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 высоко-

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.

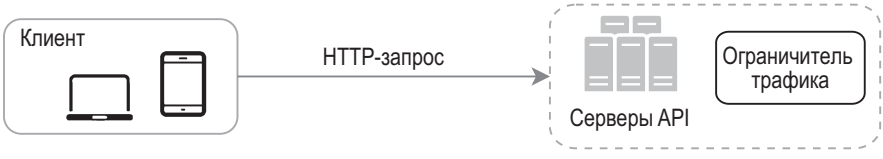

Рис. 4.1

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.

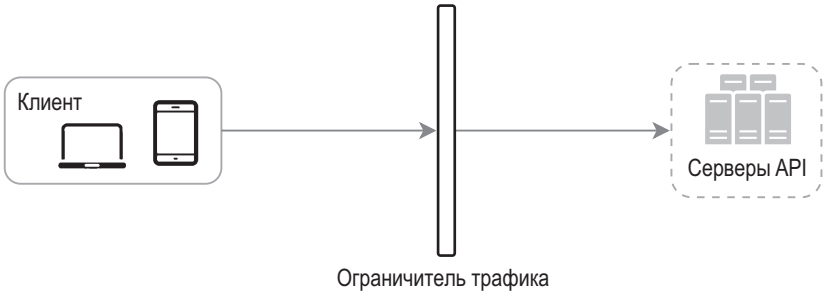

Рис. 4.2

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.

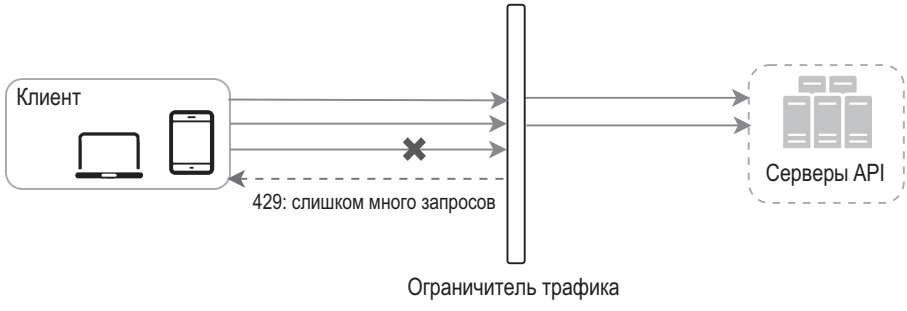

Рис. 4.3

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.

