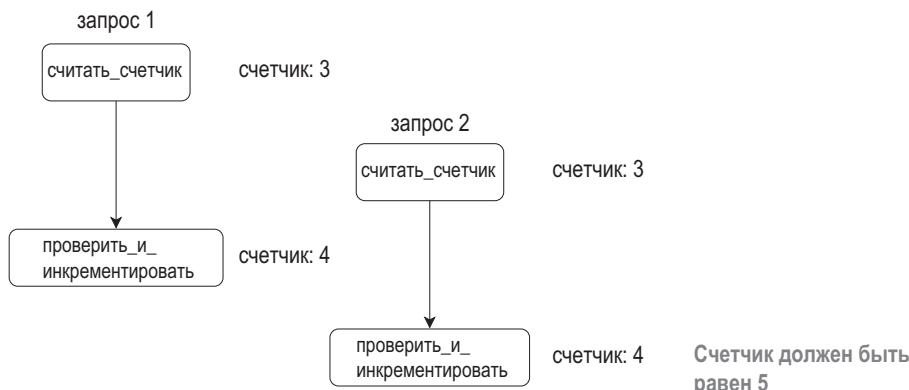


Рис. 4.14

Предположим, что значение счетчика в Redis равно 3. Если оно будет параллельно считано двумя разными запросами, прежде чем один из них успеет записать его обратно, каждый из них инкрементирует счетчик на 1 и сохранит полученное значение, не сверившись с другим потоком выполнения. Оба запроса (потока) считают, что их значение счетчика, 4, является правильным. Но на самом деле счетчик должен быть равен 5.

Очевидным решением для устранения состояния гонки являются блокировки. Но они существенно замедлят вашу систему. В таких случаях часто используют две стратегии: скрипт Lua [13] или структуру данных «упорядоченные множества», доступную в Redis [8]. Если они вас заинтересовали, можете обратиться к справочным материалам [8] [13].

### Сложность синхронизации

Синхронизация — это еще один важный фактор, который нужно учитывать в распределенном окружении. Для обработки запросов, которые генерируют миллионы пользователей, одного сервера с ограничителем трафика может не хватить. При использовании нескольких таких серверов требуется синхронизация. Например, в левой части рис. 4.15 клиент 1 отправляет запрос ограничителю трафика 1, а клиент 2 — ограничителю трафика 2. Поскольку веб-уровень не хранит свое состояние, клиенты могут направлять свои запросы разным ограничителям, как показано в правой части рис. 4.15. В случае отсутствия синхронизации у ограничителя трафика 1 нет никаких данных о клиенте 2, поэтому он не может как следует выполнить свою работу.

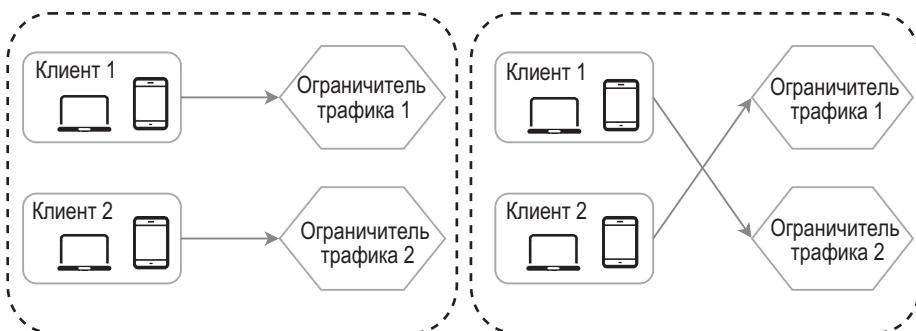


Рис. 4.15

Мы можем воспользоваться липкими сессиями. Это позволит клиенту отправлять запросы одному и тому же ограничителю трафика. Но этому решению не хватает ни масштабируемости, ни гибкости, поэтому использовать его не рекомендуется. Более разумный подход состоит в применении централизованных хранилищ данных, таких как Redis. Соответствующая конфигурация показана на рис. 4.16.

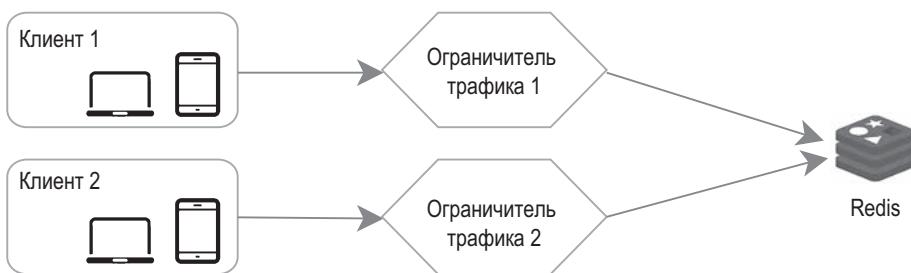


Рис. 4.16

### Оптимизация производительности

На интервью по проектированию ИТ-систем часто поднимается вопрос оптимизации производительности. Мы рассмотрим две области, которые можно улучшить.

Во-первых, ограничитель трафика обязательно должен быть распределен по разным центрам обработки данных, ведь чем дальше пользователь находится от ЦОД, тем выше латентность. Большинство поставщиков облачных услуг размещают пограничные серверы по всему миру. Например, по состоянию на 20 мая 2020 года у Cloudflare есть 194 географически распределенных пограничных сервера [14]. Для снижения латентности трафик автоматически направляется к ближайшему из них.



Рис. 4.17 (источник: [10])

Во-вторых, данные должны синхронизироваться в соответствии с моделью отложенной согласованности. Если вы с ней плохо знакомы, можете почитать раздел «Согласованность» в главе 6, посвященной проектированию хранилища вида «ключ–значение».

## **Мониторинг**

После реализации ограничителя трафика необходимо собрать аналитические данные, чтобы проверить, насколько он эффективен. Нас в основном интересует эффективность:

- алгоритма ограничения трафика;
- правил ограничения трафика.

Например, правила ограничения трафика слишком строгие, из-за чего будет теряться много корректных запросов. В этом случае их следует немножко ослабить. Ограничитель трафика также может становиться неэффективным во время внезапных всплесков активности, таких как распродажи. Чтобы этого не произошло, можно воспользоваться другим алгоритмом, который лучше справляется с такими условиями. Хорошим вариантом будет алгоритм маркерной корзины.

## **ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ**

В этой главе мы обсудили разные алгоритмы ограничения трафика и их достоинства/недостатки:

- алгоритм маркерной корзины;
- алгоритм дырявого ведра;
- счетчик фиксированных интервалов;
- журнал скользящих интервалов;
- счетчик скользящих интервалов.

Затем мы рассмотрели архитектуру системы, реализацию ограничителя трафика в распределенном окружении, оптимизацию производительности и мониторинг. Как и с любыми другими вопросами, которые задаются

во время интервью по проектированию ИТ-систем, вы можете отметить следующие аспекты (если время позволяет).

- Жесткое и гибкое ограничение трафика:
  - ◆ жесткое: количество запросов не может превысить лимит;
  - ◆ гибкое: запросы могут ненадолго превысить лимит.
- Ограничение трафика на разных уровнях. В этой главе мы обсудили только прикладной уровень (HTTP: уровень 7). Но трафик можно ограничивать, к примеру, по IP-адресам, используя Iptables [15] (IP: уровень 3). Отметим, что модель взаимодействия открытых систем (Open Systems Interconnection, OSI) имеет 7 уровней [16]: физический (уровень 1), канальный (уровень 2), сетевой (уровень 3), транспортный (уровень 4), сеансовый (уровень 5), уровень представления (уровень 6) и прикладной (уровень 7).
- Способы избежать ограничения трафика. Проектируйте свой клиент с учетом следующих рекомендаций:
  - ◆ используйте клиентский кэш, чтобы не выполнять API-вызовы слишком часто;
  - ◆ разберитесь, в чем состоит ограничение, и не отправляйте слишком много запросов за короткий промежуток времени;
  - ◆ предусмотрите код для обработки исключений или ошибок, чтобы ваш клиент мог как следует с ними справляться;
  - ◆ задержка между повторными вызовами должна быть достаточно длинной.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Rate-limiting strategies and techniques: <https://cloud.google.com/solutions/rate-limiting-strategies-techniques>
- [2] Twitter rate limits: <https://developer.twitter.com/en/docs/basics/rate-limits>
- [3] Google docs usage limits: <https://developers.google.com/docs/api/limits>
- [4] IBM microservices: <https://www.ibm.com/cloud/learn/microservices>

- [5] Throttle API requests for better throughput: <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html>
- [6] Stripe rate limiters: <https://stripe.com/blog/rate-limiters>
- [7] Shopify REST Admin API rate limits: <https://help.shopify.com/en/api/reference/rest-admin-api-rate-limits>
- [8] Better Rate Limiting With Redis Sorted Sets: <https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/>
- [9] System Design – Rate limiter and Data modelling: <https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250>
- [10] How we built rate limiting capable of scaling to millions of domains: <https://blog.cloudflare.com/counting-things-a-lot-of-different-things/>
- [11] Веб-сайт Redis: <https://redis.io/>
- [12] Lyft rate limiting: <https://github.com/lyft/ratelimit>
- [13] Scaling your API with rate limiters: <https://gist.github.com/ptarjan/e38f45f2dfe601419ca3af937fff574d#request-rate-limiter>
- [14] What is edge computing: <https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/>
- [15] Rate Limit Requests with Iptables: <https://blog.programster.org/rate-limit-requests-with-iptables>
- [16] Модель OSI: [https://ru.wikipedia.org/wiki/Сетевая\\_модель\\_OSI#Уровни\\_модели\\_OSI](https://ru.wikipedia.org/wiki/Сетевая_модель_OSI#Уровни_модели_OSI)

---

5

---

# СОГЛАСОВАННОЕ ХЕШИРОВАНИЕ

Для обеспечения горизонтального масштабирования запросы/данные должны распределяться между серверами эффективно и равномерно. Для этого зачастую используется согласованное хеширование. Давайте сначала подробно поговорим о самой проблеме.

## ПРОБЛЕМА ПОВТОРНОГО ХЕШИРОВАНИЯ

Если у вас есть  $n$  кэширующих серверов, балансирование нагрузки обычно обеспечивается с помощью следующего метода хеширования:

$$\text{serverIndex} = \text{hash}(\text{key}) \% N,$$

где  $N$  – размер пула серверов.

Рассмотрим пример того, как это работает. В табл. 5.1 перечислено 4 сервера и 8 строковых ключей вместе с хешами.

**Таблица 5.1**

Ключ	Хеш	Хеш % 4
ключ 0	18358617	1
ключ 1	26143584	0
ключ 2	18131146	2
ключ 3	35863496	0
ключ 4	34085809	1
ключ 5	27581703	3
ключ 6	38164978	2
ключ 7	22530351	3

Чтобы получить сервер, на котором хранится ключ, мы выполняем операцию взятия остатка  $f(key) \% 4$ . Например,  $\text{hash}(\text{key}0) \% 4 = 1$  означает, что для получения закэшированных данных клиент должен обратиться к серверу 1. На рис. 5.1 показано распределение ключей на основе табл. 5.1.

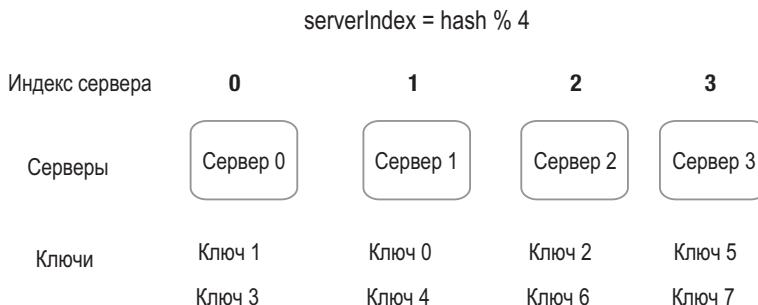


Рис. 5.1

Такой подход работает хорошо, когда пул серверов имеет фиксированный размер, а данные распределены равномерно. Но при добавлении новых или удалении существующих серверов возникают проблемы. Например, если сервер 1 выйдет из строя, размер пула станет равным 3. Используя имеющуюся функцию хеширования, мы получим то же значение хеша ключа. Но при взятии остатка индексы серверов изменятся, так как их количество уменьшилось на 1. Применение хеша  $\% 3$  даст результаты, показанные в табл. 5.2.

Таблица 5.2

Ключ	Хеш	Хеш $\% 3$
ключ 0	18358617	0
ключ 1	26143584	0
ключ 2	18131146	1
ключ 3	35863496	2
ключ 4	34085809	1
ключ 5	27581703	0
ключ 6	38164978	1
ключ 7	22530351	0

На рис. 5.2 показано распределение ключей на основе табл. 5.2.

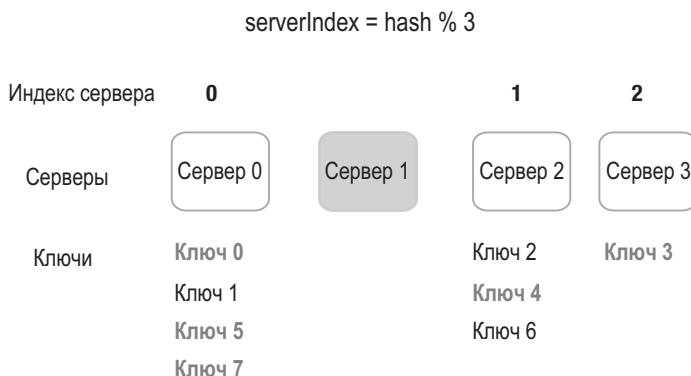


Рис. 5.2

Как показано на рис. 5.2, большая часть ключей была распределена заново: изменение коснулось не только тех ключей, которые хранились на вышедшем из строя сервере (сервер 1). Это означает, что при выпадении из пула сервера 1 большинство клиентов начнут извлекать закэшированные данные не из тех серверов. Это приводит к целой лавине кэш-промахов. Согласованное хеширование — эффективный метод борьбы с этой проблемой.

## СОГЛАСОВАННОЕ ХЕШИРОВАНИЕ

Цитата из Википедии: «Согласованное хеширование (англ. consistent hashing) — особый вид хеширования, отличающийся тем, что когда хеш-таблица перестраивается, только  $K/n$  ключей в среднем должны быть переназначены, где  $K$  — число ключей и  $n$  — число слотов. В противоположность этому, в большинстве традиционных хеш-таблиц изменение количества слотов вызывает переназначение почти всех ключей» [1].

### Пространство и кольцо хеширования

Итак, мы поняли, что такое согласованное хеширование. Теперь давайте посмотрим, как оно работает. Предположим, что в качестве хеш-функции  $f$

используется SHA-1, а ее выходной диапазон имеет вид  $x0, x1, x2, x3, \dots, xn$ . В криптографии пространство хеширования SHA-1 находится между 0 и  $2^{160} - 1$ . Это означает, что  $x0$  соответствует 0,  $xn$  соответствует  $2^{160} - 1$ , а все остальные промежуточные значения находятся между 0 и  $2^{160} - 1$ . Это пространство хеширования показано на рис. 5.3.



Рис. 5.3

Соединив оба конца, как показано на рис. 5.4, мы получим кольцо хеширования.

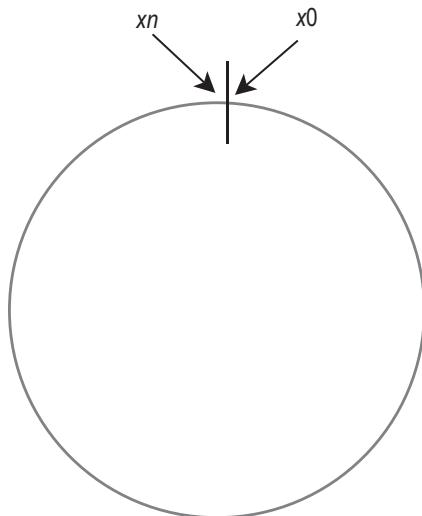


Рис. 5.4

## Хеш-серверы

Используя ту же хеш-функцию  $f$ , мы наносим серверы на кольцо с учетом их IP-адресов или имен.

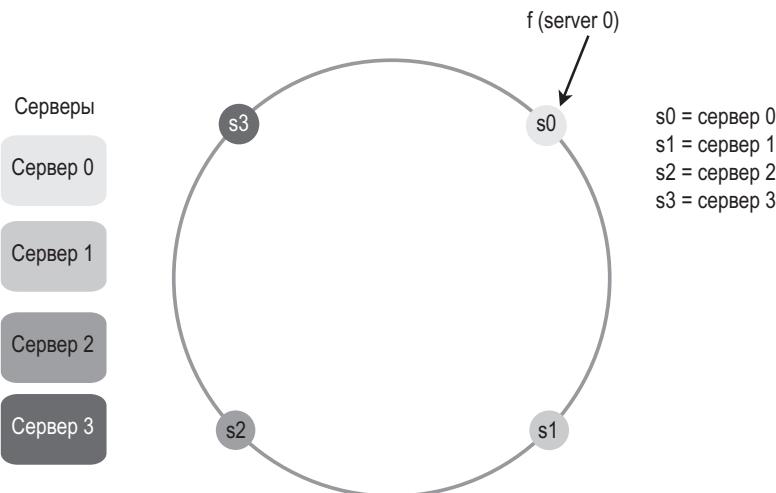


Рис. 5.5

### Хеш-ключи

Стоит упомянуть, что эта хеш-функция отличается от той, которая использовалась в «проблеме повторного хеширования», и что здесь нет операции взятия остатка. Как видно на рис. 5.6, на кольце хеширования нанесено 4 ключа (ключ 0, ключ 1, ключ 2 и ключ 3).

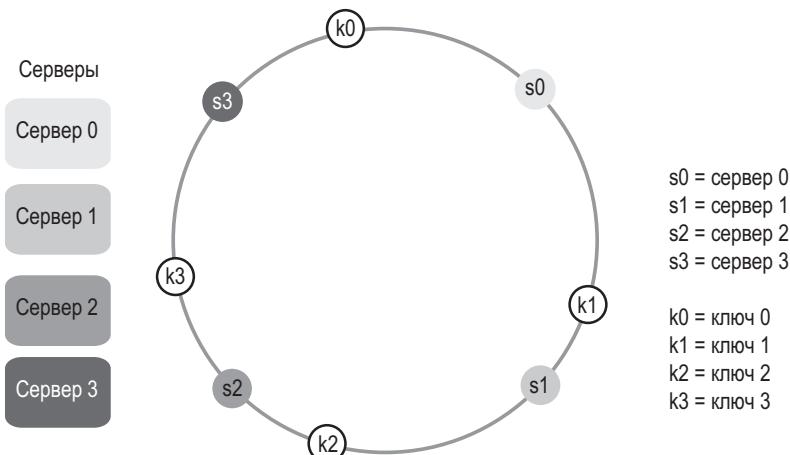


Рис. 5.6

## Поиск серверов

Чтобы определить, на каком сервере хранится ключ, мы идем по часовой стрелке, начиная с позиции ключа на кольце, пока не найдем сервер. Этот процесс показан на рис. 5.7. Если двигаться по часовой стрелке, ключ 0 находится на сервере 0, ключ 1 находится на сервере 1, ключ 2 находится на сервере 2, а ключ 3 находится на сервере 3.

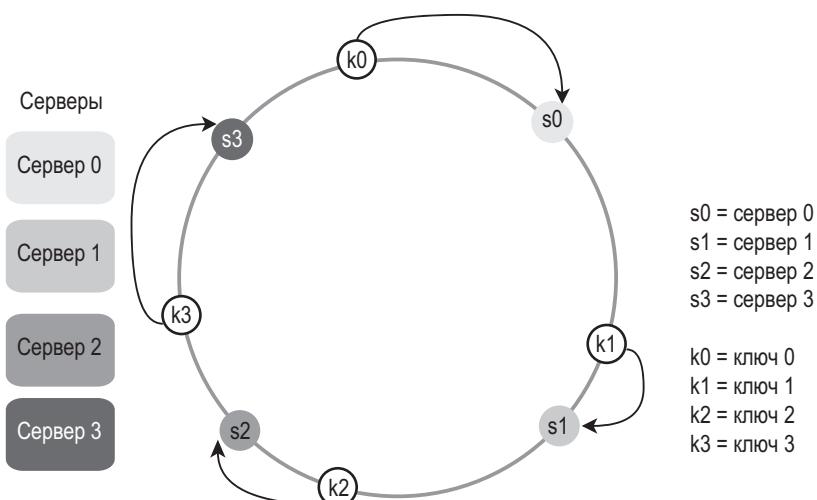


Рис. 5.7

## Добавление сервера

Исходя из логики, описанной выше, добавление нового сервера потребует перераспределения лишь небольшой части ключей.

Как видно на рис. 5.8, после добавления сервера 4 перераспределению подлежит только ключ 0. Ключи 1–3 остаются на тех же серверах.

Давайте подробнее рассмотрим эту логику. Пока не появилсяся сервер 4, ключ 0 находился на сервере 0. Теперь же он будет храниться на сервере 4, так как именно он встречается первым при прохождении по часовой стрелке от позиции ключа 0 на кольце. В соответствии с алгоритмом согласованного хеширования, остальные ключи не перераспределяются.

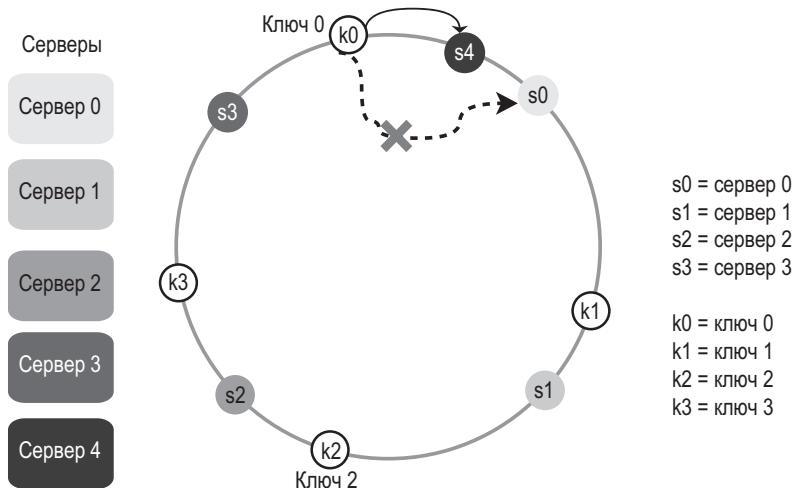


Рис. 5.8

### Удаление сервера

При использовании согласованного хеширования удаление сервера потребует перераспределения лишь небольшой части ключей. Как видно на рис. 5.9, при удалении сервера 1 необходимо перенести только ключ 1 на сервер 2. Остальные ключи остаются на месте.

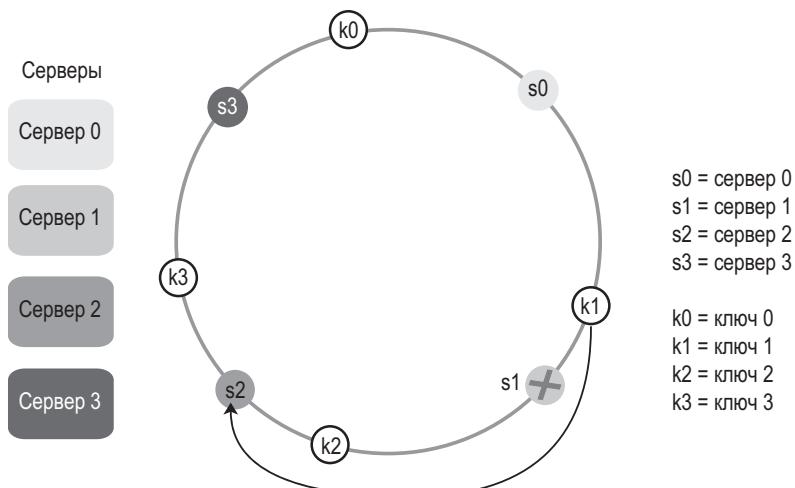


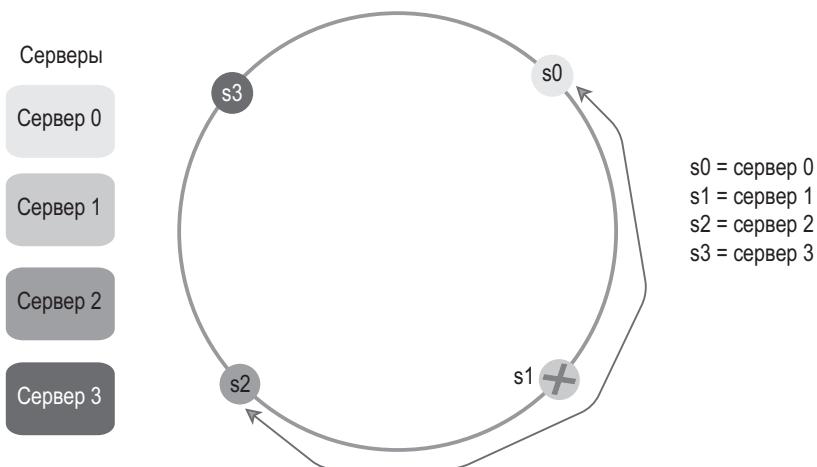
Рис. 5.9

## Две проблемы базового подхода

Алгоритм согласованного хеширования был представлен Каргером и др. в МИТ (Массачусетском технологическом институте) [1]. Он состоит из двух основных этапов:

- серверы и ключи наносятся на кольцо с использованием равномерно распределенной хеш-функции;
- чтобы определить, какому серверу принадлежит ключ, нужно пройти по часовой стрелке от позиции ключа к ближайшему серверу на кольце.

У этого подхода есть две проблемы. Первая: учитывая, что серверы могут добавляться и удаляться, их отрезки на кольце не могут иметь фиксированный размер. Отрезок — это пространство хеширования между двумя соседними серверами. Размер отрезков, назначаемых каждому серверу, может оказаться как очень маленьким, так и достаточно большим. Как видно на рис. 5.10, в случае удаления  $s_1$  отрезок  $s_3$  (выделенный двунаправленными стрелками) станет в два раза больше, чем отрезки  $s_0$  и  $s_3$ .



Вторая проблема состоит в том, что распределение ключей на кольце может быть неравномерным. Например, если серверы имеют позиции,

как на рис. 5.11, большинство ключей окажутся на сервере 2, а серверы 1 и 3 будут пустовать.

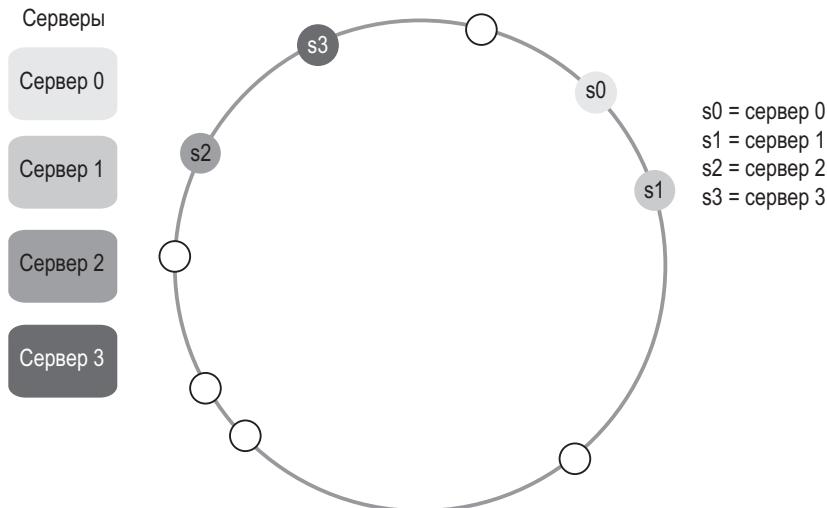


Рис. 5.11

Для решения этих проблем используется методика, известная как виртуальные узлы или реплики.

### Виртуальные узлы

Виртуальный узел ссылается на настоящий; каждый сервер представлен на кольце несколькими виртуальными узлами. Как показано на рис. 5.12, у сервера 0 и сервера 1 есть по три виртуальных узла. Число 3 выбрано произвольно; в реальных системах виртуальных узлов значительно больше. Теперь сервер 0 представлен на кольце не как  $s0$ , а как  $s0_0, s0_1$  и  $s0_2$ . Точно так же сервер 1 имеет на кольце обозначения  $s1_0, s1_1$  и  $s1_2$ . Благодаря виртуальным узлам каждый сервер отвечает сразу за несколько отрезков. Отрезки (грани) с меткой  $s0$  принадлежат серверу 0, а отрезки с меткой  $s1$  — серверу 1.

Чтобы узнать, на каком сервере хранится ключ, мы переходим в его позицию на кольце и двигаемся по часовой стрелке к ближайшему виртуальному узлу. Как показано на рис. 5.13, чтобы определить сервер, на

котором находится  $k_0$ , мы двигаемся по часовой стрелке от его позиции к виртуальному узлу  $s1\_1$ , который ссылается на сервер 1.

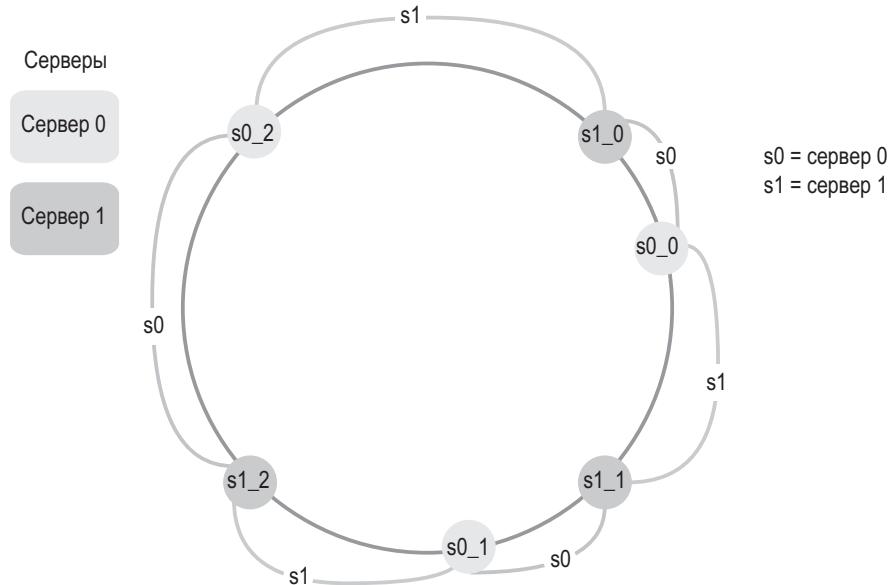


Рис. 5.12

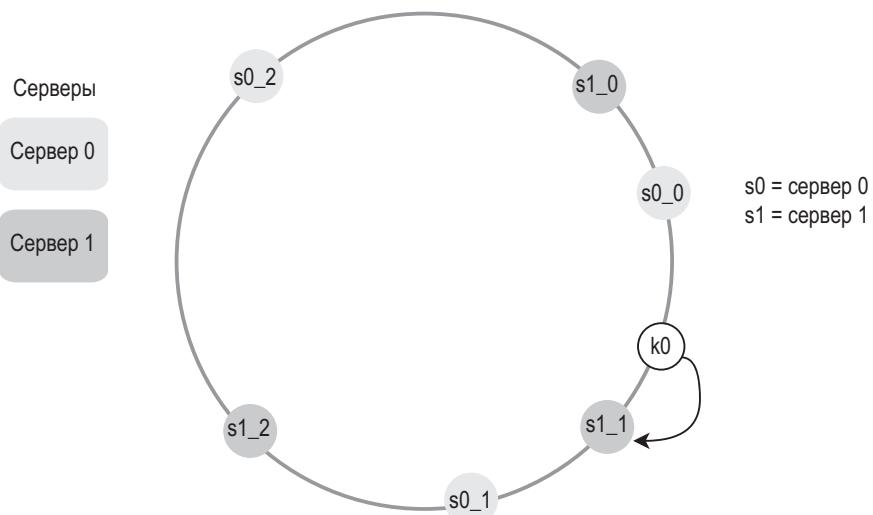


Рис. 5.13

Чем больше виртуальных узлов, тем равномернее становится распределение ключей. Это вызвано уменьшением стандартного отклонения, благодаря которому данные распределяются более сбалансированно. Стандартное отклонение определяет, как распределены данные. Результаты эксперимента, проведенного в ходе онлайн-исследования [2], показывают, что стандартное отклонение от среднего составляет 5 % и 10 % для 200 и, соответственно, 100 виртуальных узлов. Чем больше виртуальных узлов, тем меньше отклонение. Но при этом нужно больше места для хранения данных о виртуальных узлах. Мы можем подобрать такое количество, которое лучше всего соответствует требованиям нашей системы.

### Поиск затронутых ключей

При добавлении или удалении сервера часть данных нужно перераспределить. Как определить диапазон затронутых ключей?

На рис. 5.14 на кольце наносится сервер 4. Затронутый диапазон начинается с  $s_4$  (добавленного узла) и идет по кольцу против часовой стрелки до ближайшего сервера ( $s_3$ ). Таким образом, ключи, размещенные между  $s_3$  и  $s_4$ , необходимо перенести на  $s_4$ .

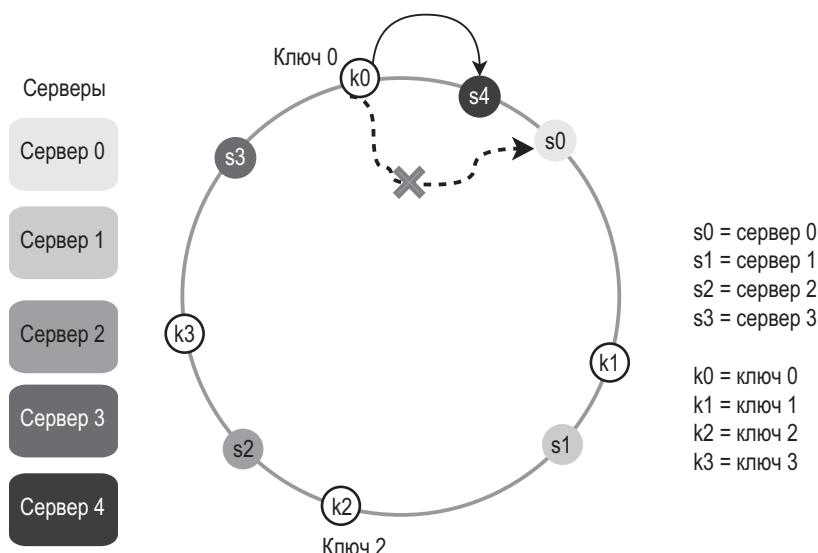


Рис. 5.14

Когда сервер ( $s_1$ ) удаляется (как показано на рис. 5.15), затронутый диапазон начинается с  $s_1$  (удаленного узла) и идет по кольцу против часовой стрелки до ближайшего сервера ( $s_0$ ). Таким образом, ключи, размещенные между  $s_0$  и  $s_1$ , необходимо перенести на  $s_2$ .

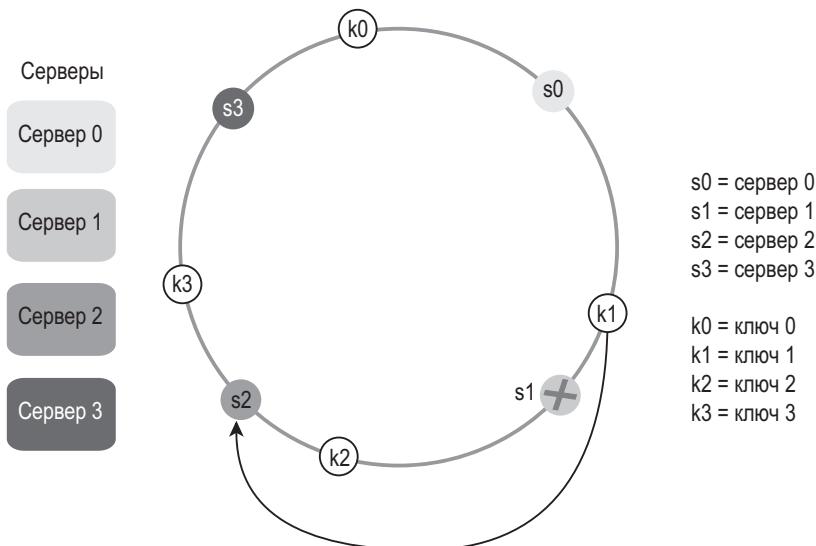


Рис. 5.15

## ИТОГИ

В этой главе мы подробно обсудили согласованное хеширование и объяснили, для чего оно нужно и как оно работает. Этот подход имеет следующие преимущества.

- При добавлении или удалении серверов перераспределяется минимальное количество ключей.
- Его легко горизонтально масштабировать, так как данные распределены более равномерно.
- Минимизация проблемы «горячих» ключей. Чрезмерный доступ к какому-то определенному сегменту может привести к перегрузке сервера. Представьте, что информация о Кэтти Перри, Джастине

Бибере и Леди Гаге очутилась в одном и том же сегменте. Согласованное хеширование помогает бороться с этой проблемой за счет более равномерного распределения данных.

Согласованное хеширование широко применяется в реальных системах, среди которых можно выделить следующие:

- компонент секционирования данных в БД Dynamo от Amazon [3];
- разбиение данных по кластеру в Apache Cassandra [4];
- система обмена сообщениями Discord [5];
- сеть доставки содержимого Akamai [6];
- сетевой балансировщик нагрузки Maglev [7].

Поздравляем, вы проделали длинный путь и можете гордиться собой. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Согласованное хеширование: [https://ru.wikipedia.org/wiki/Согласованное\\_хеширование](https://ru.wikipedia.org/wiki/Согласованное_хеширование)
- [2] Consistent Hashing: <https://tom-e-white.com/2007/11/consistent-hashing.html>
- [3] Dynamo: Amazon's Highly Available Key-value Store: <https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
- [4] Cassandra - A Decentralized Structured Storage System: <http://www.cs.cornell.edu/Projects/ladis2009/papers/Lakshman-ladis2009.PDF>
- [5] How Discord Scaled Elixir to 5,000,000 Concurrent Users: <https://blog.discord.com/scaling-elixir-f9b8e1e7c29b>
- [6] CS168: The Modern Algorithmic Toolbox Lecture #1: Introduction and Consistent Hashing: <http://theory.stanford.edu/~tim/s16/l/l1.pdf>
- [7] Maglev: A Fast and Reliable Software Network Load Balancer: <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44824.pdf>

---

# 6

---

## ПРОЕКТИРОВАНИЕ ХРАНИЛИЩА ТИПА «КЛЮЧ–ЗНАЧЕНИЕ»

Хранилище типа «ключ–значение» — это нереляционная база данных. Каждый уникальный идентификатор хранится в виде ключа и имеет отдельное значение.

В паре «ключ–значение» ключ должен быть уникальным, а соответствующее значение может быть получено через ключ. Ключами могут выступать как обычный текст, так и хешированные значения. Короткие ключи обеспечивают лучшую производительность. Как они выглядят? Вот несколько примеров:

- ключ в виде обычного текста: last\_logged\_in\_at;
- хешированный ключ: 253DDEC4.

Значение в паре может быть строкой, списком, объектом и т. д. В таких хранилищах, как Amazon Dynamo [1], Memcached [2], Redis [3] и т. д., значения обычно считают непрозрачными объектами.

Вот фрагмент данных из хранилища типа «ключ–значение»:

**Таблица 6.1**

ключ	значение
145	john
147	bob
160	julia

В этой главе вам предложено спроектировать хранилище типа «ключ–значение» с поддержкой следующих операций:

- `put(ключ, значение)` // вставить «значение», связанное с «ключом»;
- `get(ключ)` // получить «значение», связанное с «ключом».

## ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Идеальной архитектуры не существует. Каждое решение подразумевает определенные компромиссы между чтением, записью и расходом памяти. Двумя другими факторами, между которыми нужно найти баланс, являются согласованность и доступность. В этой главе мы спроектируем хранилище типа «ключ–значение», обладающее следующими характеристиками:

- малый размер пар «ключ–значение»: меньше 10 Кб;
- возможность хранения объемных данных;
- высокая доступность — система должна быстро отвечать даже во время сбоев;
- хорошая масштабируемость — система должна масштабироваться для поддержки объемных наборов данных;
- автоматическое масштабирование — серверы должны автоматически добавляться/удаляться в зависимости от трафика;
- регулируемая согласованность;
- низкая латентность.

## ХРАНИЛИЩЕ ТИПА «КЛЮЧ–ЗНАЧЕНИЕ» НА ОДНОМ СЕРВЕРЕ

Разработать хранилище типа «ключ–значение», находящееся в пределах одного сервера, довольно просто. Очевидное решение заключается в хранении пар в хеш-таблице, содержимое которой находится в памяти. Достать данные оттуда довольно легко, но их объем ограничен из-за лимита памяти. С этой проблемой можно бороться двумя путями:

- сжимать данные;
- хранить в памяти только часто используемые данные, а все остальное записывать на диск.

Но даже с этими мерами отдельно взятый сервер может очень быстро исчерпать свои ресурсы. Для поддержки крупных данных хранилище типа «ключ–значение» должно быть распределенным.

## РАСПРЕДЕЛЕННОЕ ХРАНИЛИЩЕ ТИПА «КЛЮЧ–ЗНАЧЕНИЕ»

Распределенное хранилище типа «ключ–значение» иногда называют распределенной хеш-таблицей. Оно распределяет пары «ключ–значение» между множеством серверов. При проектировании распределенной системы необходимо понимать теорему CAP (Consistency, Availability, Partition Tolerance – «согласованность, доступность, устойчивость к секционированию»).

### Теорема CAP

Теорема CAP гласит, что распределенная система может обеспечивать не больше двух из следующих трех свойств: согласованность, доступность и устойчивость к секционированию. Дадим несколько определений.

- **Согласованность.** Означает, что все клиенты одновременно видят одни и те же данные, к какому бы узлу они ни подключились.
- **Доступность.** Означает, что любой клиент, запрашивающий данные, получает ответ, даже если некоторые из узлов недоступны.
- **Устойчивость к секционированию.** Секционирование свидетельствует о нарушении связи между двумя узлами. Устойчивость к секционированию означает, что система продолжает работать вопреки нарушению связи в сети.

Согласно теореме CAP, одним из этих свойств необходимо пожертвовать, чтобы обеспечить поддержку двух других (рис. 6.1).

Сейчас хранилища типа «ключ–значение» классифицируются в зависимости от того, какие две характеристики САР они поддерживают:

- **Системы СР (согласованность и устойчивость к секционированию).** Жертвуют доступностью.
- **Системы АР (доступность и устойчивость к секционированию).** Жертвуют согласованностью.
- **Системы СА (согласованность и доступность).** Жертвуют устойчивостью к секционированию. Поскольку сетевые сбои неизбежны, распределенные системы должны справляться с разделением сети. В связи с этим системы СА не существуют в реальных условиях.



Рис. 6.1

Все это лишь определения. Чтобы вам было легче понять, о чем идет речь, рассмотрим несколько конкретных примеров. В распределенных системах данные обычно реплицируются больше одного раза. Предположим, что у нас есть три узла-реплики: n1, n2 и n3, как показано на рис. 6.2.

### Идеальная ситуация

В идеальном мире секционирование сети не происходит. Данные, записанные в n1, автоматически реплицируются в n2 и n3. Этим достигается как согласованность, так и доступность.

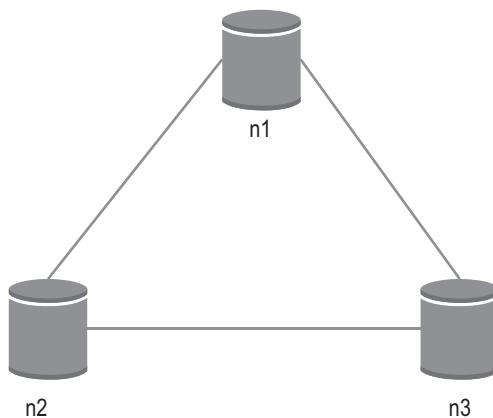


Рис. 6.2

### Реальные распределенные системы

В распределенной системе разделение неизбежно, и, когда оно происходит, мы должны сделать выбор между согласованностью и доступностью. На рис. 6.3 узел n3 отказывает и больше не может взаимодействовать с n1 и n2. Данные, которые клиенты записывают в n1 или n2, не могут дойти до n3. Если же кто-то запишет данные в n3 и они не успеют дойти до n1 и n2, это будет означать, что содержимое n1 и n2 неактуально.

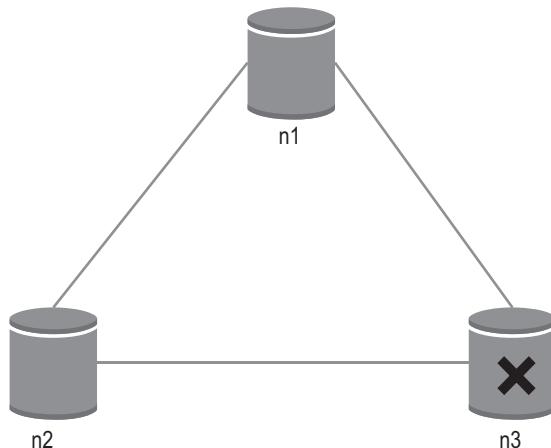


Рис. 6.3

Если мы предпочтем согласованность вместо доступности (система CP), нам придется заблокировать все операции записи на узлах n1 и n2, чтобы избежать рассинхронизации данных между этими тремя серверами. Этим мы сделаем систему недоступной. Чрезвычайно высокие требования к согласованности обычно имеют банковские системы. Например, для банка крайне важно отобразить самую актуальную информацию о балансе клиента. Если из-за разделения сети произойдет рассинхронизация, банковская система начнет возвращать ошибки, пока проблема не будет устранена.

Если же мы отдадим предпочтение доступности перед согласованностью (система AP), система продолжит принимать операции чтения, несмотря на то что она может вернуть устаревшие данные. Что касается операций записи, то они останутся доступными на узлах n1 и n2, а после устранения сетевых неполадок данные будут синхронизированы с n3.

Выбор правильных CAP, которые подходят для вашей задачи, — важный этап создания распределенного хранилища типа «ключ–значение». Вы можете обсудить это со своим интервьюером и спроектировать систему соответствующим образом.

## Компоненты системы

В этом разделе мы обсудим основные компоненты и методики, которые используются для создания хранилища типа «ключ–значение»:

- секционирование данных;
- репликация данных;
- согласованность;
- устранение несогласованности;
- обработка сбоев;
- диаграмма архитектуры системы;
- маршрут записи;
- маршрут чтения.

Следующий материал во многом основан на трех популярных системах хранения данных типа «ключ–значение»: Dynamo [4], Cassandra [5] и BigTable [6].

## Секционирование данных

В крупных приложениях все данные не могут поместиться на одном сервере. Проще всего было бы разделить их на части (секции) меньшего размера и хранить на разных серверах. При секционировании данных необходимо решить две проблемы:

- равномерно распределить их между разными серверами;
- минимизировать их перемещение при добавлении или удалении узлов.

В качестве решения отлично подойдет согласованное хеширование, рассмотренное в главе 5. Давайте вспомним в общих чертах, как оно работает.

- Сначала серверы наносятся на кольцо хеширования. На рис. 6.4 показано кольцо, где есть 8 серверов, обозначенных как  $s_0, s_1, \dots, s_7$ .
- Затем на том же кольце хешируется ключ, который сохраняется на ближайшем сервере по часовой стрелке. Например, по этой логике ключ 0 сохраняется в  $s_1$ .

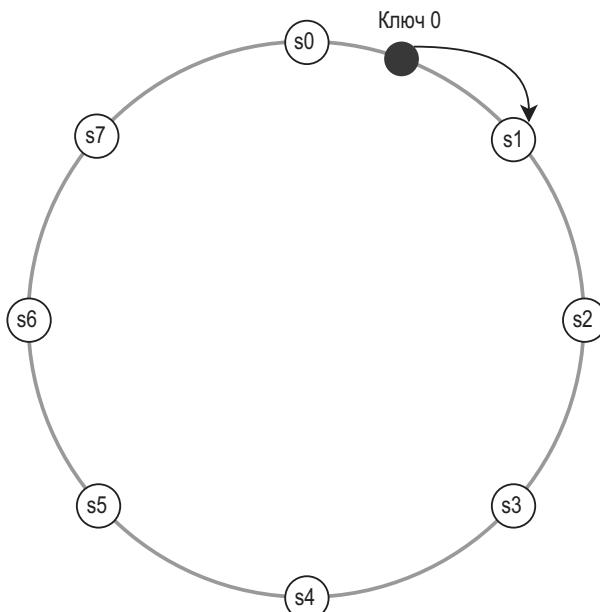


Рис. 6.4

Использование согласованного хеширования для секционирования данных имеет следующие преимущества.

- **Автоматическое масштабирование.** Серверы можно добавлять и удалять автоматически в зависимости от загрузки.
- **Гетерогенность.** Количество виртуальных узлов сервера пропорционально его емкости. Например, серверам с большей емкостью назначают больше виртуальных узлов.

## Репликация данных

Чтобы достичь высокой доступности и надежности, данные должны асинхронно реплицироваться по  $N$  серверам, причем параметр  $N$  можно настраивать. Эти  $N$  серверов выбираются по такому принципу: после нанесения ключа на кольцо хеширования мы двигаемся по часовой стрелке от его позиции и выбираем ближайшие  $N$  серверов для хранения копий данных. На рис. 6.5  $N = 3$  и ключ 0 реплицируется между серверами s1, s2 и s3.

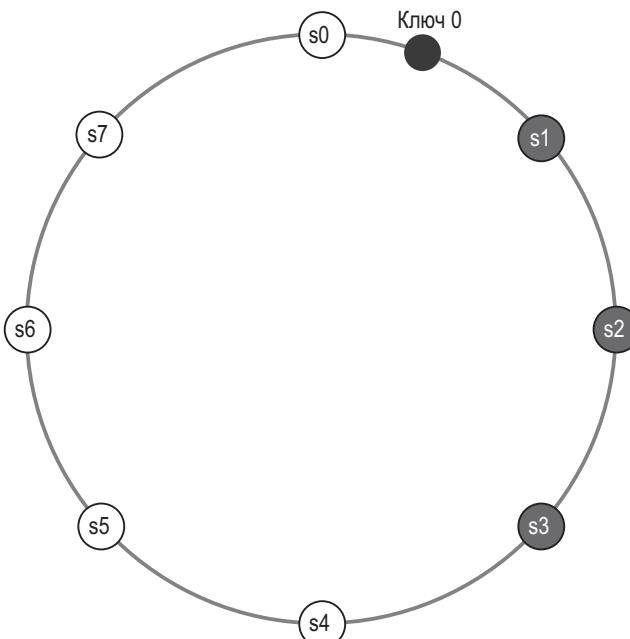


Рис. 6.5

Первые  $N$  виртуальных узлов могут принадлежать физическим серверам, количество которых меньше  $N$ . Чтобы избежать этой проблемы, при прохождении по часовой стрелке выбираются только уникальные серверы.

Узлы, находящиеся в одном центре обработки данных, зачастую выходят из строя одновременно в результате отключения электричества, неполадок в сети, стихийных бедствий и т. д. Для повышения надежности системы реплики размещаются в разных ЦОД, между которыми установлены высокоскоростные сетевые соединения.

