DOI 10.37882/2223–2966.2020.04–2.14

МЕТОДЫ ЗАЩИТЫ ОТ ПЕРЕГРУЗОК В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ ОБРАБОТКИ ИНФОРМАЦИИ

HANDLING OVERLOADS IN DISTRIBUTED SYSTEMS

D. Rubtsov

Summary. This article is devoted to the study of various method of overload handling. Method of request retry is widely used in modern distributed systems to maintain high availability, due to the increase of network interaction between different components of systems. However, if system is overloaded, this method can lead to a complete failure of entire system. The articles discusses various methods and algorithms that allow systems to handle load spikes staying highly available.

Keywords: distributed systems, handling overloads, microservice architecture, highly available systems.

Рубцов Дмитрий Викторович
Руководитель группы разработки, Яндекс, г. Москва
rubtsov.dmv@gmail.com

Аннотация. Статья посвящена исследованию различных методов защиты от перегрузок. В современных распределенных система, в связи с увеличением сетевого взаимодействия между различным компонентами систем, все чаще применяют методику повторения запросов для увеличения показателя доступности системы, но в условиях перегрузки системы данный метод может приводить к полному отказу всей системы. В статье рассматриваются различные алгоритмы и методы, которые позволяют системе переживать всплески нагрузки, не жертвуя при этом показателями доступности.

Ключевые слова: распределенные системы, защита от перегрузок, микросервисная архитектура, системы высокой доступности.

Введение

По мере развития Интернета к распределённым системам предъявлялись все более высокие требования производительности и отказоустойчивости. Сейчас уже трудно себе представить систему, которая бы могла уместиться на одном физическом сервере, даже самые небольшие системы для отказоустойчивости используют несколько виртуальных или физических серверов для сервера-приложения, распределяя запросы от пользователя с помощью балансировщика нагрузки.

Но, помимо этого, все чаще в современных системах начинают применять микросервисную архитектуру. [3, с. 4]

Основная особенность микросервисной архитектуры заключается в том, что вся система разделяется на множество небольших сервисов, которые отвечают за ограниченный участок бизнес-логики системы, называемые микросервисами. При этом микросервисы работают на разных физических или виртуальных серверах и обмениваются информацией между собой по сети TCP/IP, например с помощью протокола HTTP. И если в традиционной архитектуре основное взаимодействие между разными участками бизнес-логики системы происходит в памяти одного физического сервера, то в микросервисной архитектуре основное взаимодействие происходит по сети. Сообщения в сети могут быть потеряны, или может быть нарушена маршрутизация или даже весь сервер мог быть отключен от сети. В сети постоянно происходят различные случайные события, которые могут привести к ошибке обработки запроса, и чем более распределенной является система, тем сильнее проявляется эффект от передачи данных по сети.

Для того, чтобы защититься от такого рода ошибок в распределенных системах применяют механизм повтора запроса — если ответ от определенного сервера не был получен или была получена ошибка, то клиент повторяет запрос.

Например, представим, что есть система, состоящая из 3 серверов приложений и клиента, нагрузка между серверами приложений распределяется равномерно балансировщиком (рисунок 1). Клиент посылает запрос, который отправляется на первый сервер. В момент обработки запроса сервер перестает работать и клиент получает ошибку. Вместо того, чтобы показать ошибку пользователю, клиент повторяет запрос и балансировщик отправляет запрос на второй сервер, где он успешно выполняется. Данный метод особенно эффективен в более сложных топологиях, так при увеличении количества сетевого взаимодействия на один запрос пользователя, увеличивается и вероятность ошибки, и в системах, где требуется высокий уровень доступности, без перезапросов достаточно сложно достичь уровня 99,99% («четыре девятки») и более.