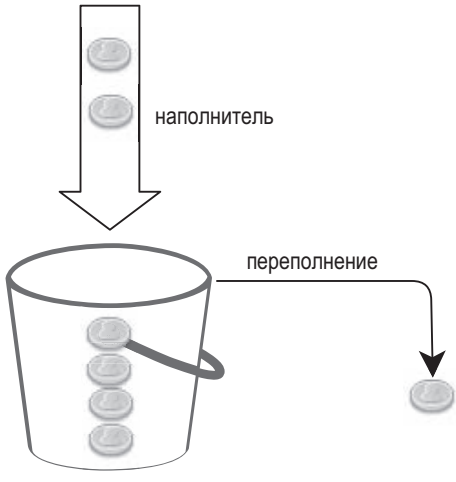
наполнитель
переполнение

Рис. 4.4

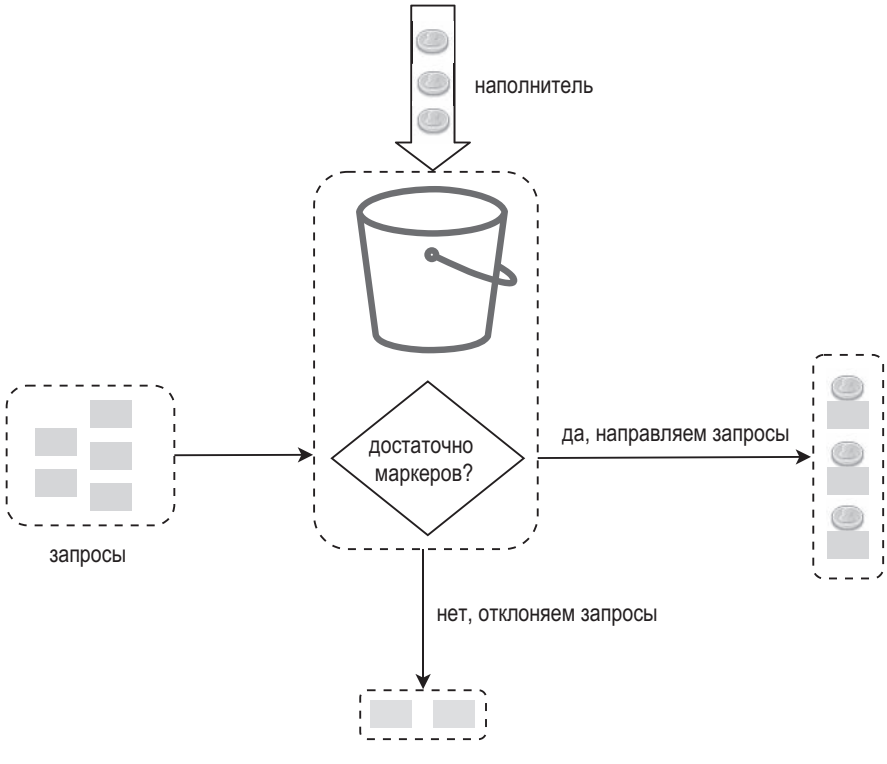
наполнитель
достаточно маркеров?
да, направляем запросы
запросы
нет, отклоняем запросы

Рис. 4.5

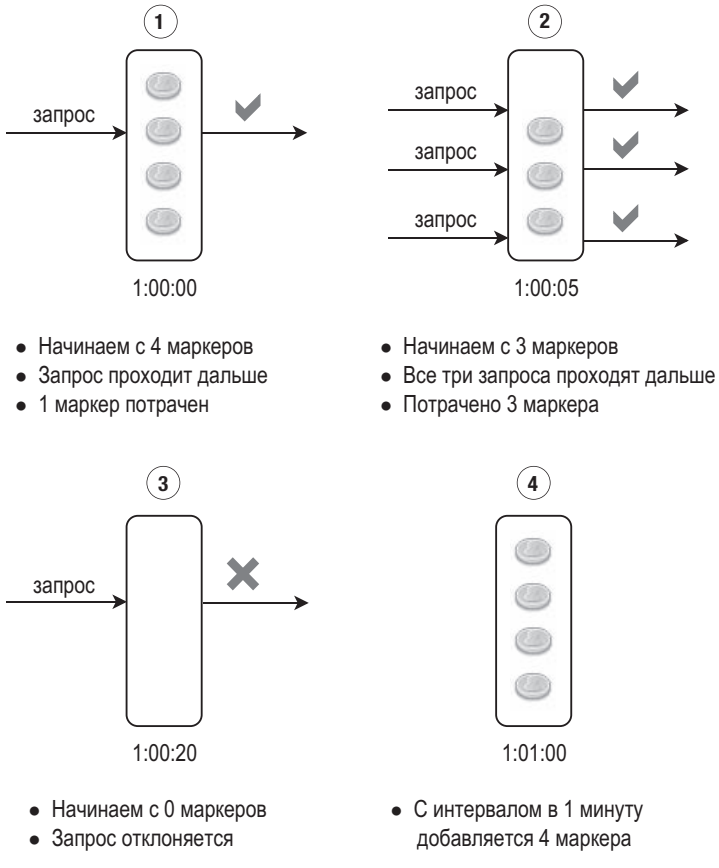

Рис. 4.6

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).

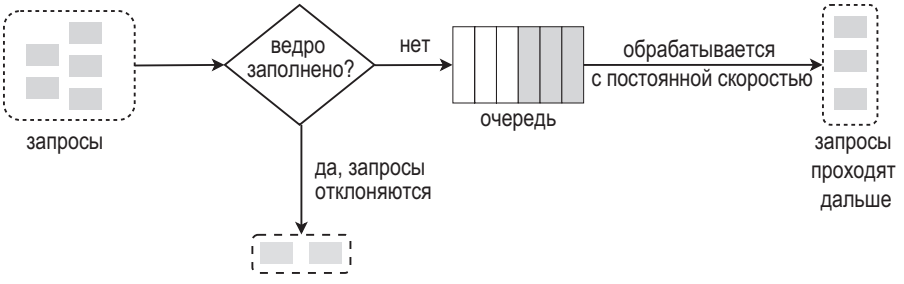

Рис. 4.7

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:

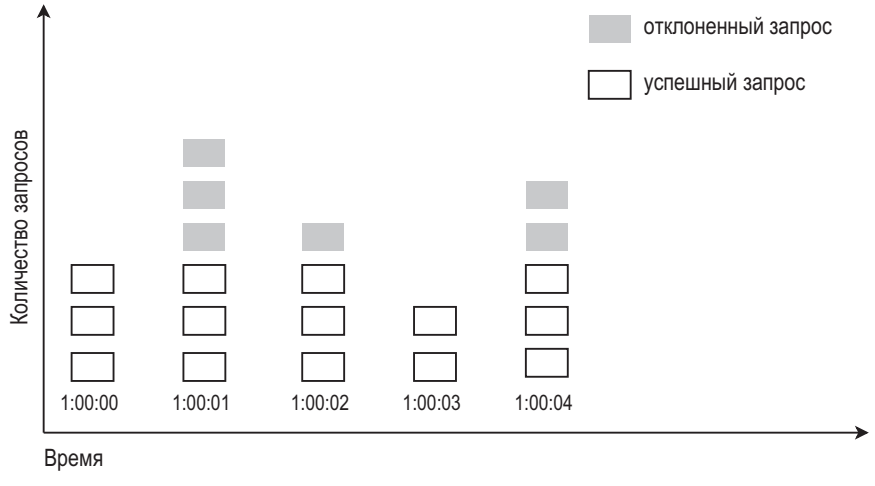

Рис. 4.8

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.

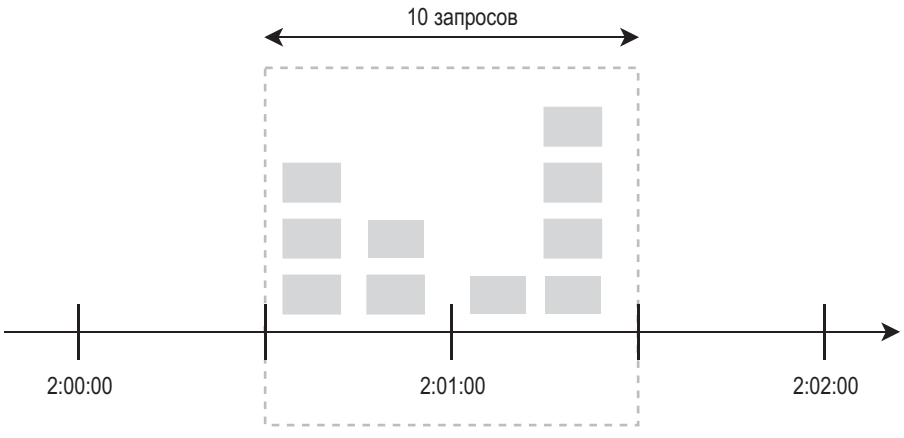

Рис. 4.9

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.

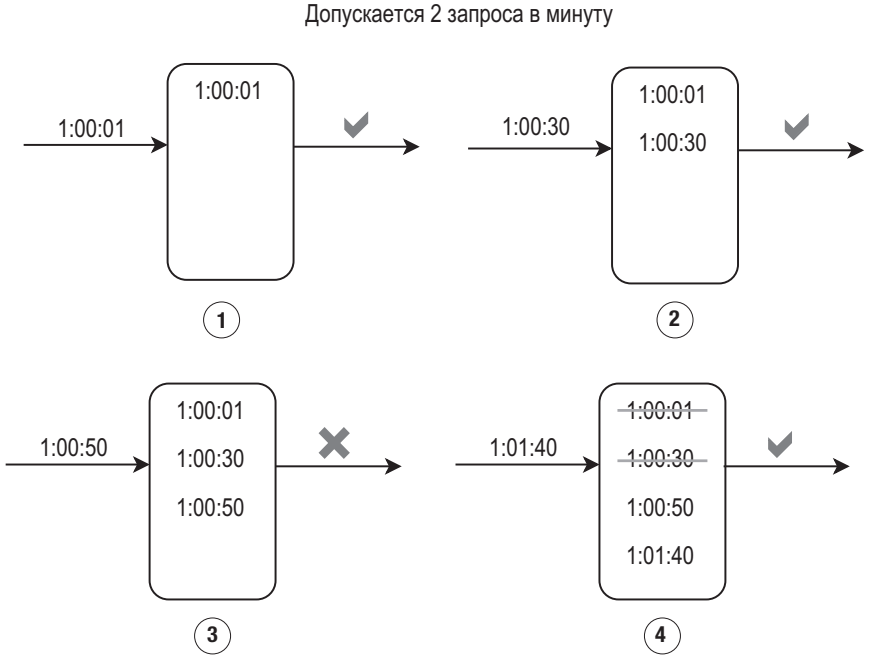

Рис. 4.10

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.