### Согласованность

Так как данные реплицируются по нескольким узлам, они должны синхронизироваться между репликами. Консенсус кворума может обеспечить согласованность как чтения, так и записи. Для начала перечислим несколько определений.

- **$N$**  = количество реплик.
- **$W$**  = кворум записи размера  $W$ . Операция записи считается успешной, только если она подтверждена  $W$  репликами.
- **$R$**  = кворум чтения размера  $R$ . Чтобы операцию записи можно было считать успешной, необходимо дождаться ответа как минимум от  $R$  реплик.

Рассмотрим пример на рис. 6.6 с  $N = 3$ .

$W = 1$  не означает, что данные записаны на одном сервере. Например, в конфигурации, представленной на рис. 6.6, данные реплицируются по  $s0, s1$  и  $s2$ . Значение  $W = 1$  говорит о том, что, прежде чем считать операцию записи успешной, координатор должен получить как минимум одно подтверждение. То есть если сервер  $s1$  подтвердит операцию, нам больше не нужно ждать подтверждений от  $s0$  и  $s2$ . Координатор выступает прокси-сервером между клиентом и узлами.

Выбор значений для  $W$ ,  $R$  и  $N$  — это типичный компромисс между латентностью и согласованностью. Если  $W = 1$  или  $R = 1$ , операция завершается быстро, так как координатору нужно ждать ответа только от одной из реплик. Если же  $W$  или  $R$  больше 1, система становится более согласованной, но при этом координатору придется ждать ответа от самой медленной реплики, что замедлит выполнение запросов.

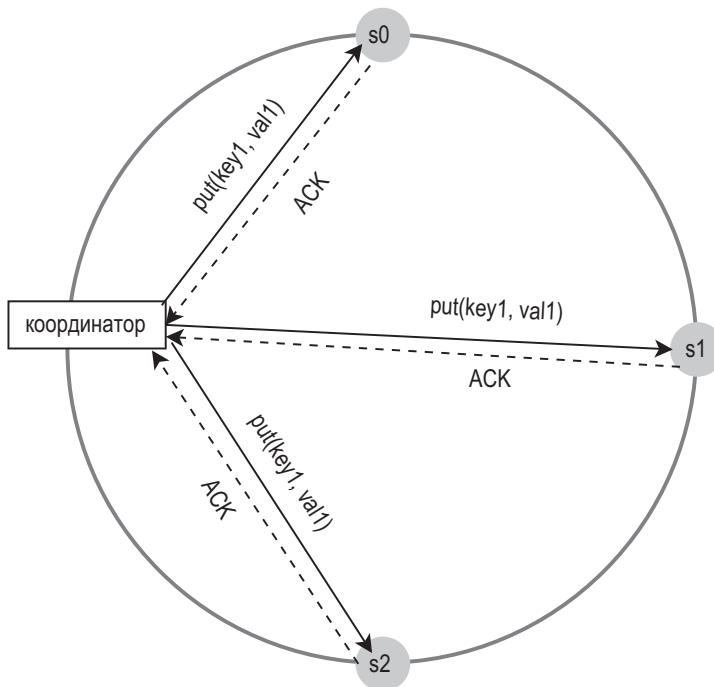


Рис. 6.6 (ACK = квитирование)

$W + R > N$  гарантирует строгую согласованность, поскольку в системе должен быть как минимум один узел с тем же минимальным набором данных.

Как сконфигурировать  $N$ ,  $W$  и  $R$  для наших задач? Вот несколько возможных вариантов:

- если  $R = 1$  и  $W = N$ , система оптимизирована для быстрого чтения;
- если  $W = 1$  и  $R = N$ , система оптимизирована для быстрой записи;
- если  $W + R > N$ , гарантируется строгая согласованность (обычно  $N = 3$ ,  $W = R = 2$ );
- если  $W + R \leq N$ , строгая согласованность не гарантируется.

В зависимости от требований значения  $W$ ,  $R$ ,  $N$  можно оптимизировать для получения нужного уровня согласованности.

## Модели согласованности

Модель согласованности — еще один важный фактор, который следует учитывать при проектировании хранилища типа «ключ–значение». Она определяет степень согласованности данных и имеет широкий спектр разновидностей.

- Строгая согласованность. Любая операция чтения возвращает значение, соответствующее результату самой последней операции записи. Клиент всегда получает актуальные данные.
- Слабая согласованность. Последующие операции чтения могут и не вернуть самое последнее значение.
- Согласованность в конечном счете. Это разновидность слабой согласованности. Рано или поздно все обновления распространяются по системе и все реплики становятся согласованными.

Жесткая согласованность обычно достигается за счет того, что операции чтения/записи принимаются только после подтверждения текущей записи всеми репликами. Это не самый оптимальный подход для высокодоступных систем, так как он может блокировать новые операции. В Dynamo и Cassandra используется отложенная согласованность, и именно эту модель мы рекомендуем для нашего хранилища. Она допускает поступление в систему несогласованных значений, заставляя клиента их прочитать и согласовать. В следующем разделе мы рассмотрим процесс согласования на основе версионирования.

## Устранение несогласованности: версионирование

Репликация обеспечивает высокую доступность, но при этом делает реплики несогласованными. Для решения этой проблемы применяются версионирование и векторные часы. Версионирование — это когда каждое обновление данных приводит к появлению их новой неизменяемой версии. Прежде чем переходить к этой теме, обсудим пример возникновения несогласованности.

Как показано на рис. 6.7, узлы-реплики n1 и n2 имеют одно и то же значение. Назовем его *исходным*. Сервер 1 и сервер 2 получают одно и то же значение при выполнении операции `get("name")`.

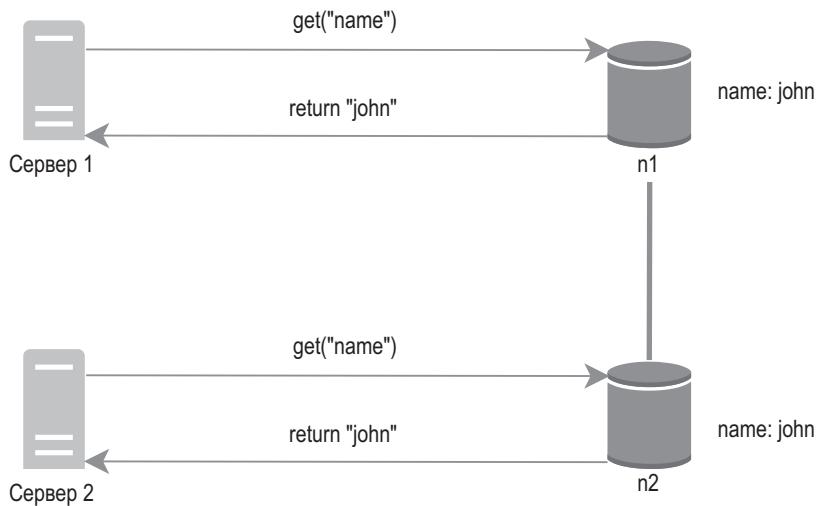


Рис. 6.7

Далее, как видно на рис. 6.8, сервер 1 меняет значение name на `johnSanFrancisco`, а сервер 2 меняет то же значение на `johnNewYork`. Эти два изменения производятся одновременно. Мы получаем два конфликтующих значения, которые называются версиями `v1` и `v2`.

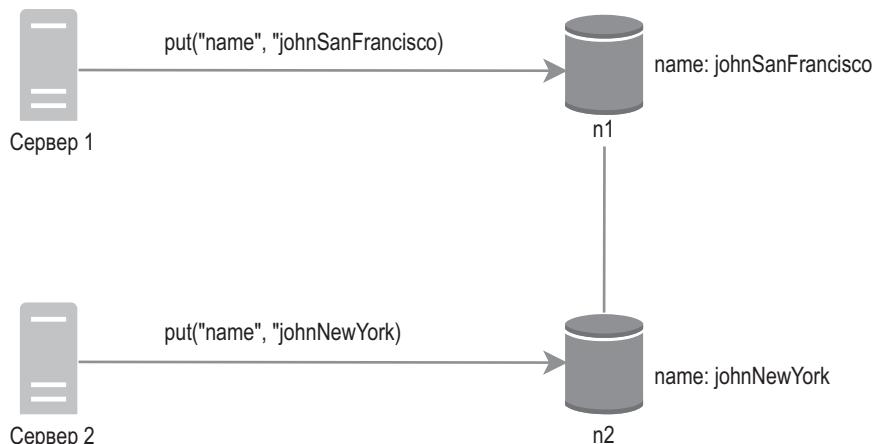


Рис. 6.8

В этом примере исходное значение можно игнорировать, так как на нем были основаны изменения. Однако конфликт между двумя последними версиями нельзя разрешить каким-то очевидным способом. Нам нужна система версионирования, способная обнаруживать и улаживать конфликты. Для решения этой проблемы часто применяют методику, известную как векторные часы. Давайте посмотрим, как она работает.

Векторные часы — это пара [сервер, версия], связанная с элементом данных. С ее помощью можно проверить, какая из двух версий более новая и есть ли между ними конфликт.

Допустим, у нас есть векторные часы вида  $D([S1, v1], [S2, v2], \dots, [Sn, vn])$ , где  $D$  — элемент данных,  $v1$  — номер версии, а  $s1$  — номер сервера. Когда элемент данных  $D$  записывается на сервер  $Si$ , система должна выполнить одно из следующих действий:

- инкрементировать  $vi$ , если  $[Si, vi]$  существует;
- в противном случае создать новую запись  $[Si, vi]$ .

Конкретный пример использования этой абстрактной логики показан на рис. 6.9.

1. Клиент записывает в систему элемент данных D1, и запись обрабатывается сервером Sx, у которого теперь есть векторные часы D1[(Sx, 1)].
2. Другой клиент считывает последнюю версию D1, обновляет ее до D2 и записывает обратно. D2 происходит от элемента D1 и поэтому записывается вместо него. Предполагается, что запись обрабатывается тем же сервером Sx, векторные часы которого теперь выглядят как D2([Sx, 2]).
3. Еще один клиент считывает последнюю версию D2, обновляет ее до D3 и записывает обратно. Предполагается, что запись обрабатывается тем же сервером Sy, векторные часы которого теперь выглядят как D3([Sx, 2], [Sy, 1])).
4. Еще один клиент считывает последнюю версию D2, обновляет ее до D4 и записывает обратно. Предполагается, что запись обрабатывается тем же сервером Sz, векторные часы которого теперь выглядят как D4([Sx, 2], [Sz, 1])).

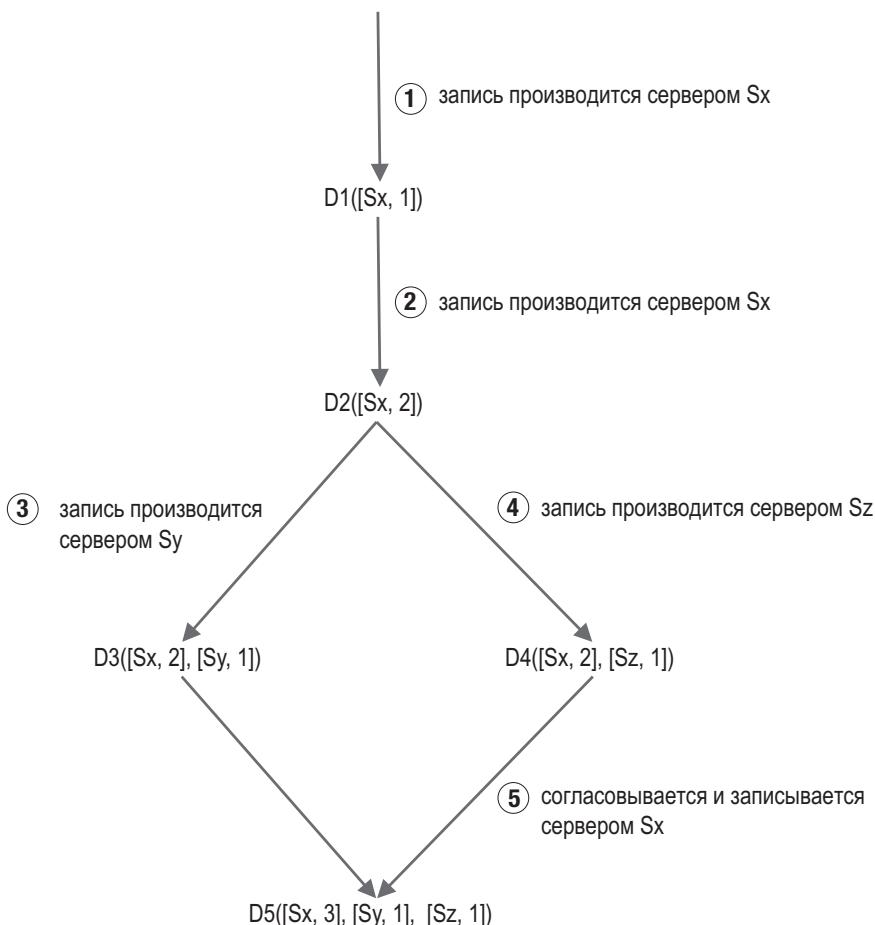


Рис. 6.9

5. Следующий клиент, который считывает D3 и D4, обнаруживает конфликт, вызванный тем, что элемент данных D2 был изменен двумя серверами: Sy и Sz. Клиент разрешает этот конфликт и отправляет на сервер обновленные данные. Предполагается, что запись обрабатывается тем же сервером Sx, который теперь имеет значение D5([Sx, 3], [Sy, 1], [Sz, 1]). О том, как обнаруживать конфликты, мы поговорим чуть позже.

Если номер версии каждого члена векторных часов  $Y$  больше или равен номерам версий в  $X$ , это означает, что  $X$  является наследником  $Y$  и, сле-

довательно, эти версии не конфликтуют. Например, векторные часы  $D([s0, 1], [s1, 1])$  являются предшественником  $D([s0, 1], [s1, 2])$ , поэтому конфликт не записывается.

Точно так же очевидно, что  $X$  и  $Y$  находятся на одном уровне (то есть конфликтуют между собой), если любой член векторных часов  $Y$  имеет версию ниже, чем у соответствующего члена  $X$ . Например, векторные часы  $D([s0, 1], [s1, 2])$  и  $D([s0, 2], [s1, 1])$  сигнализируют о конфликте.

Векторные часы могут разрешать конфликты, но у них есть два заметных недостатка. Во-первых, они усложняют клиент, так как в нем должна быть реализована логика разрешения конфликтов.

Во-вторых, пары [сервер, версия] в векторных часах могут очень быстро накапливаться. Чтобы это исправить, мы можем указать максимальную длину, при превышении которой самая старая пара удаляется. Это может снизить эффективность процесса согласования, так как больше нельзя будет точно сказать, является ли версия потомком. Но, судя по исследованию [4], компания Amazon еще не сталкивалась с этой проблемой в ходе промышленной эксплуатации Dynatrace; следовательно, это решение должно быть приемлемым для большинства организаций.

## Обработка сбоев

В любых крупномасштабных системах сбои являются не только неизбежным, но и довольно распространенным явлением. Их обработка имеет большое значение. В этом разделе мы сначала познакомимся с методами обнаружения сбоев, а затем пройдемся по распространенным стратегиям их обработки.

### Обнаружение сбоев

В распределенной системе о поломке сервера нельзя судить только по сигналам с другого сервера. Обычно для этого требуется подтверждение от двух независимых источников.

На рис. 6.10 показано мультивещание вида «все ко всем». Это простое и понятное решение, но при большом количестве серверов в системе оно становится неэффективным.

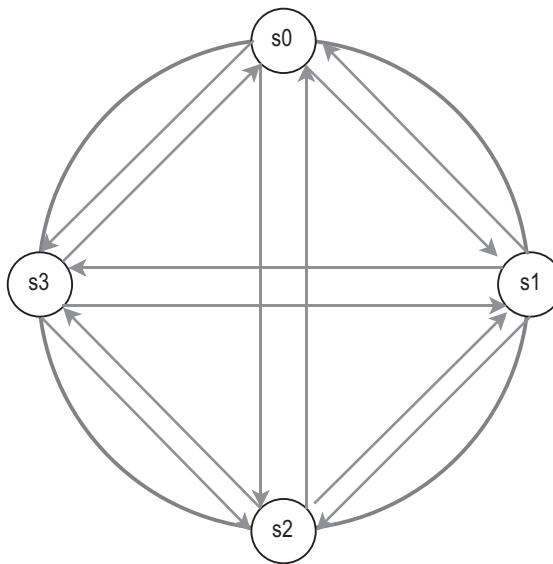


Рис. 6.10

Более оптимальное решение состоит в использовании децентрализованных методов обнаружения сбоев, таких как протокол сплетен (gossip protocol). Он работает следующим образом.

- Каждый узел хранит список узлов-участников, состоящий из идентификаторов и счетчиков пульсации.
- Каждый узел периодически инкрементирует счетчик пульсации.
- Каждый узел периодически шлет пульс группе произвольных узлов, которые в свою очередь передают его другой группе.
- При получении пульса узлы обновляют список участников до последней версии.
- Если счетчик пульсации не увеличивается на протяжении заранее определенного периода, участник считается недоступным.

Как показано на рис. 6.11:

- Узел  $s_0$  хранит список участников, показанных слева.
- Узел  $s_0$  замечает, что счетчик пульсации узла  $s_2$  (ID участника 2) уже давно не увеличивался.

- Узел s0 шлет группе произвольных узлов пульсы с информацией об s2. Когда другие узлы подтверждают, что счетчик пульсации s2 давно не обновлялся, узел s2 будет помечен как недоступный и информация об этом будет передана другим узлам.

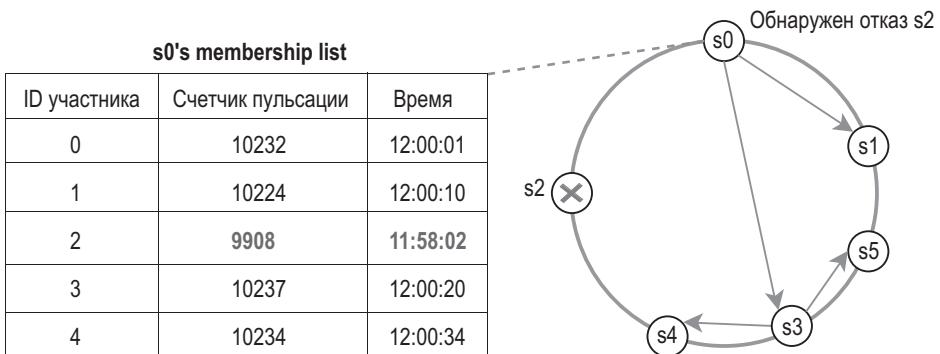


Рис. 6.11

### Обработка временных сбоев

После обнаружения сбоя по протоколу сплетеи система должна задействовать определенные механизмы, чтобы обеспечить доступность. Если используется строгий кворум, операции чтения и записи могут быть заблокированы, как было проиллюстрировано в разделе о консенсусе кворума.

Для повышения доступности используется методика, известная как нестрогий кворум [4]. Вместо обеспечения кворума система выбирает на кольце хеширования первые  $W$  исправных серверов для записи и первые  $R$  исправных серверов для чтения. Недоступные серверы игнорируются.

Если один сервер недоступен из-за сетевых неполадок, вместо него обработкой запросов временно займется другой. Когда сеть возобновит нормальную работу, изменения будут переданы ранее недоступному серверу, чтобы восстановить согласованность данных. Этот процесс называют прозрачной передачей. На рис. 6.12 сервер s2 недоступен, поэтому операции чтения и записи будут временно обрабатываться сервером s3. Когда s2 снова появится в сети, s3 передаст ему измененные данные.

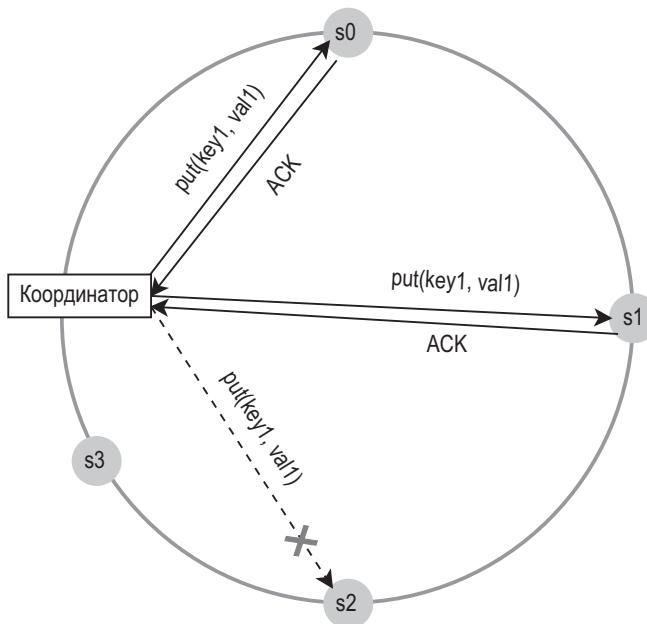


Рис. 6.12

### Обработка бессрочных сбоев

Прозрачная передача используется при временных сбоях. Но что, если реплика пропадает безвозвратно? Чтобы справиться с этой ситуацией, нужно реализовать протокол для предотвращения энтропии, который будет синхронизировать реплики. Это подразумевает сравнение элементов данных, хранящихся на репликах, и обновление каждой реплики до самой новой версии. Для обнаружения несогласованности и минимизации объема передаваемых данных используется дерево Меркля.

Цитата из Википедии [7]: «Хеш-деревом, деревом Меркля (Merkle tree), называют полное двоичное дерево, в листовые вершины которого помещены хеши от блоков данных, а внутренние вершины содержат хеши от сложения значений в дочерних вершинах. Хеш-деревья обеспечивают эффективную и безопасную проверку содержимого крупных структур данных».

Если наши ключи находятся в диапазоне от 1 до 12, построить дерево Меркля можно с помощью нескольких шагов (выделенные ячейки обозначают несогласованность).

Шаг 1: разделить диапазон ключей на бакеты (в нашем примере их 4), как показано на рис. 6.13. Бакет выступает корневым узлом, что позволяет ограничить глубину дерева.

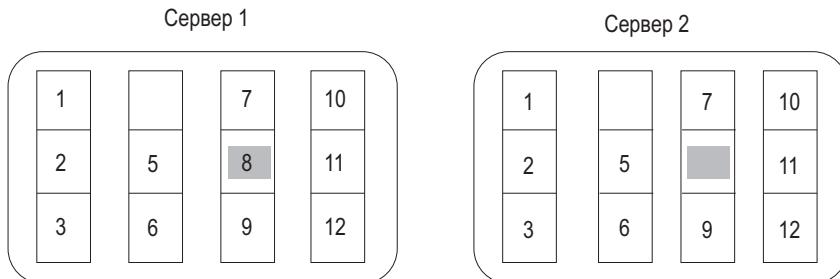


Рис. 6.13

Шаг 2: после создания бакетов захешировать для каждого из них ключ с помощью одного и того же метода хеширования (рис. 6.14).

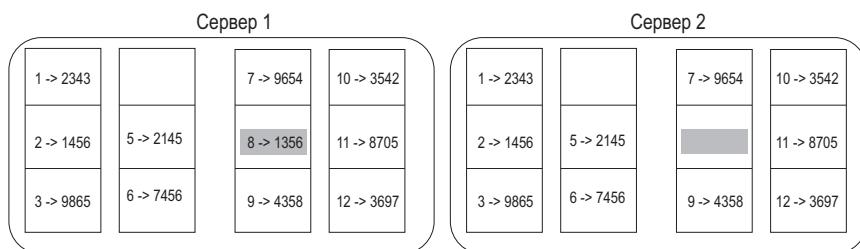


Рис. 6.14

Шаг 3: создать по одному хеш-узлу для каждого бакета (рис. 6.15).

Шаг 4: построить дерево снизу вверх до самого корня путем вычисления дочерних хешей (рис. 6.16).

Сравнение двух деревьев Меркля начинается с их корневых хешей. Если корневые хеши совпадают, серверы содержат одни и те же данные. В противном случае сравниваются дочерние хеши слева направо. В процессе осмотра дерева мы определяем несогласованные бакеты и синхронизируем только их.

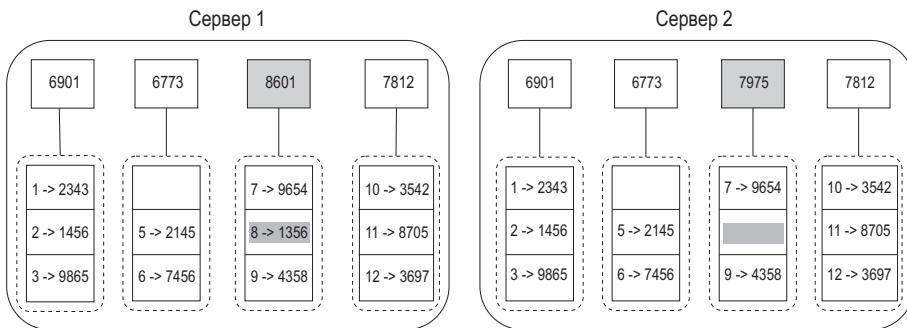


Рис. 6.15

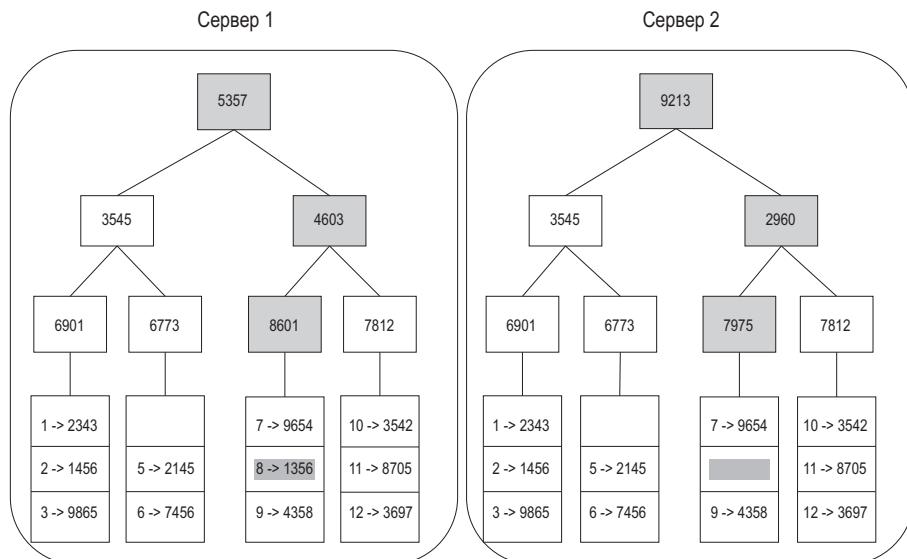


Рис. 6.16

При использовании деревьев Меркля количество данных, которые нужно синхронизировать, прямо пропорционально различиям между двумя репликами и не зависит от того, сколько всего данных в них хранится. В реальных системах бакеты довольно большие. Например, на один миллион бакетов может приходиться по одному миллиарду ключей — то есть 1000 ключей в каждом бакете.

## Обработка неполадок уровня ЦОД

Неполадки уровня ЦОД могут быть вызваны отключениями электричества, разрывами сети, стихийными бедствиями и т. д. Чтобы создать систему, способную с ними справиться, данные необходимо реплицировать по нескольким ЦОД. Даже если один ЦОД станет полностью недоступным, пользователи по-прежнему смогут получить данные из других центров обработки.

## Диаграмма архитектуры системы

Итак, мы обсудили разные технические аспекты проектирования хранилища типа «ключ–значение». Теперь можно перейти к диаграмме архитектуры, показанной на рис. 6.17.

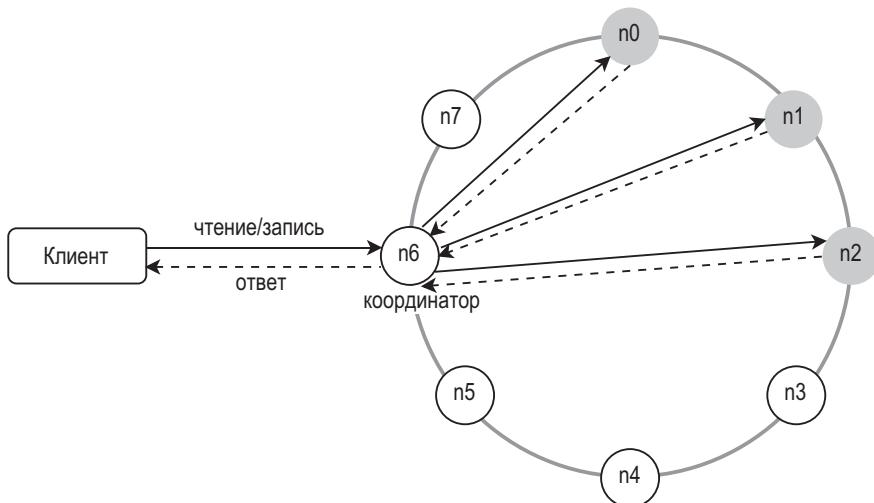


Рис. 6.17

Ниже перечислены основные особенности этой архитектуры.

- Клиенты взаимодействуют с хранилищем типа «ключ–значение» через простые API: `get(ключ)` и `put(ключ, значение)`.
- Координатор — это узел, который выступает прокси-сервером между клиентом и хранилищем.

- Узлы распределяются по кольцу с использованием согласованного хеширования.
- Система полностью децентрализована, поэтому добавление и удаление узлов можно проводить автоматически.
- Данные реплицируются по разным узлам.
- Нет единой точки отказа, так как у каждого узла один и тот же набор обязанностей.

Поскольку архитектура децентрализована, каждый узел выполняет множество задач (рис. 6.18).



Рис. 6.18

### Маршрут записи

На рис. 6.19 показано, что происходит, когда запрос на запись направляется к определенному узлу. Пожалуйста, обратите внимание на то, что предложенная архитектура маршрутов записи/чтения во многом позаимствована у Cassandra [8].

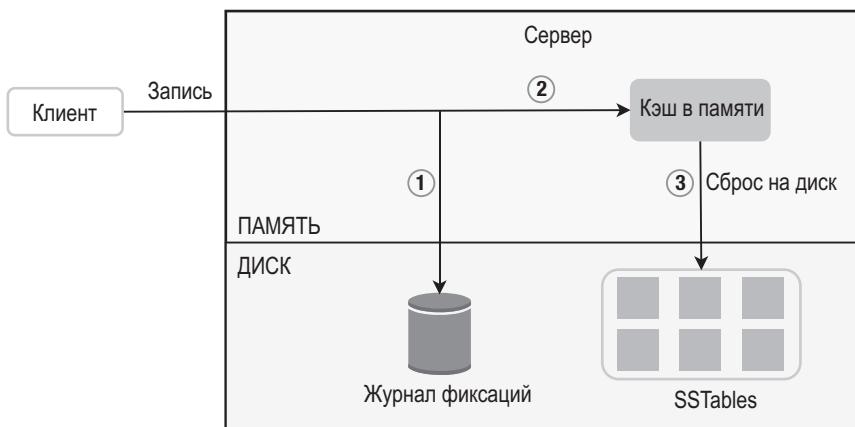


Рис. 6.19

1. Запрос на запись сохраняется в файл с логом коммитов.
2. Данные записываются в кэш, размещенный в памяти.
3. Когда кэш полностью заполняется или достигает определенного лимита, данные сбрасываются на диск в SSTable [9]. *Примечание:* SSTable (sorted-string table – «таблица с сортированием строк») – это упорядоченный список пар <ключ, значение>. Если вы хотите узнать больше об SSTable, обратитесь к справочному материалу [9].

### Маршрут чтения

Когда запрос на чтение направляется к определенному узлу, система сначала проверяет, находятся ли требуемые данные в кэше. Если это так, система возвращает их клиенту, как показано на рис. 6.20.

Если в памяти данных нет, они берутся с диска. Нам нужен эффективный способ поиска таблицы SSTable, в которой содержится ключ. Для этого часто используется фильтр Блума [10].

На рис. 6.21 показан маршрут чтения, когда данных нет в памяти.

1. Сначала система проверяет, есть ли данные в памяти, и если это не так, выполняется второй пункт.

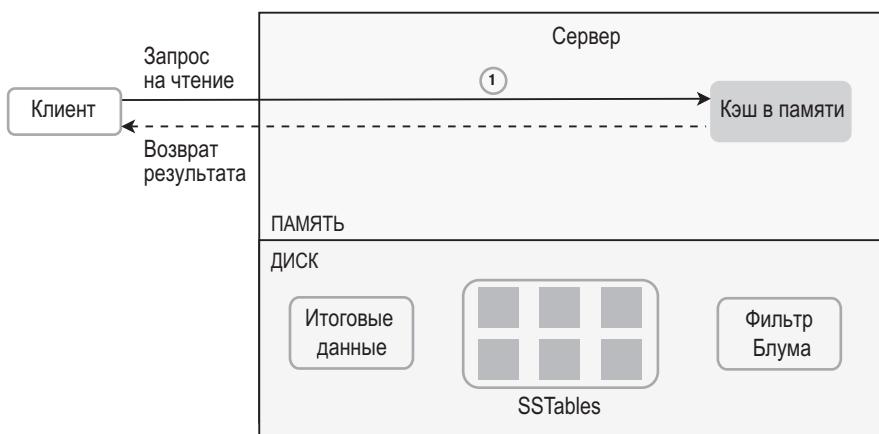


Рис. 6.20

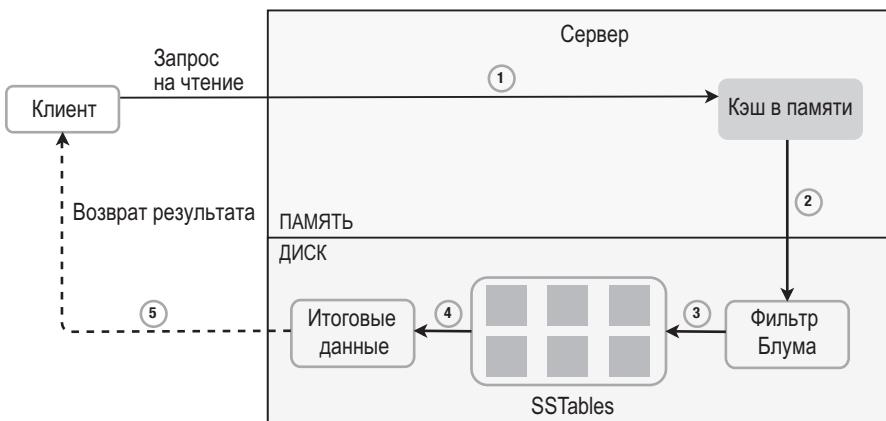


Рис. 6.21

2. Система проверяет фильтр Блума.
3. Фильтр Блума позволяет определить, в какой из таблиц SSTable может находиться ключ.
4. SSTable возвращает итоговый набор данных.
5. Итоговый набор данных возвращается клиенту.

## ИТОГИ

В этой главе было рассмотрено множество концепций и техник. Чтобы вы точно все запомнили, ознакомьтесь с таблицей ниже, где даны характеристики распределенного хранилища типа «ключ–значение» и способы их реализации.

**Таблица 6.2**

Цель/задача	Методика
Возможность хранить объемные данные	Использование согласованного хеширования для распределения нагрузки между серверами
Высокая доступность при чтении	Репликация данных Конфигурация с несколькими ЦОД
Высокая доступность при записи	Версионирование и разрешение конфликтов с использованием векторных часов
Секционирование наборов данных	Согласованное хеширование
Инкрементальная масштабируемость	Согласованное хеширование
Гетерогенность	Согласованное хеширование
Регулируемая согласованность	Консенсус кворума
Обработка временных сбоев	Нестрогий кворум и прозрачная передача
Обработка бессрочных сбоев	Дерево Меркла
Обработка сбоев уровня ЦОД	Репликация между разными ЦОД

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Amazon DynamoDB: <https://aws.amazon.com/dynamodb/>
- [2] memcached: <https://memcached.org/>
- [3] Redis: <https://redis.io/>
- [4] Dynamo: Amazon's Highly Available Key-value Store: <https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
- [5] Cassandra: <https://cassandra.apache.org/>

- [6] Bigtable: A Distributed Storage System for Structured Data: <https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf>
- [7] Дерево Меркля: [https://ru.wikipedia.org/wiki/Дерево\\_хешей](https://ru.wikipedia.org/wiki/Дерево_хешей)
- [8] Cassandra architecture: <https://cassandra.apache.org/doc/latest/architecture/>
- [9] SSTable: <https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveledb/>
- [10] Фильтр Блума: [https://ru.wikipedia.org/wiki/Фильтр\\_Блума](https://ru.wikipedia.org/wiki/Фильтр_Блума)

7

---

# ПРОЕКТИРОВАНИЕ ГЕНЕРАТОРА УНИКАЛЬНЫХ ИДЕНТИФИКАТОРОВ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

В этой главе вам предлагается спроектировать генератор уникальных идентификаторов в распределенной системе. Возможно, использование первичного ключа с атрибутом `auto_increment` в традиционной базе данных станет первым, что придет вам в голову. Однако этот подход не работает в распределенных окружениях, так как отдельно взятый сервер БД слишком мал. К тому же генерировать уникальные ID в рамках нескольких БД с минимальными задержками не так уж просто.

Вот несколько примеров уникальных идентификаторов:

user_id
1227238262110117894
1241107244890099715
1243643959492173824
1247686501489692673
1567981766075453440

Рис. 7.1

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Постановка уточняющих вопросов — это первый шаг к решению задачи на любом интервью по проектированию ИТ-систем. Вот пример диалога между кандидатом и интервьюером:

**Кандидат:** «Какие характеристики должны быть у уникальных ID?»

**Интервьюер:** «ID должны быть уникальными и подлежать сортировке».

**Кандидат:** «Инкрементируется ли ID на 1 при добавлении каждой новой записи?»

**Интервьюер:** «ID инкрементируется по времени, но необязательно на 1. Идентификаторы, созданные вечером, больше тех, которые были получены утром того же дня».

**Кандидат:** «ID имеют только числовые значения?»

**Интервьюер:** «Да, именно так».

**Кандидат:** «Какие требования к длине ID?»

**Интервьюер:** «ID должны умещаться в 64 бита».

**Кандидат:** «Какой масштаб системы?»

**Интервьюер:** «Система должна быть способна генерировать 10 000 ID в секунду».

Это лишь некоторые из вопросов, которые можно задать интервьюеру. Необходимо разобраться в требованиях и прояснить непонятные моменты. В этом примере к системе предъявляются следующие требования:

- ID должны быть уникальными;
- ID должны быть сугубо числовыми;
- ID должны умещаться в 64 бита;
- ID должны быть упорядочены по дате;
- система должна генерировать 10 000 ID в секунду.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Есть разные методы генерации уникальных ID в распределенных системах. Мы рассмотрели следующие варианты:

- репликация с несколькими источниками;
- универсальный уникальный идентификатор (universally unique identifier, UUID);
- сервер тиков;
- Twitter snowflake ID — подход «снежного кома» Twitter.

Давайте обсудим каждый из них. Рассмотрим принципы их работы, а также их слабые и сильные стороны.

### Репликация с несколькими источниками

На рис. 7.2 показан первый подход — репликация с несколькими источниками.

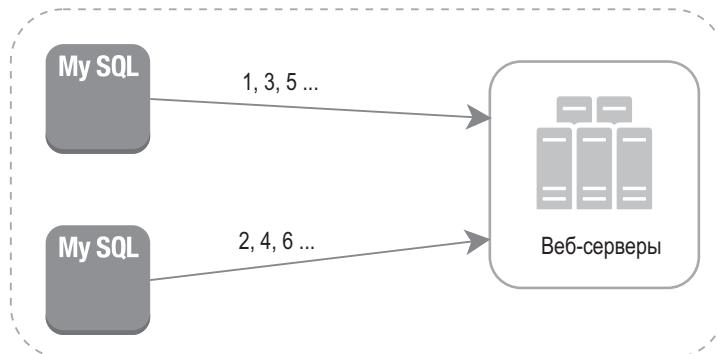


Рис. 7.2

Здесь используется свойство баз данных `auto_increment`. ID увеличивается не на 1, а на число  $k$ , равное количеству используемых БД. Как проиллюстрировано на рис. 7.2, следующий ID равен предыдущему, сгенерированному на том же сервере, плюс 2. Это позволяет избежать определенных проблем с масштабированием, так как идентификаторы

могут увеличиваться вместе с количеством серверов БД. Однако у этой стратегии есть серьезные недостатки:

- сложность масштабирования в конфигурации с несколькими центрами обработки данных;
- ID не увеличиваются хронологически в пределах нескольких серверов;
- плохая масштабируемость при добавлении или удалении сервера.

## UUID

Это еще один простой способ получения уникальных идентификаторов. UUID – это 128-битное число, которое используется для идентификации данных в компьютерных системах. UUID имеют очень низкую вероятность повторения. Цитата из Википедии: «После генерации 1 миллиарда UUID в секунду на протяжении примерно 100 лет вероятность получения одного дубликата достигает 50 %» [1].

UUID может выглядеть как 09c93e62-50b4-468d-bf8a-c07e1040fb2. Эти идентификаторы можно генерировать независимо друг от друга без координации между серверами. Архитектура UUID представлена на рис. 7.3.

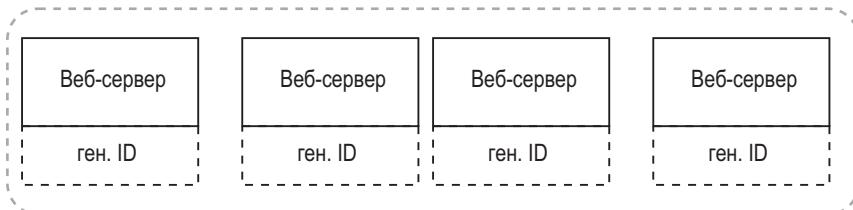


Рис. 7.3

В этой конфигурации каждый веб-сервер содержит генератор идентификаторов, работающий независимо от остальных серверов.

Преимущества:

- простота генерации. Не нужно никакой координации между серверами, что исключает проблемы с синхронизацией;

- систему легко масштабировать, так как каждый сервер отвечает за генерацию идентификаторов, которые он потребляет. Генератор может легко масштабироваться вместе с веб-серверами.

Недостатки:

- ID имеют длину 128 бит, а нам нужно 64 бита;
- ID не увеличиваются со временем;
- ID могут быть нечисловыми.

### Сервер тиков

Серверы тиков<sup>1</sup> — это еще один интересный способ генерации уникальных ID. Их разработала компания Flicker для получения распределенных первичных ключей [2]. Давайте посмотрим, как они работают.

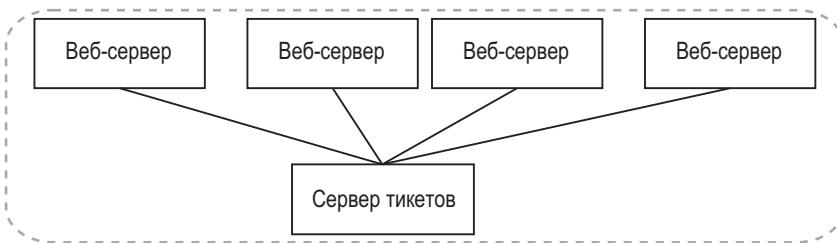


Рис. 7.4

Суть в том, что мы используем функцию `auto_increment` в отдельно взятом сервере баз данных (сервере тиков). Подробнее об этом можно почитать в статье блога разработчиков Flicker [2].

Преимущества:

- числовые ID;
- этот метод легко реализовать, и он подходит для небольших и средних приложений.

<sup>1</sup> В русскоязычных описаниях также встречается вариант «сервер билетов». — Примеч. ред.

Недостатки:

- единая точка отказа. Сервер тиков существует в единственном экземпляре, и если он выйдет из строя, проблемы возникнут у всех систем, которые на него полагаются. Чтобы этого избежать, можно предусмотреть несколько серверов тиков, но это вызовет новые трудности, такие как синхронизация данных.

### Twitter snowflake ID

Методики, упомянутые выше, позволяют получить некоторое представление о работе разных систем генерации ID. Но ни одна из них не отвечает нашим требованиям, поэтому нам нужен другой подход. Интересным вариантом, способным удовлетворить наши нужды, является система генерации уникальных ID от Twitter под названием snowflake.

Разделяй и властвуй — вот что нам нужно. Вместо того чтобы генерировать идентификатор напрямую, мы разделяем его на части. На рис. 7.5 показана структура 64-битного ID.



Рис. 7.5

Каждая часть описана ниже.

- Бит знака: 1 бит. Всегда равен 0 и зарезервирован на будущее. С его помощью потенциально можно различать знаковые и беззнаковые числа.
- Временная метка: 41 бит. Количество миллисекунд, прошедших с начала эпохи Unix или какого-то другого момента. В Twitter snowflake начальной точкой по умолчанию является Ноя 04, 2010, 01:42:54 UTC, что эквивалентно 1288834974657. Мы воспользуемся этим значением.
- ID ЦОД: 5 бит, что дает нам  $2^5 = 32$  центра обработки данных.
- ID компьютера: 5 бит, что дает нам  $2^5 = 32$  компьютера в каждом ЦОД.

- Номер последовательности: 12 бит. При генерации каждого ID на отдельно взятом компьютере или процессе номер последовательности инкрементируется на 1. Каждую миллисекунду этот номер обнуляется.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

При обсуждении общих аспектов проектирования мы перечислили разные способы генерации уникальных ID в распределенных системах и остановились на подходе, основанном на генераторе snowflake ID от Twitter. Давайте подробно рассмотрим эту архитектуру. Чтобы освежить память, еще раз приведем диаграмму snowflake ID.



Рис. 7.6

ID центра обработки данных и компьютера выбираются в момент запуска системы и обычно не меняются во время выполнения. Любые обновления этих идентификаторов требуют тщательного анализа, так как неосторожное внесение изменений может привести к конфликтам. Временные метки и номера последовательностей генерируются в ходе работы системы.

### Временная метка

Самую важную часть идентификатора составляет 41-битная временная метка. Поскольку метки увеличиваются со временем, ID можно сортировать в хронологическом порядке. На рис. 7.7 показан пример преобразования двоичного представления в UTC. Обратное преобразование можно выполнить аналогичным образом.

Максимальная временная метка, которую можно представить с помощью 41 бита, равна  $2^{41} - 1 = 2199023255551$  миллисекундам (мс), что дает нам  $\sim 69$  лет =  $2199023255551$  мс / 1000 / 365 дней / 24 часов / 3600 секунд. Это означает, что этот генератор ID будет работать на протяжении 69 лет, и чем ближе начало эпохи к текущей дате, тем позже

произойдет переполнение. По прошествии этого времени нужно будет установить новую эпоху или внедрить другой метод для миграции идентификаторов.



Рис. 7.7

### Номер последовательности

Номер последовательности занимает 12 бит, что дает нам  $2^{12} = 4096$  комбинаций. Это поле не равно нулю только в случае, если на одном и том же сервере с одну миллисекунду сгенерировано больше одного ID. Теоретически компьютер может генерировать до 4096 новых ID в миллисекунду.

### ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы обсудили разные подходы к проектированию генератора уникальных идентификаторов: репликация с несколькими источниками, UUID, сервер тикетов и генератор вида Twitter snowflake. Мы останови-

лись на последнем варианте, так как он отвечает всем нашим требованиям и способен масштабироваться в распределенном окружении.

Если в конце интервью остается время, можно обсудить дополнительные вопросы.

- Синхронизация часов. В нашей архитектуре предполагается, что у серверов, генерирующих ID, часы синхронизированы. Это может быть не так, если мы используем многоядерный компьютер или конфигурацию с несколькими серверами. В этой книге способы синхронизации часов не рассматриваются; просто знайте, что такая проблема существует. Самым популярным ее решением является протокол NTP (Network Time Protocol — «протокол сетевого времени»). Если вас это заинтересовало, можете обратиться к справочному материалу [4].
- Оптимизация длины отдельных частей. Например, выделение большего числа битов для временной метки за счет уменьшения номера последовательности хорошо подходит для приложений, которые долго работают и имеют низкую степень параллелизма.
- Высокая доступность. Поскольку генератор ID является независимой системой, он должен быть высокодоступным.

Поздравляем, вы проделали длинный путь и можете гордиться собой. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] UUID: <https://ru.wikipedia.org/wiki/UUID>
- [2] Ticket Servers: Distributed Unique Primary Keys on the Cheap: <https://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/>
- [3] Announcing Snowflake: [https://blog.twitter.com/engineering/en\\_us/a/2010/announcing-snowflake.html](https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html)
- [4] Протокол NTP: <https://ru.wikipedia.org/wiki/NTP>

---

# 8

---

## ПРОЕКТИРОВАНИЕ СИСТЕМЫ ДЛЯ СОКРАЩЕНИЯ URL-АДРЕСОВ

В этой главе мы рассмотрим интересную классическую задачу, с которой можно столкнуться на интервью по проектированию ИТ-систем: разработка сервиса для сокращения URL-адресов по примеру tinyurl.

### ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

На интервью по проектированию ИТ-систем специально дают задачи, допускающие свободную трактовку. Чтобы разработать хорошо продуманную систему, необходимо задавать уточняющие вопросы.

**Кандидат:** «Можете ли привести пример работы сервиса для сокращения URL-адресов?»

**Интервьюер:** «Допустим, <https://www.systeminterview.com/q=chatsystem&c=loggedin&v=v3&l=long> — это исходный URL-адрес. Ваш сервис должен создать ссылку покороче: <https://tinyurl.com/y7keocwj>. Клик по этой ссылке должен перенаправлять к исходному URL-адресу».

**Кандидат:** «Какой объем трафика?»

**Интервьюер:** «100 миллионов сгенерированных URL-адресов в день».

**Кандидат:** «Какая длина должна быть у сокращенного URL-адреса?»

**Интервьюер:** «Как можно короче».

**Кандидат:** «Какие символы допускаются в сокращенном URL-адресе?»

**Интервьюер:** «Сокращенный URL-адрес может содержать цифры (0–9) и буквы (a–z, A–Z)».

**Кандидат:** «Допускается ли удаление или обновление сокращенного URL-адреса?»

**Интервьюер:** «Чтобы не усложнять, предположим, что сокращенный URL-адрес не подлежит удалению или обновлению».

Вот типичные сценарии использования:

1. Сокращение URL-адресов: дается длинный URL-адрес => возвращается намного более короткий URL-адрес.
2. Перенаправление URL-адресов: дается сокращенный URL-адрес => пользователь перенаправляется к исходному URL-адресу.
3. Высокая доступность, масштабируемость и устойчивость к сбоям.

### Приближенные оценки

- Операции записи: генерируется 100 миллионов URL-адресов в день.
- Операций записи в секунду:  $100 \text{ миллионов} / 24 / 3600 = 1160$ .
- Операции чтения: если предположить, что операции чтения и записи имеют соотношение 10 к 1, за одну секунду будет выполняться  $1160 * 10 = 11\,600$  операций чтения.
- Если предположить, что сервис для сокращения URL-адресов проработает 10 лет, мы должны поддерживать хранение 100 миллионов \* 365 \* 10 = 365 миллиардов записей.
- Пусть длина среднего URL-адреса составляет 100 символов.
- Требования к хранилищу в ближайшие 10 лет: 365 миллиардов \* 100 байт \* 10 лет = 365 Тб.