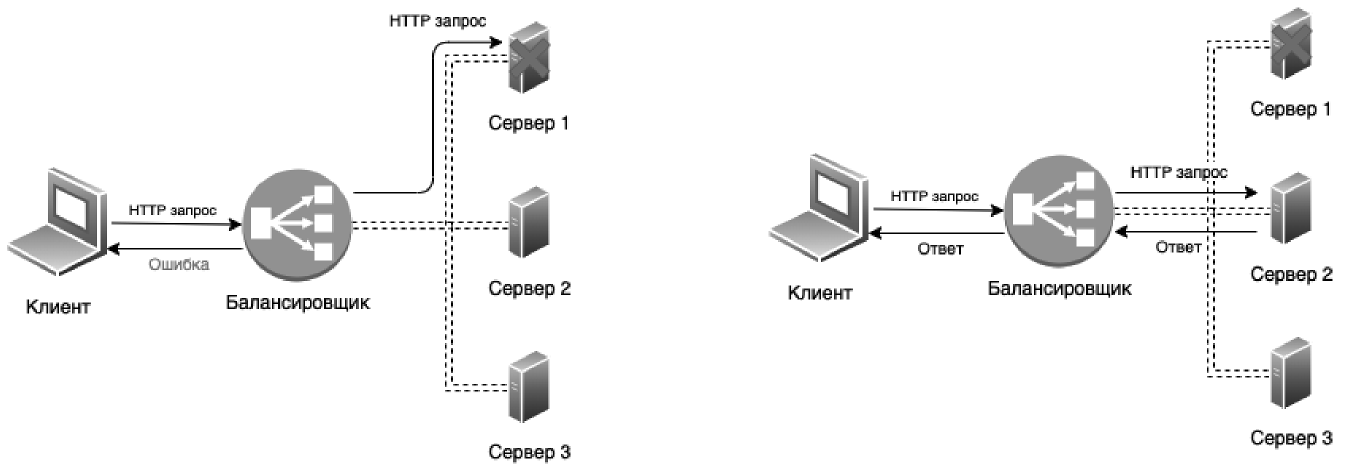

Рис. 1. Отказ сервера под балансировщиком нагрузки

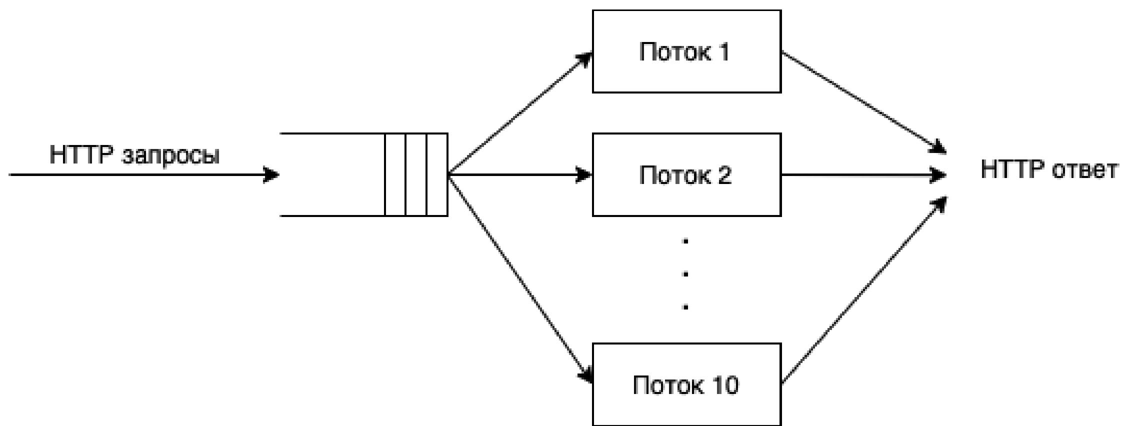

Рис. 2. Представление сервера приложения в виде M/M/10 СМО

Но у данного метода есть один большой недостаток — он увеличивает суммарную нагрузку на всю систему, что может в итоге привести к перегрузкам и отказам.