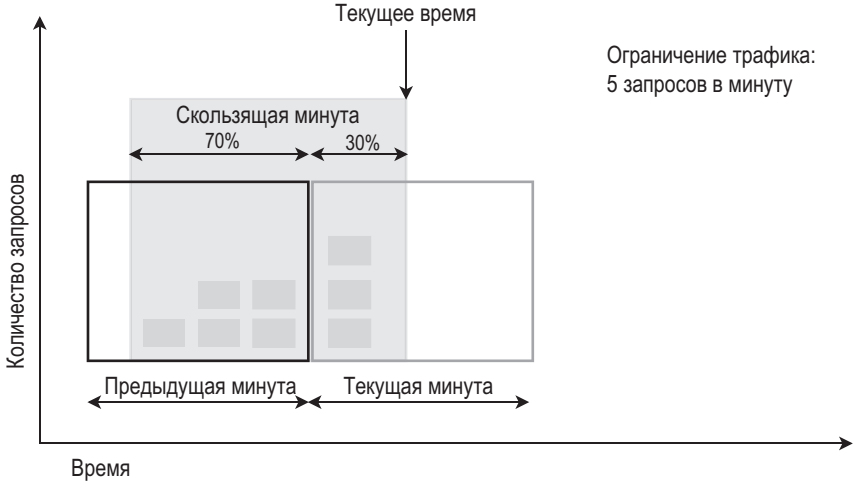

Рис. 4.11

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:

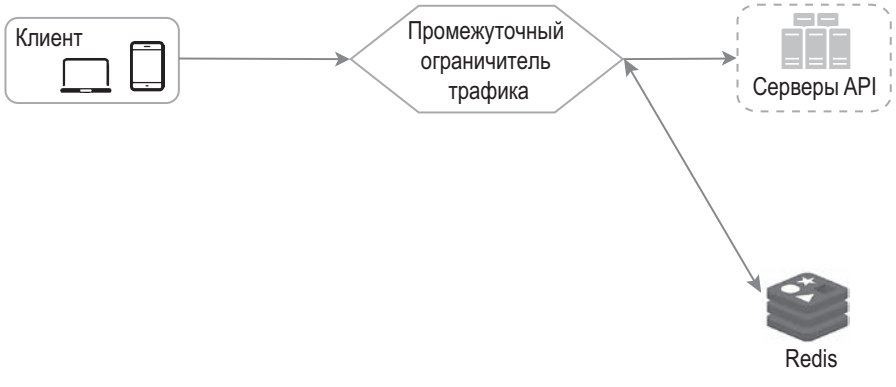

Рис. 4.12

- 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.