Вы должны обсудить эти предположения и расчеты с интервьюером и убедиться в том, что вы поняли друг друга правильно.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

В этом разделе мы обсудим конечные точки API, перенаправление и сокращение URL-адресов.

## Конечные точки API

Конечные точки API обеспечивают взаимодействие между клиентами и серверами. Наш API будет в стиле REST. Если вы не знакомы с этим стилем, можете обратиться к справочным материалам (например, [1]). Сервису сокращения URL-адресов нужны две основные конечные точки:

- Сокращение URL-адреса. Чтобы создать новый сокращенный URL-адрес, клиент отправляет POST-запрос с одним параметром: исходным длинным URL-адресом. Конечная точка выглядит так:
 

```
POST api/v1/data/shorten
```

  - параметр запроса: {longUrl: longURLString};
  - возвращается shortURL.
- Перенаправление URL-адреса. Чтобы перенаправить сокращенную ссылку к соответствующему длинному URL-адресу, клиент отправляет GET-запрос. Конечная точка выглядит так:
 

```
GET api/v1/shortUrl
```

  - возвращается longURL для HTTP-перенаправления.

## Перенаправление URL-адресов

На рис. 8.1 показано, что происходит при вводе в браузере URL-адреса tinyurl. Получив запрос, сервер меняет короткий URL-адрес на длинный, используя перенаправление с кодом 301.



Рис. 8.1

Подробное взаимодействие между клиентами и серверами показано на рис. 8.2.

короткий URL: <https://tinyurl.com/qfj5opu>

длинный URL: [https://www.amazon.com/dp/B017V4NTFA?ref=adblp13nvvx\\_0\\_2\\_im](https://www.amazon.com/dp/B017V4NTFA?ref=adblp13nvvx_0_2_im)

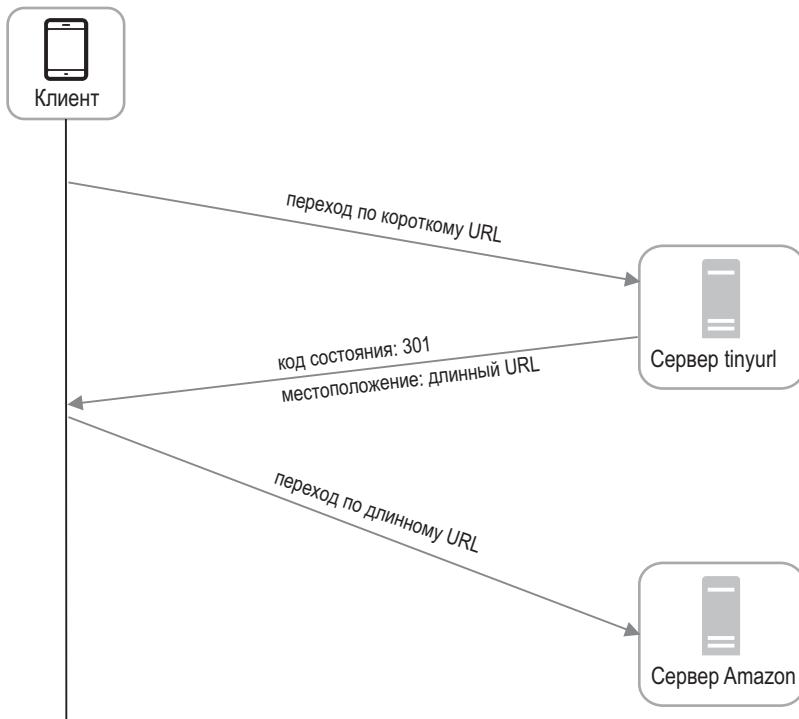


Рис. 8.2

Здесь стоит обсудить отличия между кодами перенаправления 301 и 302.

- **Перенаправление 301.** Код состояния 301 означает, что запрошенный URL-адрес «навсегда» перемещен по длинному URL-адресу. Так как перенаправление постоянное, браузер кэширует ответ и последующие запросы по тому же адресу не будут направляться к нашему сервису. Вместо этого браузер сразу откроет сокращенный URL-адрес.
- **Перенаправление 302.** Код состояния 302 означает, что URL-адрес «временно» перемещен по длинному URL-адресу. То есть после-

дующие запросы того же URL-адреса будут сначала отправляться нашему сервису, а затем перенаправляться к серверу длинного URL-адреса.

У обоих методов перенаправления есть свои плюсы и минусы. Если нам в первую очередь нужно снизить нагрузку на сервер, лучше использовать код состояния 301, так как для каждого сокращенного URL-адреса сервис будет получать только первый запрос. Если же нам нужна аналитическая информация, стоит выбрать код состояния 302, так как он упрощает отслеживание частоты и источника переходов по ссылке.

Самый очевидный способ реализации перенаправления URL-адресов заключается в использовании хеш-таблиц. Если предположить, что хеш-таблица хранит пары `<shortURL, longURL>`, перенаправление можно организовать следующим образом:

- получаем `longURL`: `longURL = hashTable.get(shortURL);`
- получив `longURL`, выполняем перенаправление.

### Сокращение URL-адресов

Допустим, сокращенный URL-адрес выглядит как `www.tinyurl.com/{hashValue}`. Чтобы его получить, мы должны найти функцию  $f_x$ , которая привязывает длинный URL-адрес к `hashValue`, как показано на рис. 8.3.

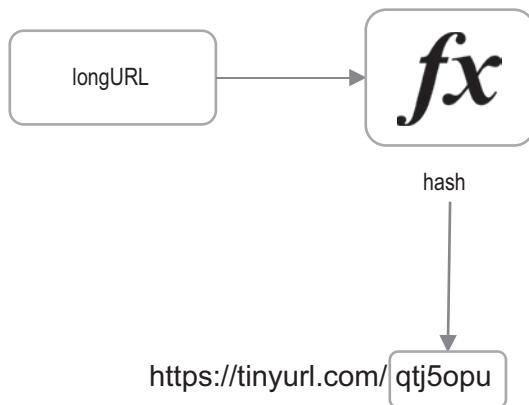


Рис. 8.3

К функции хеширования предъявляются следующие требования:

- каждое значение `longURL` должно иметь один хеш `hashValue`;
- каждое значение `hashValue` должно указывать обратно на `longURL`.

Подробная архитектура функции хеширования обсуждается в следующем разделе.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

До сих пор мы обсуждали общие вопросы проектирования сервиса для сокращения и перенаправления URL-адресов. Здесь же мы подробно рассмотрим модель данных, функцию хеширования и процессы сокращения/перенаправления.

### Модель данных

При обсуждении общей архитектуры мы решили хранить все в хеш-таблице. Это хорошая отправная точка, но в реальных системах такой подход непрактичен, поскольку ресурсы памяти ограничены и дороги. Пары `<shortURL, longURL>` лучше хранить в реляционной базе данных. На рис. 8.4 показана схема простой таблицы БД, состоящей из трех столбцов: `id`, `shortURL`, `longURL`.

url	
PK	id
	<b>shortURL</b> <b>longURL</b>

Рис. 8.4

## Функция хеширования

Функция хеширования используется для получения из длинного URL-адреса хеша — `hashValue`.

### Длина `hashValue`

`hashValue` состоит из символов [0-9, a-z, A-Z], число которых равно  $10 + 26 + 26 = 62$ . Чтобы определить длину `hashValue`, нужно найти наименьшее  $n$ , при котором  $62^n \geq 365$  миллиардов. Судя по этим прикидкам, система должна поддерживать до 365 миллиардов URL-адресов. В табл. 8.1 показаны разные значения длины `hashValue` и соответствующее максимальное количество поддерживаемых URL-адресов.

Таблица 8.1

n	Максимальное количество URL-адресов
1	$62^1 = 62$
2	$62^2 = 3\,844$
3	$62^3 = 238\,328$
4	$62^4 = 14\,776\,336$
5	$62^5 = 916\,132\,832$
6	$62^6 = 56\,800\,235\,584$
7	$62^7 = 3\,521\,614\,606\,208 \sim 3,5$ триллиона
8	$62^8 = 218\,340\,105\,584\,896$

3,5 триллиона (когда  $n = 62^7$ ) более чем достаточно для хранения 365 миллиардов URL-адресов, поэтому длина `hashValue` будет равна 7.

Мы исследуем два вида функций хеширования для сокращения URL-адресов: «хеш + разрешение конфликтов» и «преобразование base62».

### Хеш + разрешение конфликтов

Чтобы сократить длинный URL-адрес, нужно реализовать хеш-функцию, которая хеширует его в строку из 7 символов. Очевидное решение состоит

в использовании общеизвестных функций хеширования вроде CRC32, MD5 или SHA-1. В следующей таблице приводится сравнение хешей, полученных с помощью разных функций из URL-адреса [https://en.wikipedia.org/wiki/Systems\\_design](https://en.wikipedia.org/wiki/Systems_design).

Таблица 8.2

Хеш-функция	Значение хеша (шестнадцатеричное)
CRC32	5cb54054
MD5	5a62509a84df9ee03fe1230b9df8b84e
SHA-1	0eeae7916c06853901d9ccbefbfcaf4de57ed85b

Как видно из табл. 8.2, даже самое короткое значение хеша (из CRC32) получается слишком большим (больше 7 символов). Как его сократить?

В качестве одного из решений можно взять первые 7 символов хеша, но это чревато конфликтами. Чтобы значения хеша не дублировались, мы можем рекурсивно добавлять к ним заданную строку, пока они не станут уникальными. Этот процесс проиллюстрирован на рис. 8.5.

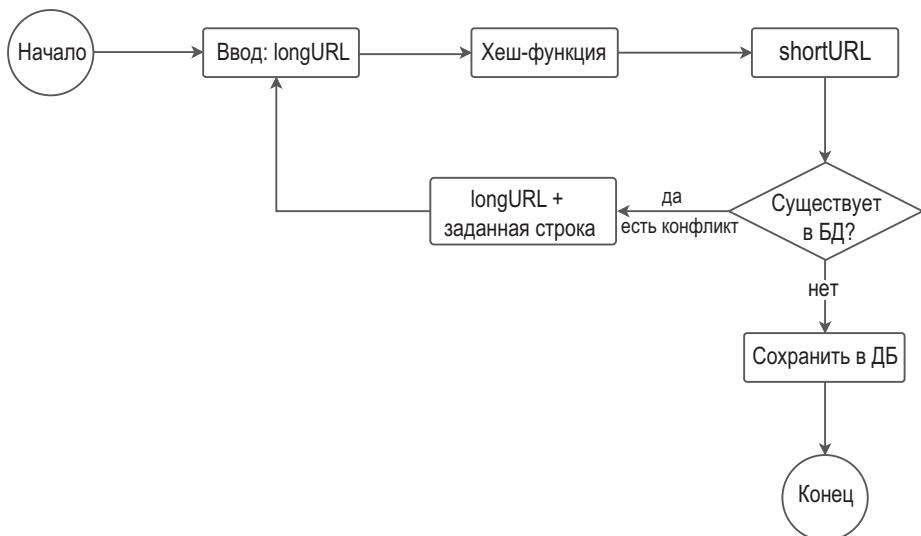


Рис. 8.5

Этот метод может избавить нас от конфликтов, но обращаться к базе данных при каждом запросе, чтобы проверить наличие в ней shortURL, довольно расточительно. Для улучшения производительности можно использовать фильтр Блума [2]. Это вероятностная методика с эффективным использованием пространства, которая позволяет проверить, входит ли элемент в множество. Подробнее об этом читайте в справочных материалах [2].

### Преобразование base62

Еще одним распространенным методом сокращения URL-адресов является преобразование значения в другую систему счисления. Поскольку для `hashValue` используется набор из 62 символов, мы выберем алгоритм base62. Чтобы понять, как происходит это преобразование, переведем десятичное число  $11157_{10}$  в вид с основанием 62.

- Как понятно из названия, base62 – это способ кодирования с использованием 62 символов. Они соотносятся как 0-0, ..., 9-9, 10-a, 11-b, ..., 35-z, 36-A, ..., 61-Z, где «a» соответствует 10, «Z» соответствует 61 и т. д.
- $11157_{10} = 2 \times 62^2 + 55 \times 62^1 + 59 \times 62^0 = [2, 55, 59] \rightarrow [2, T, X]$  в представлении base62. Процесс преобразования показан на рис. 8.6.

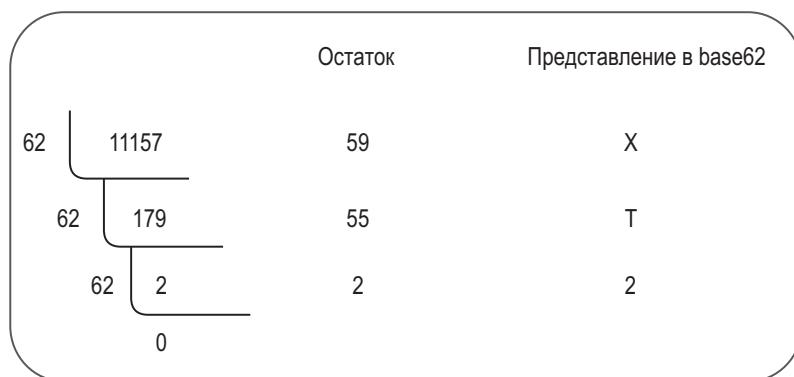


Рис. 8.6

- Таким образом, сокращенный URL-адрес выглядит как <https://tinyurl.com/2TX>.

## Сравнение двух подходов

В табл. 8.3 приводятся различия этих двух подходов.

Таблица 8.3

Хеш + разрешение конфликтов	Преобразование base62
Фиксированная длина сокращенного URL-адреса	Сокращенный URL-адрес имеет переменную длину, которая увеличивается вместе с ID
Не требует генератора уникальных ID	Этот подход использует генератор уникальных ID
Возможны конфликты, которые нужно разрешать	Конфликты исключены, так как ID уникальные
Невозможно определить, каким будет следующий сокращенный URL-адрес, так как он не зависит от ID	Если ID всегда инкрементируется на 1, можно легко определить следующий доступный URL-адрес. Это может стать угрозой безопасности

## Тщательный анализ сокращения URL-адресов

Процесс сокращения URL-адресов является одним из ключевых элементов системы, поэтому мы хотим сделать его простым и функциональным. В нашей архитектуре используется преобразование base62. Принцип работы показан на следующей диаграмме (рис. 8.7).

1. longURL подается на вход.
2. Система проверяет, есть ли longURL в базе данных.
3. Если да, это означает, что значение longURL уже было преобразовано в shortURL. В этом случае мы извлекаем shortURL из базы данных и возвращаем его клиенту.
4. Если нет, longURL является новым адресом и для него генерируется новый уникальный ID (первичный ключ).
5. Преобразуем ID в shortURL методом base62.
6. Создаем в БД новую запись с ID, shortURL и longURL.

Чтобы вам было проще понять этот процесс, рассмотрим конкретный пример.

- Предположим, longURL имеет значение [https://en.wikipedia.org/wiki/Systems\\_design](https://en.wikipedia.org/wiki/Systems_design).

- Генератор уникальных ID возвращает 2009215674938.
- Преобразуем ID в shortURL методом base62. 2009215674938 пре-вращается в zn9edcu.
- Сохраняем ID, shortURL и longURL в базу данных, как показано в табл. 8.4.

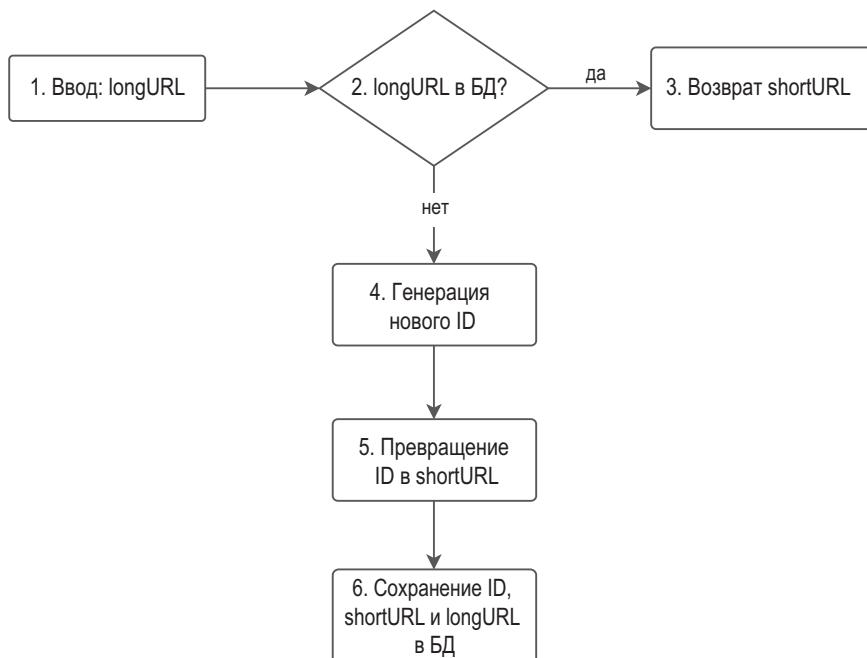


Рис. 8.7

Таблица 8.4

<b>id</b>	<b>shortURL</b>	<b>longURL</b>
2009215674938	zn9edcu	<a href="https://en.wikipedia.org/wiki/Systems_design">https://en.wikipedia.org/wiki/Systems_design</a>

Стоит упомянуть о распределенном генераторе уникальных ID. Его основная задача состоит в получении идентификаторов, которые используются для создания значений **shortURL** и не повторяются в рамках всей

системы. Реализация генератора уникальных ID в сильно распределенном окружении является непростой задачей. К счастью, мы уже обсудили несколько решений в главе 7 «Проектирование генератора уникальных идентификаторов в распределенных системах». Перечитайте ее, если хотите освежить память.

### Тщательный анализ перенаправления URL-адресов

На рис. 8.8 показана подробная схема перенаправления URL-адресов. Поскольку чтение происходит чаще, чем запись, пара `<shortURL, longURL>` хранится в кэше для улучшения производительности.

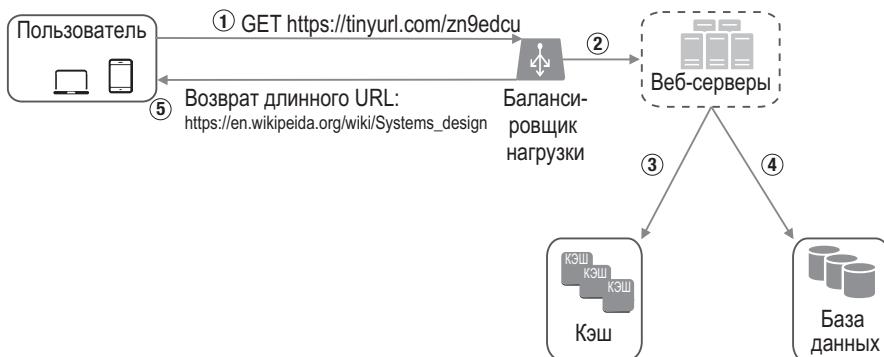


Рис. 8.8

Ниже кратко изложен процесс перенаправления URL-адресов:

- Пользователь кликает по короткой ссылке `https://tinyurl.com/zn9edcu`.
- Балансировщик нагрузки направляет запрос к веб-серверам.
- Если `shortURL` уже есть в кэше, то сразу возвращается `longURL`.
- Если `shortURL` нет в кэше, то `longURL` извлекается из базы данных. Если этого адреса нет в БД, пользователь, скорее всего, ввел его неправильно.
- Пользователю возвращается `longURL`.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы поговорили о проектировании API, модели данных, функции хеширования, а также о сокращении и перенаправлении URL-адресов.

Если в конце интервью остается время, можно обсудить дополнительные вопросы.

- Ограничитель трафика. Мы можем столкнуться с потенциальной проблемой безопасности: злоумышленники могут послать чрезмерно большое количество запросов на сокращение URL-адреса. Ограничитель трафика помогает фильтровать запросы с учетом IP-адреса или других правил. Если вам нужно вспомнить эту тему, вернитесь к главе 4 «Проектирование ограничителя трафика».
- Масштабирование веб-серверов. Поскольку веб-уровень не хранит свое состояние, его легко масштабировать, добавляя или удаляя веб-серверы.
- Масштабирование базы данных. Репликация и сегментирование базы данных являются распространенными методиками.
- Аналитика. Данные играют все более важную роль в успехе бизнеса. Интеграция аналитической системы в сервис для сокращения URL-адресов поможет получить такие ценные сведения, как количество пользователей, перешедших по ссылке, точное время перехода и т. д.
- Доступность, согласованность и надежность. Эти характеристики являются ключом к успеху любой крупной системы. Мы подробно обсуждали их в главе 1. Пожалуйста, вспомните эту тему.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Руководство по REST: <https://wwwレストапитюриал.ком/index.html>
- [2] Фильтр Блума: [https://ru.wikipedia.org/wiki/Фильтр\\_Блума](https://ru.wikipedia.org/wiki/Фильтр_Блума)

---

# 9

---

## ПРОЕКТИРОВАНИЕ ПОИСКОВОГО РОБОТА

В этой главе мы сосредоточимся на интересной классической задаче, которую можно встретить на интервью по проектированию ИТ-систем, — создании поискового робота.

Поисковый робот еще называют веб-пауком или веб-краулером. Он широко используется в поисковых системах для обнаружения нового или обновленного контента в Сети. Это могут быть веб-страницы, изображения, видео, PDF-файлы и т. д. Сначала поисковый робот собирает несколько веб-страниц, а затем проходит по всем ссылкам, которые они содержат, чтобы собрать новый контент. Пример этого процесса показан на рис. 9.1.

Поисковый робот применяется для множества разных задач.

- Индексация в поисковой системе. Это самый распространенный сценарий использования. Робот собирает веб-страницы, чтобы создать локальный индекс для поисковой системы. Например, в поисковой системе Google эту роль играет Googlebot.
- Веб-архивация. Это процесс сбора информации с веб-страниц для дальнейшего хранения и использования. Например, архивацией веб-сайтов занимаются многие национальные библиотеки. В качестве известных примеров можно привести Библиотеку Конгресса США [1] и Веб-архив ЕС [2].
- Извлечение веб-данных. Сверхбыстрое развитие интернета создает беспрецедентную возможность для сбора информации. Извлечение веб-данных помогает собирать в Сети ценные сведения. Например, ведущие финансовые фирмы используют поисковые роботы для загрузки стенограмм собраний акционеров и ежегодных отчетов для анализа ключевых инициатив компаний.

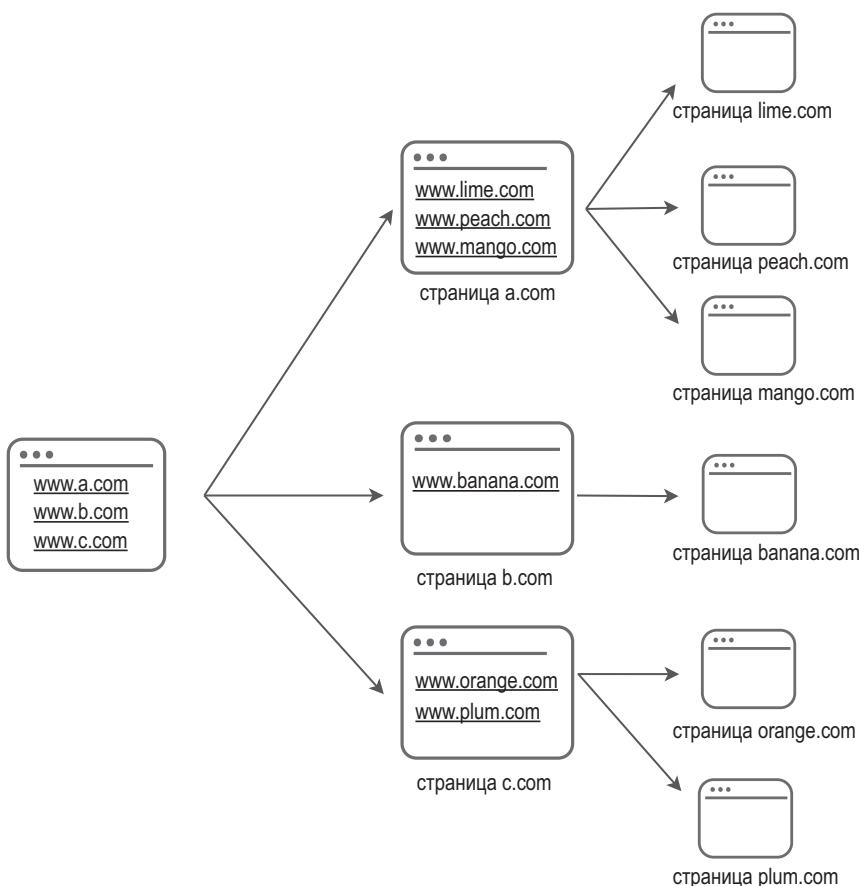


Рис. 9.1

- Веб-мониторинг. Поисковые роботы помогают отслеживать нарушения авторских прав и незаконное использование торговых марок в интернете. Например, Digimarc [3] таким образом ищет пиратские копии и отчеты.

Сложность разработки поискового робота зависит от того, какой масштаб он должен поддерживать. Это может быть как скромный школьный проект, который можно закончить за пару часов, так и гигантская система, требующая постоянного внимания со стороны целой команды инженеров. Поэтому сначала нужно определиться с масштабом и функциями, которые должны поддерживаться.

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Поисковый робот работает по простому принципу:

1. На вход подается список URL-адресов, после чего загружаются соответствующие веб-страницы.
2. Из веб-страниц извлекаются URL-адреса.
3. Новые URL-адреса добавляются в список для дальнейшей загрузки.  
Эти три шага повторяются заново.

Ограничивается ли работа поискового робота этим простым алгоритмом? Не совсем. Спроектировать широкомасштабный поисковый робот очень сложно. Вряд ли кому-то удастся это сделать во время интервью. Прежде чем переходить к архитектуре, необходимо задать вопросы о требованиях и о масштабе системы.

**Кандидат:** «Каково основное назначение поискового робота? Он используется для индексации в поисковой системе, сбора данных или для чего-то другого?»

**Интервьюер:** «Для индексации в поисковой системе».

**Кандидат:** «Сколько веб-страниц поисковый робот собирает в месяц?»

**Интервьюер:** «1 миллиард страниц».

**Кандидат:** «Какого рода контент мы ищем? Только HTML или другие ресурсы вроде PDF-файлов и изображений?»

**Интервьюер:** «Только HTML».

**Кандидат:** «Следует ли учитывать недавно добавленные или отредактированные веб-страницы?»

**Интервьюер:** «Да».

**Кандидат:** «Нужно ли хранить собранные HTML-страницы?»

**Интервьюер:** «Да. Срок хранения — 5 лет».

**Кандидат:** «Что делать с веб-страницами, содержимое которых дублируется?»

**Интервьюер:** «Их следует игнорировать».

Это лишь некоторые из тех вопросов, которые можно задать интервьюеру. Необходимо разобраться в требованиях и прояснить непонятные моменты. Даже если вас попросят спроектировать такой простой продукт, как поисковый робот, у вас с интервьюером может быть разное понимание того, что он должен собой представлять.

Помимо функциональности, которую нужно согласовать с интервьюером, необходимо учитывать следующие характеристики поискового робота.

- **Масштабируемость.** Интернет огромен. В нем миллиарды веб-страниц. Поисковый робот должен быть чрезвычайно эффективным и использовать параллельные вычисления.
- **Устойчивость.** Интернет полон ловушек. Вам постоянно будет встречаться некорректный HTML-код, неотзывчивые серверы, сбои, вредоносные ссылки и т. д. Поисковый робот должен справляться со всеми этими пограничными случаями.
- **Вежливость.** Поисковый робот не должен отправлять веб-сайту слишком много запросов за короткий промежуток времени.
- **Расширяемость.** Система должна быть гибкой, чтобы для поддержки новых видов контента не приходилось вносить масштабные изменения. Например, если в будущем нам понадобится собирать графические файлы, это не должно привести к переработке всей системы.

## Приближенные оценки

Следующие оценки основаны на множестве предположений, поэтому вы должны убедиться в том, что вы с интервьюером правильно поняли друг друга.

- Предположим, что каждый месяц загружается 1 миллиард веб-страниц.
- QPS:  $1\ 000\ 000\ 000 / 30 \text{ дней} / 24 \text{ часа} / 3600 \text{ секунд} = \sim 400$  страниц в секунду.
- Пиковый показатель QPS =  $2 * \text{QPS} = 800$ .
- Предположим, что средний размер веб-страницы составляет 500 Кб.

- 1 миллиард страниц \* 500 Кб = 500 Тб в месяц для хранения. Если вы плохо ориентируетесь в единицах измерения данных, перечитайте раздел «Степень двойки» в главе 2.
- Если данные хранятся на протяжении пяти лет, 500 Тб \* 12 месяцев \* 5 лет = 30 Пб. Для контента, собранного за пять лет, нужно хранилище размером 30 Пб.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Определившись с требованиями, мы переходим к общим архитектурным вопросам. На рис. 9.2 изображена предлагаемая архитектура, вдохновленная исследованиями [4] [5].

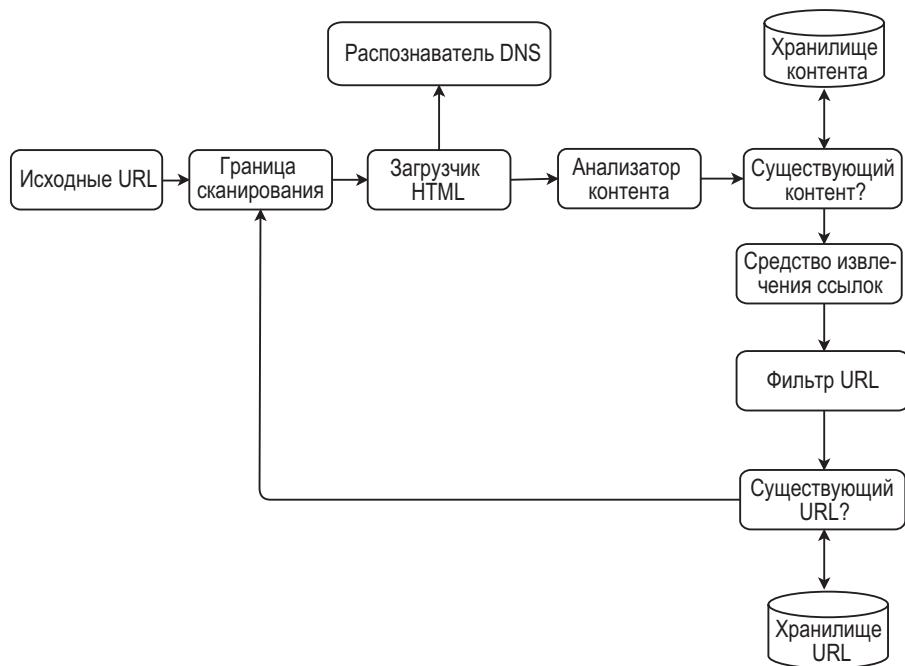


Рис. 9.2

Сначала мы исследуем каждый компонент архитектуры, чтобы понять его функции, а затем шаг за шагом рассмотрим принцип работы поискового робота.

## Исходные URL-адреса

В процессе поиска в качестве отправной точки используются исходные URL-адреса. Например, чтобы перебрать все веб-страницы на университете веб-сайте, нам, очевидно, следует начать с доменного имени университета.

Для обхода всего интернета нам нужно творчески подойти к выбору исходных URL-адресов. Хороший исходный URL-адрес позволит перебрать как можно большее количество ссылок. Общая стратегия — разделить пространство адресов на отдельные части. Первый предложенный подход основан на местоположении, так как популярность тех или иных веб-сайтов может зависеть от страны. Исходные URL-адреса также можно выбирать по тематике. Например, адресное пространство можно разделить на покупки, спорт, медицину и т. д. Выбор исходных URL-адресов зависит от разных факторов. От вас не ожидают идеального решения. Просто размышляйте вслух.

## Граница сканирования

Большинство современных поисковых роботов делят контент на уже загруженный и ожидающий загрузки. Компонент, хранящий URL-адреса, предназначенные для загрузки, называется границей сканирования. Его можно считать очередью вида «первым пришел, первым ушел». Углубленную информацию об этом компоненте можно найти в разделе, посвященном углубленному проектированию.

## Загрузчик HTML

Этот компонент загружает веб-страницы из интернета. URL-адреса этих веб-страниц предоставляются границей сканирования.

## Распознаватель DNS

Чтобы загрузить веб-страницу, URL сначала нужно перевести в IP-адрес. Для этого загрузчик HTML может обратиться к распознавателю DNS. Например, по состоянию на 5.3.2019 URL [www.wikipedia.org](http://www.wikipedia.org) преобразуется в IP-адрес 198.35.26.96.

## **Анализатор контента**

После загрузки веб-страницы ее нужно проанализировать: вдруг у нее некорректный HTML-код, который вызовет проблемы и только займет место в хранилище? Если разместить анализатор контента на одном сервере с поисковым роботом, это замедлит процесс сбора данных. В связи с этим он имеет вид отдельного компонента.

## **Существующий контент?**

Как показывает онлайн-исследование [6], 29 % от всех веб-страниц являются дубликатами, что может привести к повторному сохранению одного и того же контента. Мы используем структуру данных «Существующий контент?», чтобы избежать дублирования информации и сократить время обработки. Она помогает обнаруживать новый контент, который система уже сохранила. Но это медленный подход, занимающий много времени, особенно если речь идет о миллиардах веб-страниц. Для эффективного выполнения этой задачи следует сравнивать не сами веб-страницы, а их хеши [7].

## **Хранилище контента**

Это система хранения HTML. Ее выбор зависит от таких факторов, как тип и размер данных, частота доступа, время жизни и т. д. Используется как диск, так и память.

- Большая часть контента хранится на диске, так как набор данных слишком большой и не умещается в памяти.
- Востребованный контент хранится в памяти для снижения латентности.

## **Средство извлечения ссылок**

Этот компонент анализирует HTML-страницы и извлекает из них ссылки. Пример этого процесса показан на рис. 9.3. Относительные пути преобразуются в абсолютные URL-адреса путем добавления префикса <https://> <en.wikipedia.org>.

```
<html class="client-nojs" lang="en" dir="ltr">
  <head>
    <meta charset="UTF-8"/>
    <title>Wikipedia, the free encyclopedia</title>
  </head>
  <body>
    <li><a href="/wiki/Cong_Weixi" title="Cong Weixi">Cong Weixi</a></li>
    <li><a href="/wiki/Kay_Hagan" title="Kay Hagan">Kay Hagan</a></li>
    <li><a href="/wiki/Vladimir_Bukovsky" title="Vladimir Bukovsky">Vladimir Bukovsky</a></li>
    <li><a href="/wiki/John_Conyers" title="John Conyers">John Conyers</a></li>
  </body>
</html>
```

**Извлеченные  
ссылки:**

[https://en.wikipedia.org/wiki/Cong\\_Weixi](https://en.wikipedia.org/wiki/Cong_Weixi)  
[https://en.wikipedia.org/wiki/Kay\\_Hagan](https://en.wikipedia.org/wiki/Kay_Hagan)  
[https://en.wikipedia.org/wiki/Vladimir\\_Bukovsky](https://en.wikipedia.org/wiki/Vladimir_Bukovsky)  
[https://en.wikipedia.org/wiki/John\\_Conyers](https://en.wikipedia.org/wiki/John_Conyers)

Рис. 9.3

### Фильтр URL-адресов

Фильтр URL-адресов отклоняет определенные типы контента, расширения файлов, ссылки на страницы с ошибками и URL-адреса, занесенные в черный список.

### Существующий URL-адрес?

«Существующий URL-адрес?» — это структура данных, которая отслеживает URL-адреса, которые уже посещались или находятся в границе сканирования. Это помогает избежать повторного открытия одного и того же URL-адреса, которое чревато повышенной нагрузкой на сервер и потенциальными бесконечными циклами.

Для реализации этого компонента обычно применяют такие методики, как фильтр Блума и хеш-таблица. Мы не станем в них углубляться. Дополнительную информацию можно найти в справочных материалах [4] [8].

## Хранилище URL-адресов

Это хранилище уже посещенных URL-адресов.

Итак, мы рассмотрели каждый отдельный компонент. Теперь мы соберем их воедино и обсудим принцип работы системы в целом.

## Принцип работы поискового робота

Чтобы как следует объяснить каждый этап рабочего процесса, мы добавили на диаграмму архитектуры порядковые номера (рис. 9.4).

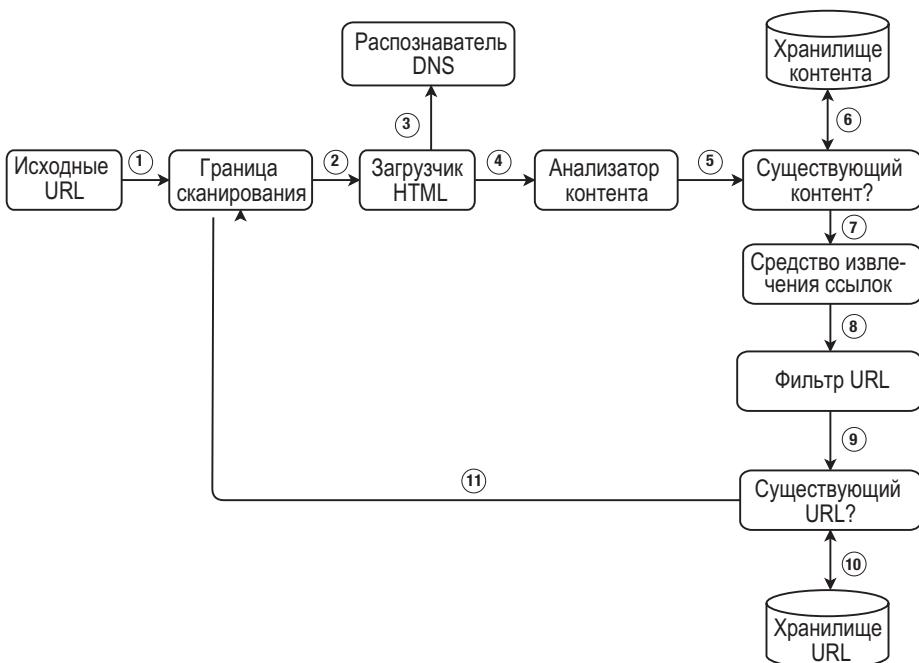


Рис. 9.4

- Шаг 1. Добавляем исходные URL-адреса в границу сканирования.
- Шаг 2. Загрузчик HTML берет список URL-адресов из границы сканирования.

- Шаг 3. Загрузчик HTML получает соответствующие IP-адреса от распознавателя DNS и начинает загрузку.
- Шаг 4. Анализатор контента разбирает HTML-страницы и проверяет их корректность.
- Шаг 5. После разбора и проверки контент передается компоненту «Существующий контент?».
- Шаг 6. Компонент «Существующий контент?» проверяет, есть ли данная HTML-страница в хранилище:
  - ◆ если есть, это означает, что этот контент находится по другому URL-адресу и мы его уже обработали. В этом случае HTML-страница отклоняется;
  - ◆ если нет, система еще не обрабатывала этот контент, поэтому он передается средству извлечения ссылок.
- Шаг 7. Из HTML-страниц извлекаются ссылки.
- Шаг 8. Извлеченные ссылки передаются фильтру URL-адресов.
- Шаг 9. После фильтрации ссылки передаются компоненту «Существующий URL-адрес?».
- Шаг 10. Компонент «Существующий URL-адрес?» проверяет, находится ли URL в хранилище. Если да, то он уже обрабатывался и больше ничего делать не нужно.
- Шаг 11. Если URL-адрес еще не обрабатывался, он добавляется в границу сканирования.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

До сих пор мы обсуждали общие аспекты архитектуры. Теперь рассмотрим подробно самые важные составные компоненты и характеристики:

- границу сканирования;
- загрузчик HTML;
- устойчивость;
- расширяемость;
- обнаружение и отклонение проблемного контента.

## DFS и BFS

Интернет можно считать направленным графом, вершинами которого являются веб-страницы, а ребрами — гиперссылки (URL-адреса). Процесс поиска можно рассматривать как обход направленного графа, начиная с корневой вершины. Для этого существует два распространенных алгоритма: поиск в глубину (deep-first search, DFS) и поиск в ширину (breadth-first search, BFS). При этом DFS обычно является не самым удачным вариантом, так как граф может быть очень глубоким.

Поисковые роботы, как правило, используют алгоритм BFS, который реализуется с помощью очереди FIFO. URL-адреса достаются из очереди в том порядке, в котором они добавлялись. Но у этой реализации есть две проблемы.

- Большинство ссылок, размещенных на одной веб-странице, указывают на один и тот же сетевой узел. На рис. 9.5 все ссылки на странице внутренние, из-за чего поисковый робот вынужден

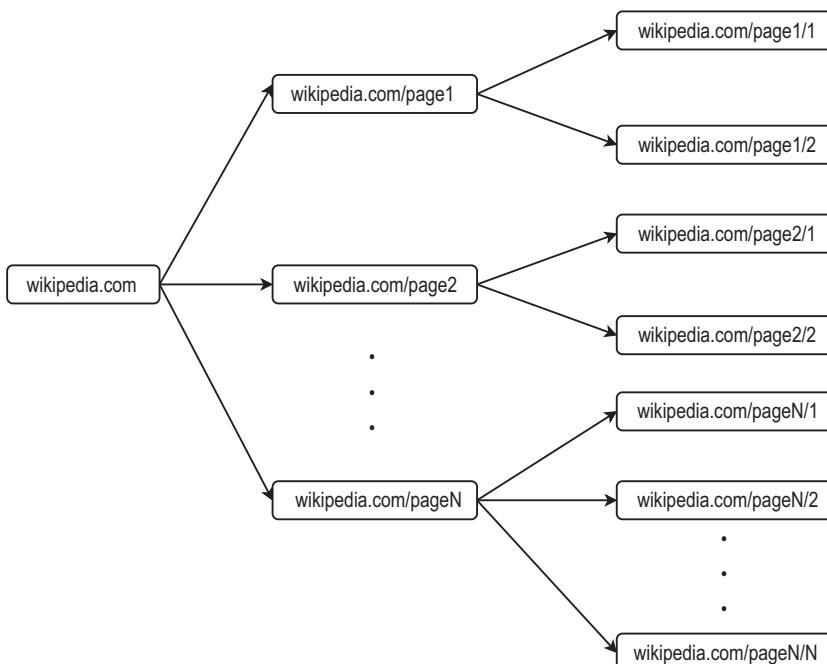


Рис. 9.5

обрабатывать URL-адреса в одном домене ([wikipedia.com](https://wikipedia.com)). Если он попытается загрузить веб-страницы параллельно, серверы Википедии будут завалены запросами. Это «невежливо».

- Стандартная версия BFS не учитывает приоритет URL-адреса. В интернете миллиарды веб-страниц, и не все они имеют одинаковый уровень качества и значимости. Поэтому приоритет обработки URL-адресов может зависеть от их рейтинга, популярности, частоты обновления и т. д.

## Граница сканирования

Эти проблемы помогает решить граница сканирования. Это структура данных, хранящая URL-адреса, которые нужно загрузить. Она играет важную роль в соблюдении вежливости, расстановке приоритетов и обеспечении актуальности страниц. Несколько исследований, посвященных границе сканирования и заслуживающих вашего внимания, упоминается в справочных материалах [5][9]. Их результаты приводятся ниже.

### Вежливость

В целом поисковый робот не должен отправлять серверу слишком много запросов за короткий промежуток времени. Такое поведение считается невежливым и даже может быть воспринято как DoS-атака. Например, если не предусмотреть никаких ограничений, поисковый робот может отправлять одному и тому же веб-сайту тысячи запросов в секунду и тем самым перегрузит серверы.

Основная идея соблюдения вежливости заключается в последовательной загрузке страниц с одного сервера. Между загрузками можно сделать задержку. Чтобы это реализовать, доменные имена веб-сайтов нужно привязывать к потокам (рабочим узлам) загрузки. У каждого потока-загрузчика есть отдельная очередь FIFO, и все URL-адреса он достает из нее. На рис. 9.6 показан механизм, который делает наш поисковый робот вежливым.

- Маршрутизатор очереди. Следит за тем, чтобы каждая очередь ( $b_1, b_2, \dots, b_n$ ) содержала URL-адреса только с одним доменным именем.
- Таблица связывания. Привязывает каждое доменное имя к очереди.

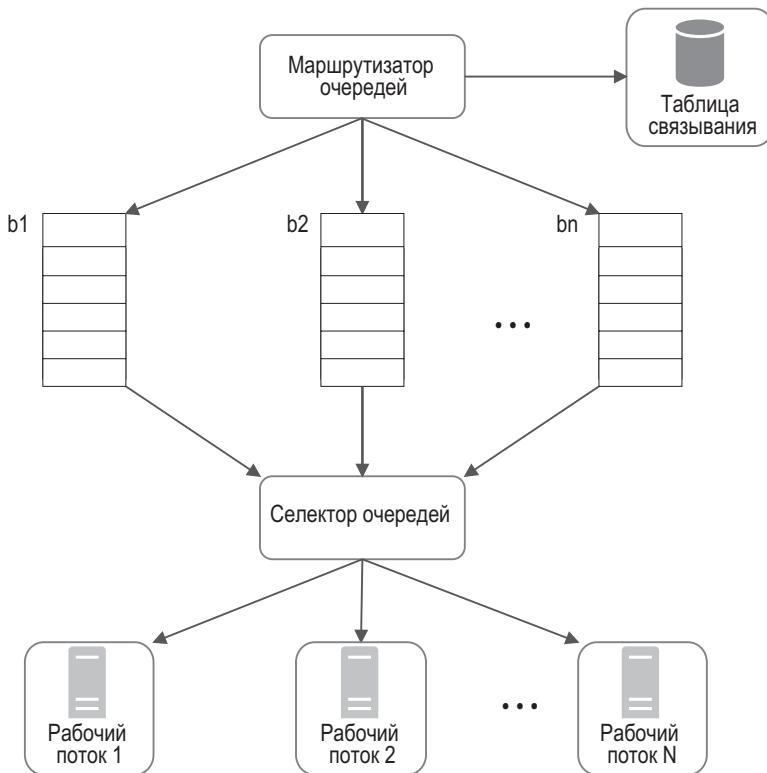


Рис. 9.6

Таблица 9.1

Доменное имя	Очередь
wikipedia.com	b1
apple.com	b2
...	...
nike.com	bn

- Очереди FIFO b1, b2, ..., bn. Каждая очередь содержит URL-адреса только с одним доменным именем.
- Селектор очередей. Каждый рабочий поток привязан к очереди FIFO, и все URL-адреса загружает только из нее. За логику выбора подходящей очереди отвечает селектор очередей.

- Рабочие потоки от 1 до N. Рабочий поток последовательно загружает веб-страницы из одного и того же сервера. Между загрузками можно предусмотреть задержку.

### Приоритет

Случайное сообщение на форуме, посвященном продукции Apple, по своей значимости отличается от сообщений, опубликованных на официальном сайте этой компании. Несмотря на общее для них ключевое слово Apple, поисковому роботу лучше начинать с последнего.

Мы назначаем URL-адресам приоритеты в зависимости от их ценности. При этом можно учитывать PageRank [10], популярность веб-сайта, частоту обновления и т. д. За это отвечает средство расстановки приоритетов. Более детальную информацию на эту тему можно найти в справочных материалах [5] [10].

Механизм, назначающий приоритеты URL-адресам, показан на рис. 9.7.

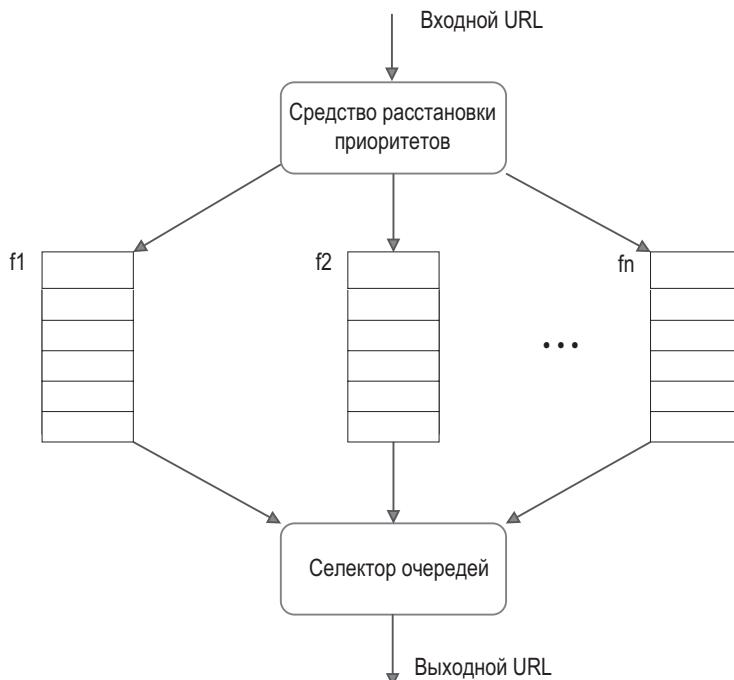


Рис. 9.7

- Средство расстановки приоритетов. Принимает на вход URL-адреса и вычисляет их приоритеты.
- Очереди с  $f_1$  по  $f_n$ . Каждой очереди назначается приоритет. Чем выше приоритет, тем больше вероятность того, что очередь будет выбрана.
- Селектор очередей. Выбирает произвольную очередь, отдавая предпочтение очередям с повышенным приоритетом.

На рис. 9.8 показано, как устроена граница сканирования. Она состоит из двух модулей:

- лицевые очереди: отвечают за расстановку приоритетов;
- тыльные очереди: отвечают за соблюдение вежливости.

### **Актуальность**

Веб-страницы постоянно добавляются, удаляются и редактируются. Поисковый робот должен периодически проходить по уже загруженным веб-страницам, чтобы поддерживать набор данных в актуальном состоянии. Для повторного перебора всех URL-адресов нужно много времени и ресурсов. Существует две стратегии оптимизации этого процесса:

- учитывать историю обновлений веб-страниц;
- назначать URL-адресам приоритеты, чтобы в первую очередь (и чаще) загружать важные страницы.

### **Хранилище для границы сканирования**

В настоящих поисковых системах количество URL-адресов на границе сканирования может достигать сотен миллионов [4]. Хранение всего этого в памяти плохо оказывается как на надежности, так и на масштабируемости. На диске тоже лучше не хранить все подряд, так как он медленный и легко может стать узким местом поискового робота.

Мы выбрали гибридный поход. Большая часть URL-адресов находится на диске, поэтому размер хранилища не составляет проблемы. Чтобы избежать накладных расходов, связанных с чтением и записью на диск, мы храним в памяти буфер для добавления/удаления данных из очереди. Содержимое буфера периодически сбрасывается на диск.