Далее в статье будут рассмотрены методы, которые позволяют достичь высокого уровня доступности, в условиях распределенных систем, и переживать возможные перегрузки с минимальными потерями в доступности системы.

Ограничение частоты запросов

Отсутствие ограничение на частоту входящих запросов — одна из наиболее часто встречающихся причин полного отказа систем в момент перегрузок, а в условиях, когда в системе используется метод перезапросов, вероятность полного отказа при перегрузках кратно возрастает из-за положительной обратной связи (данная тема будет подробнее освещена в следующем разделе).

Так почему же отсутствие ограничения на частоту запросов приводит к полному отказу? Рассмотрим пример. Возьмем сервер веб-приложения, который обрабатывает HTTP запросы от пользователя в 10 потоков, среднее время обработки одного запроса без нагрузки составляет 50мс. Будем считать, что для комфортной работы пользователя запрос должен укладываться в 200мс, если запрос превышает 1с, пользовательский опыт начинает сильно страдать, 10с ожидания пользователи воспринимают, как отказ в обслуживании.

Рассмотрим работу данного сервера, как M/M/10 систему массового обслуживания (СМО) с бесконечной очередью, чтобы определить, какой поток запросов приведет к отказу сервера.

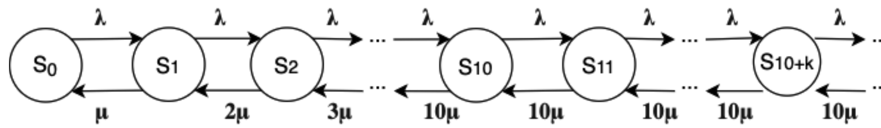

Рис. 3. Граф состояний СМО

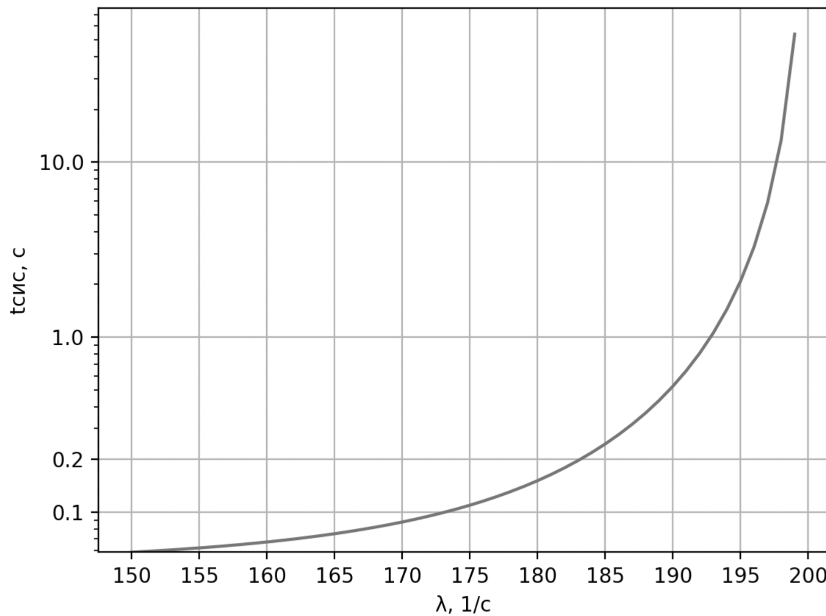

Рис. 4. Зависимость времени ответа системы от входящей нагрузки

Для расчета интенсивности входящих запросов λ, которая приведет к отказу, рассчитаем зависимость среднего времени обслуживания запроса системой $\overline{t}_{\text{сист}}$ от λ. Построим граф состояния СМО (рисунок 3), где

$$\mu = \frac{1}{0.05} = 20 \text{ (запросов/с)}—$$

интенсивность обработки запросов.