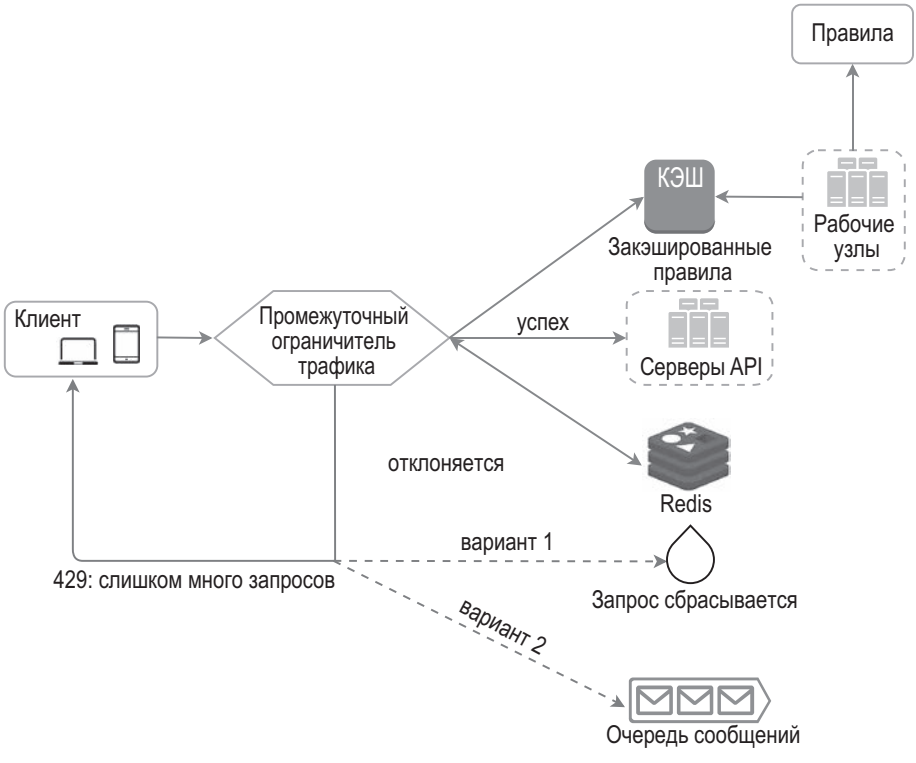

Рис. 4.13

- 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.

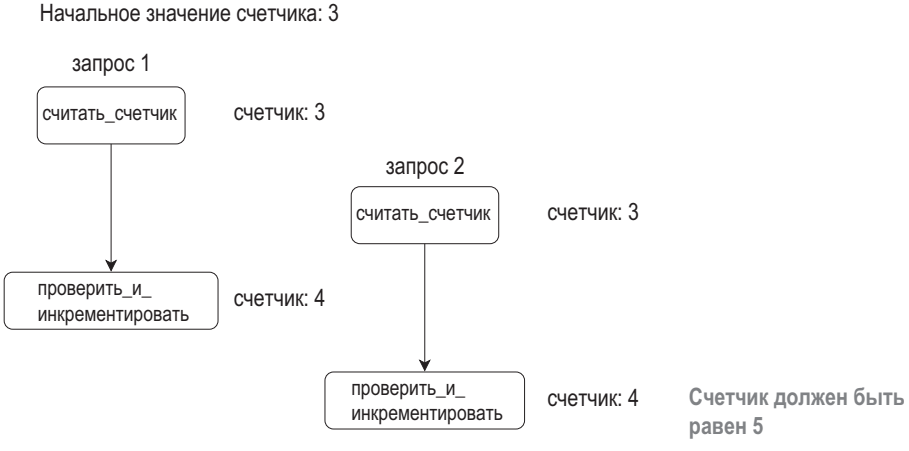

Рис. 4.14

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.

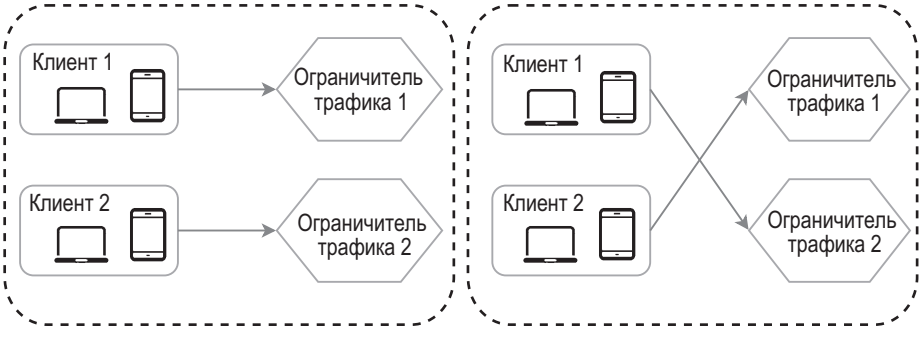

Рис. 4.15

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.

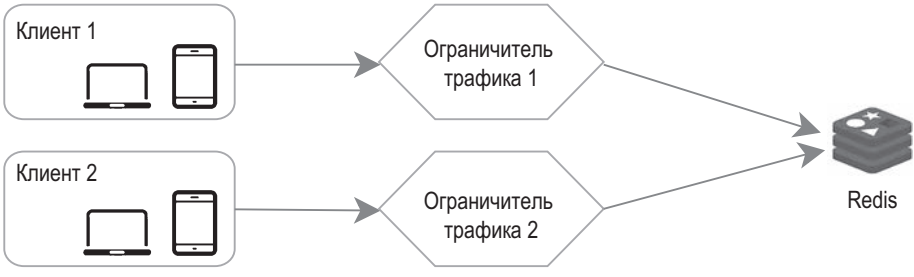

Рис. 4.16

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 немного 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