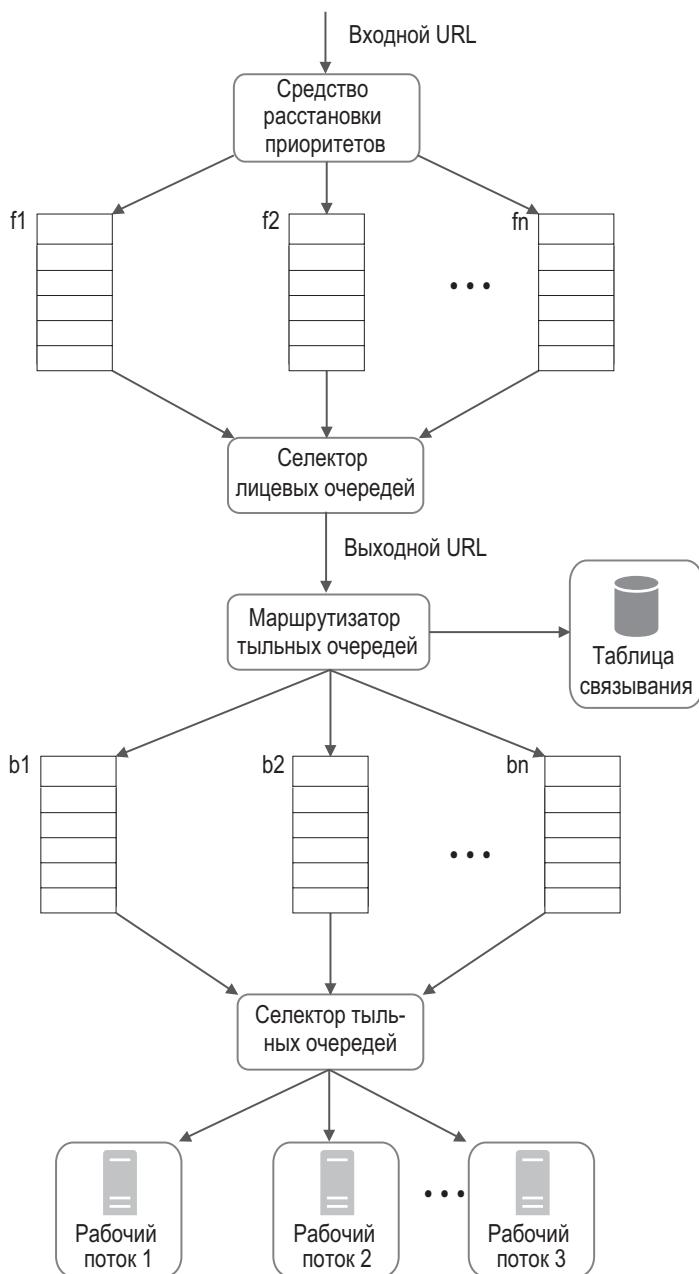


Рис. 9.8

## Загрузчик HTML

Этот компонент загружает веб-страницы из интернета по протоколу HTTP. Прежде чем его обсуждать, рассмотрим стандарт исключений для роботов.

### Robots.txt

Файл robots.txt (так называемый стандарт исключений для роботов) позволяет веб-сайтам взаимодействовать с поисковыми системами. В нем можно указать страницы, которые позволено загружать поисковым роботам. Прежде чем начинать обход веб-сайта, поисковый робот должен сначала проверить соответствующий файл robots.txt и следовать его правилам.

Чтобы не загружать robots.txt повторно, мы кэшируем его содержимое. Файл периодически загружается и сохраняется в кэш. Вот фрагмент, взятый из <https://www.amazon.com/robots.txt>. Некоторые директории, такие как creatorhub, закрыты для робота Google.

```
User-agent: Googlebot
Disallow: /creatorhub/*
Disallow: /rss/people/*/reviews
Disallow: /gp/pdp/rss/*/reviews
Disallow: /gp/cdp/member-reviews/
Disallow: /gp/aw/c r/
```

Еще одним важным аспектом загрузчика HTML является оптимизация производительности, речь о которой пойдет дальше.

### Оптимизация производительности

Ниже перечислены оптимизации загрузчика HTML.

1. **Распределенный поиск.** Для обеспечения высокой производительности процесс поиска должен быть распределен по разным серверам, каждый из которых работает в несколько потоков. Адресное пространство делится на части, чтобы каждый загрузчик отвечал за отдельное подмножество URL-адресов. Пример распределенного поиска показан на рис. 9.9.
2. **Кэширующий распознаватель DNS.** Распознаватель DNS является узким местом поискового робота, так как многие интерфейсы DNS синхронны по своей природе, и DNS-запросы могут занимать какое-то время. Время DNS-ответа варьируется от 10 мс до 200 мс. Когда поток поискового робота обращается к DNS-серверу, другие

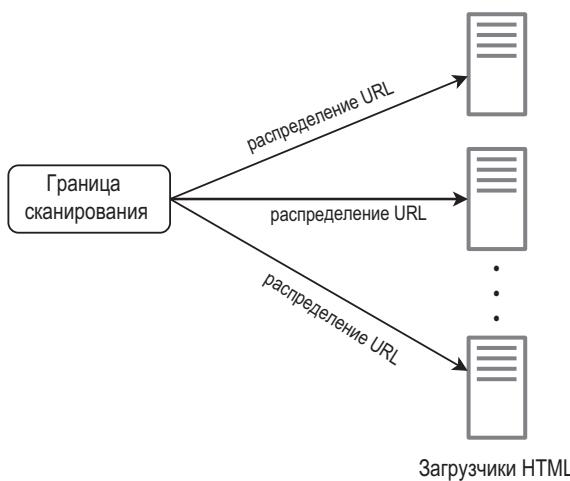


Рис. 9.9

потоки блокируются, пока не завершится первый запрос. В качестве эффективного метода повышения скорости, который позволит избежать частого обращения к DNS, можно использовать кэш DNS. Наш кэш хранит соответствия между доменными именами и IP-адресами и периодически обновляется с помощью заданий cron.

3. **Локальность.** Распределите серверы поискового робота географически. Чем ближе они к серверам веб-сайта, тем короче время загрузки. Локальность относится к большинству компонентов системы: серверам поискового робота, кэшу, очереди, хранилищу и т. д.
4. **Короткое время ожидания.** Некоторые серверы отвечают медленно или вовсе не отвечают. Чтобы не ждать их слишком долго, указывается максимальное время ожидания. Если сервер не ответит в отведенный промежуток времени, поисковый робот остановит задание и попробует загрузить какие-то другие страницы.

## Устойчивость

Наряду с оптимизацией производительности важную роль играет устойчивость. Вот несколько подходов к улучшению устойчивости системы.

- Согласованное хеширование. Оно помогает распределить нагрузку между загрузчиками и позволяет добавлять и удалять серверы загрузки. Подробнее об этом — в главе 5 «Согласованное хеширование».

- Сохранение состояния и данных поискового робота. Чтобы уберечься от сбоев, состояния и данные поискового робота записываются в систему хранения. В случае нарушения работы поисковый робот можно легко перезапустить, загрузив сохраненные состояния и данные.
- Обработка исключений. В крупномасштабных системах ошибки являются неизбежным и распространенным явлением. Поисковый робот должен как следует справляться с исключениями, не нарушая работу системы.
- Проверка корректности данных. Это важная мера для предотвращения системных ошибок.

## Расширяемость

Почти всегда по мере развития системы разработчики стремятся сделать ее гибкой и обеспечить поддержку новых типов контента. Поисковый робот можно расширить за счет подключения новых модулей. На рис. 9.10 показано, как это делается.

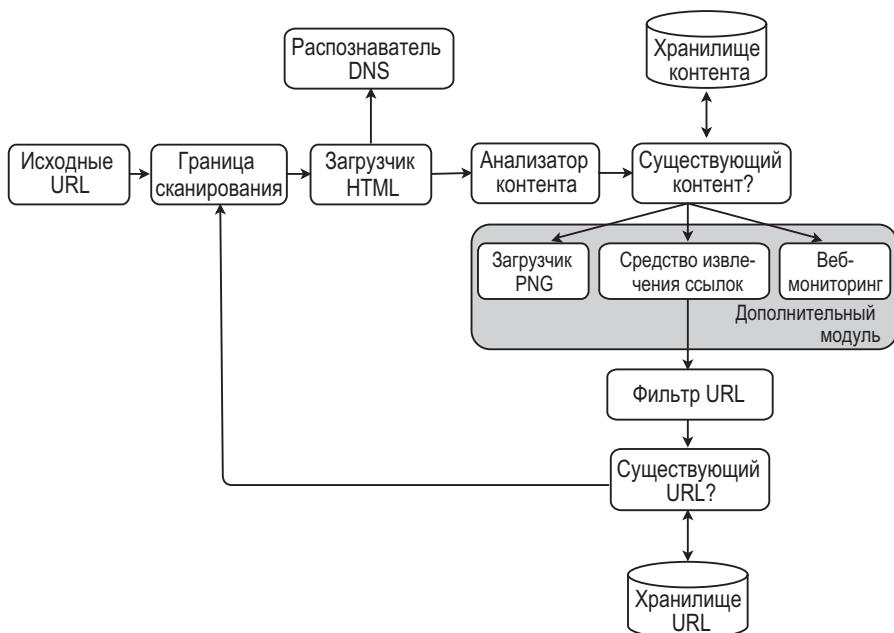


Рис. 9.10

- Загрузчик PNG — это подключаемый модуль для загрузки PNG-файлов.
- Модуль веб-мониторинга подключается для отслеживания и предотвращения нарушений авторских прав и незаконного использования товарных знаков в интернете.

## Обнаружение и отклонение проблемного контента

В этом разделе обсуждается обнаружение и предотвращение загрузки лишнего, бессмысленного или вредоносного контента.

1. **Лишний контент.** Как говорилось выше, около 30 % всех страниц являются дубликатами. Для их обнаружения можно использовать хеши или контрольные суммы [11].
2. **Ловушки.** Речь идет о веб-страницах, которые заставляют поискового робота входить в бесконечный цикл. Например, вот ссылка на бесконечно глубокую структуру директорий: [www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/....](http://www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/....)

Таких ловушек можно избежать, если установить максимальную длину URL-адресов. Но вообще, для их обнаружения не существует какого-то универсального метода. Веб-сайты с ловушками можно легко определить по необычно большому количеству страниц, которые они содержат. Сложно написать алгоритм, который будет автоматически избегать таких ловушек; это можно сделать вручную, а затем либо занести соответствующие веб-сайты в черный список, либо применить собственные фильтры для URL-адресов.

3. **Бессмысленные данные.** Некоторый контент несет в себе мало смысла, а порой его нет вообще. Это относится к рекламе, листингам кода, спаму и т. д. Такой контент бесполезен для поисковых роботов, и его по возможности следует отклонять.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

Мы начали эту главу с обсуждения характеристик хорошего поискового робота — масштабируемости, вежливости, расширяемости и устойчивости. Затем мы предложили вариант архитектуры и рассмотрели ее

ключевые компоненты. Создание масштабируемого поискового робота — нетривиальная задача, ведь интернет огромен и полон ловушек. И хотя нам удалось охватить множество аспектов, без внимания осталось еще много важного.

- Генерация страниц на стороне сервера. Многие веб-сайты генерируют ссылки на лету, используя JavaScript, AJAX и т. д. Если мы будем загружать и разбирать веб-страницы напрямую, нам не удастся извлечь динамически сгенерированные ссылки. Чтобы решить эту проблему, перед разбором веб-страницы мы генерируем ее на сервере [12].
- Фильтрация нежелательных страниц. Поскольку емкость хранилища и ресурсы поискового робота ограничены, будет полезно использовать компонент для борьбы со спамом, который будет фильтровать низкокачественные и рекламные страницы [13] [14].
- Репликация и сегментирование базы данных. Такие методики, как репликация и сегментирование, позволяют улучшить доступность, масштабируемость и надежность уровня данных.
- Горизонтальное масштабирование. Для крупномасштабного обхода загрузкой должны заниматься сотни или даже тысячи серверов. Главное, чтобы они не хранили свое состояние.
- Доступность, согласованность и надежность. Это ключевые характеристики успеха любой крупной системы. Мы подробно обсудили их в главе 1.
- Аналитика. Сбор и анализ данных — важная часть системы и ключевой элемент ее оптимизации.

Поздравляем, вы проделали длинный путь и можете гордиться собой. Отличная работа!

## **СПРАВОЧНЫЕ МАТЕРИАЛЫ**

- [1] Библиотека Конгресса США: <https://www.loc.gov/websites/>
- [2] Веб-архив ЕС: <http://data.europa.eu/webarchive>
- [3] Digimarc: <https://www.digimarc.com/products/digimarc-services/piracy-intelligence>

- [4] Heydon A., Najork M. Mercator: A scalable, extensible web crawler World Wide Web, 2 (4) (1999), pp. 219-229
- [5] By Christopher Olston, Marc Najork: Web Crawling. [http://infolab.stanford.edu/~olston/publications/crawling\\_survey.pdf](http://infolab.stanford.edu/~olston/publications/crawling_survey.pdf)
- [6] 29% Of Sites Face Duplicate Content Issues: <https://tinyurl.com/y6tmh55y>
- [7] Rabin M.O., et al. Fingerprinting by random polynomials Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. (1981)
- [8] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.
- [9] Donald J. Patterson, Web Crawling: <https://www.ics.uci.edu/~lopes/teaching/cs221W12/slides/Lecture05.pdf>
- [10] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Technical Report, Stanford University, 1998.
- [11] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), pages 422–426, July 1970.
- [12] Google Dynamic Rendering: <https://developers.google.com/search/docs/guides/dynamic-rendering>
- [13] T. Urvoy, T. Lavergne, and P. Filoche, “Tracking web spam with hidden style similarity,” in Proceedings of the 2nd International Workshop on Adversarial Information Retrieval on the Web, 2006.
- [14] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, “IRLbot: Scaling to 6 billion pages and beyond,” in Proceedings of the 17th International World Wide Web Conference, 2008.

# 10

## ПРОЕКТИРОВАНИЕ СИСТЕМЫ УВЕДОМЛЕНИЙ

В последние годы система уведомлений активно используется во многих приложениях. Она сообщает пользователям важную информацию вроде актуальных новостей, обновлений продуктов, событий, коммерческих предложений и т. д. Уведомления стали неотъемлемой частью нашей жизни. В этой главе вам предложено спроектировать систему уведомлений. Речь идет не только о мобильных push-уведомлениях, но и об SMS и электронных письмах. Примеры каждого типа уведомлений показаны на рис. 10.1.

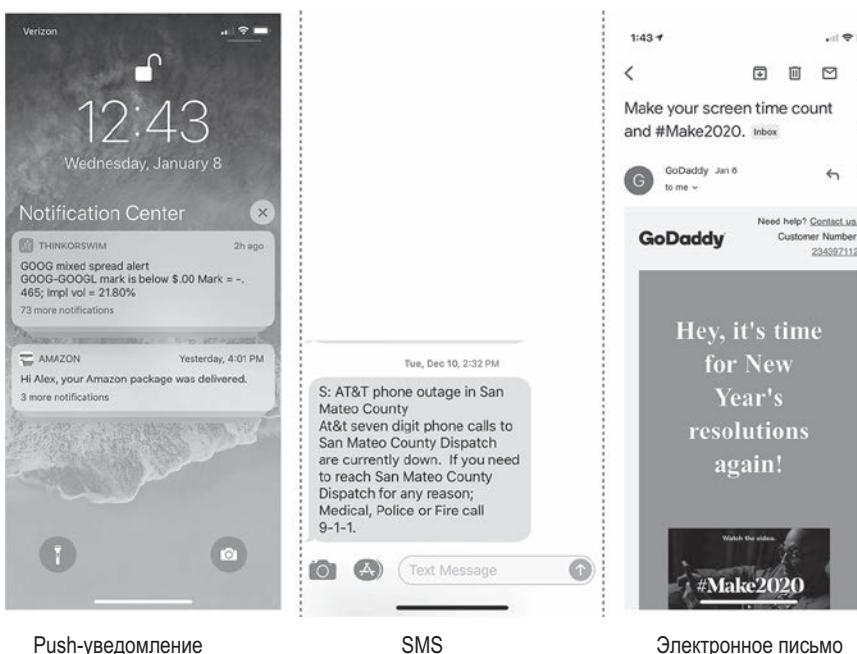


Рис. 10.1

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Создание масштабируемой системы, которая ежедневно отправляет миллионы уведомлений, — непростая задача, которая требует глубокого понимания соответствующей экосистемы. Эта задача специально сформулирована так, чтобы допускать свободную трактовку. Чтобы прояснить все требования, вам необходимо задать уточняющие вопросы.

**Кандидат:** «Какого рода уведомления поддерживает система?»

**Интервьюер:** «Push-уведомления, SMS-сообщения и электронные письма».

**Кандидат:** «Это система реального времени?»

**Интервьюер:** «Пусть это будет система мягкого реального времени. Мы хотим, чтобы пользователь получал уведомления как можно раньше. Но, если система сильно нагружена, допускаются небольшие задержки».

**Кандидат:** «Какие устройства поддерживаются?»

**Интервьюер:** «Устройства iOS и Android, а также ноутбуки и ПК».

**Кандидат:** «Что инициирует уведомления?»

**Интервьюер:** «Уведомления могут быть инициированы клиентскими приложениями или запланированы на стороне сервера».

**Кандидат:** «Есть ли у пользователей возможность отписаться от уведомлений?»

**Интервьюер:** «Да, пользователи, решившие отписаться, больше не будут получать уведомления».

**Кандидат:** «Сколько уведомлений отправляется ежедневно?»

**Интервьюер:** «10 миллионов мобильных push-уведомлений, 1 миллион SMS-сообщений и 5 миллионов электронных писем».

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

В этом разделе представлена общая архитектура с поддержкой разных типов уведомлений: push-уведомлений для iOS и Android, SMS-сообщений и электронных писем. Мы рассмотрим следующие темы:

- разные типы уведомлений;
- процесс сбора контактной информации;
- процесс отправки/получения уведомлений.

## Разные типы уведомлений

Для начала рассмотрим общий принцип работы уведомлений каждого типа.

### Push-уведомления для iOS

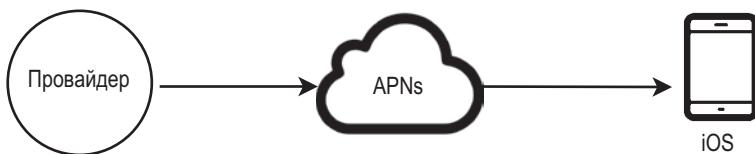


Рис. 10.2

Отправка push-уведомлений для iOS требует наличия трех основных компонентов.

- Провайдер. Провайдер формирует и отправляет запрос сервису APN (Apple Push Notification). Для создания уведомления он использует следующие данные:
  - ◆ маркер устройства – уникальный идентификатор, который используется для отправки push-уведомлений;
  - ◆ полезные данные — словарь JSON с содержимым уведомления.
 Например:

```
{
  "aps": {
    "alert": {
      "title": "Game Request",
      "body": "Bob wants to play chess",
      "action-loc-key": "PLAY"
    },
    "badge": 5
  }
}
```

- APN – удаленный сервис, предоставляемый компанией Apple для доставки push-уведомлений на устройства iOS.
- Устройство iOS – конечный клиент, который получает push-уведомления.

### Push-уведомления для Android

Для отправки уведомлений устройствам Android используется похожий механизм, но вместо APN обычно применяется сервис FCM (Firebase Cloud Messaging).

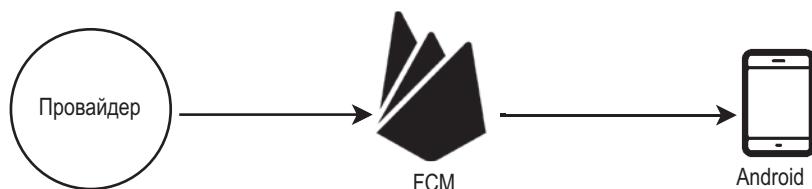


Рис. 10.3

### SMS-сообщения

Для рассылки SMS-сообщений обычно используются такие сторонние сервисы, как Twilio [1], Nexmo [2] и многие другие. Большинство из них являются коммерческими.

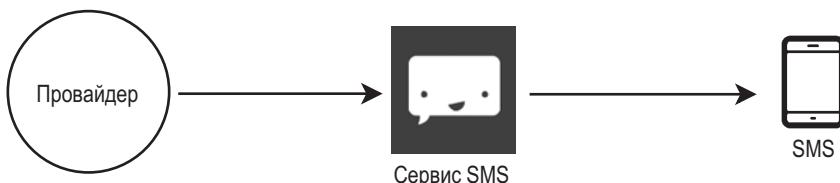


Рис. 10.4

### Электронные письма

У компаний есть возможность сконфигурировать собственные почтовые серверы, но многие из них отдают предпочтение коммерческим сервисам. Sendgrid [3] и Mailchimp [4] – одни из самых популярных; они обеспе-

чидают повышенную скорость доставки и включают системы анализа данных.



Рис. 10.5

На рис. 10.6 показана архитектура с поддержкой всех сторонних сервисов.

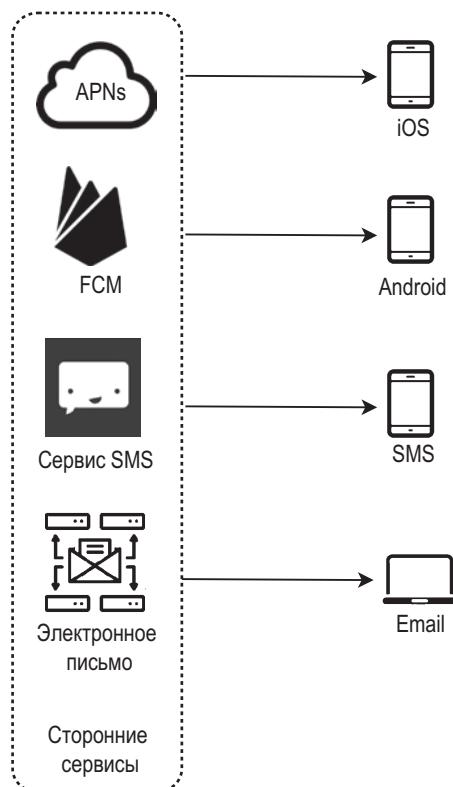


Рис. 10.6

## Процесс сбора контактной информации

Для отправки уведомлений нам нужно собрать маркеры, телефонные номера или адреса электронной почты мобильных устройств. Как показано на рис. 10.7, при установке нашего приложения или во время регистрации пользователь предоставляет контактную информацию, которую серверы API сохраняют в базе данных.

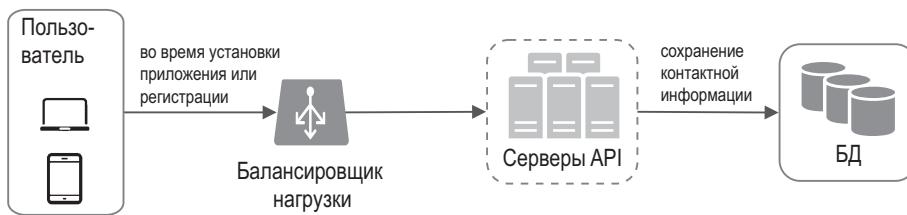


Рис. 10.7

На рис. 10.8 показаны упрощенные таблицы базы данных для хранения контактной информации. Адреса электронной почты и телефонные номера хранятся в таблице user, а маркеры устройств — в таблице device. У пользователя может быть несколько устройств, а уведомления могут отправляться каждому из них.

user	
user_id	bigint
email	varchar
country_code	integer
phone_number	integer
created_at	timestamp

device	
id	bigint
device_token	varchar
user_id	bigint
last_logged_in_at	timestamp

Рис. 10.8

## Процесс отправки/получения уведомлений

Сначала рассмотрим общую архитектуру, а затем предложим некоторые оптимизации.

## Общая архитектура

На рис. 10.9 представлена общая архитектура. Ниже мы разберем каждый ее компонент.

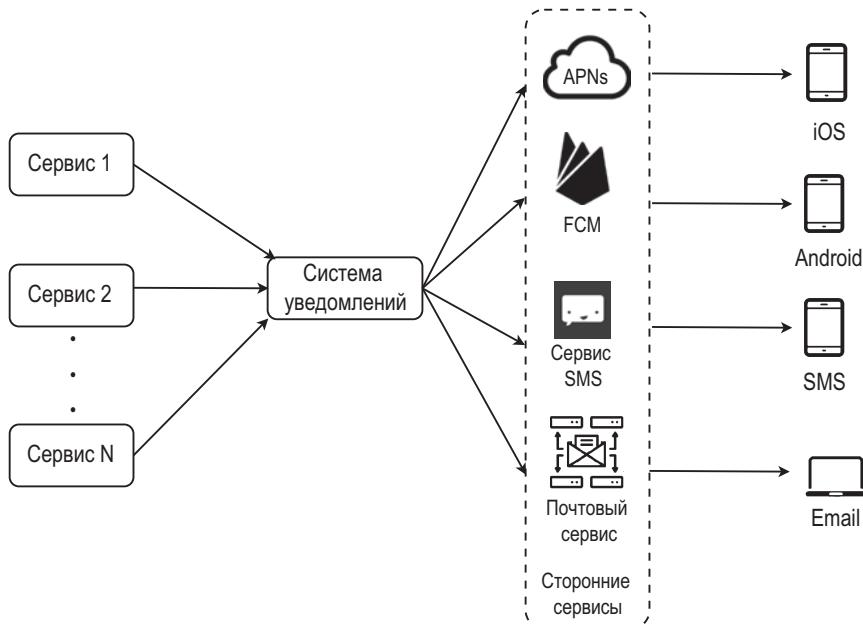


Рис. 10.9

- **Сервисы от 1 до N.** Это могут быть микросервисы, задания cron или распределенная система, которая инициирует отправку уведомлений с помощью событий. Например, система биллинга отправляет клиентам электронные письма с напоминаниями о необходимости оплаты, а интернет-магазин отправляет покупателям SMS-сообщения с информацией о дате доставки товара.
- **Система уведомлений.** Это главный механизм отправки/получения уведомлений. Чтобы начать с чего-то простого, будем использовать один сервер, который предоставляет API-интерфейсы для сервисов от 1 до N и формирует содержимое уведомлений для сторонних сервисов.

- **Сторонние сервисы.** Сторонние сервисы отвечают за доставку уведомлений пользователем. При интеграции с ними нужно уделять внимание расширяемости. Благодаря расширяемости система становится гибкой и позволяет легко подключать и отключать сторонние сервисы. Еще один важный фактор: сторонний сервис может стать недоступным в будущем или при выходе на новые рынки. Например, сервис FCM недоступен в Китае, поэтому вместо него там используются такие альтернативы, как Jpush, PushY и т. д.
- **iOS, Android, SMS, электронные письма.** Пользователи получают уведомления на свои устройства.

У этой архитектуры есть три проблемы.

- Единая точка отказа из-за наличия лишь одного сервера.
- Плохая масштабируемость. Все, что касается push-уведомлений, происходит на одном сервере. Нам будет сложно масштабировать базы данных, кэши и компоненты обработки разных уведомлений по отдельности.
- Узкое место производительности. На обработку и отправку уведомлений может уходить много ресурсов. Например, генерация HTML-страниц и ожидание ответов от сторонних сервисов может занять какое-то время. Система, которая берет на себя всю работу, может оказаться перегруженной, особенно в часы пик.

### Улучшенная общая архитектура

С учетом недостатков исходной архитектуры внесем следующие улучшения:

- выносим базы данных и кэш за пределы сервера уведомлений;
- добавляем больше серверов уведомлений и настраиваем автоматическое горизонтальное масштабирование;
- добавляем очереди сообщений для разделения компонентов системы.

Улучшенная общая архитектура показана на рис. 10.10.

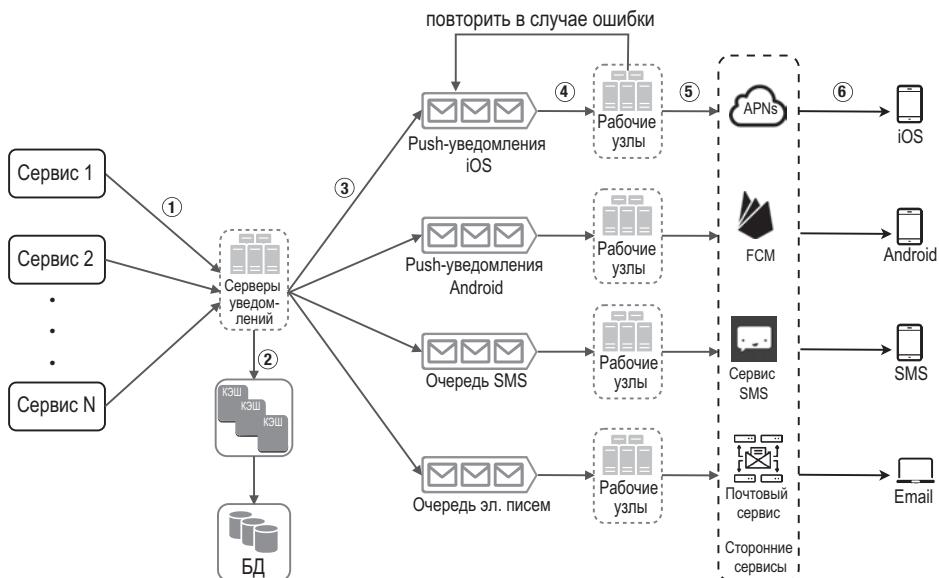


Рис. 10.10

Лучше разобрать эту диаграмму слева направо.

- **Сервисы от 1 до N** представляют разные сервисы, которые используют API серверов уведомлений.
- **Серверы уведомлений.** Они дают следующие возможности:
  - ◆ API для отправки уведомлений, доступные только внутри или для проверенных клиентов (чтобы предотвратить спам);
  - ◆ базовая проверка адресов электронной почты, телефонных номеров и т. д.;
  - ◆ извлечение из БД или информации, необходимой для формирования уведомления;
  - ◆ запись уведомлений в очередь сообщений для параллельной обработки.

Вот пример API-интерфейса для отправки электронного письма:

```
POST https://api.example.com/v/sms/send
```

Тело запроса:

```
{  
  "to": [  
    {  
      "user_id": 123456  
    }  
  ],  
  "from": {  
    "email": "from_address@example.com"  
  },  
  "subject": "Hello, World!"  
  "content": [  
    {  
      "type": "text/plain",  
      "value": "Hello, World"  
    }  
  ]  
}
```

- **Кэш** — пользовательские данные, информация об устройстве и шаблоны уведомлений.
- **БД** хранит данные о пользователях, уведомлениях, настройках и пр.
- **Очереди сообщений** позволяют избавиться от зависимостей между компонентами и играют роль буферов, когда отправляется большой объем уведомлений. Каждому типу уведомлений назначается отдельная очередь, чтобы отказ одного стороннего сервиса не сказался на отправке уведомлений других типов.
- **Рабочие узлы** — список серверов, которые достают события об уведомлениях из очереди сообщений и отправляют их соответствующим сторонним сервисам.
- **Сторонние сервисы.** Они уже обсуждались в описании исходной архитектуры.
- **iOS, Android, SMS, электронные письма.** Они уже обсуждались в описании исходной архитектуры.

Теперь давайте посмотрим, как совместная работа всех этих компонентов позволяет отправить уведомление.

1. Сервис вызывает API, предоставленные серверами уведомлений.

2. Серверы уведомлений извлекают из БД метаданные, такие как информация о пользователе, маркер устройства и настройки.
3. В соответствующую очередь отправляется событие об уведомлении для последующей обработки. Например, событие о push-уведомлении для iOS попадает в очередь iOS PN.
4. Рабочие узлы достают события об уведомлениях из очередей сообщений.
5. Рабочие узлы отправляют уведомления сторонним сервисам.
6. Сторонние сервисы отправляют уведомления на устройства пользователей.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

В разделе, посвященном общей архитектуре, мы обсудили разные типы уведомлений, процесс сбора контактной информации и процесс отправки/получения уведомлений. Теперь углубимся в следующие темы:

- Надежность.
- Дополнительные компоненты и функции: шаблон уведомлений, параметры уведомлений, ограничение трафика, механизм повторных вызовов, безопасность push-уведомлений, мониторинг ожидающих уведомлений и отслеживание событий.
- Обновленная архитектура.

### Надежность

При проектировании системы уведомлений в распределенном окружении необходимо ответить на несколько важных вопросов.

#### Как предотвратить потерю данных?

Одно из важнейших требований к системе уведомлений — она не должна терять данные. Уведомления, как правило, могут задерживаться или приходить в другом порядке, но они никогда не теряются. Для удовлетворения этого требования система записывает уведомления в лог, который

хранится в базе данных, и реализует механизм повторных вызовов. Это показано на рис. 10.11.



Рис. 10.11

### Доходят ли уведомления до получателя строго в единственном экземпляре?

Если коротко, то нет. В большинстве случаев уведомление приходит ровно один раз, но из-за распределенной природы нашей системы порой случается дублирование. Чтобы это происходило реже, мы используем механизм устранения дубликатов и тщательно обрабатываем каждый неудачный случай. Логика в этом случае проста. При поступлении события об уведомлении мы сначала проверяем его ID, чтобы узнать, приходило ли оно раньше. Если мы его уже встречали, оно отклоняется. В противном случае мы его отправляем. Если вас интересует, почему мы не можем гарантировать доставку строго в единственном экземпляре, обратитесь к справочному материалу [5].

### Дополнительные компоненты и функции

Мы уже обсудили то, как собирать контактную информацию пользователей и отправлять/получать уведомления. Но система уведомлений этим не ограничивается. Здесь речь пойдет о дополнительных компонентах и функциях, таких как повторное использование шаблонов, параметры уведомлений, отслеживание событий, система мониторинга, ограничение трафика и т. д.

## Шаблон уведомлений

Крупные системы ежедневно отправляют миллионы уведомлений, многие из которых имеют похожий формат. Чтобы не генерировать их все с нуля, мы используем шаблоны. Шаблон содержит отформатированный текст с возможностью изменения параметров, стилей, ссылок для отслеживания и т. д. Это позволяет создавать уникальные уведомления. Пример показан ниже.

### ТЕЛО СООБЩЕНИЯ:

Вы об этом мечтали. Мы на это решились. [НАЗВАНИЕ ПРОДУКТА] снова в продаже — только до [ДАТА].

### СТА:

Заказать сейчас. Или сохранить [НАЗВАНИЕ ПРОДУКТА].

К преимуществам использования шаблонов уведомлений можно отнести неизменность формата, уменьшение количества ошибок и экономию времени.

## Параметры уведомлений

Количество уведомлений, которые ежедневно получают пользователи, может легко превысить пределы разумного. В связи с этим многие сайты и приложения дают пользователям возможность гибкого управления своими уведомлениями. Эта информация хранится в таблице с параметрами уведомлений, состоящей из следующих полей:

```
user_id bigint
channel varchar # push-уведомления, электронные письма или SMS
opt_in boolean # получать уведомления
```

Прежде чем отправлять пользователю какие-либо уведомления, мы проверяем, желает ли он их получать.

## Ограничение трафика

Чтобы не заваливать пользователя уведомлениями, мы можем ограничить их количество. Это важно, потому что пользователь может вообще отключить все уведомления, если мы будем отправлять их слишком часто.

## Механизм повторных вызовов

Уведомление, которое не удается отправить стороннему сервису, добавляется в очередь сообщений для повторной попытки. Если проблема остается, информация об этом передается разработчикам.

## Безопасность push-уведомлений

В приложениях для iOS и Android защита API-интерфейсов push-уведомлений основана на appKey и appSecret [6]. Отправлять push-уведомления с помощью этих API-интерфейсов могут только аутентифицированные или проверенные клиенты. Подробнее об этом можно почитать в справочном материале [6].

## Мониторинг отложенных уведомлений

Одной из ключевых метрик мониторинга является общее число ожидающих уведомлений. Если оно становится большим, это означает, что рабочие узлы не успевают обрабатывать события об уведомлениях. Чтобы не задерживать доставку уведомлений, нужно добавить больше рабочих узлов. На рис. 10.12 (см. [7]) показан пример сообщений, ожидающих обработки.

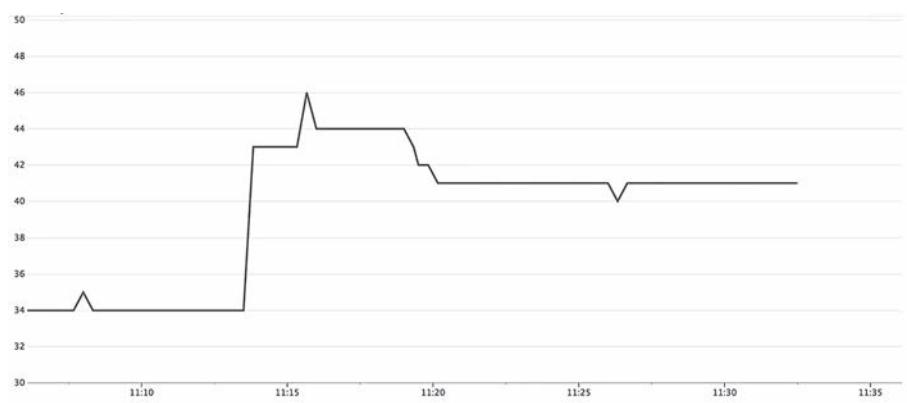


Рис. 10.12

## Отслеживание событий

Чтобы лучше понимать поведение пользователей, необходимо отслеживать такие метрики, как процент открытия уведомлений, процент кликов

и вовлеченность. Отслеживание событий реализует аналитический сервис, который обычно должен быть интегрирован в систему уведомлений. На рис. 10.13 показан пример событий, которые могут отслеживаться для последующего анализа.

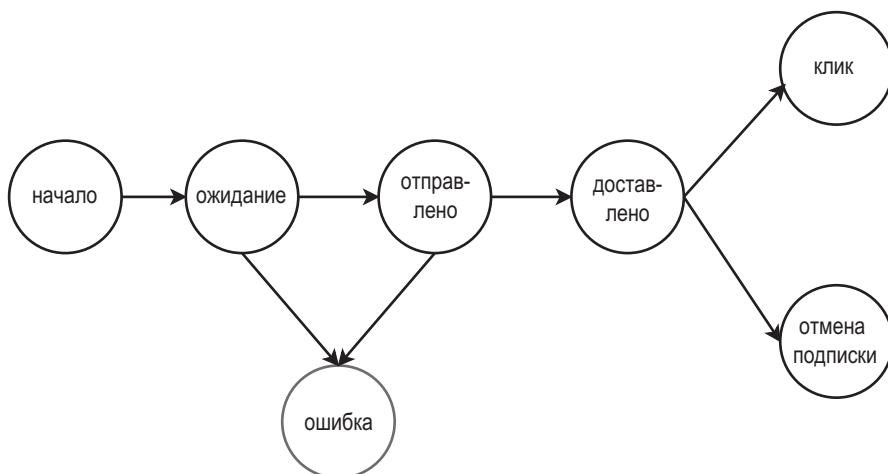


Рис. 10.13

### Обновленная архитектура

Если собрать все воедино, получится обновленная архитектура системы уведомлений, показанная на рис. 10.14.

По сравнению с предыдущей архитектурой здесь появилось много новых компонентов.

- Серверы уведомлений снабжены еще двумя важными функциями — аутентификацией и ограничением трафика.
- Мы также добавили механизм повторных вызовов для обработки ошибок при отправке уведомлений. Уведомления, которые не удается отправить, помещаются обратно в очередь сообщений, а рабочие узлы выполняют определенное количество повторных попыток.
- Шаблоны позволяют сделать процесс создания уведомлений согласованным и эффективным.

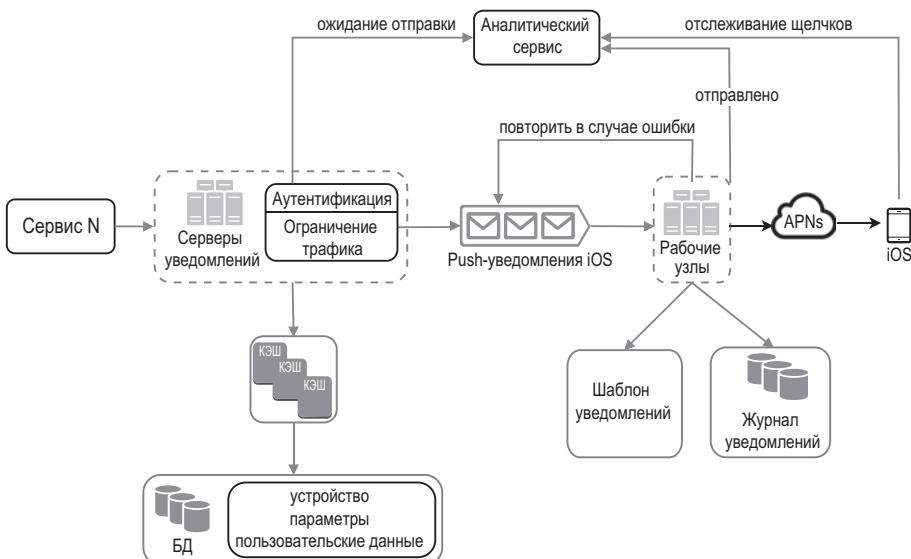


Рис. 10.14

- Наконец, были добавлены механизмы мониторинга и отслеживания для проверки работоспособности системы и ее дальнейшего улучшения.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

Уведомления позволяют оперативно доставлять важную информацию, что делает их незаменимыми. Это может быть push-уведомление о вашем любимом фильме Netflix, электронное письмо о скидках на новые продукты или сообщение с подтверждением оплаты в онлайн-магазине.

В этой главе мы обсудили проектирование масштабируемой системы уведомлений с поддержкой нескольких форматов: push-уведомлений, SMS-сообщений и электронных писем. Для разделения компонентов системы использовалась очередь сообщений.

Определившись с общей архитектурой, мы подробно рассмотрели дополнительные компоненты и оптимизации.

- Надежность. Мы предложили надежный механизм повторных вызовов для минимизации частоты сбоев.
- Безопасность. Чтобы уведомления могли отправлять только проверенные клиенты, мы использовали пару AppKey/appSecret.
- Отслеживание и мониторинг. Эти механизмы могут быть реализованы на любом этапе процесса отправки уведомлений для получения полезной статистики.
- Соблюдение пользовательских настроек. Пользователи могут отказаться от рассылки. Прежде чем отправлять уведомления, наша система проверяет пользовательские настройки.
- Ограничение трафика. Пользователи будут призательны, если уведомления не будут сыпаться на них ежеминутно.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Twilio SMS: <https://www.twilio.com/sms>
- [2] Nexmo SMS: <https://www.nexmo.com/products/sms>
- [3] Sendgrid: <https://sendgrid.com/>
- [4] Mailchimp: <https://mailchimp.com/>
- [5] You Cannot Have Exactly-Once Delivery: <https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/>
- [6] Security in Push Notifications: <https://cloud.ibm.com/docs/services/mobilepush?topic=mobilepushnotification-security-in-push-notifications>
- [7] Key metrics for RabbitMQ: [www.datadoghq.com/blog/rabbitmq-monitoring](http://www.datadoghq.com/blog/rabbitmq-monitoring)

---

# 11

---

## ПРОЕКТИРОВАНИЕ ЛЕНТЫ НОВОСТЕЙ

В этой главе вам предлагается спроектировать ленту новостей. Согласно справке Facebook, «Лента новостей — это постоянно обновляемый список историй в центральной части вашей домашней страницы. Она включает обновления информации о состоянии, фотографии, видео, ссылки, активность приложений и лайки, приходящие от людей, страниц и групп, на которые вы подписаны в Facebook» [1]. Эта задача часто встречается на интервью. Другие ее разновидности могут заключаться в проектировании новостной ленты Facebook, ленты Instagram, потока сообщений Twitter и т. д.



Рис. 11.1

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Когда вас просят спроектировать ленту новостей, начинать следует с уточняющих вопросов, которые помогут вам понять, что именно имеет в виду интервьюер. Вам как минимум нужно определиться со списком возможностей, которые должны поддерживаться. Вот пример диалога между кандидатом и интервьюером:

**Кандидат:** «Это приложение должно быть мобильным, браузерным или и тем и другим?»

**Интервьюер:** «И тем и другим».

**Кандидат:** «Какие его основные возможности?»

**Интервьюер:** «Пользователь может публиковать посты и просматривать ленту новостей с постами его друзей».

**Кандидат:** «Лента новостей сортируется в обратном хронологическом порядке или по какому-то другому принципу? Например, статьи от близких друзей могут иметь повышенный приоритет».

**Интервьюер:** «Чтобы не усложнять, предположим, что лента сортируется в обратном хронологическом порядке».

**Кандидат:** «Сколько друзей может быть у пользователя?»

**Интервьюер:** «5000».

**Кандидат:** «Какой объем трафика?»

**Интервьюер:** «10 миллионов DAU».

**Кандидат:** «Может ли лента в дополнение к тексту содержать изображения и видео?»

**Интервьюер:** «Да, она может содержать медиафайлы, включая изображения и видео».

Уточнив требования, мы можем переходить к проектированию системы.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Архитектура состоит из двух процессов — публикации постов и формирования новостной ленты.

- Публикация постов. Когда пользователь публикует пост, соответствующие данные записываются в кэш и в базу данных. Затем пост появляется в ленте новостей друзей пользователя.
- Составление новостной ленты. Чтобы не усложнять, предположим, что лента формируется путем агрегации постов друзей в обратном хронологическом порядке.

## API ленты новостей

API ленты новостей — это основной механизм взаимодействия клиентов с серверами. API основаны на HTTP и позволяют клиентам выполнять такие действия, как публикация обновлений состояния, получение новостной ленты, добавление друзей и т. д. Мы обсудим два основных действия: публикацию постов и получение ленты.

### API для публикации постов

Чтобы опубликовать пост, нужно отправить серверу HTTP-запрос типа POST. Пример:

```
POST /v1/me/feed
```

Параметры:

- content: текст поста;
- auth\_token: используется для аутентификации API-запросов.

### API для получения ленты

API для получения ленты новостей выглядит так:

```
GET /v1/me/feed
```

Параметры:

- auth\_token: используется для аутентификации API-запросов.

## Публикация статей

На рис. 11.2 показан общий принцип публикации ленты.



Рис. 11.2

- Пользователь может просматривать ленты новостей в браузере или в мобильном приложении. Чтобы опубликовать пост с текстом Hello, используется следующий API-вызов:  
`/v1/me/feed?content=Hello&auth_token={auth_token}`
- Балансирующий нагрузки распределяет трафик между серверами.

- Веб-серверы перенаправляют трафик к разным внутренним сервисам.
- Сервис постов сохраняет статьи в базе данных и кэше.
- Сервис ветвления добавляет новый контент в новостные ленты друзей. Данные новостной ленты хранятся в кэше, чтобы их можно было быстро извлечь.
- Сервис уведомлений. Сообщает друзьям о появлении нового контента и рассыпает push-уведомления.

### Составление ленты новостей

В этом разделе мы обсудим внутренний процесс составления новостной ленты. В общих чертах он показан на рис. 11.3.

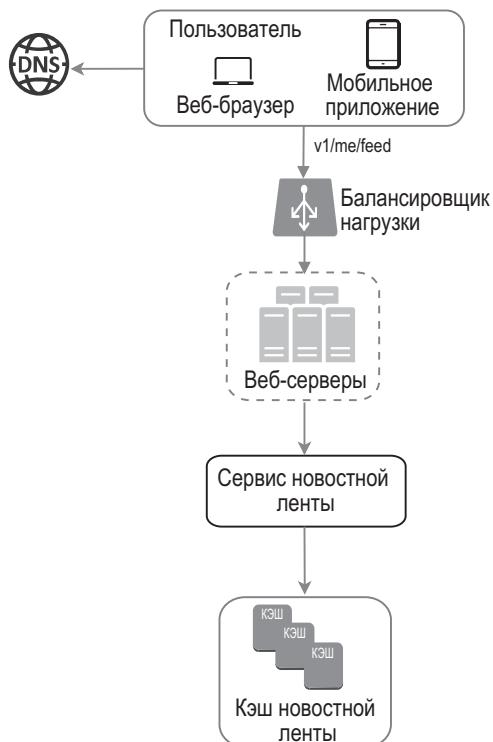


Рис. 11.3

- Пользователь отправляет запрос для получения своей ленты новостей. Запрос имеет вид /v1/me/feed.
- Балансировщик нагрузки направляет трафик к веб-серверам.
- Веб-серверы направляют запросы к серверу новостной ленты.
- Сервис новостной ленты извлекает ленту новостей из кэша.
- Кэш новостной ленты хранит идентификаторы статей, необходимые для составления ленты новостей.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

В предыдущем разделе вкратце описывалось два процесса: публикация постов и составление ленты новостей. Здесь мы подробнее обсудим их.

### Подробно о публикации статей

На рис. 11.4 подробно изображен процесс публикации постов. Большинство компонентов было рассмотрено при обсуждении общей архитектуры. Здесь же мы сосредоточимся на двух из них: веб-серверах и сервисе ветвления.

#### Веб-серверы

Помимо взаимодействия с клиентами, веб-серверы отвечают за аутентификацию и ограничение трафика. Публиковать посты разрешено только пользователям, которые предоставили при входе в систему действительное значение `auth_token`. Система ограничивает количество постов, которое можно публиковать за определенный промежуток времени. Это незаменимое средство борьбы со спамом и нежелательным контентом.

#### Сервис ветвления

Сервис ветвления доставляет посты всем вашим друзьям. Он может быть основан на двух моделях: ветвление при записи (пассивная модель) и ветвление при чтении (активная модель). Оба варианта имеют свои преимущества и недостатки. Давайте посмотрим, как они работают, и выберем тот, который лучше всего подходит для нашей системы.

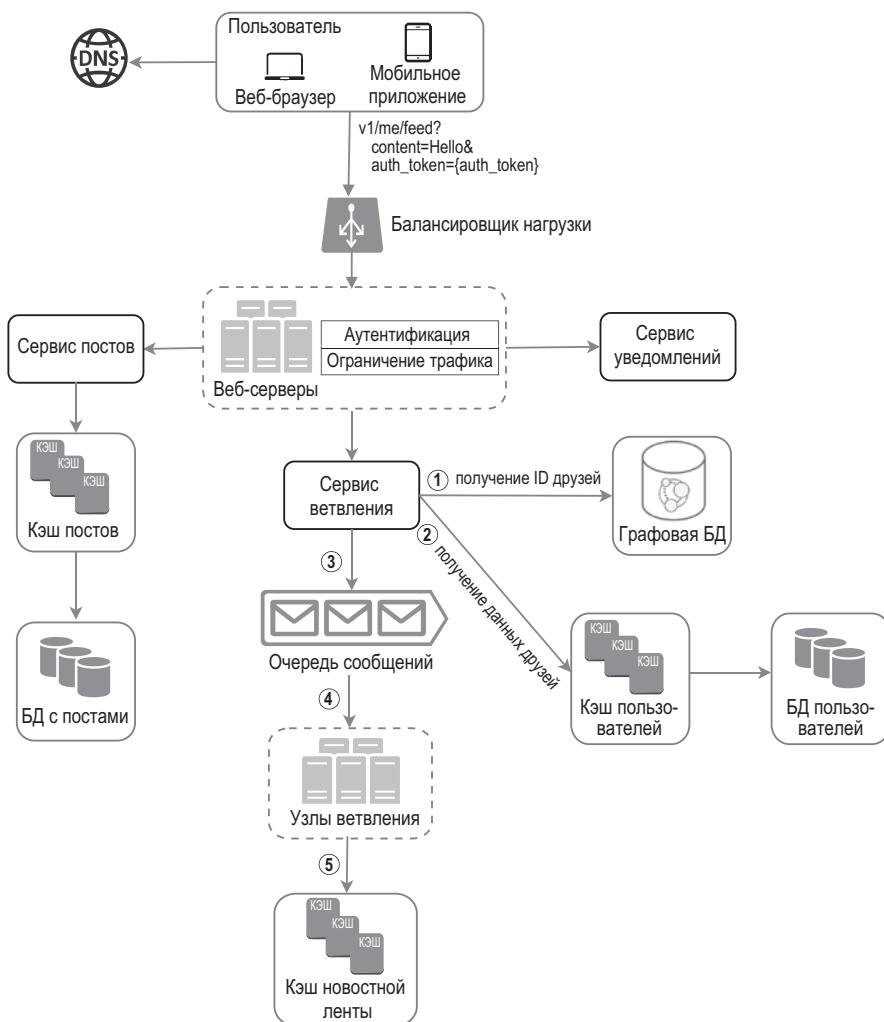


Рис. 11.4

**Ветвление при записи.** В рамках этого подхода лента новостей составляется во время записи. После публикации новый пост сразу же помещается в кэш друзей.

Преимущества:

- новостная лента генерируется в реальном времени и сразу становится доступной для друзей;

- получение новостной ленты происходит быстро, так как она составляется во время записи.

Недостатки:

- если у пользователя много друзей, получение их списка и генерация новостной ленты для каждого из них будет происходить медленно. Это называется проблемой горячих клавиш;
- если пользователь неактивен или редко входит в систему, составление его новостной ленты будет пустой тратой вычислительных ресурсов.

**Ветвление при чтении.** Лента новостей генерируется во время чтения. Последние посты загружаются, когда пользователь загружает домашнюю страницу.

Преимущества:

- ветвление при чтении лучше подходит для пользователей, которые не очень активны или редко входят в систему, так как при этом на них не тратятся лишние ресурсы;
- данные не заносятся в кэш каждого друга, поэтому при большом количестве друзей проблем не возникает.

Недостатки:

- загрузка новостной ленты происходит медленно, поскольку она не составляется заранее.

Мы применим гибридный поход, чтобы получить преимущества обеих моделей и избежать их недостатков. Нам крайне важно, чтобы ленту новостей можно было получить быстро, поэтому для большинства пользователей предусмотрена модель push. Контент знаменитостей и пользователей с большим количеством друзей/подписчиков можно запрашивать по требованию, чтобы не перегружать систему. Согласованное хеширование позволяет справляться с большим количеством друзей за счет более равномерного распределения запросов/данных.

Давайте подробно рассмотрим сервис ветвления, показанный на рис. 11.5.

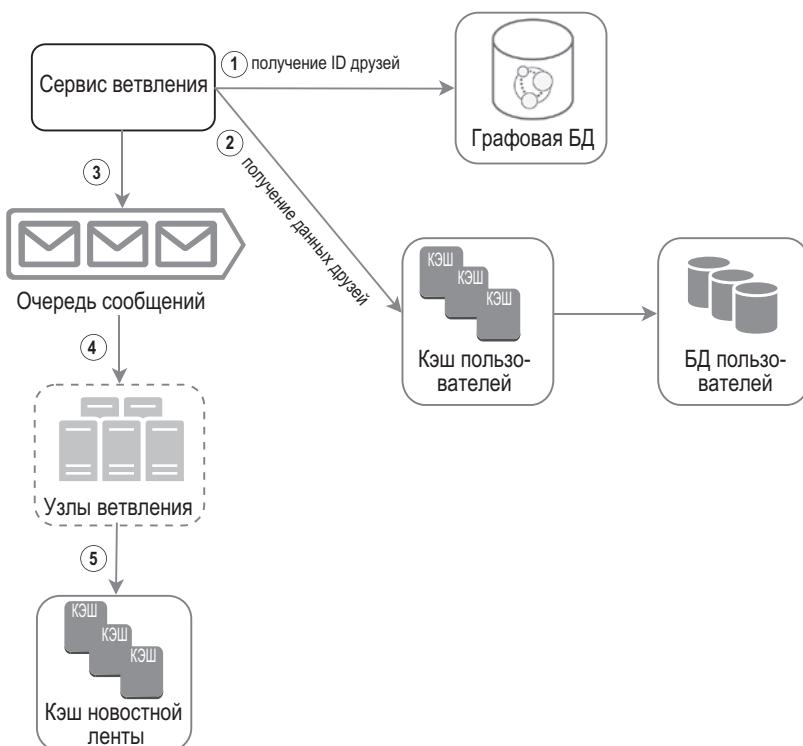


Рис. 11.5

Сервис ветвления работает следующим образом:

1. Извлекаем идентификаторы друзей из графовой базы данных. Графовая БД подходит для управления связями между друзьями и рекомендации новых друзей. Если хотите узнать больше, ознакомьтесь со справочным материалом [2].
2. Получаем информацию о друзьях из кэша пользователей. Система фильтрует полученный список с учетом пользовательских настроек. Например, если вы решили «заглушить» одного из своих друзей, его посты не попадут в вашу ленту. Пост также может быть скрыт по той причине, что пользователь решил поделиться информацией лишь с определенным кругом друзей или скрыть его от других людей.
3. Отправляем список друзей и ID новой статьи в очередь сообщений.

4. Узлы ветвления достают данные из очереди сообщений и сохраняют содержимое ленты новостей в кэше. Кэш ленты новостей можно представить в виде хеш-таблицы `<post_id, user_id>`. Новые посты добавляются в нее в момент создания, как показано на рис. 11.6. Если хранить в кэше данные о пользователях и содержимое постов, расход памяти сильно вырастет. Поэтому мы храним лишь идентификаторы. Чтобы ограничить расход памяти, мы устанавливаем лимит, который можно настраивать. Вероятность того, что пользователь будет прокручивать тысячи постов, невысока. Большинство людей интересуются самым новым контентом, поэтому доля промахов кэша остается низкой.
5. Сохраняем `<post_id, user_id>` в кэш ленты новостей. На рис. 11.6 показан пример того, как может выглядеть закэшированная новостная лента.

post_id	user_id
post_id	user_id

Рис. 11.6

### Подробно о получении ленты новостей

На рис. 11.7 подробно проиллюстрирован процесс получения ленты новостей.

Как видно на рис. 11.7, медиаконтент (изображения, видео и т. д.) хранится в CDN, чтобы его можно было быстро извлекать. Давайте посмотрим, как клиент получает ленту новостей.

1. Пользователь отправляет запрос вида `/v1/me/feed`, чтобы получить свою ленту новостей.

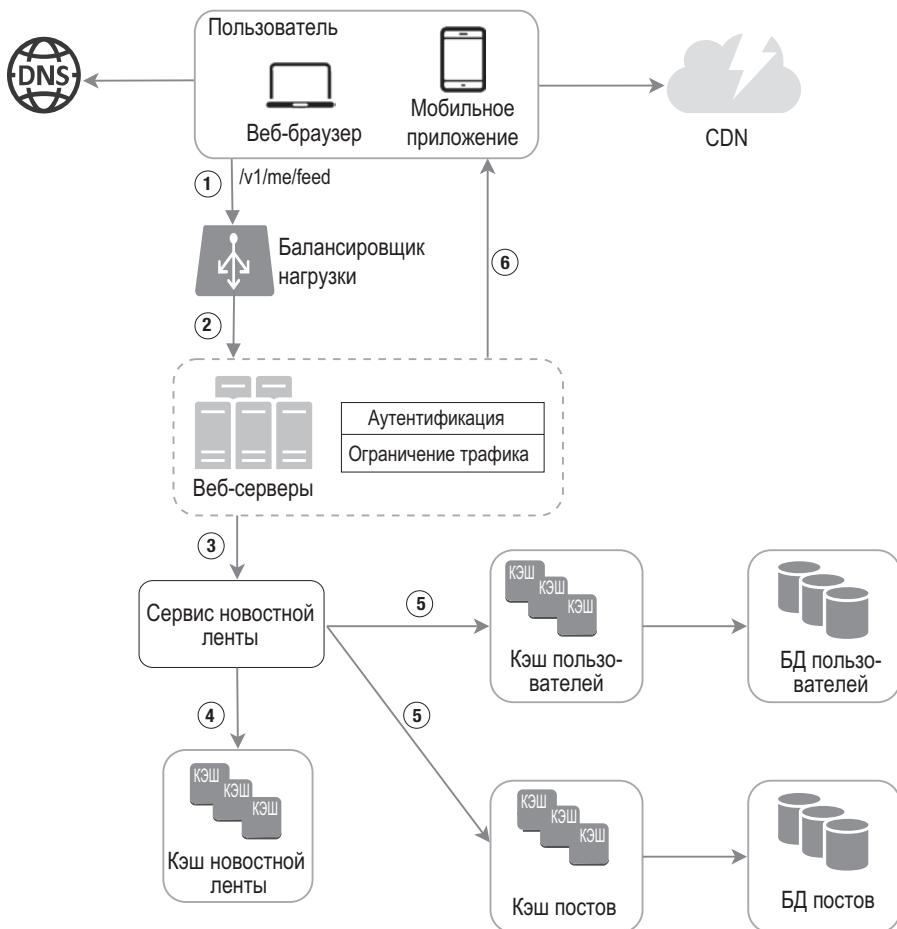


Рис. 11.7

2. Балансировщик нагрузки распределяет запросы между веб-серверами.
3. Веб-серверы обращаются за новостной лентой к соответствующему сервису.
4. Сервис извлекает из кэша новостной ленты список с идентификаторами постов.
5. Лента новостей пользователя не ограничивается списком идентификаторов. Она содержит имена пользователей, аватары, текст

постов, изображения и т. д. Таким образом, сервис новостной ленты извлекает полные данные о пользователях и постах из соответствующих кэшей, чтобы составить полноценную автоматизированную ленту новостей.

- Полноценная лента новостей возвращается на клиент в формате JSON для дальнейшего отображения.

## Архитектура кэширования

Кэш имеет очень большое значение для системы новостных лент. Мы делим его на 5 слоев, как показано на рис. 11.8.



Рис. 11.8

- Лента новостей хранит идентификаторы постов.
- Контент хранит данные каждого поста. Популярный контент находится в горячем кэше.
- Социальный граф хранит данные об отношениях между пользователями.
- Действие хранит информацию о действиях пользователя по отношению к посту: лайкнул, оставил ответ или что-то другое.

- Счетчики включают счетчики лайков, ответов, подписчиков, тех, на кого пользователь подписан, и т. д.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы спроектировали ленту новостей. Наша архитектура состоит из двух процессов: публикации постов и получения новостной ленты.

У задач, которые встречаются на интервью по проектированию ИТ-систем, не бывает идеальных решений, и эта система не исключение. У каждой компании есть уникальные ограничения, которые необходимо учитывать при проектировании. Вы должны понимать сильные и слабые стороны архитектурных и технологических аспектов системы. Если у вас еще остается несколько минут, можете затронуть вопросы масштабирования. Чтобы не повторяться, ниже перечислены только основные тезисы.

Масштабирование базы данных:

- вертикальное и горизонтальное масштабирование;
- SQL и NoSQL;
- репликация вида «ведущий–ведомый»;
- реплики для чтения;
- модели согласованности;
- шардинг базы данных.

Несколько советов:

- не храните состояние веб-уровня;
- кэшируйте данные как можно активнее;
- обеспечьте поддержку разных центров обработки данных;
- ослабьте связанность компонентов с помощью очередей сообщений;
- отслеживайте ключевые метрики, такие как QPS, в часы пик и латентность в момент, когда пользователи обновляют свои новостные ленты.

Поздравляем, вы проделали длинный путь и можете гордиться собой.  
Отличная работа!

## **СПРАВОЧНЫЕ МАТЕРИАЛЫ**

- [1] How News Feed Works: <https://www.facebook.com/help/327131014036297/>
- [2] Friend of Friend recommendations Neo4j and SQL Server: <https://web.archive.org/web/20210116003626/http://geekswithblogs.net/brendonpage/archive/2015/10/26/friend-of-friend-recommendations-with-neo4j.aspx>

---

# 12

---

## ПРОЕКТИРОВАНИЕ СИСТЕМЫ МГНОВЕННОГО ОБМЕНА СООБЩЕНИЯМИ

В этой главе мы исследуем архитектуру системы мгновенного обмена сообщениями или чата. Чатами пользуются почти все. На рис. 12.1 показаны некоторые из самых популярных приложений на рынке.

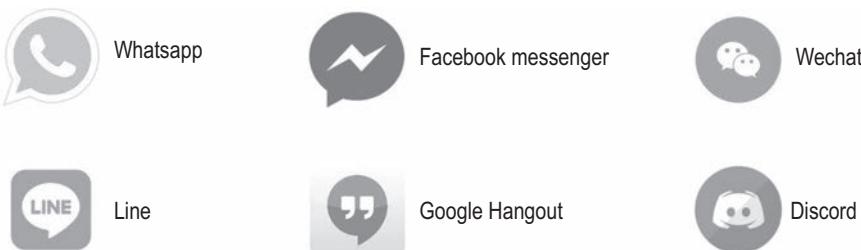


Рис. 12.1

Люди используют чаты для разных задач. Мы обязательно должны определить точные требования к нашей системе. Например, вам вряд ли захочется спроектировать групповой чат, если интервьюер имеет в виду обмен сообщениями между двумя пользователями. Очень важно понять, какие возможности нужно предусмотреть.

### ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Вы обязательно должны определиться с тем, какого рода приложение нужно проектировать. На рынке есть приложения для прямой переписки, такие как Facebook Messenger, WeChat и WhatsApp, офисные системы для группового обмена сообщениями вроде Slack и игровые сервисы наподо-

бие Discord, которые делают акцент на взаимодействии больших групп пользователей и голосовом общении с низкой латентностью.

Первая череда уточняющих вопросов должна прояснить, что именно имеет в виду интервьюер под системой мгновенного обмена сообщениями. Вам как минимум необходимо понять, на взаимодействии какого типа следует сосредоточиться: приватном (между двумя пользователями) или групповом. Ниже даны примерные вопросы.

**Кандидат:** «Какого рода чат мы будем проектировать: приватный (один на один) или групповой?»

**Интервьюер:** «Чат должен поддерживать оба вида взаимодействия».

**Кандидат:** «Это приложение должно быть мобильным, браузерным или и тем и другим?»

**Интервьюер:** «И тем и другим».

**Кандидат:** «Каков масштаб этого приложения? Стартап или огромная система?»

**Интервьюер:** «Оно должно поддерживать 50 миллионов ежедневных активных пользователей (DAU)».

**Кандидат:** «Что касается группового чата, насколько большой может быть группа?»

**Интервьюер:** «Не больше 100 человек».

**Кандидат:** «Какими важными возможностями обладает система? Поддерживает ли она вложения?»

**Интервьюер:** «Приватный и групповой чат, индикатор присутствия в сети. Система поддерживает только текстовые сообщения».

**Кандидат:** «Ограничен ли размер сообщений?»

**Интервьюер:** «Да, длина текста не должна превышать 100 000 символов».

**Кандидат:** «Требуется ли сквозное шифрование?»

**Интервьюер:** «Пока что нет, но мы это обсудим, если время позволит».

**Кандидат:** «Как долго должна храниться история переписки?»

**Интервьюер:** «Вечно».

В этой главе мы сосредоточимся на проектировании приложения наподобие Facebook Messenger с акцентом на следующие возможности:

- приватный чат с низкой латентностью доставки сообщений;
- небольшой групповой чат (не больше 100 человек);
- индикатор сетевого статуса;
- поддержка нескольких устройств; один и тот же пользователь может быть аутентифицирован сразу на нескольких устройствах;
- push-уведомления.

Также необходимо определиться с масштабом системы. Наша архитектура должна поддерживать 50 миллионов DAU.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Чтобы разработать высококачественную архитектуру, необходимо иметь общее представление о взаимодействии между клиентами и серверами. Клиентские приложения могут быть либо мобильными, либо браузерными. Они не взаимодействуют друг с другом напрямую. Вместо этого каждый клиент подключается к сервису чата, который поддерживает все вышеупомянутые возможности. Давайте обсудим основные операции. Сервис чата должен поддерживать следующие функции:

- прием сообщений от других клиентов;
- поиск подходящего получателя для каждого сообщения и передача сообщений получателям;
- временное хранение сообщений на сервере, если их получатель не в сети.

На рис. 12.2 показаны отношения между клиентами (отправителем и получателем) и сервисом чата.

Когда клиент хочет начать общение, он соединяется с сервисом чата, используя один или несколько сетевых протоколов. Для сервиса чата выбор протокола имеет значение. Давайте обсудим это с интервьюером.

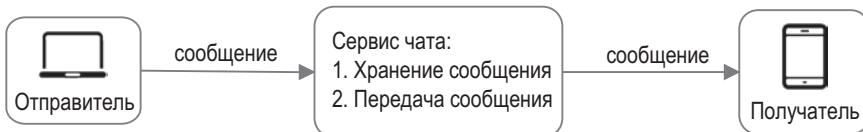


Рис. 12.2

В большинстве клиент-серверных приложений запросы инициируются клиентом. То же самое относится и к отправляющей стороне нашего приложения. На рис. 12.2 видно, что для передачи сообщения отправитель использует проверенный временем протокол HTTP, который чаще всего встречается в интернете. В этом случае клиент открывает HTTP-соединение с сервисом чата и отправляет сообщение, которое сервис должен передать получателю. Для этого хорошо подходит заголовок `Keep-Alive`, который позволяет клиенту поддерживать постоянное соединение с сервисом чата. Он также уменьшает количество TCP-согласований. Протокол HTTP хорошо работает на стороне сервера, и многие системы обмена сообщениями, такие как Facebook [1], изначально использовали именно его.

Однако на стороне получателя все немного сложнее. Поскольку в HTTP соединение инициирует клиент, отправить сообщение с сервера не так-то просто. С годами было выработано множество способов имитации соединения, инициированного сервером: HTTP-опрос, длинный HTTP-опрос и WebSocket. Это важные методики, которые широко применяются в интервью по проектированию ИТ-систем. Давайте рассмотрим каждую из них.

## HTTP-опрос

Как видно на рис. 12.3, HTTP-опрос (polling) состоит в том, что клиент периодически спрашивает сервер о наличии новых сообщений. В зависимости от частоты опроса этот подход может быть затратным. Драгоценные ресурсы сервера могут уходить на возвращение ответа, который в большинстве случаев не несет в себе никакой полезной информации.

## Длинный HTTP-опрос

Поскольку HTTP-опрос может быть неэффективным, его логическим развитием является длинный HTTP-опрос (long polling) (рис. 12.4).

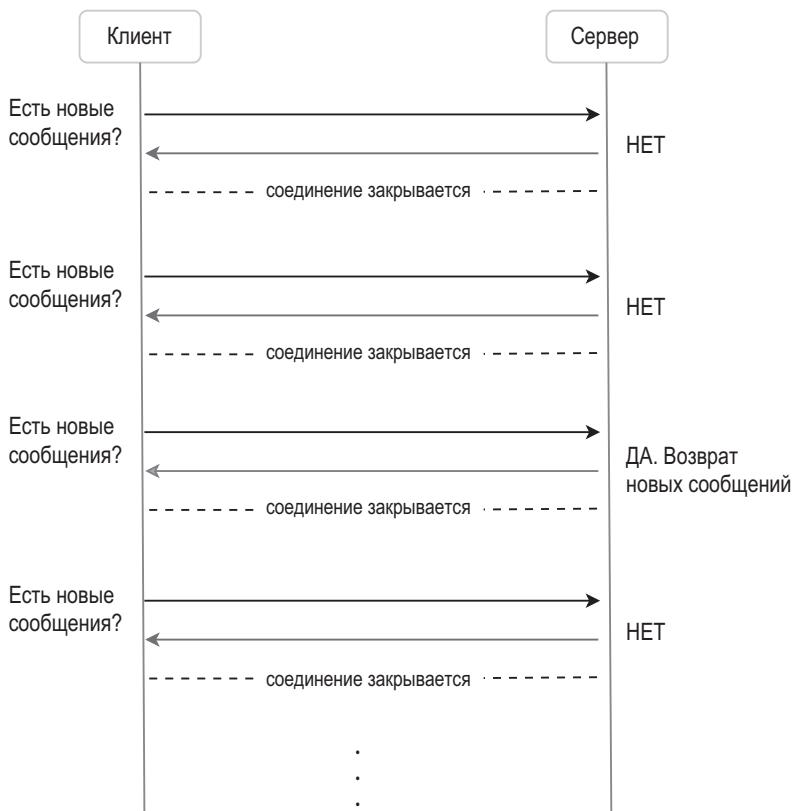


Рис. 12.3

При длинном HTTP-опросе клиент оставляет соединение открытым, пока не появятся новые сообщения или пока не истечет время ожидания. Получив новые сообщения, клиент немедленно отправляет серверу еще один запрос, повторяя весь процесс заново. У длинного HTTP-опроса есть несколько недостатков.

- Отправитель и получатель могут быть подключены к разным серверам чата. HTTP-серверы обычно не хранят свое состояние. Если вы балансируете нагрузку путем циклического перебора, у сервера, принявшего сообщение, может не быть соединения с клиентом, которому это сообщение направлено.
- У сервера нет хорошего механизма для определения того, отключился ли клиент.

- Это неэффективный подход. Если пользователь не слишком активен, время ожидания будет периодически истекать и соединение будет устанавливаться заново.



Рис. 12.4

## WebSocket

WebSocket — это наиболее распространенное решение для передачи асинхронных обновлений от сервера к клиенту. Принцип его работы показан на рис. 12.5.

Соединение по WebSocket инициируется клиентом. Оно является двунаправленным и постоянным. Все начинается с HTTP-соединения, которое можно «модернизировать» до WebSocket с помощью определенной процедуры согласования. По этому постоянному соединению сервер может отправлять обновления клиенту. Соединения по WebSocket обычно работают даже при наличии брандмауэра. Все благодаря тому, что они используют порты 80 или 443, принадлежащие протоколам HTTP/HTTPS.

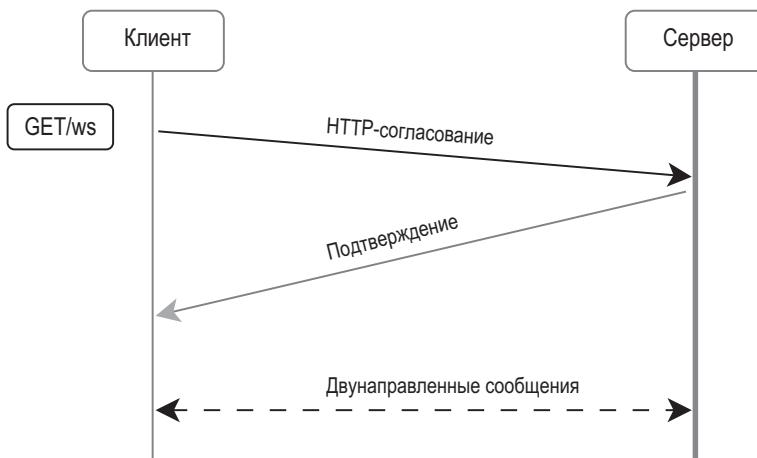


Рис. 12.5

Ранее мы упоминали, что HTTP хорошо подходит для использования на стороне отправителя, однако протокол WebSocket является двунаправленным, поэтому с технической точки зрения нам ничего не мешает применять его и для получения сообщений. На рис. 12.6 показано, как WebSocket (ws) работает на стороне отправителя и получателя.

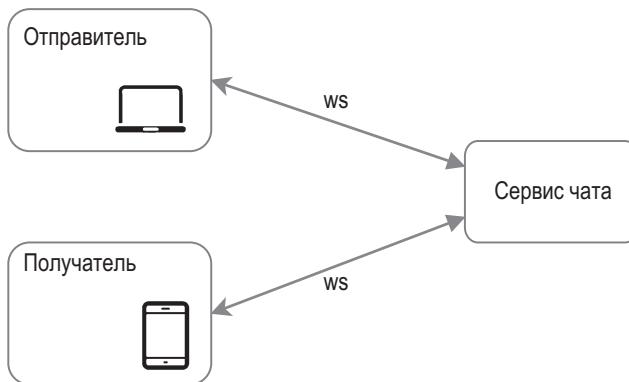


Рис. 12.6

Использование WebSocket как для отправки, так и для получения сообщений упрощает архитектуру и делает реализацию клиента и сервера более

понятной. Поскольку соединение по WebSocket является постоянным, нам нужно позаботиться об его эффективном управлении на серверной стороне.

## Общая архитектура

Мы только что упомянули, что выбор WebSocket в качестве основного протокола взаимодействия между клиентом и сервером обусловлен его двунаправленностью. Но необходимо также отметить, что его необязательно использовать для всего остального. На самом деле большинство возможностей приложения (регистрация, вход в систему, профили пользователей и т. д.) могут быть реализованы в традиционном стиле «запрос–ответ» по протоколу HTTP. Давайте немного углубимся в этот вопрос и поговорим о том, из каких компонентов состоит наша система.

Как показано на рис. 12.7, чат разделен на три основных части: сервисы без состояния, сервисы с состоянием и интеграция со сторонними сервисами.

### **Сервисы без сохранения состояния**

Сервисы без сохранения состояния традиционно используются для взаимодействия с клиентами по принципу «запрос–ответ». На их основе реализованы такие функции, как вход в систему, регистрация, профили пользователей и т. д. Эти возможности присутствуют во многих веб-сайтах и приложениях.

Сервисы без сохранения состояния находятся за балансировщиком нагрузки. Последний отвечает за маршрутизацию запросов и выбор подходящего сервиса с учетом указанного пути. Эта часть системы может быть как монолитной, так и разделенной на отдельные микросервисы. Многие из этих сервисов не нужно создавать с нуля, так как на рынке уже есть решения, которые можно легко интегрировать. Мы подробно остановимся на механизме обнаружения сервисов. Основная задача этого компонента состоит в возвращении клиенту списка доменных имен, принадлежащих серверам чата, к которым можно подключиться.

### **Сервисы с сохранением состояния**

Единственный сервис, который хранит свое состояние, — это чат. Это обусловлено тем, что каждый клиент поддерживает постоянное сетевое

соединение с сервером чата. Пока сервер остается доступным, клиент обычно не переходит на другой сервер. Чтобы серверы не перегружались, механизм обнаружения сервисов координирует свою работу с чатом. В следующем разделе мы обсудим это подробнее.



Рис. 12.7

## Интеграция со сторонними сервисами

Для такого приложения, как чат, самым важным сторонним сервисом являются push-уведомления. Они позволяют информировать пользователей о новых сообщениях, даже когда приложение не работает. Интеграция push-уведомлений крайне важна. Подробнее об этом можно почитать в главе 10 «Проектирование системы уведомлений».

## Масштабируемость

Если масштаб небольшой, все сервисы, перечисленные выше, можно уместить на одном сервере. Но даже при таком масштабе все пользовательские соединения теоретически могут обрабатываться одним современным облачным сервером. Ограничивающим фактором, скорее всего, будет количество параллельных соединений. В нашем случае речь идет об 1 миллионе активных пользователей; если предположить, что каждое пользовательское соединение занимает 10 Кб (это очень грубая оценка, которая сильно зависит от выбранного языка программирования), то для того, чтобы все они уместились на одном сервере, потребуется 10 Гб оперативной памяти.

Если мы предложим архитектуру, в которой все находится на одном сервере, это может очень сильно насторожить интервьюера. Ни один разработчик не стал бы использовать единственный сервер для системы такого масштаба, и для этого есть много причин. Самая главная из них — единая точка отказа.

И все же односерверная архитектура вполне может послужить отправной точкой. Просто дайте интервьюеру понять, что это лишь начало. Если собрать воедино все, о чем упоминалось выше, получится улучшенная общая архитектура, показанная на рис. 12.8.

На рис. 12.8 клиент поддерживает постоянное соединение с сервером чата по WebSocket, чтобы обеспечить обмен сообщениями в реальном времени.

- Серверы чата отвечают отправкой/получением сообщений.
- Серверы присутствия следят за тем, находятся ли пользователи в сети.
- Серверы API занимаются всем остальным, включая вход в систему, регистрацию, редактирование профиля и т. д.
- Серверы уведомлений отправляют push-уведомления.

- И наконец, хранилище типа «ключ–значение» используется для хранения истории переписки. Когда пользователь появляется в сети, он видит все предыдущие сообщения.

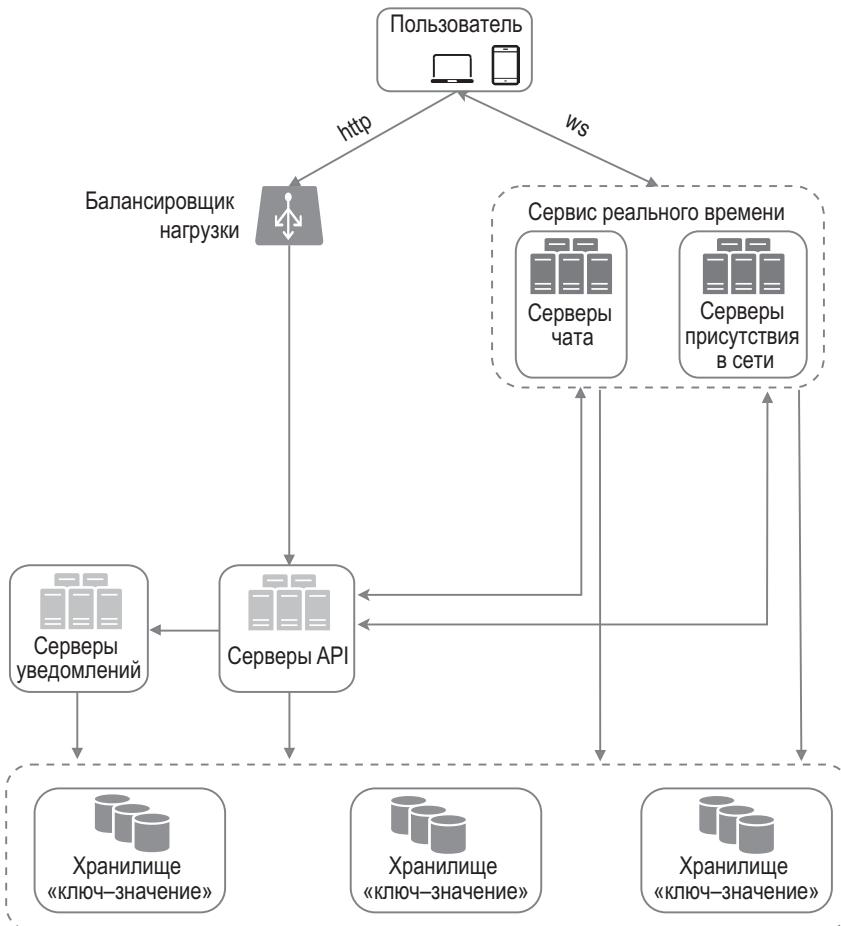


Рис. 12.8

## Хранилище

Итак, мы подготовили наши серверы, запустили сервисы и завершили интеграцию со сторонними системами. Глубоко внутри технологического стека находится уровень данных, для корректного проектирования которого обычно нужно приложить некоторые усилия. Очень важно опреде-

льиться с тем, какая база данных нам лучше подходит: реляционная или NoSQL. Чтобы сделать обоснованный выбор, необходимо исследовать типы данных и модель чтения/записи.

В типичной системе мгновенного обмена сообщениями существует два вида данных. Первый вид, обобщенный, включает в себя профили пользователей, настройки и списки друзей. Такие данные хранятся в устойчивой и надежной реляционной БД. Для соответствия требованиям доступности и масштабируемости обычно применяются репликация и сегментирование.

Второй вид данных встречается только в системах мгновенного обмена сообщениями. Здесь важно понять модель чтения/записи.

- Такие системы обрабатывают огромные объемы данных. Согласно исследованию [2], через Facebook Messenger и WhatsApp проходит больше 60 миллиардов сообщений в день.
- Активный доступ осуществляется только к недавним чатам. Пользователи обычно не возвращаются к старым перепискам.
- В большинстве случаев просматривается только самая свежая история сообщений, но пользователи могут обращаться к функциям, требующим произвольного доступа, таким как поиск, просмотр сообщений, в которых упоминается ваше имя, переход к определенным сообщениям и т. д. Все эти возможности должны поддерживаться на уровне доступа к данным.
- В приватных чатах чтение и запись происходят примерно с одинаковой частотой.

Выбор подходящей системы хранения, которая поддерживает все эти сценарии использования, имеет большое значение. Мы советуем использовать хранилища типа «ключ–значение» по следующим причинам:

- хранилища типа «ключ–значение» легко поддаются горизонтальному масштабированию;
- хранилища типа «ключ–значение» имеют низкую латентность обращения к данным;
- реляционные БД плохо справляются с длинными последовательностями данных [3]. С увеличением индекса замедляется произвольный доступ.

- хранилища типа «ключ–значение» применяются в других надежных системах мгновенного обмена сообщениями, проверенных временем. Например, они используются в Facebook Messenger (HBase [4]) и Discord (Cassandra [5]).

## Модели данных

Мы только что обсудили использование хранилищ типа «ключ–значение» в качестве уровня данных. Самыми важными данными являются сообщения. Давайте рассмотрим их подробнее.

### Таблица сообщений для приватного чата

На рис. 12.9 показана таблица сообщений для приватного чата. Первичный ключ, `message_id`, помогает определить порядок следования сообщений. Мы не можем полагаться в этом на поле `created_at`, поскольку два разных сообщения могут быть созданы одновременно.

message	
<code>message_id</code>	<code>bigint</code>
<code>message_from</code>	<code>bigint</code>
<code>message_to</code>	<code>bitint</code>
<code>content</code>	<code>text</code>
<code>created_at</code>	<code>timestamp</code>

Рис. 12.9

### Таблица сообщений для группового чата

На рис. 12.10 показана таблица сообщений для группового чата. Составной первичный ключ имеет вид (`channel_id`, `message_id`). В этом контексте канал и группа являются синонимами. Все запросы в групповом чате выполняются в рамках канала, поэтому ключом раздела является `channel_id`.

group_message	
<b>channel_id</b>	bigint
<b>message_id</b>	bigint
user_id	bigint
content	text
created_at	timestamp

Рис. 12.10

### ID-сообщения

Стоит поговорить о том, как генерируется `message_id`. Это поле обеспечивает правильный порядок вывода сообщений. Для этого оно должно удовлетворять следующим двум требованиям:

- идентификаторы должны быть уникальными;
- идентификаторы должны поддерживать сортировку по времени, то есть у новых строк идентификаторы должны быть больше, чем у старых.

Как реализовать эти два свойства? Первое, что приходит на ум, это ключевое слово `auto_increment` из MySQL. Однако в базах данных NoSQL такой возможности обычно нет.

Еще один подход состоит в использовании глобального генератора последовательных 64-битных чисел вроде Snowflake [6]. Это обсуждалось в главе 7 «Проектирование генератора уникальных идентификаторов в распределенных системах».

Последним вариантом будет использование локального генератора последовательных чисел. Локальным его делает то, что идентификаторы уникальны только в пределах группы. Этот подход работает, потому что сообщения достаточно упорядочивать на уровне приватного канала или группы. Локальные ID легче реализовать по сравнению с глобальными.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

В ходе интервью по проектированию ИТ-систем от кандидата обычно ожидают подробного анализа некоторых компонентов общей архитектуры. В случае с системой мгновенного обмена сообщениями отдельного внимания заслуживают механизм обнаружения сервисов, маршруты прохождения сообщений и индикатор сетевого статуса собеседника.

### Обнаружение сервисов

Основная задача механизма обнаружения сервисов — предложить клиенту лучший сервер чата с учетом таких критериев, как географическое местоположение, емкость сервера и т. д. Популярное решение — система с открытым исходным кодом Apache Zookeeper [7]. Она регистрирует все доступные серверы чата и выбирает из них тот, который лучше всего соответствует заранее заданным критериям.

Принцип работы обнаружения сервисов (на примере Zookeeper) показан на рис. 12.11.

1. Пользователь А пытается войти в приложение.
2. Балансировщик нагрузки передает запрос входа в систему серверам API.
3. Когда внутренняя часть системы аутентифицирует пользователя, механизм обнаружения сервисов подберет для него наиболее подходящий сервер чата. В этом примере выбран сервер 2, и информация о нем возвращается обратно пользователю.
4. Пользователь А подключается к серверу чата 2 по WebSocket.

### Маршруты прохождения сообщений

Давайте посмотрим, как проходят данные по системе мгновенного обмена сообщениями. В этом разделе мы исследуем маршрут прохождения сообщений в приватном и групповом чатах, а также синхронизацию сообщений между разными устройствами.

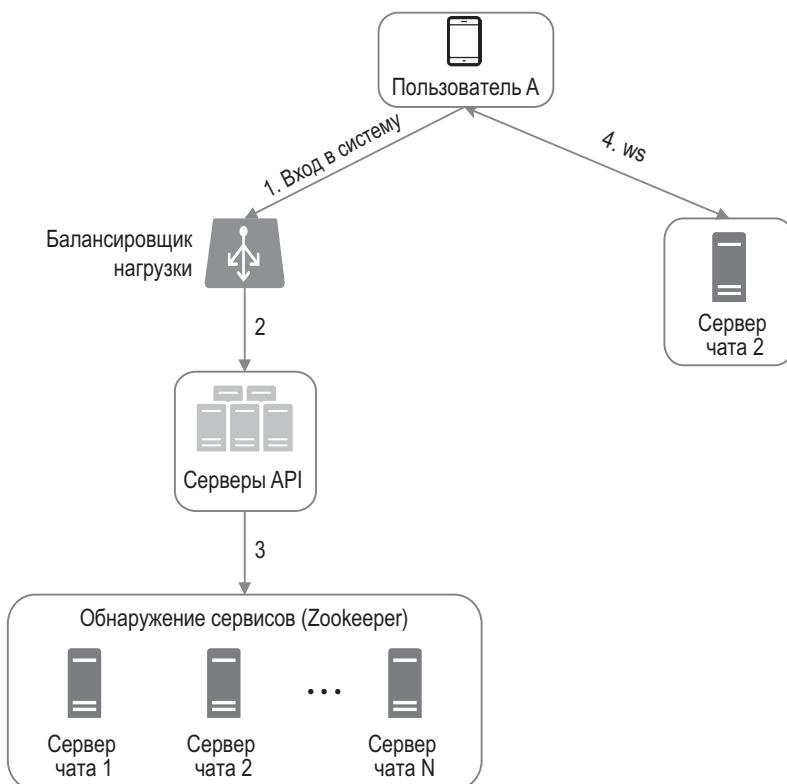


Рис. 12.11

### Маршрут прохождения сообщений в приватном чате

На рис. 12.12 можно видеть, что происходит, когда пользователь А отправляет сообщение пользователю Б.

- Пользователь А отправляет мгновенное сообщение на сервер чата 1.
- Сервер чата 1 получает ID сообщения из генератора идентификаторов.
- Сервер чата 1 передает сообщение в очередь синхронизации сообщений.
- Сообщение записывается в хранилище типа «ключ–значение».

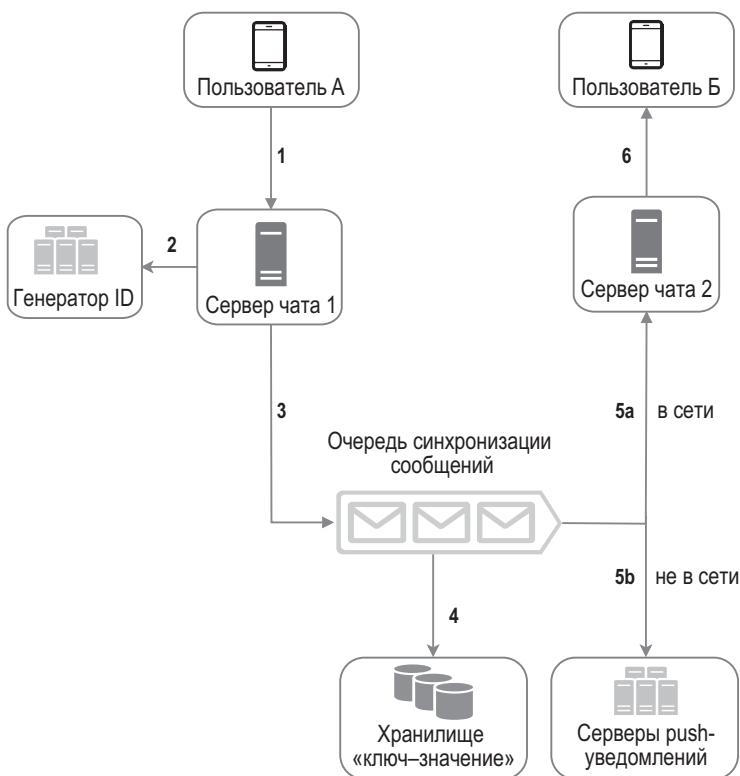


Рис. 12.11

5. а. Если пользователь Б в сети, сообщение направляется на сервер 2, к которому он подключен.
6. Если пользователь Б не в сети, отправляется push-уведомление.
6. Сервер чата 2 передает сообщение пользователю Б. Между пользователем Б и сервером чата 2 установлено постоянное соединение по WebSocket.

### Синхронизация сообщений между несколькими устройствами

У многих пользователей есть сразу несколько устройств. Ниже объясняется, как в таких условиях происходит синхронизация сообщений (рис. 12.13).

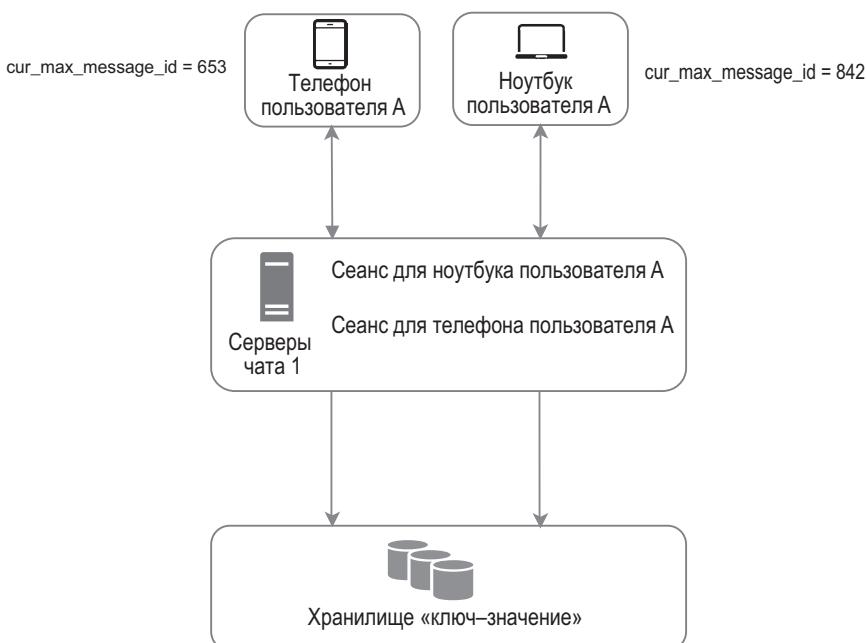


Рис. 12.13

На рис. 12.13 у пользователя А есть два устройства: телефон и ноутбук. Когда он входит в чат со своего телефона, приложение устанавливает по WebSocket соединение с сервером чата 1. Аналогичным образом с сервером чата 1 соединяется и ноутбук.

Каждое устройство использует переменную под названием `cur_max_message_id` для отслеживания ID последнего сообщения. Сообщения считаются новыми, если они соответствуют следующим двум условиям:

- ID получателя совпадает с ID текущего аутентифицированного пользователя.
- ID сообщения в хранилище типа «ключ–значение» больше, чем `cur_max_message_id`.

Каждое устройство имеет свое значение `cur_max_message_id` и может получить новые значения из хранилища, что упрощает синхронизацию.

## Маршрут прохождения сообщений в небольшом групповом чате

Логика группового чата сложнее по сравнению с приватным. Маршрут прохождения сообщений проиллюстрирован на рис. 12.14 и 12.15.

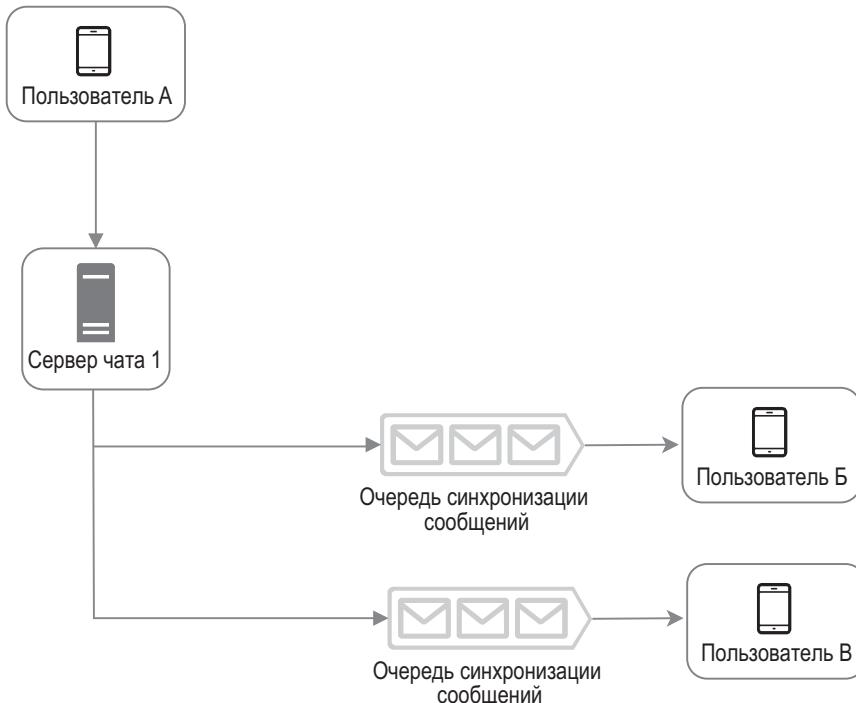


Рис. 12.14

На рис. 12.14 показано, что происходит, когда пользователь А отправляет сообщение в групповой чат. Предположим, группа состоит из трех участников (пользователей А, Б и В). Сначала сообщение пользователя А копируется в очередь синхронизации сообщений каждого участника группы: одно для пользователя Б, другое для пользователя В. Очередь синхронизации сообщений можно считать почтовым ящиком получателя. Такая архитектура хорошо подходит для небольших групп, потому что:

- она упрощает процесс синхронизации, так как для получения новых сообщений каждый клиент должен проверять собственный почтовый ящик;

- в небольших группах хранение копии сообщения в почтовом ящике каждого получателя забирает мало ресурсов.

В системе WeChat используется похожий подход: количество участников группы в ней не превышает 500 [8]. Но если группа насчитывает много пользователей, хранение копий сообщений для каждого из них будет неприемлемым.

Получателю могут поступать сообщения от множества пользователей. У каждого получателя есть почтовый ящик (очередь синхронизации сообщений) с сообщениями от разных отправителей. Эта архитектура проиллюстрирована на рис. 12.15.

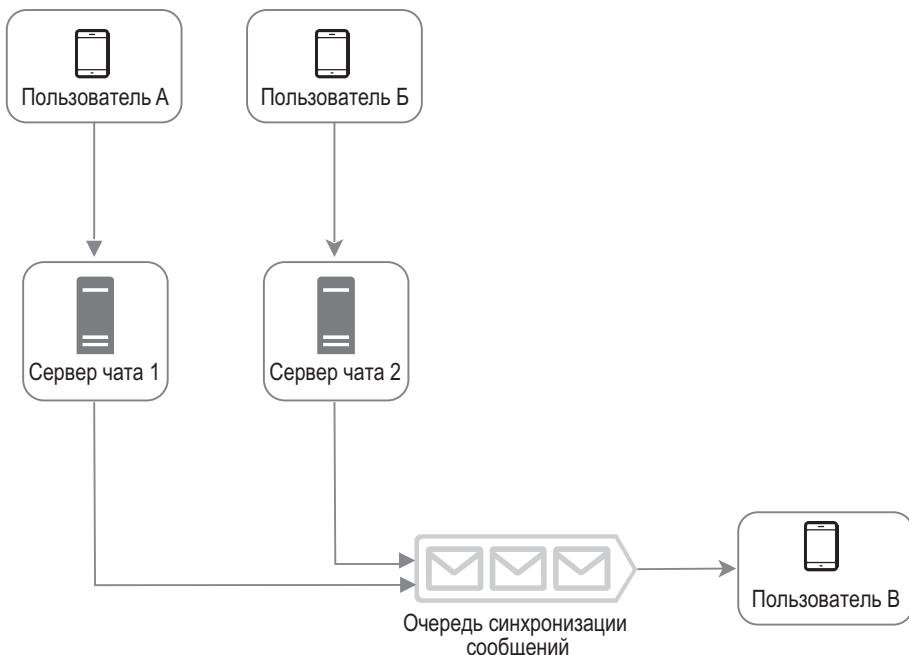


Рис. 12.15

### Сетевой статус

Индикатор сетевого статуса — неотъемлемая часть многих приложений для мгновенного обмена сообщениями. Обычно он имеет вид зеленой

точки рядом с аватаром или именем пользователя. В этом разделе вы узнаете, как этот компонент устроен изнутри.

В нашей общей архитектуре за управление сетевым состоянием и взаимодействие с клиентами по WebSocket отвечают серверы сетевого статуса. Его изменение может быть инициировано несколькими путями. Давайте рассмотрим каждый из них.

### Вход пользователя в систему

Процесс входа пользователя в систему был описан в разделе «Обнаружение сервисов». После установления соединения на основе WebSocket между клиентом и сервисом реального времени сетевой статус пользователя A и временная метка `last_active_at` записываются в хранилище типа «ключ–значение». После входа в систему индикатор показывает, что пользователь находится в сети.

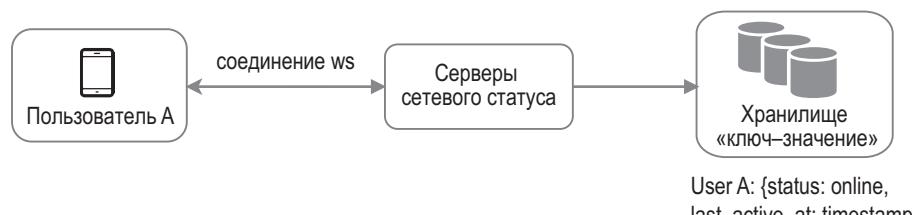


Рис. 12.16

### Выход из системы

При выходе пользователя из системы происходит процесс, показанный на рис. 12.17. В хранилище типа «ключ–значение» сетевой статус меняется на `offline`. Индикатор присутствия показывает, что пользователя нет в сети.

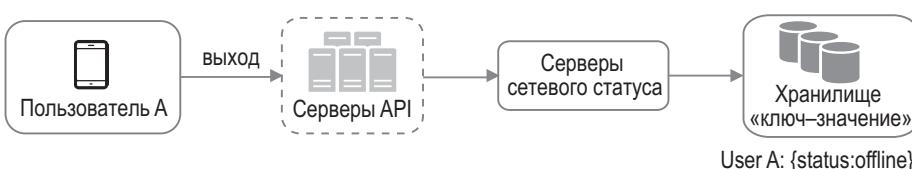


Рис. 12.17

### Отключение пользователя от сети

Всем нам хочется, чтобы интернет-соединение всегда было стабильным и надежным, но в реальности такого не бывает. Поэтому проблему следует отразить в нашей архитектуре. Когда пользователь отключается от интернета, постоянное соединение между клиентом и сервером теряется. Мы могли бы указать, что пользователь не в сети, и затем поменять его состояние на противоположное, когда соединение восстановится, но это решение слишком примитивное и имеет серьезный недостаток. Пользователи могут отключаться и подключаться заново по многу раз за короткое время — это распространённое явление. Например, сетевое соединение может временно разорваться, когда пользователь проезжает по тоннелю. Если обновлять сетевое состояние при каждом разрыве и переподключении, индикатор сетевого статуса будет мигать слишком часто, что отрицательно скажется на взаимодействии с пользователями.

Для решения этой проблемы мы воспользуемся механизмом пульсации. Время от времени клиент шлет серверам сетевого статуса события. Если в течение какого-то времени (скажем,  $x$  секунд) серверы получили событие пульсации, они считают, что пользователь находится в сети. В противном случае пользователь недоступен.

На рис. 12.18 клиент отправляет серверу событие пульсации раз в 5 секунд. После отправки третьего события клиент отключается на  $x = 30$  секунд (это число выбрано произвольно, чтобы продемонстрировать логику). Сетевой статус меняется на «не в сети».

### Распространение информации о сетевом статусе

Как друзья пользователя А узнают об изменении его сетевого статуса? На рис. 12.19 показано, как это работает. Серверы сетевого статуса используют модель «издатель–подписчик», в которой между каждой парой друзей существует канал. Когда сетевое состояние пользователя А меняется, он публикует соответствующее событие в три канала: А-Б, А-В и А-Г. На эти три канала подписаны пользователи Б, В и, соответственно, Г. Таким образом эти друзья могут легко получать обновления о статусе. Взаимодействие между клиентами и серверами происходит в реальном времени по WebSocket.

Приведенная архитектура подходит для небольших групп пользователей. Например, в WeChat используется похожий подход, так как группы в этой системе могут включать не более 500 участников. Но если мы имеем дело

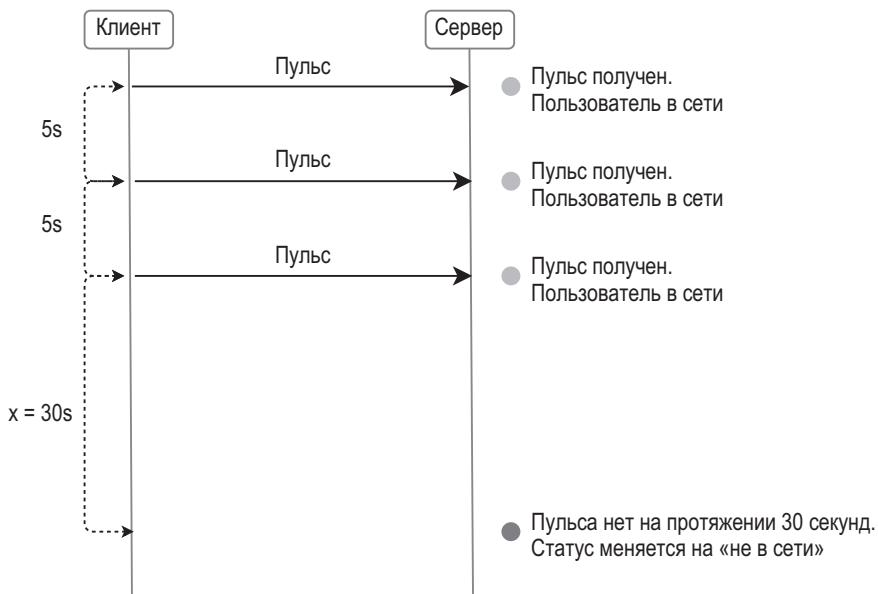


Рис. 12.18

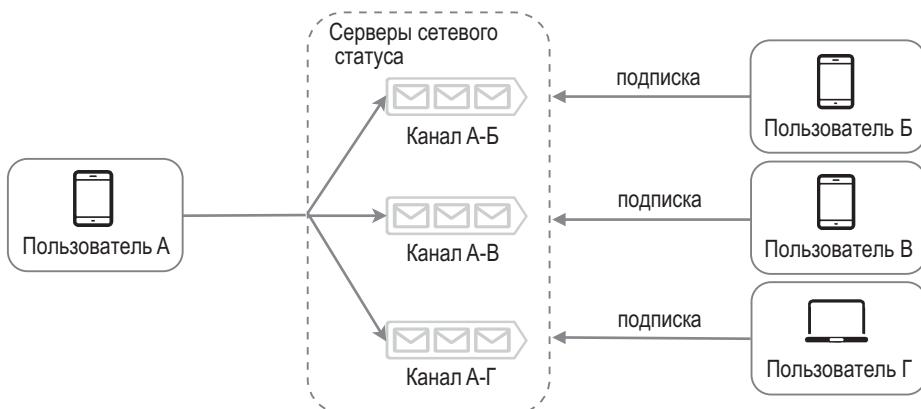


Рис. 12.19

с крупными группами, на информирование каждого участника о сетевом статусе будет уходить много ресурсов и времени. Представьте себе группу из 100 000 участников. Каждое изменение статуса будет генерировать 100 000 событий. Чтобы избавиться от этой проблемы, можно получать

сетевой статус пользователя, только когда он заходит в группу или вручную обновляет список друзей.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы представили архитектуру системы мгновенного обмена сообщениями, которая поддерживает как приватные, так и групповые чаты. Для взаимодействия в реальном времени между клиентом и сервером используется WebSocket. Система состоит из следующих компонентов: серверы чата для обмена сообщениями в реальном времени, серверы сетевого статуса для управления сетевым статусом пользователя, серверы для отправки push-уведомлений, хранилища типа «ключ–значение» для хранения истории переписки и серверы API для других функций.

Если в конце интервью еще остается время, можете затронуть дополнительные аспекты.

- Приложение можно расширить для поддержки медиафайлов, таких как фотографии и видео. Медиафайлы имеют намного больший размер по сравнению с текстом. Можно обсудить такие вещи, как сжатие, облачное хранение и миниатюрные изображения.
- Сквозное шифрование. WhatsApp поддерживает сквозное шифрование сообщений. Заинтересованные читатели могут ознакомиться со статьей в справочных материалах [9].
- Кэширование сообщений на стороне клиента — эффективный способ сокращения объема данных, передаваемых между клиентом и сервером.
- Сокращение времени загрузки. Компания Slack создала географически распределенную сеть для кэширования пользовательских данных, каналов и т. д. Это сокращает время загрузки [10].
- Обработка ошибок:
  - ◆ ошибки на сервере чата. У сервера чата могут быть сотни тысяч (или даже больше) постоянных сетевых соединений. Если он выйдет из строя, механизм обнаружения сервисов (Zookeeper) предоставит клиентам новый сервер чата, к которому они смогут подключаться;

- ♦ механизм повторной отправки сообщений. В случае ошибки сообщения обычно записываются в очередь и затем отправляются повторно.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Erlang at Facebook: <https://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf>
- [2] Messenger and WhatsApp process 60 billion messages a day: <https://www.theverge.com/2016/4/12/11415198/facebook-messenger-whatsapp-number-messages-vs-sms-f8-2016>
- [3] Длинный хвост: [https://ru.wikipedia.org/wiki/Длинный\\_хвост](https://ru.wikipedia.org/wiki/Длинный_хвост)
- [4] The Underlying Technology of Messages: <https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919/>
- [5] How Discord Stores Billions of Messages: <https://blog.discordapp.com/how-discord-stores-billions-of-messages-7fa6ec7ee4c7>
- [6] Announcing Snowflake: [https://blog.twitter.com/engineering/en\\_us/a/2010/announcing-snowflake.html](https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html)
- [7] Apache ZooKeeper: <https://zookeeper.apache.org/>
- [8] Из ничего: эволюция фоновой системы WeChat (статья на китайском): <https://www.infoq.cn/article/the-road-of-the-growth-weixin-background>
- [9] End-to-end encryption: <https://faq.whatsapp.com/en/android/28030015/>
- [10] Flannel: An Application-Level Edge Cache to Make Slack Scale: <https://slack.engineering/flannel-an-application-level-edge-cache-tomake-slack-scale-b8a6400e2f6b>

---

# 13

---

## ПРОЕКТИРОВАНИЕ СИСТЕМЫ АВТОЗАПОЛНЕНИЯ ПОИСКОВЫХ ЗАПРОСОВ

При вводе запроса в поисковиках или интернет-магазинах пользователь часто видит один или несколько всплывающих вариантов слов или их сочетаний. Эту функцию называют автозаполнением, опережающим вводом, поиском по мере ввода или инкрементальным поиском. На рис. 13.1 показан пример того, как при вводе в поле поиска слова dinner Google выводит список вариантов автозаполнения. Эта полезная возможность доступна во многих продуктах, что подводит нас к следующей задаче: спроектируйте систему автозаполнения. На интервью также встречаются ее разновидности: «Реализуйте алгоритм топ- $k$ » или «Сформируйте список из  $k$  самых популярных поисковых запросов».



Рис. 13.1

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Решение любой задачи на интервью по проектированию ИТ-систем начинается с прояснения требований. Для этого нужно задать достаточное количество уточняющих вопросов. Вот пример диалога между кандидатом и интервьюером:

**Кандидат:** «Сопоставление поддерживается только в начале поискового запроса или также в середине?»

**Интервьюер:** «Только в начале поискового запроса».

**Кандидат:** «Сколько вариантов автозаполнения должна возвращать система?»

**Интервьюер:** «5».

**Кандидат:** «Как система определяет, какие 5 вариантов нужно вернуть?»

**Интервьюер:** «В зависимости от популярности, основанной на статистике частоты запросов».

**Кандидат:** «Поддерживает ли система проверку орфографии?»

**Интервьюер:** «Проверка орфографии и автозамена не поддерживаются».

**Кандидат:** «Поисковые запросы выполняются на английском языке?»

**Интервьюер:** «Да. Если в конце останется время, мы обсудим поддержку разных языков».

**Кандидат:** «Допускаются ли прописные буквы и специальные символы?»

**Интервьюер:** «Нет, мы исходим из того, что все поисковые запросы состоят из алфавитных символов в нижнем регистре».

**Кандидат:** «Сколько пользователей у этого продукта?»

**Интервьюер:** «10 миллионов DAU».

## Требования

Вот краткий список требований.

- Короткое время ответа. Варианты автозаполнения должны отображаться достаточно быстро по мере того, как пользователь вводит поисковый запрос. В статье [1] говорится, что система автозаполнения в Facebook должна возвращать результаты в течение 100 миллисекунд. В противном случае возникают задержки.
- Актуальность. Варианты автозаполнения должны иметь отношение к поисковому запросу.
- Сортировка. Результаты, возвращаемые системой, должны быть упорядочены по популярности или с использованием других моделей ранжирования.
- Масштабирование. Система должна справляться с большим объемом трафика.
- Высокая доступность. Система должна оставаться доступной и отзывчивой, когда какая-то ее часть выходит из строя, показывает плохую производительность или испытывает неожиданные проблемы с сетью.

## Приблизительные оценки

- Ожидается 10 миллионов активных пользователей в день (DAU).
- В среднем пользователь выполняет 10 поисковых запросов в день.
- Поисковая строка занимает 20 байтов:
  - ◆ предполагается использование кодировки ASCII. Каждый символ занимает 1 байт;
  - ◆ предполагается, что запрос состоит из 4 слов, в среднем по 5 символов в каждом;
  - ◆ получается  $4 \times 5 = 20$  байтов в каждом запросе.
- При вводе в поле поиска каждого символа клиент обращается к серверу за вариантами автозаполнения. В среднем для каждой поисковой строки требуется 20 запросов. Например, при вводе слова dinner сервер получает следующие 6 запросов:

```
search?q=d
search?q=di
search?q=din
search?q=dinn
search?q=dinne
search?q=dinner
```

- ~24 000 запросов в секунду ( $QPS$ ) = 10 000 000 пользователей \* 10 запросов / день \* 20 символов / 24 часа / 3600 секунд.
- Пиковый показатель  $QPS = QPS * 2 = \sim 48\ 000$ .
- Предполагается, что ежедневно 20 % запросов являются новыми. 10 миллионов \* 10 запросов / день / 20 байт на запрос \* 20 % = 0,4 Гб. Это означает, что каждый день в хранилище записывается 0,4 Гб новых данных.

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Общая архитектура системы состоит из двух сервисов.

- Сервис сбора данных. Собирает пользовательские поисковые запросы и накапливает их в режиме реального времени. Для больших наборов данных обработка в реальном времени является нецелесообразной, но она послужит хорошей отправной точкой. Более реалистичное решение будет рассмотрено на следующем этапе.
- Сервис запросов. Возвращает 5 самых популярных строк для данного поискового запроса или его начальной части.

### Сервис сбора данных

Чтобы увидеть, как работает сервис сбора данных, рассмотрим упрощенный пример. Допустим, у нас есть частотная таблица, содержащая строки и частоту, с которой их ищут, как показано на рис. 13.2. Изначально эта таблица пустая. Затем пользователь вводит по очереди twitch, twitter, twitter и twillo. На рис. 13.2 видно, как обновляется частотная таблица.

The diagram illustrates five separate requests being processed by a service. Each request has its own frequency table:

- Запрос: twitch**: Contains one row for 'twitch' with a frequency of 1.
- Запрос: twitter**: Contains two rows: one for 'twitch' (frequency 1) and one for 'twitter' (frequency 1).
- Запрос: twitter**: Contains two rows: one for 'twitch' (frequency 1) and one for 'twitter' (frequency 2).
- Запрос: twillo**: Contains three rows: one for 'twitch' (frequency 1), one for 'twitter' (frequency 2), and one for 'twillo' (frequency 1).

Рис. 13.2

### Сервис запросов

Предположим, что у нас есть следующая частотная таблица (табл. 13.1). Она состоит из двух полей.

- «Запрос» хранит строку запроса.
- «Частота» показывает, сколько раз искали ту или иную строку.

Таблица 13.1

Запрос	Частота
twitter	35
twitch	29
twilight	25
twin peak	21
twitch prime	18
twitter search	14
twillo	10
twin peak sf	8

Когда пользователь вводит tw в поле поиска, на экране появляется пять самых популярных поисковых запросов (рис. 13.3) при условии, что частотная таблица основана на табл. 13.1.

tw
twitter
twitch
twilight
twin peak
twitch prime

Рис. 13.3

Чтобы получить 5 строк, которые ищут чаще всего, нужно выполнить следующий SQL-запрос:

```
SELECT * FROM frequency_table
WHERE query Like `prefix%`
ORDER BY frequency DESC
LIMIT 5
```

Рис. 13.4

Это приемлемое решение для небольших наборов данных. Если же данных много, обращение к БД становится узким местом. В следующем разделе мы исследуем потенциальные пути оптимизации.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

При описании общей архитектуры мы обсудили сервис сбора данных и сервис запросов. Этот подход не оптимален, но его можно взять за основу. В этом разделе мы подробно поговорим о нескольких компонентах и способах оптимизации:

- префиксное дерево;
- сервис сбора данных;
- сервис запросов;
- масштабирование хранилища;
- операции с префиксным деревом.

## Префиксное дерево

В общей архитектуре в качестве хранилища используется реляционная база данных. Но для извлечения пяти самых популярных поисковых запросов она будет малоэффективной. Чтобы решить эту проблему, воспользуемся структурой данных, известной как префиксное дерево. Поскольку этот компонент крайне важен для работы системы, мы уделим его проектированию особое внимание. Следует отметить, что некоторые представленные здесь идеи позаимствованы из статей [2] и [3].

Для решения этой задачи обязательно нужно понимать, как работает простое префиксное дерево, хотя этот аспект больше относится к структурам данных, чем к проектированию ИТ-систем. К тому же в интернете есть множество материалов на эту тему. В этой главе приведен лишь краткий обзор префиксных деревьев, а основное внимание уделяется тому, как их оптимизировать, чтобы сократить время ответа.

Префиксное дерево — это иерархическая структура данных, которая подходит для компактного хранения строк. Ее английское название, trie, происходит от слова retrieval («извлечение, поиск»); это говорит о том, что она предназначена для операций извлечения строк. Основные свойства префиксного дерева:

- префиксное дерево является иерархической структурой данных;
- корень представляет пустую строку;
- каждый узел хранит символы и имеет 26 дочерних узлов, по одному для каждой буквы английского алфавита. Для экономии места мы не показываем пустые ветви;
- каждый узел дерева представляет отдельное слово или префиксную строку.

На рис. 13.5 показано префиксное дерево с поисковыми запросами tree, try, true, toy, wish и win. Полные поисковые запросы имеют утолщенные края.

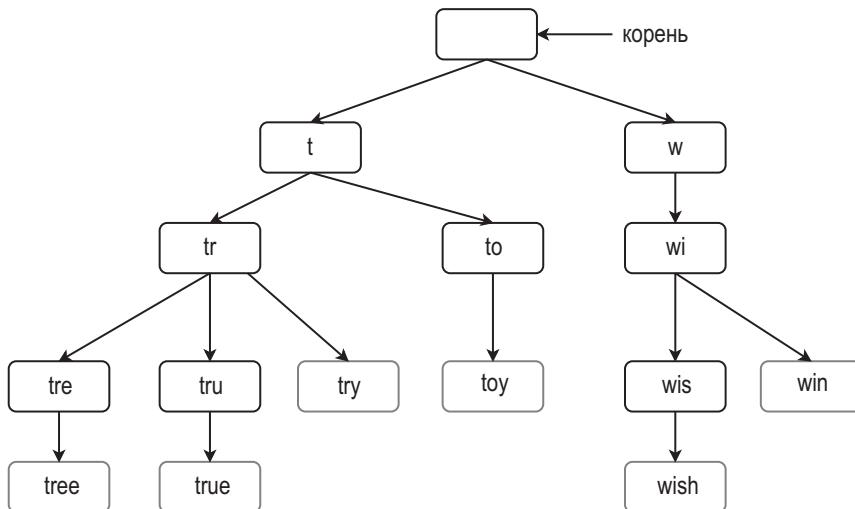


Рис. 13.5

Простое префиксное дерево хранит в своих узлах символы. Для поддержки сортировки узлы должны содержать информацию о частоте. Допустим, у нас есть следующая частотная таблица.

Таблица 13.2

Запрос	Частота
tree	10
try	29
true	35
toy	14
wish	25
win	50

После добавления в узлы информации о частоте префиксное дерево будет выглядеть, как на рис. 13.6.

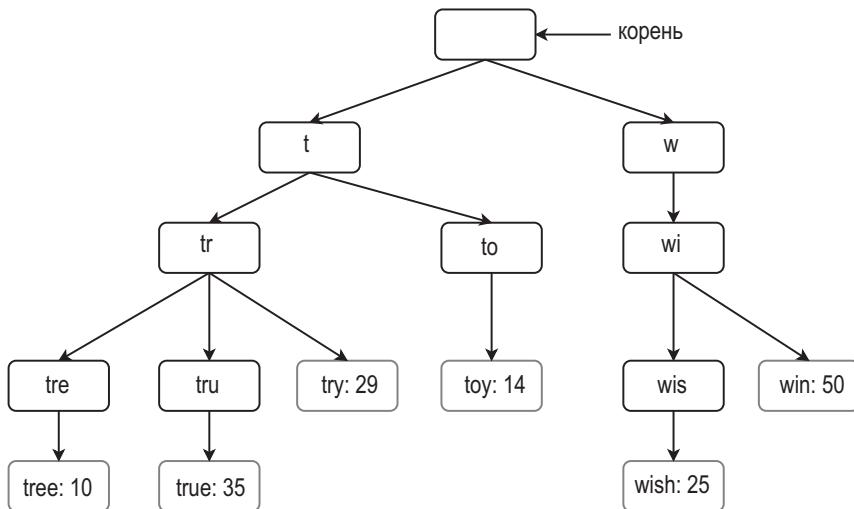


Рис. 13.6

Как работает автозаполнение при использовании префиксного дерева? Прежде чем углубляться в алгоритм, определимся с обозначениями:

- $p$  — длина префикса;
- $n$  — общее количество узлов в префиксном дереве;
- $c$  — количество потомков у заданного узла.

Ниже перечислены этапы получения  $k$  самых популярных поисковых запросов.

1. Найти префикс. Временная сложность:  $O(p)$ .
2. Пройтись по дереву, начиная с префиксного узла, чтобы получить все подходящие узлы-потомки. Потомок подходит, если он может сформировать нужную строку запроса. Временная сложность:  $O(c)$ .
3. Отсортировать узлы-потомки и получить первые  $k$ . Временная сложность:  $O(c \log c)$ .

Давайте рассмотрим этот алгоритм на примере рис. 13.7. Допустим,  $k$  равно 2, а пользователь вводит в поле поиска tr. Алгоритм работает следующим образом:

- Шаг 1. Найти префиксный узел tr.
- Шаг 2. Пройтись по поддереву, чтобы получить все подходящие дочерние узлы. В данном случае подходят узлы [tree: 10], [true: 35], [try: 29].
- Шаг 3. Отсортировать дочерние узлы и получить два верхних. Двумя самыми популярными запросами с префиксом tr являются [true: 35] и [try: 29].

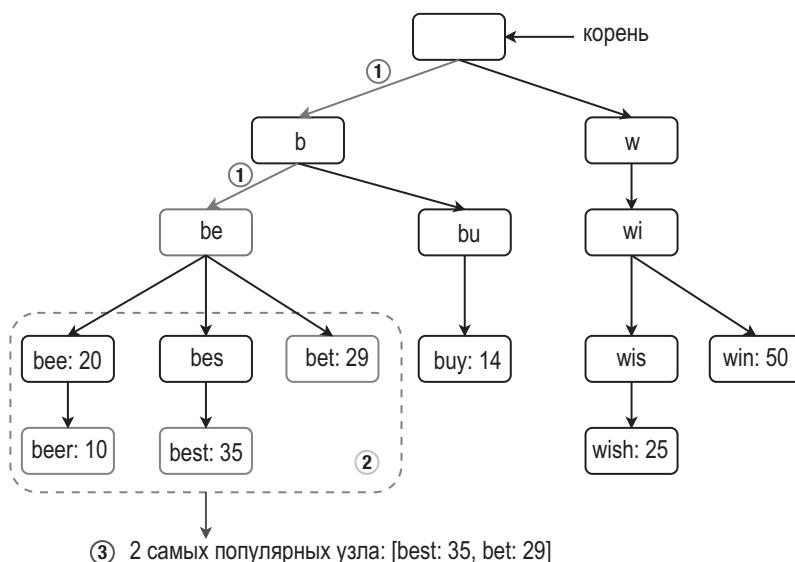


Рис. 13.7

Временная сложность этого алгоритма равна общему времени, потраченному на каждый этап, описанный выше:  $O(p) + O(c) + O(c \log c)$ .

Этот алгоритм довольно простой, но при этом слишком медленный, так как в худшем случае для получения  $k$  верхних результатов придется пройти по всему префиксному дереву. Есть два варианта оптимизации:

- 1) ограничить максимальную длину префикса;
- 2) кэшировать самые популярные поисковые запросы в каждом узле.

Давайте рассмотрим их по порядку.

### **Ограничение максимальной длины префикса**

Пользователи редко вводят длинные поисковые запросы. Поэтому можно легко предположить, что  $p$  – это небольшое целое число (скажем, 50). Если ограничить длину префикса, временную сложность его поиска можно сократить с  $O(p)$  до  $O(\text{небольшая константа})$ , например  $O(1)$ .

### **Кэширование самых популярных поисковых запросов в каждом узле**

Чтобы не выполнять обход всего дерева, мы сохраняем  $k$  наиболее часто используемых запросов в каждом узле. Пользователю будет достаточно 5–10 вариантов автозаполнения, поэтому  $k$  будет относительно небольшим числом. В нашем конкретном случае кэшируются только пять верхних поисковых запросов.

Кэшируя популярные запросы в каждом узле, мы существенно снижаем временную сложность извлечения вариантов автозаполнения. Но этот подход требует много места для хранения популярных запросов в каждом узле. Короткое время ответа крайне важно и вполне оправдывает выделение дополнительного места.

На рис. 13.8 показано обновленное префиксное дерево. В каждом узле хранится пять верхних запросов. Например, узел с префиксом *be* хранит следующее: [best: 35, bet: 29, bee: 20, be: 15, beer: 10].

Давайте проанализируем временную сложность алгоритма после применения этих двух оптимизаций:

1. Поиск префиксного узла. Временная сложность:  $O(1)$ .
2. Возвращение  $k$  верхних результатов. Поскольку популярные запросы кэшируются, временная сложность этого этапа составляет  $O(1)$ .

Временная сложность каждого шага сведена к  $O(1)$ , поэтому получение  $k$  самых популярных запросов с помощью этого алгоритма занимает всего  $O(1)$ .

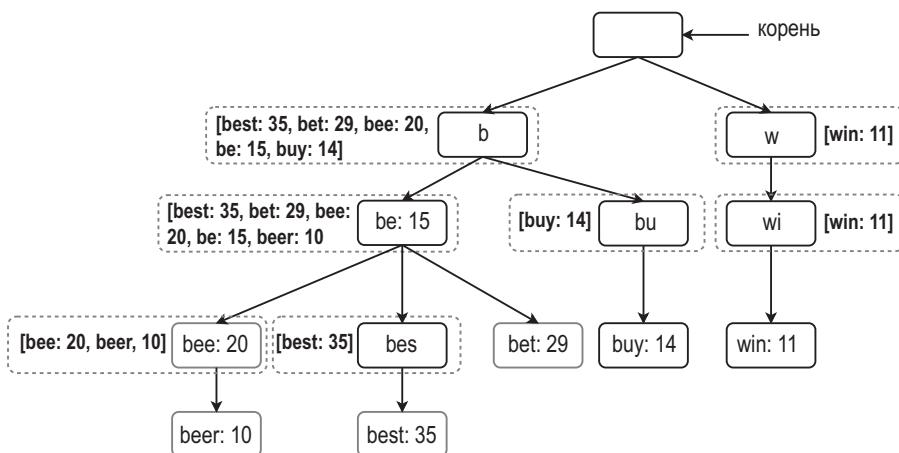


Рис. 13.8

### Сервис сбора данных

В предыдущем варианте архитектуры данные обновлялись в реальном времени по мере ввода поискового запроса. Этот подход непрактичен по двум причинам.

- Пользователи могут вводить миллиарды запросов в день. Обновление дерева при каждом вводе существенно замедляет сервис запросов.
- После того как префиксное дерево сформировано, изменение самых популярных вариантов автозаполнения может быть незначительным. Следовательно, префиксное дерево не нужно часто обновлять.

Чтобы спроектировать масштабируемый сервис сбора данных, нужно понять, откуда эти данные поступают и как они используются. Приложения вроде Twitter, работающие в реальном времени, нуждаются в актуальных вариантах автозаполнения. Но, к примеру, в Google варианты автозаполнения для многих ключевых слов могут практически не меняться изо дня в день.

Несмотря на разные сценарии использования, основной принцип работы сервиса сбора данных остается неизменным, поскольку информация, с помощью которой формируется префиксное дерево, обычно берется из сервисов анализа или логирования.

На рис. 13.9 показан переработанный сервис сбора данных. Проанализируем по очереди каждый его компонент.



Рис. 13.9

**Логи анализа.** Необработанная информация о поисковых запросах. Содержимое логов не индексируется и может только дополняться. Пример лог-файла показан в табл. 13.3.

Таблица 13.3

Запрос	Время
tree	2019-10-01 22:01:01
try	2019-10-01 22:01:05
tree	2019-10-01 22:01:30
toy	2019-10-01 22:02:22
tree	2019-10-02 22:02:42
try	2019-10-03 22:03:03

**Агрегаторы.** Логи анализа обычно очень большие, и их содержимое не соответствует нужному формату. Чтобы наша система могла легко обрабатывать эти данные, их необходимо агрегировать.

Способ агрегации данных зависит от ситуации. Для таких приложений, как Twitter, работающих в реальном времени, требуются актуальные

результаты, поэтому информация агрегируется за короткие промежутки времени. С другой стороны, менее частая агрегация (скажем, раз в неделю) может подойти для других задач. Во время интервью поинтересуйтесь, важны ли актуальные результаты. Мы исходим из того, что префиксное дерево формируется заново каждую неделю.

### Агрегированные данные

В табл. 13.4 показан пример данных, агрегируемых еженедельно. Поле «время» обозначает начало недели. Поле «частота» показывает, сколько раз за неделю вводился тот или иной запрос.

**Таблица 13.4**

Запрос	Время	Частота
tree	2019-10-01	12 000
tree	2019-10-08	15 000
tree	2019-10-15	9000
toy	2019-10-01	8500
toy	2019-10-08	6256
toy	2019-10-15	8866

**Рабочие узлы.** Это группа серверов, которые выполняют асинхронные задания с регулярной периодичностью. Они формируют префиксное дерево и сохраняют его в соответствующую БД.

**Кэш префиксного дерева.** Это распределенная система кэширования, которая хранит префиксное дерево в памяти для быстрого чтения. Она делает снимок БД на еженедельной основе.

**БД префиксного дерева.** Это постоянное хранилище одного из двух типов:

1. Документное хранилище. Поскольку новое префиксное дерево генерируется еженедельно, мы можем периодически сохранять его снимок в сериализованном виде. Для сериализованных данных хорошо подходят такие документные хранилища, как MongoDB [4].

2. Хранилище типа «ключ–значение». Префиксное дерево можно представить в виде хеш-таблицы [4], если руководствоваться следующей логикой:

- ◆ каждый префикс в дереве соответствует ключу в хеш-таблице;
- ◆ данные в каждом узле дерева соответствуют значению в хеш-таблице.

На рис. 13.10 показано, как префиксное дерево соотносится с хеш-таблицей.

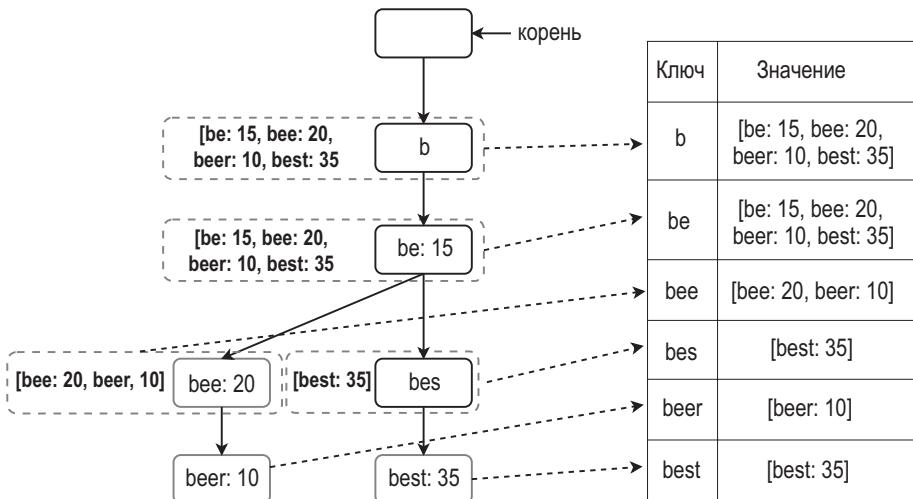


Рис. 13.10

На рис. 13.10 каждый узел префиксного дерева (слева) привязан к паре «ключ, значение» (справа). Если вам не до конца понятно, как работают хранилища типа «ключ–значение», вернитесь к главе 6 «Проектирование хранилища типа “ключ–значение”».

### Сервис запросов

В нашей общей архитектуре сервис запросов извлекает пять самых популярных результатов непосредственно из базы данных. Такой подход неэффективен. На рис. 13.11 показана улучшенная архитектура.



Рис. 13.11

1. Поисковый запрос передается балансировщику нагрузки.
2. Балансировщик нагрузки перенаправляет запрос серверам API.
3. Серверы API достают содержимое префиксного дерева из кэша и генерируют варианты автозаполнения для клиента.
4. Если данных нет в кэше, мы их дополнительно кэшируем. Таким образом, все последующие запросы для того же префикса будут возвращаться из кэша. Сбой кэша происходит, когда кэширующий сервер недоступен или у него заканчивается память.

Сервис запросов должен работать молниеносно. Мы предлагаем следующие оптимизации:

- AJAX-запросы. В веб-приложениях для извлечения результатов автозаполнения обычно используются AJAX-запросы. Их основное преимущество в том, что для отправки/получения запроса/ответа браузеру не нужно обновлять всю веб-страницу целиком.
- Кэширование на уровне браузера. Во многих приложениях варианты автозаполнения меняются не очень часто. В связи с этим их можно хранить в кэше браузера и впоследствии доставать оттуда напрямую. Такой механизм кэширования использует Google. На рис. 13.12 показан заголовок ответа, который приходит при вводе в Google запроса system design interview. Как видите, результаты кэшируются в браузере на протяжении 1 часа. Слово **private** в заголовке **cache-control** означает, что результаты предназначены для отдельного пользователя и не должны попасть в общий кэш. **max-age=3600** означает, что кэш действителен на протяжении 3600 секунд (то есть одного часа).

```

Request URL: https://www.google.com/complete/search?q&c0=0&client=psy-ab&xss1=t&gs_r1=gws-wiz&hl=en&authuser=0&pq=system%20design%20interview
Request method: GET
Remote address: [2607:f8b0:4005:8071::2004]:443
Status code: 200 OK ⓘ Edit and Resend Raw headers
Version: HTTP/2.0
Filter headers
Response headers (615 B)
alt-svc: quic="443"; ma=2592000; v="45-00-h3-Q043"; ma=2592000
cache-control: private, max-age=3600
content-disposition: attachment; filename="t.txt"
content-encoding: br
content-type: application/json; charset=UTF-8
date: Tue, 17 Dec 2019 22:52:01 GMT
expires: Tue, 17 Dec 2019 22:52:01 GMT
server: gws
strict-transport-security: max-age=31536000
trailer: X-Google-GFE-Current-Request-Cost-From-GWS
X-Firefox-Spdy: h2
x-frame-options: SAMEORIGIN
x-xss-protection: 0

```

Рис. 13.12

- Выборка данных. В крупномасштабных системах логирование каждого поискового запроса требует много вычислительных ресурсов и места. Поэтому важно выбрать те данные, которые будут сохранены. Например, наша система может записывать в лог лишь 1 из  $N$  запросов.

## Операции с префиксным деревом

Префиксное дерево — ключевой компонент системы автозаполнения. Давайте рассмотрим операции, которые оно поддерживает (создание, обновление и удаление).

### Создание

Префиксное дерево создается рабочими узлами на основе агрегированных данных. Источником данных выступает лог или БД с результатами анализа.

### Обновление

Префиксное дерево можно обновлять двумя способами.

- Способ 1: обновлять дерево еженедельно. Дерево, созданное заново, заменяет собой старое.
- Способ 2: обновлять узлы дерева напрямую. Мы стараемся избегать этой операции ввиду ее низкой скорости. Но если префиксное дерево небольшое, это приемлемое решение. При обновлении узла необходимо обновить всех его предков вплоть до корня, так как они хранят его популярные запросы. На рис. 13.13 показан пример того,

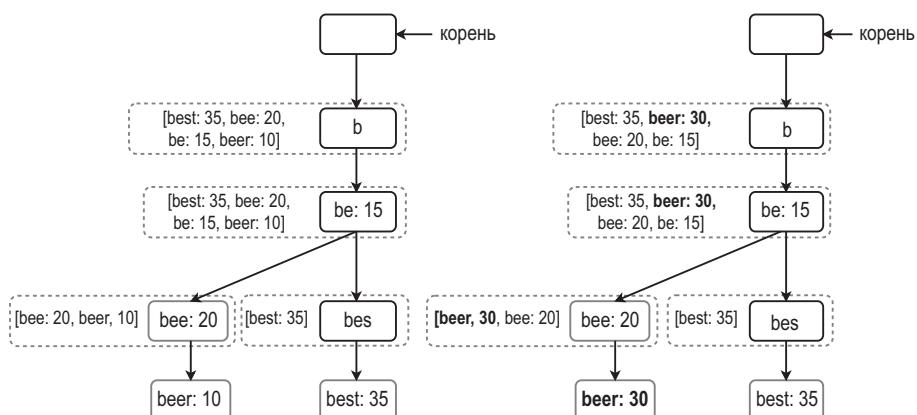


Рис. 13.13

как работает операция обновления. В левой части поисковый запрос beer имеет начальное значение 10. В правой части он обновляется до 30. Как видите, узел и его предки теперь содержат обновленное значение beer, равное 30.

### Удаление

Мы обязаны удалять варианты автозаполнения, которые разжигают ненависть, пропагандируют насилие, носят откровенно сексуальный характер или представляют опасность. Мы добавили слой фильтрации (рис. 13.14) перед кэшем префиксного дерева, чтобы отсеивать нежелательные результаты. Это позволяет фильтровать запросы на основе разных гибких правил. Данные удаляются непосредственно из БД асинхронным образом, чтобы во время следующего цикла обновления для генерации префиксного дерева использовалась походящая информация.



Рис. 13.14

### Масштабирование хранилища

Итак, мы разработали системы для автозаполнения пользовательских поисковых запросов. Пришло время решить проблему с масштабированием, которая возникает, когда префиксное дерево становится слишком большим и уже не помещается на одном сервере.

Поскольку наша система поддерживает только английский язык, для сегментирования проще всего использовать первые символы. Вот несколько примеров.

- Если для хранения данных требуется два сервера, запросы, первая буква которых находится в диапазоне от a до m, можно записывать на первый, а все остальные (от n до z) — на второй.

- Если требуется три сервера, запросы можно разделить на диапазоны от a до i, от j до g и от s до z.

Если следовать этой логике, запросы можно распределить по 26 серверам, так как английский алфавит состоит из 26 букв. Пусть это будет первый уровень сегментирования. Если 26 серверов будет недостаточно, мы можем перейти на второй или даже третий уровень. Например, запросы, начинающиеся с a, можно распределить по четырем серверам: aa-ag, ah-an, ao-au и av-az.

На первый взгляд этот подход кажется разумным, если не брать во внимание тот факт, что намного больше слов начинается с буквы с, чем с буквы x. Это приводит к неравномерному распределению.

Чтобы минимизировать несбалансированность данных, нужно проанализировать, как они обычно распределяются, и применить более элегантную логику шардинга, показанную на рис. 13.15. Диспетчер карты сегментов содержит базу данных для определения того, где должны храниться те или иные строки. Например, если результаты анализа показывают, что совокупное количество запросов для u, v, w, x, y и z сравнимо с количеством запросов для s, мы можем предусмотреть два сегмента: один для s, а другой для диапазона от u до z.

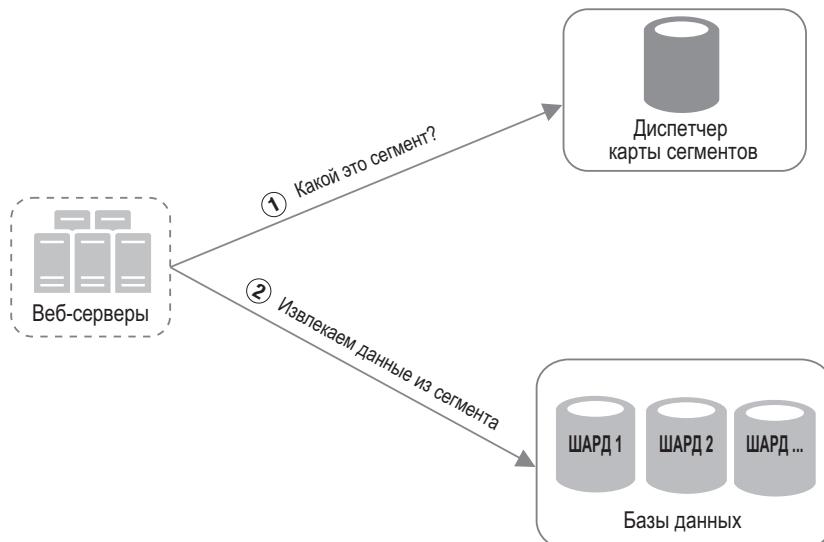


Рис. 13.15

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

Когда вы закончите подробное проектирование, интервьюер может задать вам несколько дополнительных вопросов.

**Интервьюер: «Как бы вы расширили свою архитектуру для поддержки других языков?»**

Для поддержки запросов на языках, отличных от английского, мы записываем в узлы префиксного дерева символы Unicode. Если вы не знакомы с Unicode, вот определение: «Стандарт кодирования, охватывающий все символы для всех систем письма в мире, как современных, так и древних» [5].

**Интервьюер: «Что, если список самых популярных запросов зависит от страны?»**

В этом случае мы можем генерировать разные префиксные деревья для разных стран. Чтобы сократить время ответа, их можно хранить в CDN.

**Интервьюер: «Как реализовать обновление популярных поисковых запросов в реальном времени?»**

Предположим, в результате какого-то важного события становится популярным определенный поисковый запрос. Наша исходная архитектура не подойдет, так как:

- рабочие узлы обновляют префиксное дерево по расписанию (раз в неделю) и в остальное время бездействуют;
- даже если обновление префиксного дерева запланировано, оно занимает слишком много времени.

Разработка системы автозаполнения поисковых запросов реального времени — сложная задача, которая выходит за рамки этой книги, поэтому мы дадим лишь несколько советов:

- уменьшите рабочий набор данных с помощью шардинга;
- измените модель ранжирования так, чтобы недавним поисковым запросам назначался больший вес;
- данные могут поступать в виде потоков, поэтому они могут не быть доступны целиком и сразу. Потоковые данные генерируются непрерывно. Для обработки потоков нужен другой набор систем: Apache

Hadoop MapReduce [6], Apache Spark Streaming [7], Apache Storm [8], Apache Kafka [9] и т. д. Поскольку все эти продукты требуют знаний в определенных предметных областях, мы не станем рассматривать их подробно.

Поздравляем, вы проделали длинный путь и можете гордиться собой. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] The Life of a Typeahead Query: <https://www.facebook.com/notes/facebook-engineering/the-life-of-a-typeahead-query/389105248919/>
- [2] How We Built Prefixy: A Scalable Prefix Search Service for Powering Autocomplete: <https://medium.com/@prefixyteam/how-we-built-prefixy-a-scalable-prefix-search-service-for-powering-autocomplete-c20f98e2eff1>
- [3] Prefix Hash Tree An Indexing Data Structure over Distributed Hash Tables: <https://people.eecs.berkeley.edu/~sylvia/papers/pht.pdf>
- [4] Страница MongoDB на Википедии: <https://ru.wikipedia.org/wiki/MongoDB>
- [5] Часто задаваемые вопросы о Unicode: [https://www.unicode.org/faq/basic\\_q.html](https://www.unicode.org/faq/basic_q.html)
- [6] Apache Hadoop: <https://hadoop.apache.org/>
- [7] Spark Streaming: <https://spark.apache.org/streaming/>
- [8] Apache Storm: <https://storm.apache.org/>
- [9] Apache Kafka: <https://kafka.apache.org/documentation/>

# 14

## ПРОЕКТИРОВАНИЕ YOUTUBE

В этой главе вам предложено спроектировать YouTube. У этой задачи есть и другие разновидности, например: «Спроектируйте платформу для обмена видео, такую как Netflix или Hulu», но ко всем им можно применить одно и то же решение. На рис. 14.1 показана главная страница YouTube.

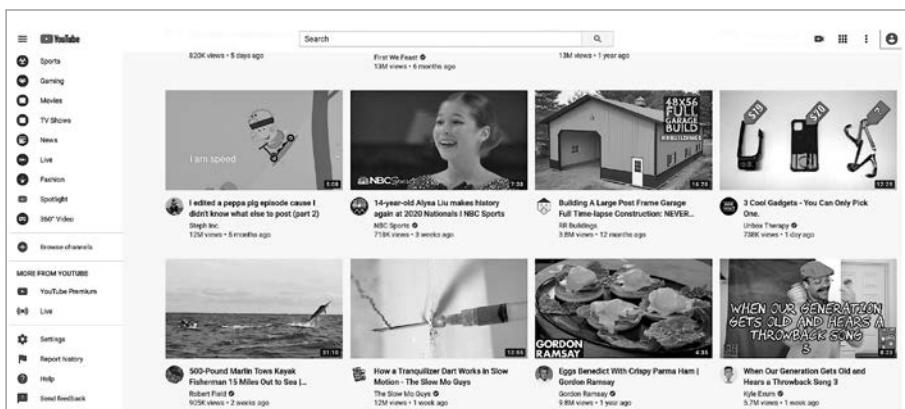


Рис. 14.1

YouTube выглядит просто: автор контента загружает видеофайлы, а зрители их воспроизводят одним щелчком мыши. Но эта простота обманчива. За ней скрывается множество сложных технологий. Ниже перечислена интересная статистика и неочевидные факты о YouTube в 2020 году [1] [2].

- Общее количество активных пользователей в месяц: 2 миллиарда.
- Количество видеороликов, просматриваемых ежедневно: 5 миллиардов.
- 73 % взрослых людей в США используют YouTube.

- На YouTube насчитывается 50 миллионов авторов.
- За полный 2019 год доход от рекламы на YouTube составил 15,1 миллиарда долларов, что на 36 % больше, чем в 2018 году.
- На YouTube приходится 37 % всего мобильного трафика в интернете.
- Веб-сайт YouTube доступен на 80 языках.

Судя по этим цифрам, YouTube – огромная глобальная система, которая приносит много денег.

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Как видно на рис. 14.1, помимо просмотра видео YouTube поддерживает много других функций. Например, под видео можно оставить комментарий, им можно поделиться с друзьями или поставить ему лайк; вы также можете сохранить его в список воспроизведения, подписаться на канал и т. д. Все это невозможно спроектировать за 45- или 60-минутное интервью. Поэтому важно определиться с тем, что именно от вас требуется.

**Кандидат:** «Какие возможности самые важные?»

**Интервьюер:** «Возможность загружать и просматривать видео».

**Кандидат:** «Какие клиенты должны поддерживаться?»

**Интервьюер:** «Мобильные приложения, веб-браузеры и Smart TV».

**Кандидат:** «Сколько у нас ежедневных активных пользователей?»

**Интервьюер:** «5 миллионов».

**Кандидат:** «Сколько времени в среднем уходит на использование этого продукта в день?»

**Интервьюер:** «30 минут».

**Кандидат:** «Нужно ли нам поддерживать пользователей из других стран?»

**Интервьюер:** «Да, существенная часть аудитории находится за рубежом».

**Кандидат:** «Какие разрешения видео должны поддерживаться?»

**Интервьюер:** «Система совместима с большинством разрешений и видеоформатов».

**Кандидат:** «Требуется ли шифрование?»

**Интервьюер:** «Да».

**Кандидат:** «Есть ли какие-либо требования к размеру видео?»

**Интервьюер:** «Наша платформа нацелена на видео небольшого и среднего размера. Максимальный размер видеофайла составляет 1 Гб».

**Кандидат:** «Можем ли мы пользоваться существующей облачной инфраструктурой, которую предоставляют Amazon, Google или Microsoft?»

**Интервьюер:** «Это очень хороший вопрос. Для большинства компаний создание всего с нуля было бы нереалистичным. Рекомендуется задействовать некоторые из существующих облачных сервисов».

В этой главе мы сосредоточимся на проектировании сервиса стримингового видео со следующими возможностями:

- быстрая загрузка видеофайлов;
- бесперебойное вещание;
- возможность изменять качество видео;
- недорогая инфраструктура;
- высокие доступность, масштабируемость и надежность;
- поддерживаемые клиенты: мобильные приложения, веб-браузер и Smart TV.

## **Приблизительные оценки**

Следующие оценки основаны на множестве предположений, поэтому вы должны убедиться в том, что вы с интервьюером поняли друг друга.

- Предположим, что у продукта 5 миллионов активных пользователей в день (DAU).
- Пользователи ежедневно просматривают по 5 видеороликов.
- 10 % пользователей загружают по 1 ролику в день.

- Пусть средний размер видео составляет 300 Мб.
- Общий объем данных, которые нужно сохранять ежедневно: 5 миллионов \* 10 % \* 300 Мб = 150 Тб.
- Стоимость CDN:
  - ◆ когда облачная сеть CDN раздает видео, вы платите за трафик, исходящий из CDN;
  - ◆ для оценки расходов возьмем CDN CloudFront от Amazon (рис. 14.2) [2]. Предположим, что весь трафик раздается из США. Средняя цена за гигабайт составляет 0,02 доллара. Для простоты будем учитывать только стоимость видеовещания;
  - ◆ 5 миллионов \* 5 видео \* 0,3 Гб \* 0,02 доллара = 150 000 долларов в день.

Исходя из этих приблизительных оценок, раздача видео из CDN будет стоить довольно дорого. Даже несмотря на то что облачные провайдеры с готовностью делают скидки для крупных клиентов, расходы будут значительными. О том, как их снизить, мы поговорим в разделе, посвященном подробному проектированию.

Per Month	United States & Canada	Europe & Israel	South Africa, Kenya, & Middle East	South America	Japan	Australia	Singapore, South Korea, Taiwan, Hong Kong, & Philippines	India
First 10TB	\$0.085	\$0.085	\$0.110	\$0.110	\$0.114	\$0.114	\$0.140	\$0.170
Next 40TB	\$0.080	\$0.080	\$0.105	\$0.105	\$0.089	\$0.098	\$0.135	\$0.130
Next 100TB	\$0.060	\$0.060	\$0.090	\$0.090	\$0.086	\$0.094	\$0.120	\$0.110
Next 350TB	\$0.040	\$0.040	\$0.080	\$0.080	\$0.084	\$0.092	\$0.100	\$0.100
Next 524TB	\$0.030	\$0.030	\$0.060	\$0.060	\$0.080	\$0.090	\$0.080	\$0.100
Next 4PB	\$0.025	\$0.025	\$0.050	\$0.050	\$0.070	\$0.085	\$0.070	\$0.100
Over 5PB	\$0.020	\$0.020	\$0.040	\$0.040	\$0.060	\$0.080	\$0.060	\$0.100

Рис. 14.2

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Как уже обсуждалось, интервьюер посоветовал использовать существующие облачные сервисы вместо того, чтобы разрабатывать все с нуля. Так что мы задействуем CDN и хранилище больших двоичных объектов

(binary large objects, BLOBs). Некоторые читатели могут задуматься, почему мы не будем создавать все это самостоятельно. Причины таковы:

- Интервью по проектированию ИТ-систем не посвящены созданию всего с нуля. Выбрать подходящие технологии за отведенный период времени важнее, чем объяснить, как они работают. Например, на интервью достаточно упомянуть, что для хранения исходных видеофайлов будет использоваться хранилище BLOB-объектов. Подробное обсуждение того, как это хранилище устроено, было бы излишним.
- Разработка масштабируемого хранилища BLOB-объектов или CDN чрезвычайно сложная и дорогая. Даже такие крупные компании, как Netflix или Facebook, не разрабатывают все сами. Netflix использует облачные сервисы Amazon [4], а Facebook применяет CDN от Akamai [5].

В целом наша система состоит из трех компонентов (рис. 14.3).

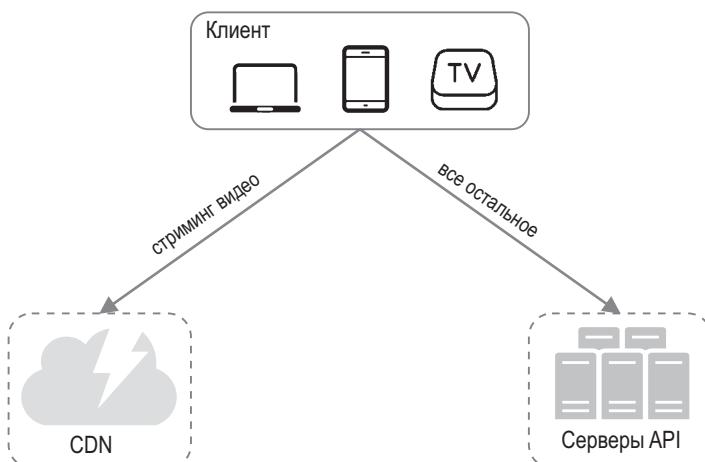


Рис. 14.3

- **Клиент.** Вы можете смотреть YouTube на своем компьютере, мобильном телефоне или Smart TV.
- **CDN.** Видеоролики хранятся в CDN. Когда вы нажимаете кнопку «Смотреть», CDN возвращает стрим.

- **Серверы API.** Все, что не относится к стримингу, проходит через серверы API. Это касается ленты рекомендаций, генерации URL-адреса для загруженного видео, обновления БД кэша с метаданными, регистрации пользователей и т. д.

Во время нашего диалога интервьюер проявил интерес к следующим двум процессам:

- загрузка видео;
- стриминг видео.

Мы обсудим общие аспекты каждого из них.

## Процесс загрузки видео

На рис. 14.4 показана общая схема загрузки видео.

Он состоит из следующих компонентов.

- Пользователь. Сматрит YouTube на таких устройствах, как компьютер, мобильный телефон или Smart TV.
- Балансировщик нагрузки. Равномерно распределяет запросы между серверами API.
- Серверы API. Все пользовательские запросы, за исключением потоковой передачи видео, проходят через серверы API.
- БД метаданных. Метаданные видеофайлов хранятся в отдельной разделяемой БД. Она реплицируется, чтобы отвечать требованиям к производительности и высокой доступности.
- Кэш метаданных. Для улучшения производительности метаданные видеофайлов и пользовательских объектов кэшируются.
- Хранилище исходного видео. Оригинальные видеофайлы размещаются в системе хранения BLOB-объектов. Вот что говорится в статье о хранилище BLOB-объектов в Википедии: «Двоичный большой объект (Binary Large Object, BLOB) — это набор двоичных данных, которые хранятся в СУБД как единое целое» [6].
- Серверы перекодирования. Перекодирование видео — это процесс преобразования видеофайла из одного формата в другой (MPEG,

HLS и т. д.) с целью предоставления оптимальных видеопотоков для разных устройств и типов сетевых соединений.

- Хранилище перекодированного видео. Это хранилище BLOB-объектов для перекодированных видеофайлов.

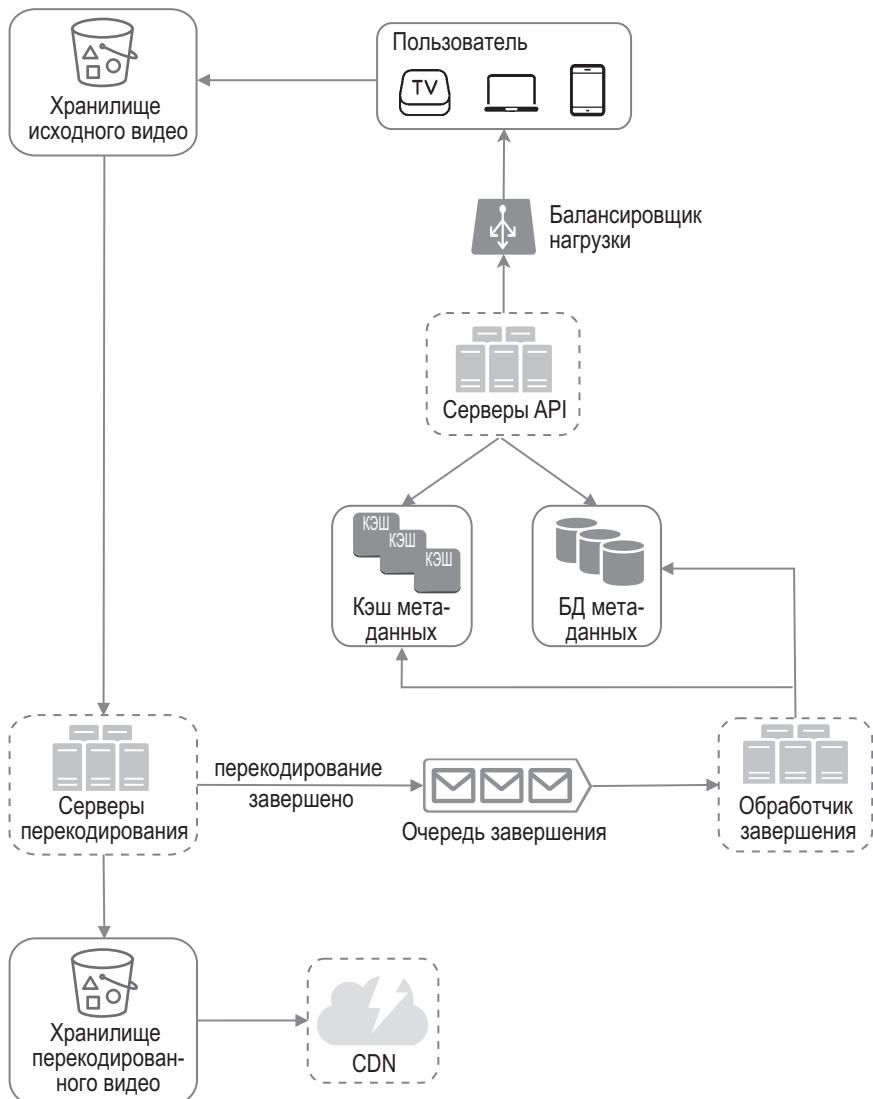


Рис. 14.4

- CDN. Видеофайлы кэшируются в CDN. Когда вы нажимаете кнопку «Смотреть», CDN возвращает видеопоток.
- Очередь завершения. Это очередь сообщений, хранящая информацию о завершении перекодирования видео.
- Обработчик завершения. Этот компонент состоит из списка рабочих узлов, которые извлекают события из очереди завершения и обновляют кэш и БД с метаданными.

Итак, мы разобрали каждый отдельный компонент. Теперь давайте посмотрим, как работает процесс загрузки видео. Он состоит из двух процедур, которые выполняются параллельно.

- А. Загрузка видеофайла.
- Б. Обновление метаданных видеофайла. Метаданные содержат сведения о URL-адресе, размере, разрешении и формате видео, а также информацию о пользователе и т. д.

### Процедура А: загрузка видео

На рис. 14.5 показано, как загружается видеофайл. Ниже представлено описание этого процесса:

1. Видеофайлы загружаются в хранилище исходного видео.
2. Серверы перекодирования извлекают видеофайлы из хранилища исходного видео и начинают их перекодировать.
3. По завершении перекодирования следующие два задания выполняются параллельно:
  - За. Перекодированные видеофайлы записываются в хранилище перекодированного видео.
  - 3б. События о завершении перекодирования записываются в очередь завершения.
- 3а.1. Перекодированное видео передается в CDN.
- 3б.1. Обработчик завершения содержит группу рабочих узлов, которые непрерывно достают события из очереди.
- 3б.1.а. и 3б.1.б. По окончании перекодирования видео обработчик завершения обновляет БД и кэш метаданных.

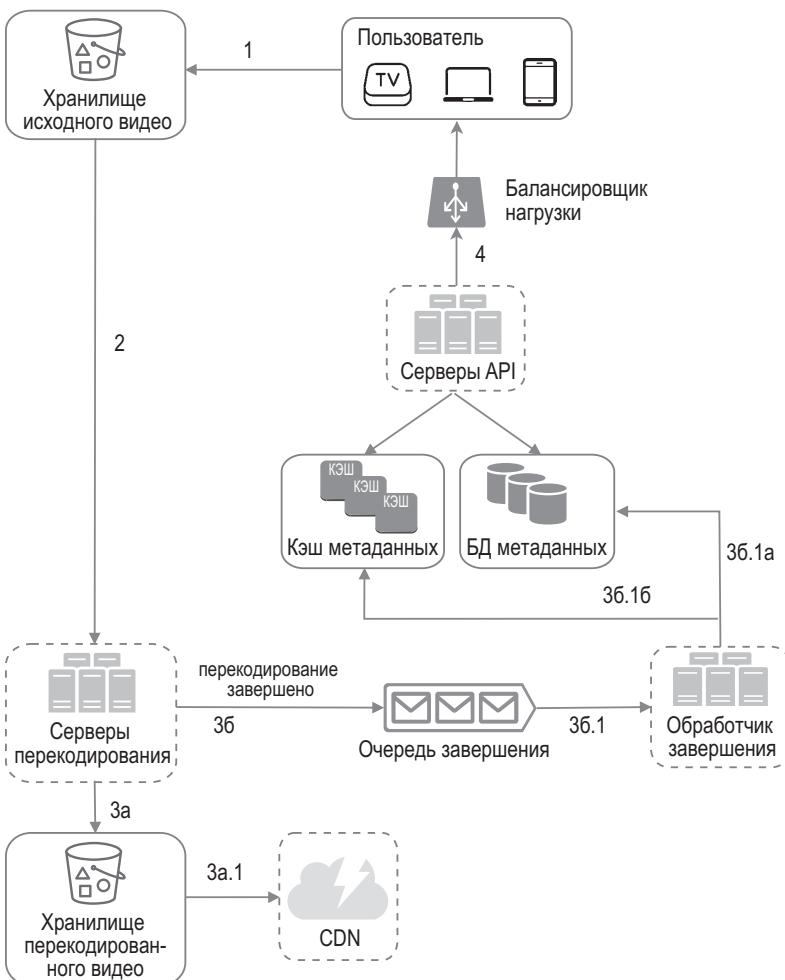


Рис. 14.5

4. Серверы API сообщают клиенту о том, что видео успешно загружено и готово для стриминга.

#### Процедура Б: обновление метаданных

Пока файл загружается в хранилище исходного видео, клиент отправляет запрос на обновление его метаданных, как показано на рис. 14.6. Этот запрос содержит такие сведения, как название видеофайла, его размер, формат и т. д. Серверы API обновляют кэш и БД метаданных.

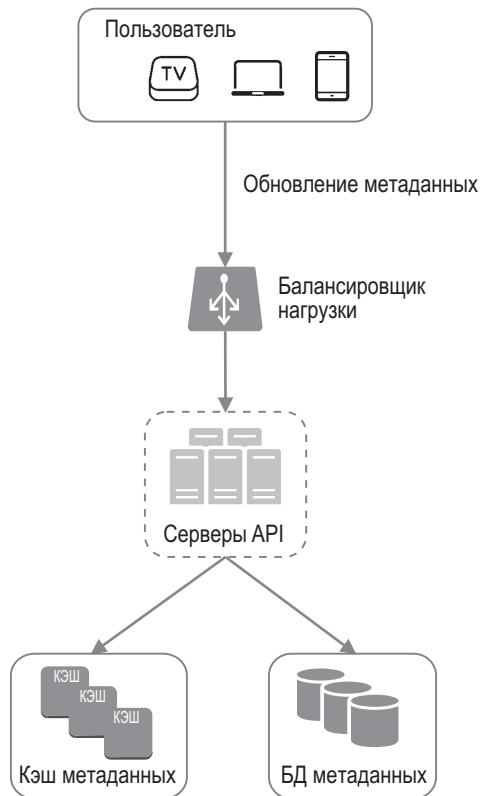


Рис. 14.6

## Стриминг видео

Обычно, когда вы открываете видео на YouTube, оно сразу же начинает воспроизводиться, и вам не нужно ждать окончания его загрузки. Загрузка — это когда весь видеофайл копируется на ваше устройство, а стриминг подразумевает, что ваше устройство получает непрерывный видеопоток из удаленного источника. При просмотре потокового видео ваш клиент загружает данные постепенно, чтобы воспроизведение началось сразу и продолжалось беспрерывно.

Прежде чем обсуждать процесс стриминга видео, давайте рассмотрим такое важное понятие, как протокол стриминга. Это стандартизованный способ управления стримингом видеоданных. Существуют такие популярные протоколы:

- MPEG–DASH. MPEG расшифровывается как Moving Picture Experts Group («экспертная группа по движущимся изображениям»), а DASH – как Dynamic Adaptive Streaming over HTTP («динамический адаптивный стриминг по HTTP»).
- Apple HLS. HLS расшифровывается как HTTP Live Streaming («стриминг по HTTP в реальном времени»).
- Microsoft Smooth Streaming.
- Adobe HTTP Dynamic Streaming (HDS).

Вам не нужно досконально разбираться в этих протоколах или даже помнить их названия, так как это низкоуровневые детали, требующие знания определенной предметной области. Но важно понимать, что разные стриминговые протоколы поддерживают разные форматы кодирования видео и доступны в разных видеоплеерах. При проектировании сервиса видеовещания необходимо выбрать протокол, подходящий для наших задач. Больше о протоколах потоковой передачи можно узнать в замечательной статье [7].

Видеопоток поступает напрямую из CDN. Его доставляет ближайший к вам пограничный сервер. Благодаря этому латентность получается очень низкой. На рис. 14.7 показана общая схема стриминга видео.

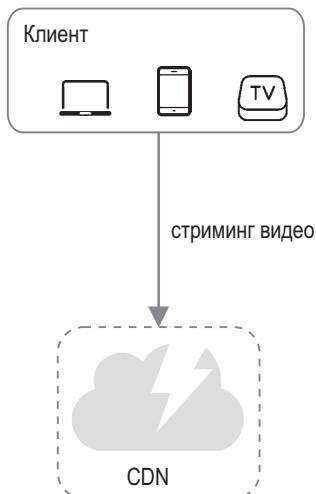


Рис. 14.7

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

При описании общей архитектуры мы разбили систему на две части: процесс загрузки видео и процесс его стриминга. В этом разделе мы улучшим оба эти процессы с помощью важных оптимизаций и добавим механизм обработки ошибок.

### Перекодирование видео

Когда вы записываете видео, ваше устройство (обычно телефон или камера) сохраняет видеофайл в определенном формате. Если вы хотите, чтобы видео как следует воспроизводилось на других устройствах, его битрейт и формат должны быть совместимы. Битрейт — это скорость, с которой обрабатываются биты; обычно чем он выше, тем лучше качество видео. Потоки с высоким битрейтом требуют больше вычислительных ресурсов и более быстрое интернет-соединение.

Перекодирование видео имеет большое значение по следующим причинам.

- Необработанное видео имеет большой размер. Часовое видео высокого разрешения с частотой 60 кадров в секунду может занимать до нескольких сотен гигабайт.
- Многие устройства и браузеры поддерживают только определенные форматы видео. Поэтому видео должно быть перекодировано в разные форматы.
- Чтобы обеспечить высокое качество и плавность воспроизведения, выбор разрешения видео следует делать с учетом скорости сетевого соединения пользователя.
- Качество сетевого соединения может меняться, особенно на мобильных устройствах. Чтобы обеспечить непрерывное воспроизведение видео, следует автоматически или вручную переключаться на поток с подходящим битрейтом в зависимости от пропускной способности сети. Это крайне важно для качественного взаимодействия с пользователем.

Существует множество форматов кодирования, но большинство из них состоят из двух частей.

- Контейнер. Содержит видеофайл, звуковую дорожку и метаданные.
- Кодеки. Это алгоритмы сжатия и распаковки, предназначенные для уменьшения размера видео с сохранением его качества. Самыми распространенными видеокодеками являются H.264, VP9 и HEVC.

### Модель направленного ациклического графа

На перекодирование видео уходит много ресурсов и времени. К тому же требования к перекодированию могут зависеть от автора видео. Например, некоторым авторам нужно, чтобы поверх видео выводились водяные знаки, прочие сами предоставляют миниатюрные изображения; одни загружают видео в высоком разрешении, а другие нет.

Для поддержки разных процедур обработки и обеспечения высокой степени распараллеливания необходимо добавить некий слой абстракции и позволить разработчикам клиентов самим выбирать, какие задания должны выполняться. Например, система потокового вещания видео в Facebook использует модель программирования на основе направленного ациклического графа (directed acyclic graph, DAG), которая разбивает задания на этапы с возможностью последовательного или параллельного выполнения [8]. В нашей архитектуре применяется похожая модель, позволяющая добиться гибкости и распараллелить вычисления. На рис. 14.8 показан DAG для перекодирования видео.

На рис. 14.8 исходный видеофайл разделяется на видео, звук и метаданные. Вот некоторые задания, которые можно применить к видеофайлу.

- Анализ. Следует убедиться в том, что видео не повреждено и имеет хорошее качество.
- Кодирование видео. Это делается для поддержки разных разрешений, кодеков, битрейтов и пр. На рис. 14.9 показан пример закодированных файлов.
- Миниатюрное изображение. Миниатюры могут быть загружены пользователем или автоматически генерированы системой.
- Водяной знак. Изображение, наносимое поверх видео и содержащее информацию, которая его идентифицирует.



Рис. 14.8

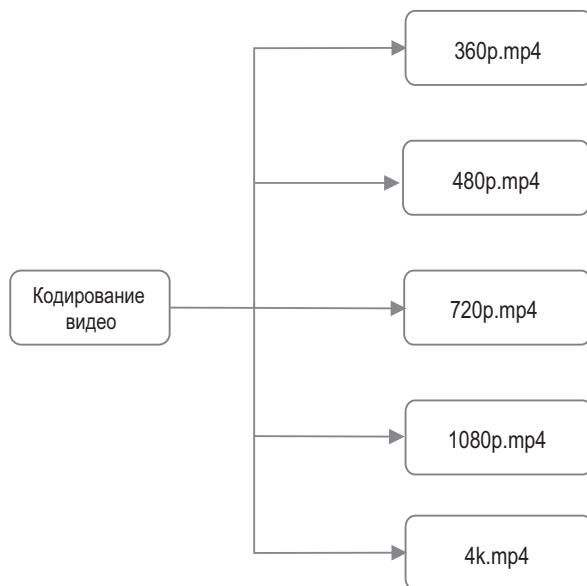


Рис. 14.9

## Архитектура перекодирования видео

На рис. 14.10 показана предложенная нами архитектура перекодирования видео, которая использует облачные сервисы.

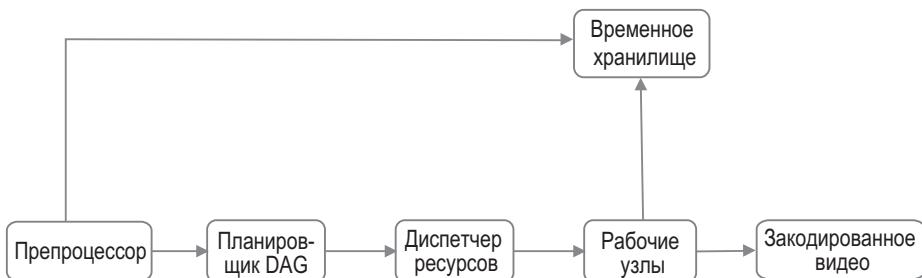


Рис. 14.10

Представленная архитектура состоит из шести основных элементов: препроцессора, планировщика DAG, диспетчера ресурсов, рабочих узлов, временного хранилища и закодированного видео в качестве вывода. Давайте подробно рассмотрим каждый из них.

### Препроцессор

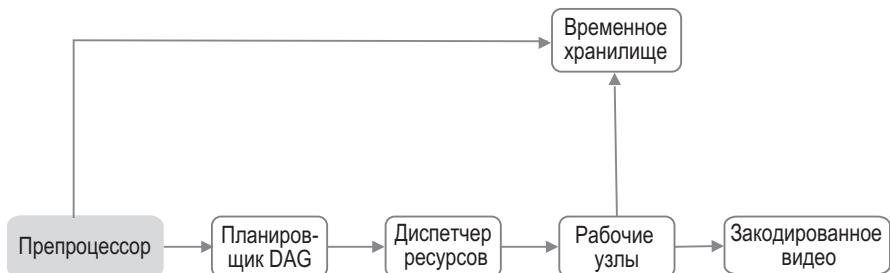


Рис. 14.11

На препроцессор возложено четыре обязанности:

1. Разделение видео. Видеопоток делится на части или более мелкие группы кадров (Group of Pictures, GOP). GOP – это группа или

блок кадров, размещенных в определенном порядке. Каждый блок является независимо воспроизводимой единицей, обычно длиной в несколько секунд.

2. Некоторые старые мобильные устройства и браузеры могут не поддерживать разделение видео на части. Специально для них препроцессор делит видеофайл на GOP.
3. Генерация DAG. Препроцессор генерирует DAG на основе конфигурационных файлов, предоставленных разработчиками клиента. На рис. 14.12 изображено упрощенное представление DAG с двумя узлами и одним ребром:



Рис. 14.12

Это представление DAG сгенерировано из двух конфигурационных файлов, показанных на рис. 14.13.

```

task {
    name 'download-input'
    type 'Download'
    input {
        url config.url
    }
    output { it->
        context.inputVideo = it.file
    }
    next 'transcode'
}

task {
    name 'transcode'
    type 'Transcode'
    input {
        input context.inputVideo
        config config.transConfig
    }
    output { it->
        context.file = it.outputVideo
    }
}
  
```

Рис. 14.13 (источник: [9])

4. Кэширование данных. Препроцессор служит кэшем для сегментированных видеофайлов. Для повышения надежности он записывает GOP и метаданные во временное хранилище. Если кодирование завершится неудачно, система сможет воспользоваться сохраненными данными для повторного выполнения операций.

### Планировщик DAG

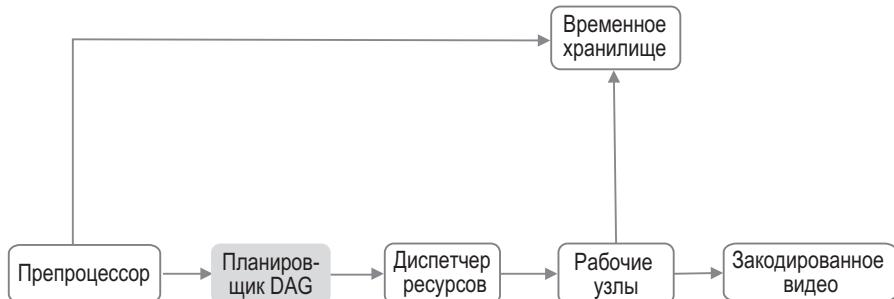


Рис. 14.14

Планировщик DAG делит задания в графе на этапы и записывает их в очередь заданий, принадлежащую диспетчеру ресурсов. На рис. 14.15 показан пример того, как работает планировщик DAG.

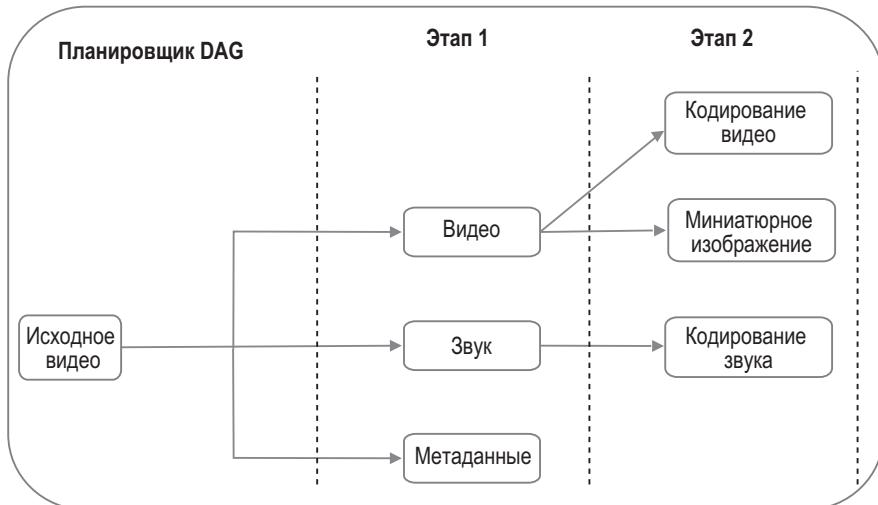


Рис. 14.15

Как видно на рис. 14.15, исходный видеофайл проходит три этапа обработки. На этапе 1 он делится на видео, звук и метаданные. На этапе 2 выполняется два процесса: кодирование видео и генерация миниатюры. В рамках этого этапа также происходит кодирование аудиофайла.

## Диспетчер ресурсов

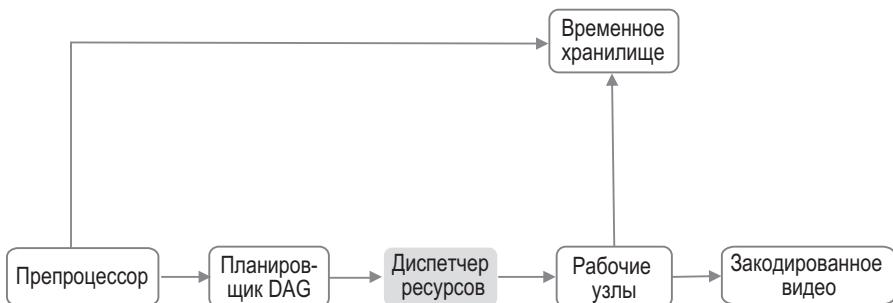


Рис. 14.16

Диспетчер ресурсов следит за тем, чтобы ресурсы выделялись эффективно. Как показано на рис. 14.17, он содержит три очереди и планировщик заданий.

- Очередь заданий. Это приоритетная очередь заданий, которые нужно выполнить.
- Очередь рабочих узлов. Это приоритетная очередь, содержащая информацию о загруженности рабочих узлов.
- Очередь выполнения. Содержит информацию о заданиях, выполняющихся в настоящий момент, и рабочих узлах, которые их выполняют.
- Планировщик заданий. Он выбирает оптимальный рабочий узел и поручает ему выполнить подходящее задание.

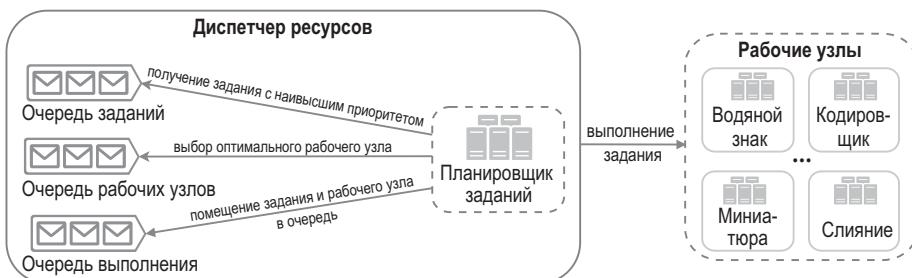


Рис. 14.17

Диспетчер ресурсов работает следующим образом:

- получает задание с наивысшим приоритетом из соответствующей очереди;
- выбирает из очереди оптимальный рабочий узел для выполнения полученного задания;
- поручает выбранному рабочему узлу выполнить задание;
- связывает между собой информацию о задании и рабочем узле и записывает ее в очередь выполнения;
- удаляет задание из очереди выполнения, как только оно завершилось.

### Рабочие узлы

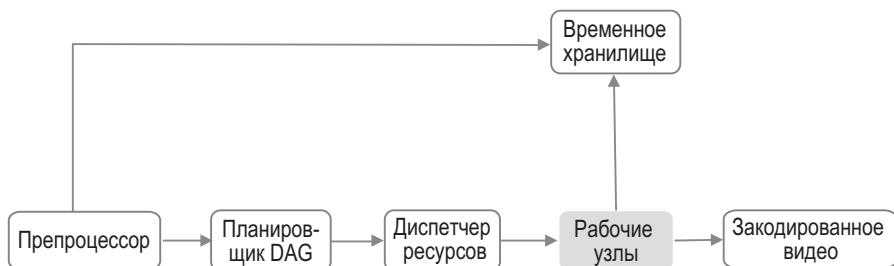


Рис. 14.18

Рабочие узлы выполняют задания, перечисленные в DAG. Разные узлы могут быть предназначены для разных заданий (рис. 14.19).

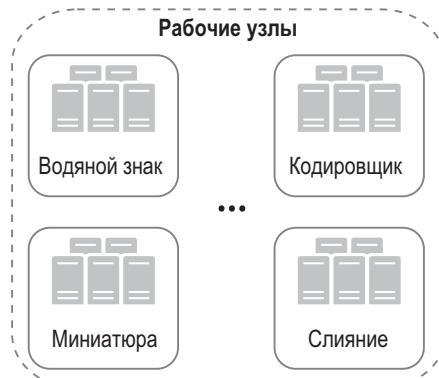


Рис. 14.19

## Временное хранилище

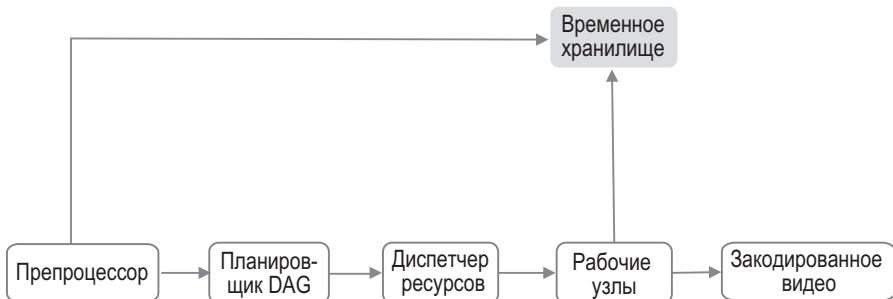


Рис. 14.20

Здесь используется несколько систем хранения данных. Выбор той или иной системы зависит от таких факторов, как тип, размер и время жизни данных, частота доступа к ним и т. д. Например, метаданные часто нужны рабочим узлам и обычно имеют небольшой размер, поэтому их целесообразно кэшировать в памяти. Видео- и аудиоданные записываются в хранилище BLOB-объектов. После завершения обработки видео соответствующие данные удаляются из временного хранилища.

## Закодированное видео

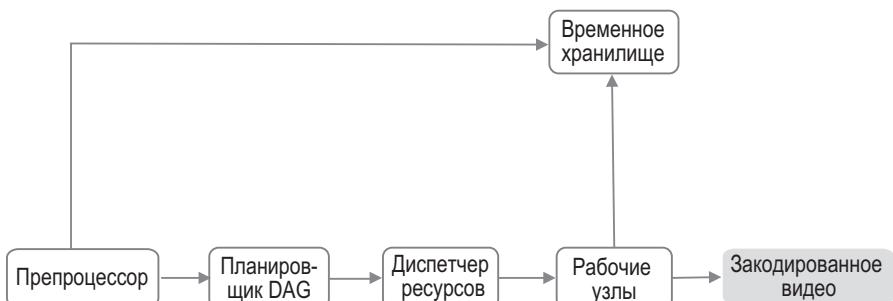


Рис. 14.21

Закодированное видео является конечным результатом процесса кодирования. Оно может иметь вид файла с именем наподобие `funny_720p.mp4`.

## Оптимизация системы

На этом этапе у вас уже должно быть четкое представление о процессах загрузки, стриминга и перекодирования видео. Теперь мы оптимизируем некоторые аспекты системы, включая ее скорость, безопасность и стоимость обслуживания.

### Оптимизация скорости: распараллеливание загрузки видео

Загружать видео как единое целое неэффективно. Мы можем разделить его на мелкие блоки, выровненные по GOP, как показано на рис. 14.22.



Рис. 14.22

Это сделает процесс загрузки быстрым и позволит его возобновить, если что-то пойдет не так. Разделение видеофайла на GOP можно выполнить на стороне клиента, чтобы ускорить загрузку (рис. 14.23).

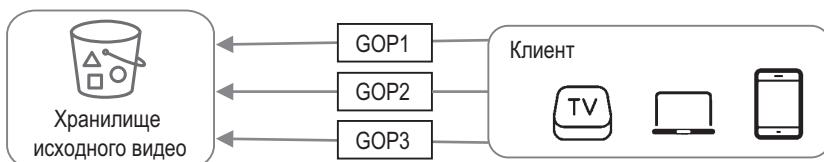


Рис. 14.23

### Оптимизация скорости: размещение центров загрузки поблизости от пользователя

Загрузку также можно ускорить за счет нескольких центров обработки данных, разбросанных по всему миру (рис. 14.24). Жители США могут загружать видео в североамериканский ЦОД, а жители Китая — в азиатский. Для этого в качестве центров загрузки мы будем использовать CDN.

### Оптимизация скорости: повсеместное распараллеливание

Обеспечить низкую латентность непросто. Еще один способ оптимизации состоит в разработке слабосвязанной системы с высокой степенью параллелизма.

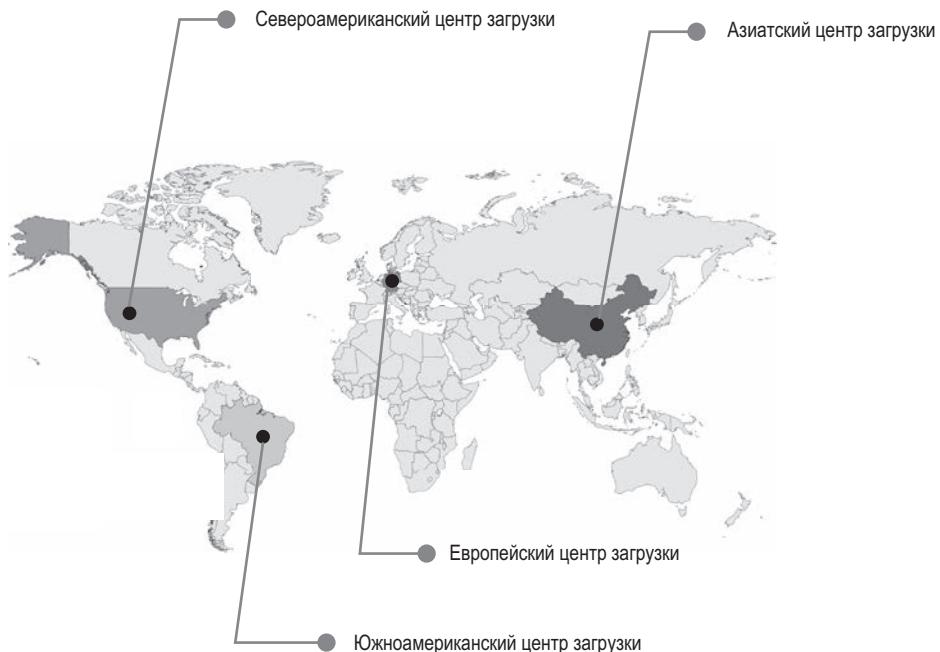


Рис. 14.24

Чтобы как следует распараллелить нашу архитектуру, в нее необходимо внести некоторые изменения. Давайте взглянем на то, как данные передаются из хранилища исходного видео в CDN. Этот процесс показан на рис. 14.25. Как видите, его результат зависит от ввода на предыдущем этапе. Эта зависимость затрудняет распараллеливание.

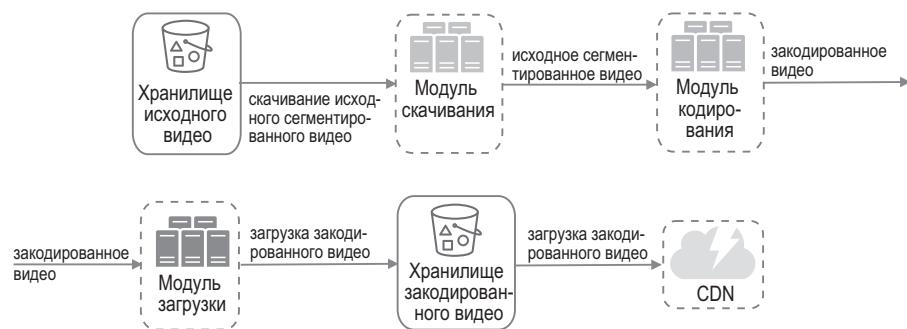


Рис. 14.25

Для ослабления связности системы мы добавили очереди сообщений, как показано на рис. 14.26. Чтобы объяснить, как очереди сообщений ослабляют связность системы, рассмотрим пример.

- Когда очереди сообщений не было, модулю кодирования приходилось ждать возвращения вывода от модуля скачивания.
- После добавления очереди сообщений модулю кодирования больше не нужно ждать, пока модуль скачивания вернет вывод. Если в очереди сообщения находятся события, модуль кодирования может выполнить соответствующие задания параллельно.

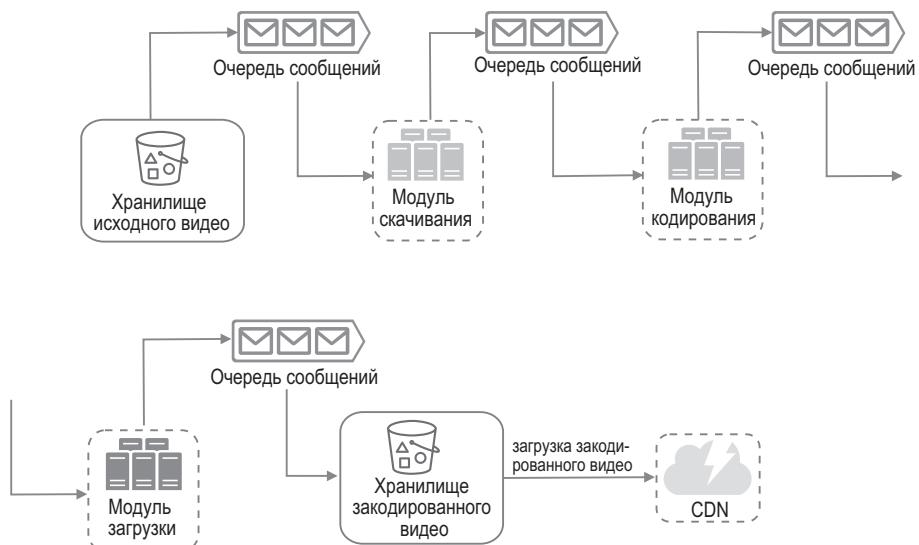


Рис. 14.26

### Оптимизация безопасности: предварительно поданный URL-адрес загрузки

Безопасность — один из важнейших аспектов любого продукта. Чтобы гарантировать, что видео загружается туда, куда нужно, и теми, кому это позволено, мы вводим понятие предварительно подписанного URL-адреса, как показано на рис. 14.27.

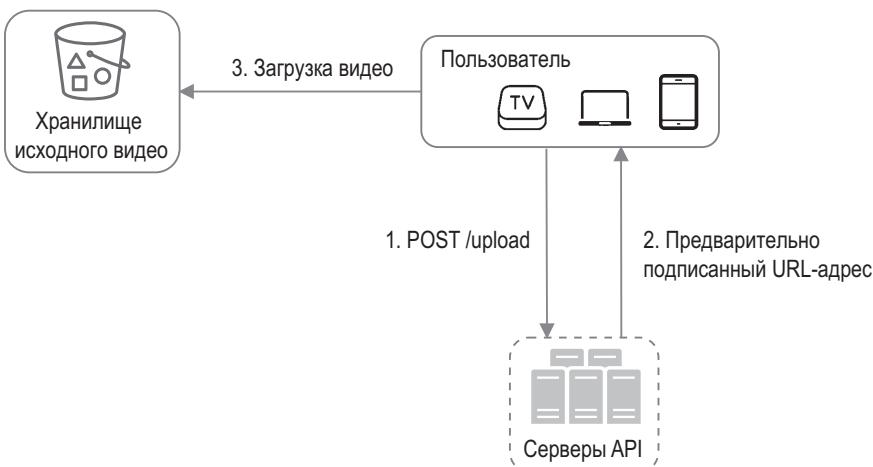


Рис. 14.27

Обновленный процесс загрузки выглядит так:

1. Клиент отправляет HTTP-запрос серверам API, чтобы получить предварительно подписанный URL-адрес, который дает доступ к соответствующему объекту. Термин «предварительно подписанный URL-адрес» встречается при загрузке файлов в Amazon S3. Другие облачные провайдеры могут называть этот механизм иначе. Например, хранилище BLOB-объектов в Microsoft Azure поддерживает ту же возможность, но называет ее подписью общего доступа [10].
2. Серверы API возвращают предварительно подписанный URL-адрес.
3. Получив ответ, клиент загружает видео с его помощью.

### Оптимизация безопасности: защита видеороликов

Многие производители контента с неохотой публикуют свои видео в интернете, опасаясь, что их оригинальная работа будет похищена. Для защиты авторских прав можно внедрить один из следующих механизмов.

- Технические средства защиты авторских прав (digital rights management, DRM). Три крупнейшие системы DRM – Apple FairPlay, Google Widevine и Microsoft PlayReady.
- Шифрование AES. Вы можете зашифровать видео и настроить политику авторизации. Зашифрованные видео будут расшифровываться во время воспроизведения. Это гарантирует, что их будут просматривать только авторизованные пользователи.
- Водяные знаки. Это изображение, которое наносится поверх видео и содержит информацию, позволяющую его идентифицировать. Это может быть логотип или название компании.

### Оптимизация стоимости обслуживания

Сеть CDN – неотъемлемый компонент нашей системы. Она обеспечивает быструю доставку видео в глобальных масштабах. Но, как показывают наши приблизительные оценки, CDN стоит дорого, особенно если данных много. Как можно снизить расходы?

Согласно проведенному ранее исследованию, видеопотоки в YouTube распределяются по принципу «длинного хвоста» [11] [12]. Это означает, что большое количество просмотров приходится на горстку популярных видеороликов, тогда как многие другие видео просматриваются редко или вообще никогда не просматриваются. Исходя из этого наблюдения, можно внести несколько оптимизаций:

1. Только самые популярные видеоролики раздаются из CDN, а все остальные доступны на серверах хранения видеофайлов большой емкости (рис. 14.28).
2. Если контент не очень популярный, нам, возможно, не придется хранить множество его закодированных версий. Короткие видеоролики могут кодироваться по требованию.
3. Некоторые видео пользуются популярностью только в определенных регионах. Их не нужно распространять за пределами этих регионов.
4. Можно создать собственную сеть CDN, как это сделала компания Netflix, и наладить партнерство с интернет-провайдерами. Построение CDN – это масштабный проект, но для крупных ком-

паний, занимающихся стримингом видео, такой вариант может подойти. Интернет-провайдеры, такие как Comcast, AT&T или Verizon, предоставляют свои услуги по всему миру и находятся близко к пользователям. Партнерство с ними может улучшить качество воспроизведения и снизить денежные расходы на сетевой трафик.

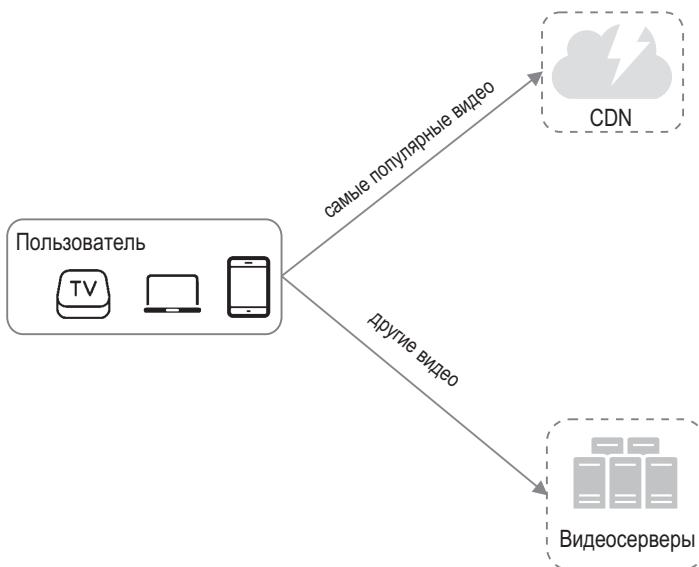


Рис. 14.28

Все эти оптимизации основаны на популярности контента, его размере, на том, как к нему обращаются пользователи, и т. д. Прежде чем что-либо оптимизировать, необходимо исследовать тенденции и закономерности в сфере видео. Вот несколько интересных статей на эту тему: [12] [13].

## Обработка ошибок

В крупномасштабных системах ошибки неизбежны. Чтобы обеспечить высокий уровень отказоустойчивости, ошибки должны обрабатываться контролируемым образом и работа после них должна быстро возобновляться. Есть два вида ошибок.

- Некритические. В качестве примера можно привести неудачное кодирование сегмента видео. В таких случаях обычно делается несколько повторных попыток выполнить операцию. Если проблема продолжает возникать и системе не удается ее исправить, клиенту возвращается код соответствующей ошибки.
- Критические. Примером может служить некорректный формат видео. В таких случаях система останавливает задание, относящееся к этому видео, и возвращает клиенту код соответствующей ошибки.

Ниже перечислены типичные ошибки, возникающие в каждом компоненте нашей системы.

- Ошибка загрузки: повторить попытку несколько раз.
- Ошибка разделения видео: если клиенты более старых версий не могут разделить видео с выравниванием по GOP, видеофайл загружается на сервер целиком и процесс разделения выполняется на стороне сервера.
- Ошибка перекодирования: повторить попытку.
- Ошибка препроцессора: заново сгенерировать диаграмму DAG.
- Ошибка планировщика DAG: запланировать задание заново.
- Отказ очереди диспетчера ресурсов: использовать реплику.
- Отказ рабочего узла: повторить выполнение задания на другом узле.
- Отказ сервера API: серверы API не хранят свое состояние, поэтому запрос перенаправляется к другому серверу API.
- Отказ сервера для кэширования метаданных: данные реплицируются несколько раз. Если один сервер выходит из строя, вы можете обратиться за данными к другим серверам. Мы можем заменить отказавший кэширующий сервер новым.
- Отказ сервера БД для кэширования метаданных:
  - ◆ отказ ведущего сервера: сделать ведущим один из ведомых серверов;
  - ◆ отказ ведомого сервера: для чтения можно использовать другой ведомый сервер. Сервер базы данных, вышедший из строя, можно заменить новым.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы представили архитектуру для сервисов стриминга видео наподобие YouTube. Если в конце интервью еще остается время, можно обсудить несколько дополнительных аспектов.

- Масштабирование уровня API. Поскольку серверы API не хранят свое состояние, их легко масштабировать горизонтально.
- Масштабирование базы данных. Можно упомянуть о репликации и сегментировании базы данных.
- Прямая трансляция. Это процесс записи и трансляции видео в реальном времени. Изначально наша система не рассчитана на такое вещание, но эта функция имеет некоторые сходства со стримингом обычного видео: оба процесса требуют загрузки, кодирования и потоковой передачи. Есть и заметные отличия:
  - ◆ у прямой трансляции повышенные требования к латентности, поэтому для нее может понадобиться другой протокол потоковой передачи;
  - ◆ у прямой трансляции не настолько высокие требования к параллельной обработке, поскольку небольшие блоки данных и так обрабатываются в реальном времени;
  - ◆ прямая трансляция требует другого подхода к обработке ошибок. Любой механизм, обрабатывающий ошибки слишком долго, не подходит.
- Запрет доступа к видео. Видеоролики, нарушающие авторские права, содержащие порнографию или нарушающие закон каким-то другим образом, должны удаляться. Некоторые из них могут быть обнаружены в процессе загрузки, о других могут сообщать сами пользователи.

Поздравляем, вы проделали длинный путь и можете гордиться собой. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] YouTube by the numbers: <https://www.omnicoreagency.com/youtube-statistics/>
- [2] 2019 YouTube Demographics: <https://blog.hubspot.com/marketing/youtube-demographics>
- [3] Цены на Amazon CloudFront: <https://aws.amazon.com/ru/cloudfront/pricing/>
- [4] Netflix на AWS: <https://aws.amazon.com/solutions/case-studies/netflix/>
- [5] Официальный веб-сайт Akamai: <https://www.akamai.com/>
- [6] Двоичные большие объекты: <https://ru.wikipedia.org/wiki/BLOB>
- [7] Here's What You Need to Know About Streaming Protocols: <https://www.dacast.com/blog/streaming-protocols/>
- [8] SVE: Distributed Video Processing at Facebook Scale: <https://www.cs.princeton.edu/~wlloyd/papers/sve-sosp17.pdf>
- [9] Архитектура обработки видео в Weibo (на китайском языке): <https://www.ipyun.com/opentalk/399.html>
- [10] Делегат доступа с общей подписью доступа: <https://docs.microsoft.com/ru-ru/rest/api/storageservices/delegate-access-with-shared-access-signature>
- [11] YouTube scalability talk by early YouTube employee: <https://www.youtube.com/watch?v=w5WVu624fY8>
- [12] Understanding the characteristics of internet short video sharing: A youtube-based measurement study. <https://arxiv.org/pdf/0707.3670.pdf>
- [13] Content Popularity for Open Connect: <https://netflixtechblog.com/content-popularity-for-open-connect-b86d56f613b>

---

# 15

---

## ПРОЕКТИРОВАНИЕ GOOGLE DRIVE

В последние годы большой популярностью пользуются такие облачные сервисы хранения данных, как Google Drive, Dropbox, Microsoft OneDrive и Apple iCloud. В этой главе вам предлагается спроектировать Google Drive.

Прежде чем переходить к проектированию, давайте немного подумаем о том, что собой представляет эта система. Google Drive — это файловое хранилище и сервис синхронизации, который позволяет хранить документы, фотографии, видео и другие файлы в облаке. Доступ к своим файлам можно осуществлять с любого компьютера, смартфона и планшета. Вы можете легко делиться этими файлами с друзьями, членами семьи и коллегами [1]. На рис. 15.1 и 15.2 показаны браузерная и, соответственно, мобильная версии Google Drive.

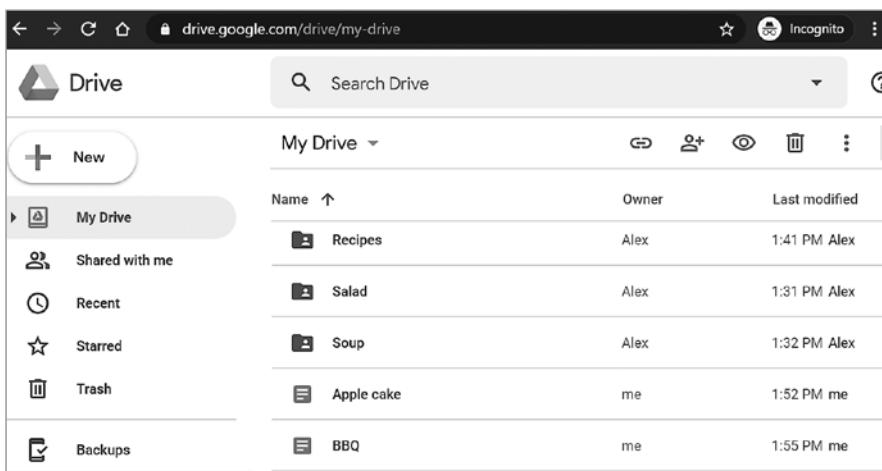


Рис. 15.1

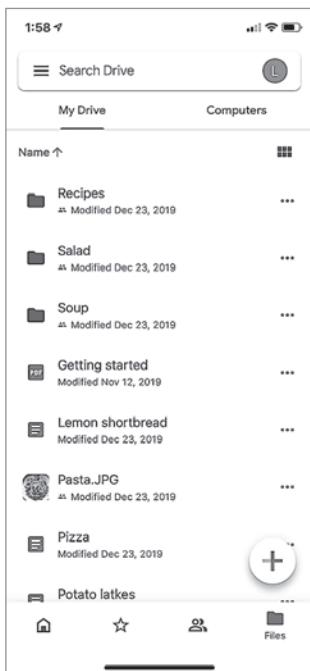


Рис. 15.2

## ШАГ 1: ПОНЯТЬ ЗАДАЧУ И ОПРЕДЕЛИТЬ МАСШТАБ РЕШЕНИЯ

Проектирование Google Drive — масштабный проект, поэтому сначала необходимо определиться с объемом работ.

**Кандидат:** «Какие функции самые важные?»

**Интервьюер:** «Загрузка, скачивание и синхронизация файлов, а также уведомления».

**Кандидат:** «Это приложение должно быть мобильным, браузерным или и тем и другим?»

**Интервьюер:** «И тем и другим».

**Кандидат:** «Какие форматы файлов поддерживаются?»

**Интервьюер:** «Любые типы файлов».

**Кандидат:** «Должны ли файлы шифроваться?»

**Интервьюер:** «Да, файлы в хранилище должны быть зашифрованы».

**Кандидат:** «Ограничено ли размер файлов?»

**Интервьюер:** «Да, размер файла не должен превышать 10 ГБ».

**Кандидат:** «Сколько пользователей у этого продукта?»

**Интервьюер:** «10 миллионов DAU».

В этой главе мы сосредоточимся на следующих возможностях.

- Добавление файлов. Самый простой способ добавить файл — это перетащить его в Google Drive.
- Скачивание файлов.
- Синхронизация файлов между разными устройствами. Файл, добавленный на одном устройстве, автоматически синхронизируется с другими устройствами.
- Просмотр истории изменений файлов.
- Обмен файлами с друзьями, семьей и коллегами.
- Отправка уведомлений при редактировании и удалении файлов, а также в случае, если кто-то поделился с вами своими файлами.

В этой главе не обсуждаются следующие возможности.

- Редактирование и совместная работа над документами Google Docs. Google Docs позволяет редактировать один и тот же документ сразу нескольким людям. Эта функция не входит в нашу архитектуру.

Помимо уточнения требований необходимо как следует понять нефункциональные характеристики системы.

- Надежность. Чрезвычайно важна для систем хранения. Потеря данных недопустима.
- Быстрая синхронизация. Если файлы синхронизируются слишком долго, пользователи теряют терпение и отказываются от продукта.
- Потребление трафика. Если продукт потребляет много трафика без реальной необходимости, пользователи будут недовольны, особенно если их мобильный тарифный план ограничен.

- Масштабируемость. Система должна справляться с большими объемами трафика.
- Высокая доступность. Пользователи не должны терять доступ к системе, даже если некоторые из ее серверов выходят из строя, демонстрируют низкую производительность или испытывают проблемы с сетью.

### Приблизительные оценки

- Предположим, что система имеет 50 миллионов зарегистрированных пользователей и 10 миллионов DAU.
- Каждый пользователь получает 10 Гб свободного места.
- Допустим, пользователи загружают в среднем по 2 файла в день. Средний размер файла составляет 500 Кб.
- Чтение и запись имеют равное соотношение.
- Общий размер выделенного пространства:  $50 \text{ миллионов} * 10 \text{ Гб} = 500 \text{ петабайтов}$ .
- QPS для API загрузки:  $10 \text{ миллионов} * 2 \text{ загрузки} / 24 \text{ часа} / 3600 \text{ секунд} = \sim 240$ .
- Пиковый показатель QPS:  $\text{QPS} * 2 = 480$ .

## ШАГ 2: ПРЕДЛОЖИТЬ ОБЩЕЕ РЕШЕНИЕ И ПОЛУЧИТЬ СОГЛАСИЕ

Вместо того чтобы демонстрировать общую диаграмму архитектуры с самого начала, мы пойдем другим путем. Начнем с простого: разместим все на одном сервере. Затем постепенно расширим систему для поддержки миллионов пользователей. Это упражнение позволит вам освежить знания некоторых важных тем, рассмотренных в книге.

Изначально наша конфигурация будет состоять из одного сервера и иметь такой вид:

- веб-сервер для загрузки и скачивания файлов;

- БД для хранения таких метаданных, как информация о пользователях, аутентификации, файлы и т. д.;
- система хранения файлов. Мы выделим для нее 1 Тб.

Мы потратим несколько часов на настройку веб-сервера Apache и базы данных MySQL. Загружаемые файлы будут храниться в директории `drive/`, внутри которой находится ряд других директорий, которые называются пространствами имен. Каждое пространство имен содержит все файлы, загруженные конкретным пользователем. Копии файлов на сервере имеют те же имена, что и оригиналы. Для однозначной идентификации файла или папки достаточно объединить пространство имен и относительный путь.

На рис. 15.3 показан пример директории `drive/` (слева) и ее развернутое представление (справа).

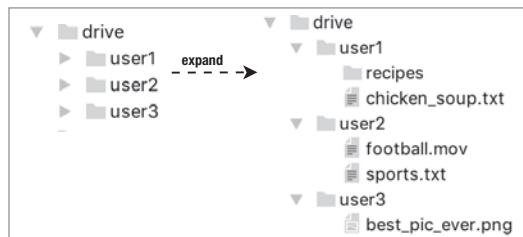


Рис. 15.3

## API

Нам в первую очередь нужно три API: для загрузки, скачивания и получения истории изменений файлов.

### 1. Загрузка файлов в Google Drive.

Поддерживаются два типа загрузки:

- простая загрузка. Предназначена для маленьких файлов;
- возобновляемая загрузка. Используется для больших файлов, когда существует высокий риск разрыва сетевого соединения.

Пример API для возобновляемой загрузки: <https://api.example.com/files/upload?uploadType=resumable>.

Параметры:

- `uploadType=resumable`;
- `data`: локальный файл, который нужно загрузить.

Процесс возобновления загрузки состоит из следующих трех шагов [2]:

- отправить начальный запрос для получения возобновляемого URL-адреса;
- загрузить данные и отследить состояние загрузки;
- возобновить загрузки в случае прерывания.

## **2. Скачивание файла из Google Drive.**

Пример API: <https://api.example.com/files/download>.

Параметры:

- `path`: путь к скачиваемому файлу. Пример:

```
{  
  "path": "/recipes/soup/best_soup.txt"  
}
```

## **3. Получение истории изменений файла.**

Пример API: [https://api.example.com/files/list\\_revisions](https://api.example.com/files/list_revisions).

Параметры:

- `path`: путь к файлу, историю изменений которого вы хотите получить;
- `limit`: максимальное количество версий файла, которые нужно вернуть. Пример:

```
{  
  "path": "/recipes/soup/best_soup.txt",  
  "limit": 20  
}
```

Все API используют HTTPS и требуют аутентификации пользователя. SSL (Secure Sockets Layer — «слой защищенных сокетов») защищает передачу данных между клиентом и серверами системы.

### Отказываемся от архитектуры с одним сервером

Файлов загружается все больше и больше, и в какой-то момент мы получим предупреждение о нехватке места, как показано на рис. 15.4.



Рис. 15.4

У нас осталось всего 10 Мб свободного места! Это чрезвычайная ситуация, так как пользователи больше не могут загружать свои файлы. Первое, что приходит на ум, это сегментировать данные так, чтобы они хранились на разных серверах. На рис. 15.5 показан пример шардинга на основе `user_id`.

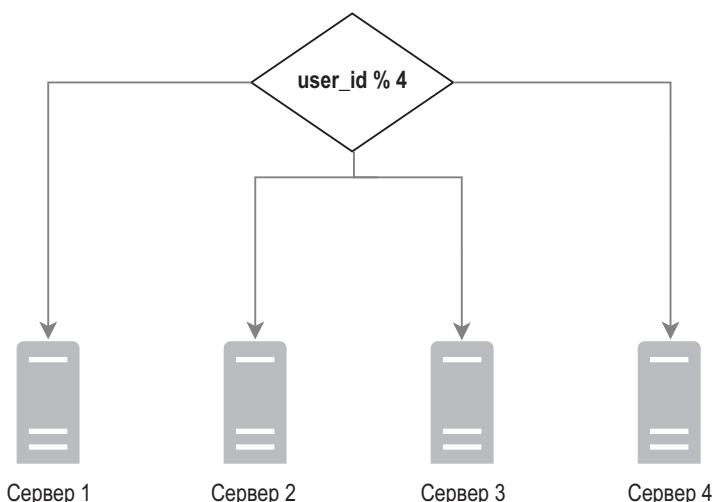


Рис. 15.5

Вы проработали всю ночь, чтобы настроить шардинг и подробный мониторинг базы данных. Все опять работает как следует. Вы потушили пожар, но вам не дает покоя мысль о том, что перебои в работе сервера хранилища могут привести к потере данных. Вы начали интересоваться мнением своих коллег, и один ваш друг, гуру в области серверных систем, сказал, что многие ведущие компании вроде Netflix и Airbnb используют для хранения данных Amazon S3. «Amazon S3 (Simple Storage Service — “простой сервис хранения данных”) — это объектное хранилище, лидирующее в сферах масштабируемости, доступности данных, безопасности и производительности» [3]. Вы решаете исследовать эту технологию, чтобы понять, подходит ли она вам.

После продолжительного чтения вы как следует разобрались в системе хранения S3 и решили разместить в ней файлы своей системы. Amazon S3 поддерживает репликацию в рамках одного или нескольких регионов. Регион — это географическая область, в которой находятся центры обработки данных AWS. Как видно на рис. 15.6, данные могут реплицироваться как в одном регионе (слева), так и между разными регионами (справа). Резервные копии файлов хранятся в нескольких регионах, чтобы предотвратить потерю данных и обеспечить их доступность. Бакет — это аналог папки в файловой системе.

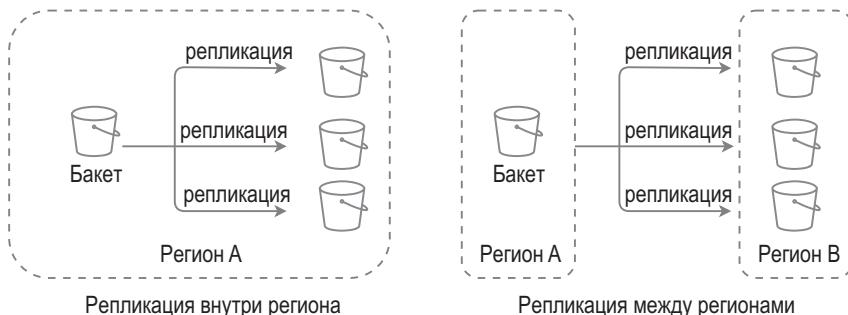


Рис. 15.6

Разместив файлы в S3, вы, наконец, можете спать спокойно, не волнуясь о потере данных. Чтобы подобные проблемы не возникали в будущем, вы проводите углубленное исследование в поиске участков системы, которые можно улучшить. Вот к каким выводам вы пришли.

- Балансировщик нагрузки. Добавление балансировщика нагрузки позволяет равномерно распределить сетевой трафик. Если сервер выходит из строя, трафик распределяется автоматически.
- Веб-серверы. Благодаря наличию балансировщика нагрузки веб-серверы можно легко добавлять/удалять в зависимости от нагрузки.
- БД метаданных. Чтобы избавиться от единой точки отказа, базу данных можно вынести за пределы сервера. Вместе с тем можно настроить репликацию и сегментирование данных в соответствии с требованиями к доступности и масштабируемости.
- Хранилище файлов. Для хранения файлов используется Amazon S3. Чтобы обеспечить доступность и устойчивость, файлы реплицируются в двух разных географических регионах.

После внесения перечисленных выше улучшений вы успешно вынесли веб-серверы, БД метаданных и хранилище файлов за пределы единого сервера. Обновленная архитектура показана на рис. 15.7.

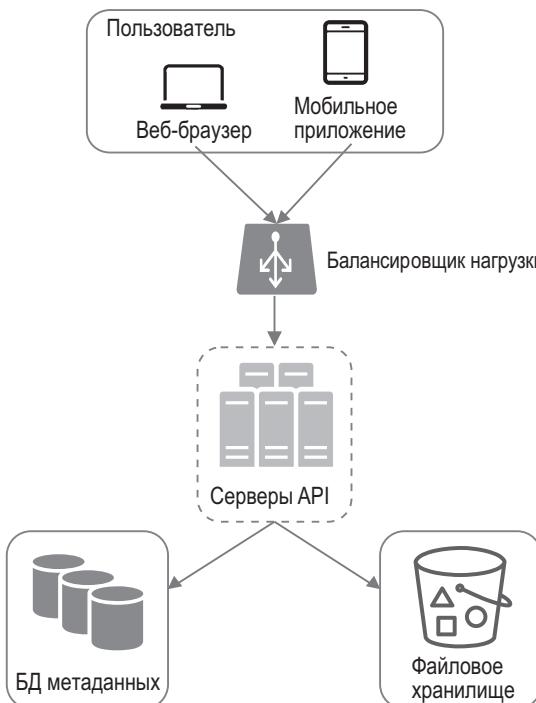


Рис. 15.7

## Конфликты синхронизации

В такой крупной системе хранения данных, как Google Drive, время от времени случаются конфликты синхронизации. Это происходит, когда два пользователя одновременно изменяют один и тот же файл или папку. Как разрешить такой конфликт? Вот наша стратегия: побеждает та версия, которая обрабатывается первой, а обработка другой завершается конфликтом. Пример конфликта синхронизации показан на рис. 15.8.

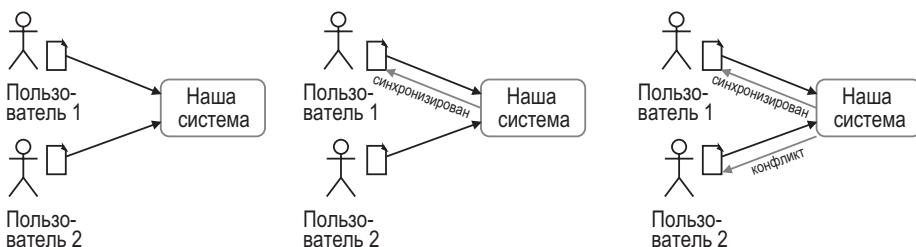


Рис. 15.8

Как видно на рис. 15.8, пользователь 1 и пользователь 2 одновременно пытаются обновить один и тот же файл, но наша система обрабатывает файл пользователем 1 первым. Операция обновления, выполненная пользователем 1, принимается, а аналогичная операция пользователя 2 приводит к конфликту синхронизации. Как разрешить этот конфликт? Наша система предлагает пользователю 2 обе версии файла: его локальную копию и последнюю версию, взятую с сервера (рис. 15.9). Пользователь 2 может выполнить слияние файлов или выбрать одну из версий.

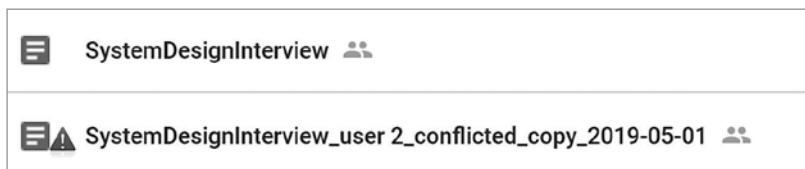


Рис. 15.9

Когда один документ редактируется сразу несколькими пользователями, это затрудняет его синхронизацию. Заинтересованные читатели могут обратиться к справочным материалам [4] [5].

## Общая архитектура

На рис. 15.10 проиллюстрирована предложенная нами общая архитектура. Давайте проанализируем каждый ее компонент.

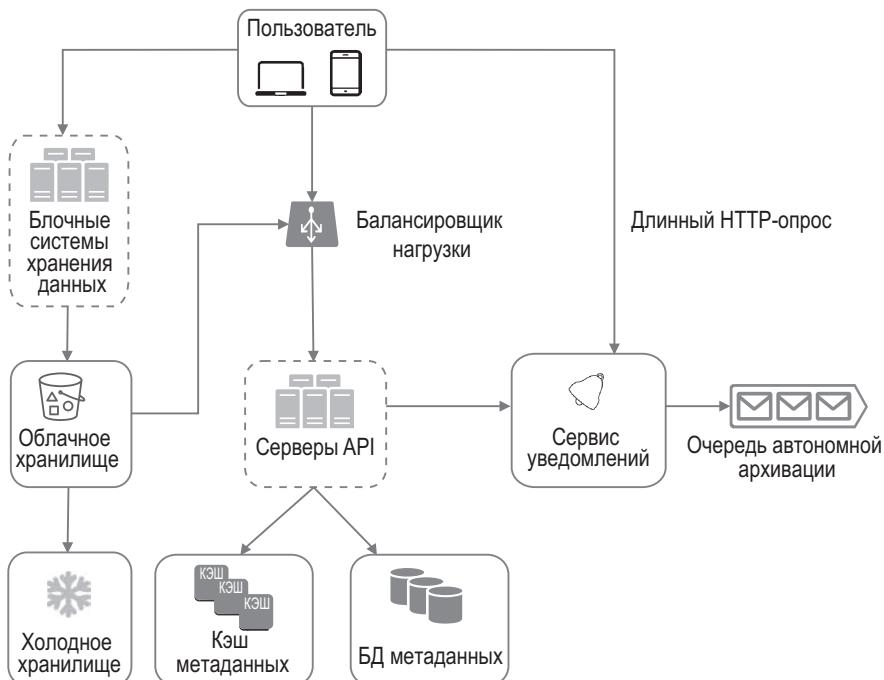


Рис. 15.10

- **Пользователь.** Взаимодействует с системой с помощью браузера или мобильного приложения.
- **Блочные системы хранения данных.** Загружают блоки данных в облачное хранилище. Блочное хранилище – это технология хранения файлов в облачных окружениях. Файл может быть разделен на несколько блоков, каждый из которых имеет уникальный хеш и хранится в нашей БД метаданных. Каждый блок обрабатывается как независимый объект и записывается в нашу систему хранения (S3). Чтобы восстановить файл, блоки соединяются в определенном порядке. Что касается размера блоков, мы возьмем за пример сервис Dropbox, в котором блок не может превышать 4 Мб [6].

- **Облачное хранилище.** Файл делится на блоки меньшего размера, которые записываются в облачное хранилище.
- **Холодное хранилище.** Компьютерная система, предназначенная для хранения неактивных данных (файлов, к которым не обращаются на протяжении длительного времени).
- **Балансировщик нагрузки.** Равномерно распределяет запросы между серверами API.
- **Серверы API.** Отвечают почти за все, кроме процесса загрузки. Их используют для аутентификации пользователей, управления пользовательскими профилями, обновления метаданных файлов и т. д.
- **БД метаданных.** Хранит метаданные пользователей, файлов, блоков, версий и т. д. Пожалуйста, обратите внимание на то, что в этой БД находятся только метаданные, а сами файлы хранятся в облаке.
- **Кэш метаданных.** Некоторые метаданные кэшируются для быстрого доступа.
- **Сервис уведомлений.** Система типа «издатель–подписчик», которая передает данные клиентам при возникновении определенных событий. В нашем случае этот сервис оповещает соответствующих пользователей о добавлении/редактировании/удалении файла кем-то другим, чтобы они могли просмотреть последние изменения.
- **Очередь автономной архивации.** Если клиент находится вне сети и не может получить последние изменения, соответствующая информация попадает в очередь автономной архивации. Это позволяет синхронизировать изменения, когда пользователь снова подключается к сети.

Мы обсудили общую архитектуру Google Drive. Некоторые из ее компонентов довольно сложные и заслуживают отдельного внимания; мы подробно рассмотрим их в следующем разделе.

## ШАГ 3: ПОДРОБНОЕ ПРОЕКТИРОВАНИЕ

В этом разделе мы подробно рассмотрим такие темы: блочные системы хранения данных, БД метаданных, процесс загрузки, процесс скачивания, сервис уведомлений, экономия места в хранилище и обработка сбоев.

## Блочные системы хранения данных

Если большой файл, который регулярно изменяется, передавать целиком при каждом обновлении, на это будет уходить много трафика. Чтобы сэкономить трафик, предлагаем две оптимизации.

- Синхронизация изменений. При редактировании файла синхронизируются только измененные блоки. Для этого используется алгоритм синхронизации [7] [8].
- Сжатие. Сжатие блоков может существенно уменьшить размер данных. Следовательно, к блокам применяется алгоритм сжатия с учетом типа файла. Например, gzip и bzip2 используются для сжатия текстовых файлов. Для сжатия изображений и видеофайлов требуются другие алгоритмы.

В нашей системе блочные системы хранения данных берут на себя основную работу по загрузке файлов. Они обрабатывают файлы, переданные клиентами, разбивая их на блоки, каждый из которых затем сжимается и шифруется. В систему хранения загружается не весь файл целиком, а лишь измененные блоки.

На рис. 15.11 показано, что делает блочная система хранения при добавлении нового файла.

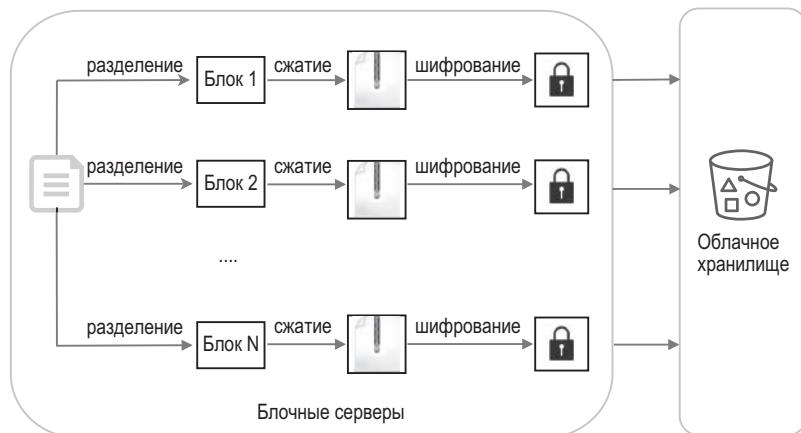


Рис. 15.11

- Файл разделяется на блоки меньшего размера.
- Каждый блок сжимается с помощью алгоритмов сжатия.
- Чтобы обеспечить безопасность, перед отправкой в облачное хранилище каждый блок шифруется.
- Блоки загружаются в облачное хранилище.

На рис. 15.12 проиллюстрирована дельта-синхронизация — процесс, в ходе которого в облачное хранилище передаются только измененные блоки, 2 и 5 (выделены на диаграмме).

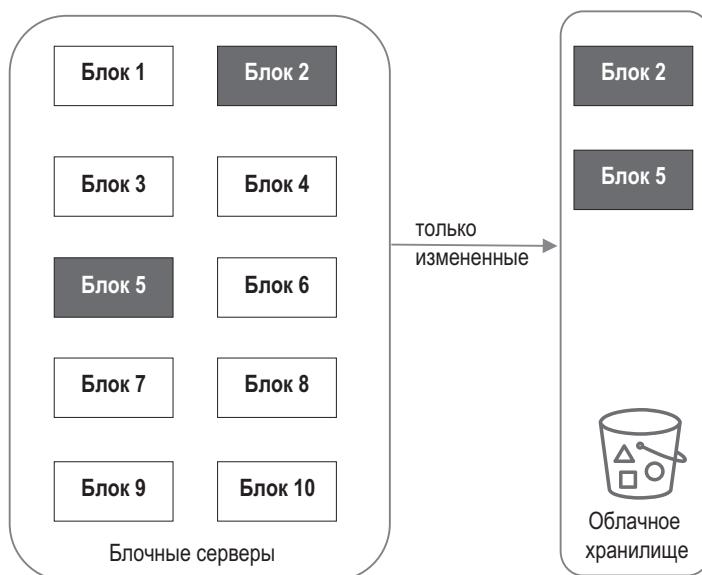


Рис. 15.12

Блочные системы хранения данных позволяют сэкономить сетевой трафик за счет синхронизации изменений и сжатия.

### Требование строгой согласованности

Наша система должна поддерживать строгую согласованность по умолчанию. Нельзя допустить, чтобы один файл одновременно выглядел

по-разному на разных клиентах. Система должна обеспечивать строгую согласованность уровней кэша и БД метаданных.

Системы кэширования на основе оперативной памяти по умолчанию используют модель отложенной согласованности; это означает, что разные реплики могут хранить разные данные. Ради строгой согласованности мы должны сделать следующее:

- позаботиться о согласованности реплик и ведущего узла;
- аннулировать кэш при записи в базу данных, чтобы закэшированные значения совпадали с теми, которые находятся в БД.

Реляционные базы данных обладают свойствами ACID (Atomicity, Consistency, Isolation, Durability — «атомарность, согласованность, изолированность, прочность»), поэтому в них легко достичь строгой согласованности [9].

А вот в NoSQL свойства ACID по умолчанию не поддерживаются, поэтому их необходимо внедрять в логику синхронизации программным образом. Мы выбрали для нашей архитектуры реляционные базы данных, так как они изначально поддерживают ACID.

## БД метаданных

На рис. 15.13 показана схема базы данных. Пожалуйста, имейте в виду, что это крайне упрощенный вариант, в котором указаны только самые важные таблицы и интересующие нас поля.

- **user.** Таблица user содержит основную информацию о пользователе, включая его имя, адрес электронной почты, аватар и т. д.
- **device.** Таблица device хранит сведения об устройстве. Поле push\_id используется для отправки и получения мобильных push-уведомлений. Обратите внимание на то, что у пользователя может быть несколько устройств.
- **namespace.** Это пространство имен — корневая директория пользователя.
- **file.** Таблица file содержит все, что относится к последней версии файла.

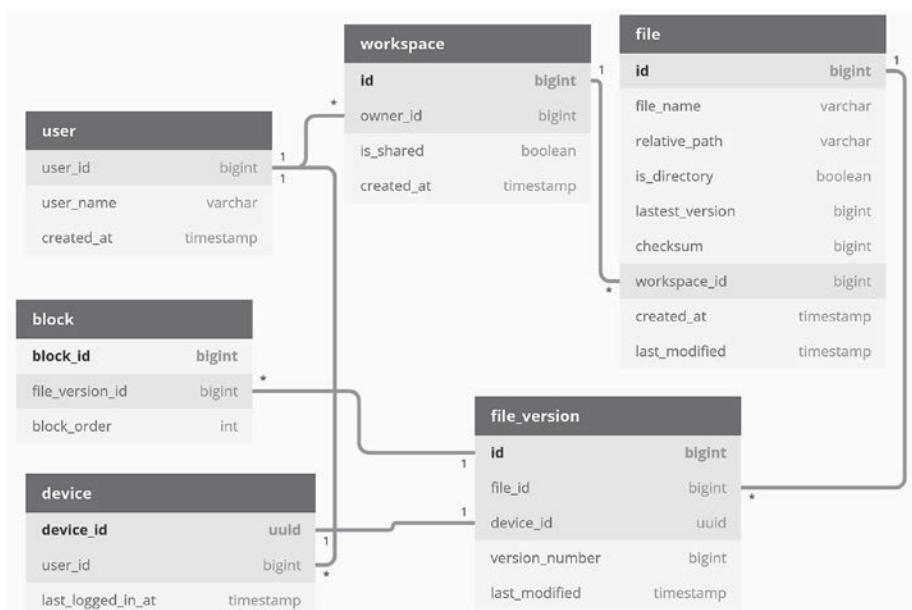


Рис. 15.13

- **file\_version**. Хранит историю изменений файла. Имеющиеся строки доступны только для чтения, благодаря чему поддерживается неизменность истории изменений файла.
- **block**. Хранит все, что связано с блоком файла. Файл любой версии можно воссоздать путем объединения всех его блоков в правильном порядке.

### Процесс загрузки

Давайте посмотрим, что происходит, когда клиент загружает файл. Чтобы лучше понять этот процесс, воспользуемся диаграммой последовательности, представленной на рис. 15.14.

На рис. 15.14 показана параллельная отправка двух запросов: «добавить метаданные файла» и «загрузить файл в облачное хранилище». Оба они исходят от клиента 1.

- Добавить метаданные файла.

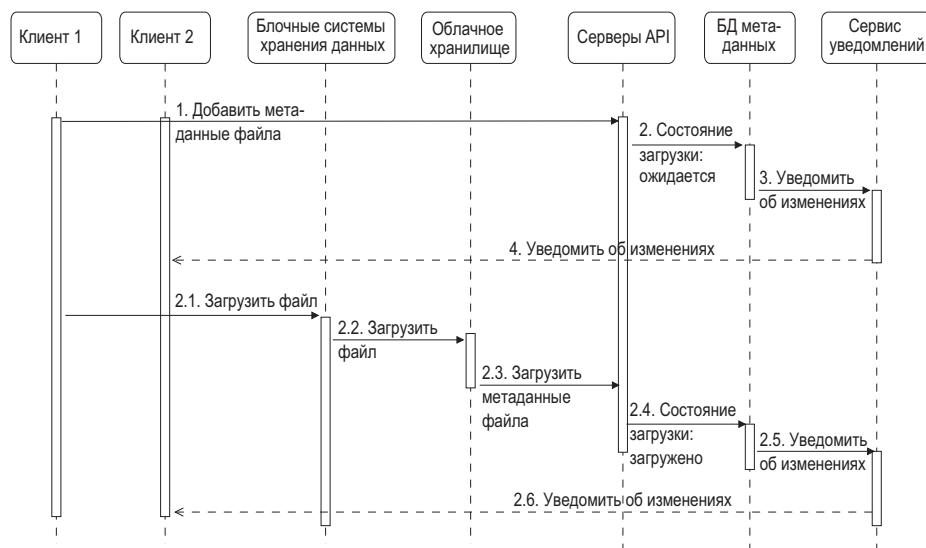


Рис. 15.14

- Клиент 1 отправляет запрос на добавление метаданных нового файла.
- Система сохраняет метаданные нового файла в БД метаданных и присваивает загружаемому файлу состояние «ожидается».
- Система оповещает сервис уведомлений о том, что файл будет загружен.
- Сервис уведомлений оповещает заинтересованные клиенты (клиент 2) о том, что файл загружается.
- Загрузить файлы в облачное хранилище.
  - Клиент 1 загружает содержимое файла на блочные системы хранения данных.
  - Блочные системы хранения данных разбивают файл на блоки, которые затем сжимаются, шифруются и загружаются в облачное хранилище.
  - Как только файл загружен, облачное хранилище инициирует обратный вызов завершения загрузки. Запрос передается серверам API.

2.4. В БД метаданных состояние файла меняется на «загружено».

2.5. Система оповещает сервис уведомлений о том, что состояние файла поменялось на «загружено».

2.6. Сервис уведомлений оповещает заинтересованные клиенты (клиент 2) о том, что файл полностью загружен.

При проектировании файла применяется аналогичный процесс, так что мы не станем повторяться.

## Процесс скачивания

Процесс скачивания инициируется при добавлении или редактировании файла на другом устройстве. Откуда клиент узнает, что файл добавляется или редактируется другим клиентом? Это может происходить двумя способами.

- Если клиент А находится в сети в момент, когда файл изменяется другим клиентом, сервис уведомлений оповестит его об этих изменениях и необходимости скачать последние данные.
- Если клиент А находится не в сети в момент, когда файл изменяется другим клиентом, данные сохраняются в кэш. Когда клиент А снова появляется в сети, он скачивает последние изменения.

Узнав об изменении файла, клиент сначала запрашивает его метаданные у серверов API, а затем загружает соответствующие блоки, чтобы восстановить его содержимое. Детали процесса показаны на рис. 15.15. Отметим, что на этой диаграмме показаны лишь самые важные компоненты, чтобы уместить ее на страницу.

1. Сервис уведомлений информирует клиент 2 о том, что файл был изменен в другом месте.
2. Поскольку клиенту 2 известно о доступных обновлениях, он шлет запрос на скачивание метаданных.
3. Серверы API обращаются к БД за метаданными изменений.
4. Метаданные возвращаются серверам API.
5. Клиент 2 получает метаданные.

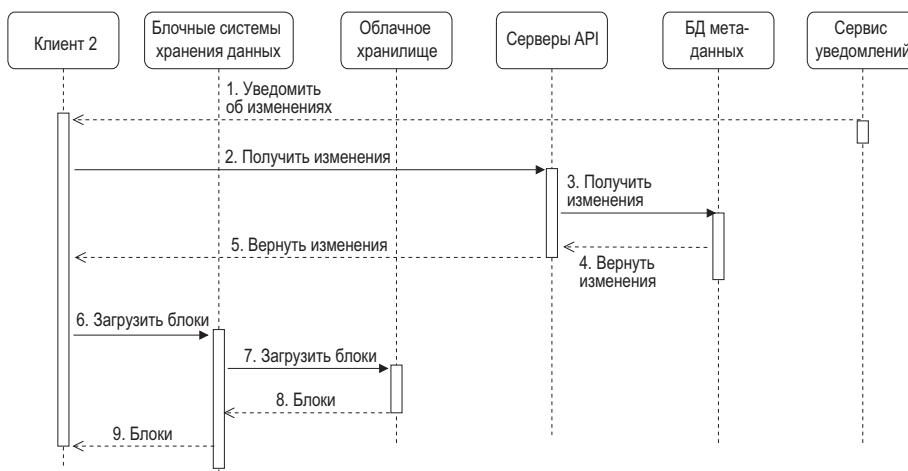


Рис. 15.15

6. Получив метаданные, клиент отправляет запрос блочным системам хранения данных, чтобы скачать блоки.
7. Блочные системы сначала скачивают блоки из облачного хранилища.
8. Облачное хранилище возвращает блоки блочным системам хранения данных.
9. Клиент 2 скачивает все новые блоки, чтобы восстановить файл.

## Сервис уведомлений

Чтобы поддерживать файлы в согласованном состоянии и уменьшить число конфликтов, любое изменение, вносимое локально, должно быть доступно другим клиентам. Сервис уведомлений создан именно с этой целью. Его основная задача состоит в передаче данных клиентам в ответ на определенные события. Это можно организовать несколькими способами.

- Длинный HTTP-опрос. Этот метод использует Dropbox [10].
- WebSocket. Этот протокол устанавливает постоянное соединение между клиентом и сервером с двунаправленным взаимодействием.

Нам подходят оба варианта, но мы выберем длинный HTTP-опрос по двум причинам.

- Взаимодействие с сервисом уведомлений не двунаправленное. Сервер отправляет клиенту информацию о файле, но обратно ничего не возвращается.
- WebSocket подходит для двунаправленного взаимодействия в реальном времени, как в случае с чатом. В Google Drive уведомления отправляются не так часто и не провоцируют всплески трафика.

При использовании длинного HTTP-опроса каждый клиент устанавливает HTTP-соединение с сервисом уведомлений. В случае внесения изменений в файл клиент закрывает соединение. Это означает, что он должен подключиться к серверу метаданных для скачивания последних изменений. После получения ответа или по истечении времени ожидания клиент немедленно отправляет новый запрос, чтобы поддерживать соединение открытым.

### **Экономия места в хранилище**

Чтобы реализовать историю изменений файлов и сделать систему надежной, мы храним разные версии одного и того же файла в нескольких центрах обработки данных. При частом сохранении каждого изменения свободное место в хранилище может быстро закончиться. Для экономии места можно предложить три способа.

- Дедупликация блоков данных. Устранение лишних блоков на уровне учетной записи позволяет легко сэкономить место. Два блока являются идентичными, если у них одинаковое значение хеша.
- Внедрение интеллектуальной стратегии резервного копирования. Здесь можно применить два подхода:
  - ◆ установить лимит. Мы можем ограничить количество хранимых версий. При достижении лимита самая старая версия заменяется новой;
  - ◆ хранение только важных версий. Некоторые файлы могут активно редактироваться. Поэтому, если сохранять каждую изме-

ненную версию, документ может записываться более 1000 раз за короткий промежуток времени. Чтобы избавиться от лишних копий, мы можем ограничить количество сохраняемых версий. Найти оптимальное значение можно методом проб и ошибок. При этом последние версии получают больший вес.

- Перемещать редко используемые данные в холодное хранилище. Холодными называют данные, к которым никто не обращался на протяжении месяцев или даже лет. Холодное хранилище, такое как Amazon S3 Glacier [11], намного дешевле, чем S3.

## Обработка сбоев

В крупномасштабных системах случаются сбои, и, чтобы с ними справляться, необходимо внедрить подходящие стратегии проектирования. Интервьюеру может быть интересно услышать о том, как вы будете обрабатывать следующие сбои системы.

- Отказ балансировщика нагрузки. Если балансировщик нагрузки выходит из строя, активируется его резервный экземпляр, который подхватывает трафик. Балансировщики нагрузки обычно следят друг за другом с помощью механизма пульсации, периодически обмениваясь сигналами. Балансировщик считается неисправным, если он не отправляет пульс на протяжении какого-то времени.
- Отказ блочного сервера. Если блочный сервер выходит из строя, другие серверы подхватывают незаконченные или ожидающие задания.
- Отказ облачного хранилища. Бакеты S3 реплицируются по нескольку раз в разных регионах. Если файл недоступен в одном регионе, его можно извлечь из другого.
- Отказ сервера API. Этот сервис не хранит свое состояние, поэтому в случае поломки одного сервера API балансировщик нагрузки перенаправит трафик к другим серверам.
- Отказ кэша метаданных. Серверы, кэширующие метаданные, реплицируются по несколько раз. Если один узел выходит из строя,

можно обращаться за данными к другим. Отказавший сервер кэша будет заменен новым.

- Отказ БД метаданных:
  - ◆ отказ ведущего узла. Если выходит из строя ведущий узел, его место занимает один из ведомых, после чего активируется новый ведомый узел;
  - ◆ отказ ведомого узла. Если выходит из строя ведомый узел, вы можете направить операции чтения другому ведомому узлу и заменить отказавший сервер базы данных.
- Отказ сервиса уведомлений. Каждый пользователь, находящийся в сети, поддерживает длинные HTTP-соединения с сервером уведомлений. Следовательно, к каждому такому серверу подключено много пользователей. Согласно презентации Dropbox 2012 года [6], на каждом компьютере открыто более миллиона соединений. Если сервер выходит из строя, все длинные HTTP-соединения теряются и клиентам приходится переподключаться к другому серверу. Несмотря на поддержку большого количества соединений, отдельно взятый сервер не в состоянии установить их все одновременно. Переподключение всех клиентов, которые потеряли связь, проходит довольно медленно.
- Отказ очереди автономной архивации. Очереди реплицируются по нескольку раз. Если одна очередь выйдет из строя, ее потребителям, возможно, придется подписаться на другую.

## ШАГ 4: ПОДВЕДЕНИЕ ИТОГОВ

В этой главе мы предложили архитектуру системы для поддержки возможностей Google Drive. Сочетание строгой согласованности, небольшого объема трафика и быстрой синхронизации делает эту архитектуру интересной. Наша система состоит из двух процессов: управления метаданными файлов и синхронизации. Еще одним важным компонентом системы является сервис уведомлений. С помощью длинного HTTP-опроса он сообщает клиентам о последних изменениях в файлах.

У задач, которые встречаются на интервью по проектированию ИТ-систем, не бывает идеальных решений. У каждой компании есть свои

уникальные ограничения, и при проектировании их необходимо учитывать. Вы должны понимать сильные и слабые стороны архитектурных и технологических аспектов системы. Если у вас еще остается несколько минут, можете обсудить различные архитектурные решения.

Например, файлы из клиента можно загружать напрямую в облачное хранилище, минуя блочные системы хранения данных. Преимущество этого подхода в том, что загрузка файлов становится быстрее, так как их нужно передавать только один раз. В нашей архитектуре файлы сначала передаются блочным системам хранения данных и только затем попадают в облачное хранилище. Тем не менее у альтернативного решения есть несколько минусов.

- Во-первых, одну и ту же логику разбиения на блоки, сжатия и шифрования нужно реализовать на разных платформах (iOS, Android, веб). Это чревато ошибками, к тому же на это уходит много времени. В нашем решении вся эта логика централизована и находится в блочных системах хранения данных.
- Во-вторых, поскольку клиент подвержен взлому и манипуляциям, реализация логики шифрования на его стороне была бы не самым оптимальным вариантом.

Еще одним интересным улучшением было бы вынесение логики управления сетевым состоянием в отдельный сервис. Назовем его сервисом присутствия в сети. Благодаря нахождению за пределами серверов уведомлений этот сервис можно было бы легко интегрировать с другими компонентами системы.

Поздравляем, вы проделали длинный путь и можете собой гордиться. Отличная работа!

## СПРАВОЧНЫЕ МАТЕРИАЛЫ

- [1] Google Drive: <https://www.google.com/drive/>
- [2] Upload file data: <https://developers.google.com/drive/api/v2/manage-uploads>
- [3] Amazon S3: <https://aws.amazon.com/s3>
- [4] Differential Synchronization <https://neil.fraser.name/writing/sync/>

- [5] Презентация на YouTube о синхронизации изменений: [https://www.youtube.com/watch?v=S2Hp\\_1jqpY8](https://www.youtube.com/watch?v=S2Hp_1jqpY8)
- [6] How We've Scaled Dropbox: <https://youtu.be/PE4gwstWhmc>
- [7] Tridgell, A., & Mackerras, P. (1996). The rsync algorithm.
- [8] Libsync <https://github.com/librsync/librsync>
- [9] ACID: <https://ru.wikipedia.org/wiki/ACID>
- [10] Dropbox security white paper: [https://www.dropbox.com/static/business/resources/Security\\_Whitepaper.pdf](https://www.dropbox.com/static/business/resources/Security_Whitepaper.pdf)
- [11] Amazon S3 Glacier: <https://aws.amazon.com/glacier/faqs/>

---

# 16

---

## ВЕК ЖИВИ — ВЕК УЧИСЬ

Чтобы спроектировать хорошую систему, нужно годами накапливать знания. Ускорить этот процесс поможет изучение архитектур существующих систем. Ниже собран список полезных материалов для чтения. Мы настоятельно рекомендуем вам обращать внимание как на общие принципы, так и на технологии, которые используются внутри. Исследование каждой технологии и понимание того, какие проблемы она решает, существенно повышает уровень ваших знаний и помогает оптимизировать процесс проектирования.

### СУЩЕСТВУЮЩИЕ СИСТЕМЫ

Следующие материалы могут помочь с пониманием общих принципов проектирования настоящих систем в разных компаниях.

- Facebook Timeline: Brought To You By The Power Of Denormalization: <https://goo.gl/FCNrmb>
- Scale at Facebook: <https://goo.gl/NGTdCs>
- Building Timeline: Scaling up to hold your life story: <https://goo.gl/8p5wDV>
- Erlang at Facebook (Facebook Chat): <https://goo.gl/zSLHrj>
- Facebook Chat: <https://goo.gl/qzSiWC>
- Finding a needle in Haystack: Facebook's photo storage: <https://goo.gl/edj4FL>
- Serving Facebook Multifeed: Efficiency, performance gains through redesign: <https://goo.gl/adFVMQ>

- Scaling Memcache at Facebook: <https://goo.gl/rZiAhX>
- TAO: Facebook's Distributed Data Store for the Social Graph: <https://goo.gl/TK1DyH>
- Amazon Architecture: <https://goo.gl/k4feoW>
- Dynamo: Amazon's Highly Available Key-value Store: <https://goo.gl/C7zxDL>
- A 360 Degree View Of The Entire Netflix Stack: <https://goo.gl/rYSDTz>
- It's All A/Bout Testing: The Netflix Experimentation Platform: <https://goo.gl/agbA4K>
- Netflix Recommendations: Beyond the 5 stars (Part 1): <https://goo.gl/A4FkYi>
- Netflix Recommendations: Beyond the 5 stars (Part 2): <https://goo.gl/XNPMXm>
- Google Architecture: <https://goo.gl/dvkDiY>
- The Google File System (Google Docs): <https://goo.gl/xj5n9R>
- Differential Synchronization (Google Docs): <https://goo.gl/9zqG7x>
- YouTube Architecture: <https://goo.gl/mCPRUF>
- Seattle Conference on Scalability: YouTube Scalability: <https://goo.gl/dH3zYq>
- Bigtable: A Distributed Storage System for Structured Data: <https://goo.gl/6NaZca>
- Instagram Architecture: 14 Million Users, Terabytes Of Photos, 100s Of Instances, Dozens Of Technologies: <https://goo.gl/s1VcW5>
- The Architecture Twitter Uses To Deal With 150M Active Users: <https://goo.gl/EwvfRd>
- Scaling Twitter: Making Twitter 10000 Percent Faster: <https://goo.gl/nYGC1k>
- Announcing Snowflake (Snowflake – это сетевой сервис для генерации уникальных числовых ID в крупных масштабах и с некоторыми простыми гарантиями): <https://goo.gl/GzVWYm>
- Timelines at Scale: <https://goo.gl/8KbqTy>

- How Uber Scales Their Real-Time Market Platform: <https://goo.gl/kGZuVy>
- Scaling Pinterest: <https://goo.gl/KtmjW3>
- Pinterest Architecture Update: <https://goo.gl/w6rRsf>
- A Brief History of Scaling LinkedIn: <https://goo.gl/8A1Pi8>
- Flickr Architecture: <https://goo.gl/dWtgYa>
- How We've Scaled Dropbox: <https://goo.gl/NjBDtC>
- The WhatsApp Architecture Facebook Bought For \$19 Billion: <https://bit.ly/2AHJnFn>

## КОРПОРАТИВНЫЕ БЛОГИ, ПОСВЯЩЕННЫЕ ПРОЕКТИРОВАНИЮ

Если вы собираетесь проходить интервью в какой-либо компании, вам будет крайне полезно почитать ее блоги, посвященные проектированию, и познакомиться с технологиями и системами, которые в ней реализованы. Кроме того, такие блоги служат бесценным источником знаний в определенных областях. Их регулярное чтение поможет вам повысить вашу квалификацию.

Вот список инженерных блогов, принадлежащих крупным общеизвестным компаниям и стартапам.

- Airbnb: <https://medium.com/airbnb-engineering>
- Amazon: <https://developer.amazon.com/blogs>
- Asana: <https://blog.asana.com/category/eng>
- Atlassian: <https://developer.atlassian.com/blog>
- BitTorrent: <http://engineering.bittorrent.com>
- Cloudera: <https://blog.cloudera.com>
- Docker: <https://blog.docker.com>
- Dropbox: <https://blogs.dropbox.com/tech>
- eBay: <http://www.ebaytechblog.com>
- Facebook: <https://code.facebook.com/posts>

- GitHub: <https://githubengineering.com>
- Google: <https://developers.googleblog.com>
- Groupon: <https://engineering.groupon.com>
- Highscalability: <http://highscalability.com>
- Instacart: <https://tech.instacart.com>
- Instagram: <https://engineering.instagram.com>
- Linkedin: <https://engineering.linkedin.com/blog>
- Mixpanel: <https://mixpanel.com/blog>
- Netflix: <https://medium.com/netflix-techblog>
- Nextdoor: <https://engblog.nextdoor.com>
- PayPal: <https://www.paypal-engineering.com>
- Pinterest: <https://engineering.pinterest.com>
- Quora: <https://engineering.quora.com>
- Reddit: <https://redditblog.com>
- Salesforce: <https://developer.salesforce.com/blogs/engineering>
- Shopify: <https://engineering.shopify.com>
- Slack: <https://slack.engineering>
- Soundcloud: <https://developers.soundcloud.com/blog>
- Spotify: <https://labs.spotify.com>
- Stripe: <https://stripe.com/blog/engineering>
- System design primer: <https://github.com/donnemartin/system-design-primer>
- Twitter: [https://blog.twitter.com/engineering/en\\_us.html](https://blog.twitter.com/engineering/en_us.html)
- Thumbtack: <https://www.thumbtack.com/engineering>
- Uber: <http://eng.uber.com>
- Yahoo: <https://yahooblog.tumblr.com>
- Yelp: <https://engineeringblog.yelp.com>
- Zoom: <https://medium.com/zoom-developer-blog>

## ПОСЛЕСЛОВИЕ

Поздравляем! Вы дочитали до конца это руководство по прохождению интервью. Вы получили навыки и знания для проектирования систем. Не всем хватает дисциплины, чтобы выучить то, что выучили вы. Можете мысленно похлопать себя по плечу. Ваш тяжелый труд себя окупит.

Поиск работы мечты — это долгий процесс, требующий много времени и усилий. Практика — путь к совершенству. Удачи!

Спасибо за то, что купили и прочли эту книгу. Без таких читателей, как вы, она никогда бы не увидела свет. Надеемся, вам понравилось!

Алекс Сюй

## System Design. Подготовка к сложному интервью

Перевел с английского А. Павлов

Заведующая редакцией	Ю. Сергиенко
Ведущий редактор	А. Юринова
Литературный редактор	Ю. Зорина
Художественный редактор	В. Мостинан
Корректоры	С. Беляева, Г. Шкатова
Верстка	Л. Егорова

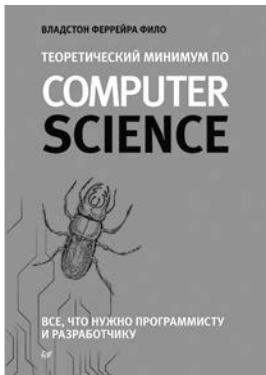
Изготовлено в России. Издатель: ООО «Прогресс книга».  
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2022. Наименование: книжная продукция. Срок годности: не ограничен.  
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные  
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.  
Подписано в печать 03.12.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.

Владстон Феррейра Фило

# ТЕОРЕТИЧЕСКИЙ МИНИМУМ ПО COMPUTER SCIENCE. ВСЕ, ЧТО НУЖНО ПРОГРАММИСТУ И РАЗРАБОТЧИКУ



Хватит тратить время на скучные академические фолианты! Изучение Computer Science может быть веселым и увлекательным занятием. Владстон Феррейра Фило знакомит нас с вычислительным мышлением, позволяющим решать любые сложные задачи. Научиться писать код — просто: пара недель на курсах, и вы «программист», но чтобы стать профи, который будет востребован всегда и везде, нужны фундаментальные знания. Здесь вы найдете только самую важную информацию, которая необходима каждому разработчику и программисту каждый день.

«Эта книга пригодится и для решения повседневных задач. Упреждающая выборка и кэширование помогут сложить рюкзак, параллелизм облегчит готовку на кухне. Ну и, разумеется, ваш программный код будет просто потрясающим». — Владстон Феррейра Фило

купить

Анналин Ын, Кеннет Су

## ТЕОРЕТИЧЕСКИЙ МИНИМУМ ПО BIG DATA. ВСЁ, ЧТО НУЖНО ЗНАТЬ О БОЛЬШИХ ДАННЫХ



Сегодня Big Data — это большой бизнес.

Нашей жизнью управляет информация, и извлечение выгоды из нее становится центральным моментом в работе современных организаций. Не важно, кто вы — деловой человек, работающий с аналитикой, начинающий программист или разработчик, — «Теоретический минимум по Big Data» позволит разобраться в основах новой и стремительно развивающейся отрасли обработки больших данных.

Хотите узнать о больших данных и механизмах работы с ними? Каждому алгоритму посвящена отдельная глава, в которой не только объясняются основные принципы работы, но и даются примеры использования в реальных задачах. Большое количество иллюстраций и простые комментарии позволяют легко разобраться в самых сложных аспектах Big Data.

«Отличная визуализация концепций машинного обучения позволяет «нетехнарям» интуитивно понять сложные абстрактные понятия. Это лаконичная и точная выжимка содержит теоретический минимум информации, необходимый для первого знакомства с Big Data». — Этан Чен, автор курса CS 102: Big Data, Стэнфордский университет

Купить