Для данного графа можно написать систему дифференциальных уравнений Колмогорова [2, с. 19] и для предельных вероятностей, приравнивая производные нулю, получить следующую систему алгебраических уравнений:

$$\begin{cases} 0 = -\lambda p_0 + \mu p_1 \\ 0 = \lambda p_0 - (\lambda + \mu) p_1 + 2\mu p_2 \\ \dots \\ 0 = \lambda p_9 - (\lambda + 10\mu) p_{10} + 11\mu p_{11} \\ \dots \\ 0 = \lambda p_{10+m-1} - [\lambda + (10+m)\mu] p_{10+m} + (11+m)\mu p_{11+m} \end{cases}$$

Вычислим значение предельных вероятностей:

$$p_0 = \left(1 + \frac{\rho}{1!} + \frac{\rho^2}{2!} + \dots + \frac{\rho^{10}}{10!} + \frac{\rho^{11}}{10!(10-\rho)}\right)^{-1}.$$

$$p_1 = \frac{\rho}{1!} p_0, \dots, p_k = \frac{\rho^k}{k!} p_0, \dots, p_{10} = \frac{\rho^{10}}{10!} p_0.$$

$$p_{11} = \frac{\rho^{11}}{10 \cdot 10!} p_0, .., p_{10+m} = \frac{\rho^{10+m}}{10^m \cdot 10!} p_0, \dots — \rho = \frac{\lambda}{\mu}$$

приведенная интенсивность потока.

Рассчитаем показатели эффективности СМО:

среднее число заявок в очереди

$$\overline{r} = \frac{\rho^{11} p_0}{10 \cdot 10! (1 - \frac{\rho}{10})^2};$$

среднее число запросов в системе

$\overline{z} = \overline{r} + \rho$;

среднее время пребывания запроса в системе

$$\overline{t}_{сис} = \frac{1}{\lambda} \overline{z}.$$

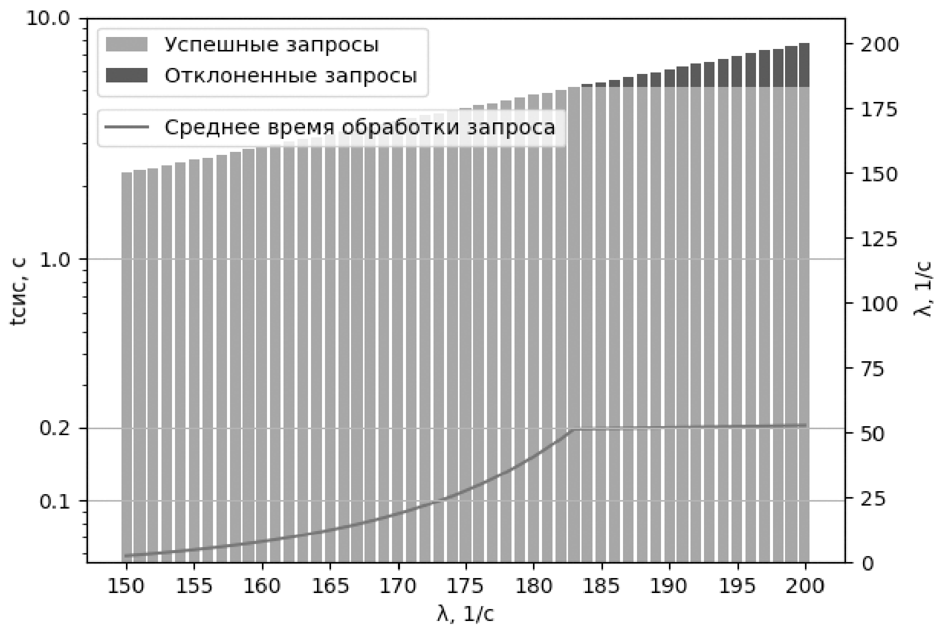

Рис. 5. Зависимость времени ответа системы от входящей нагрузки с ограничением частоты запросов

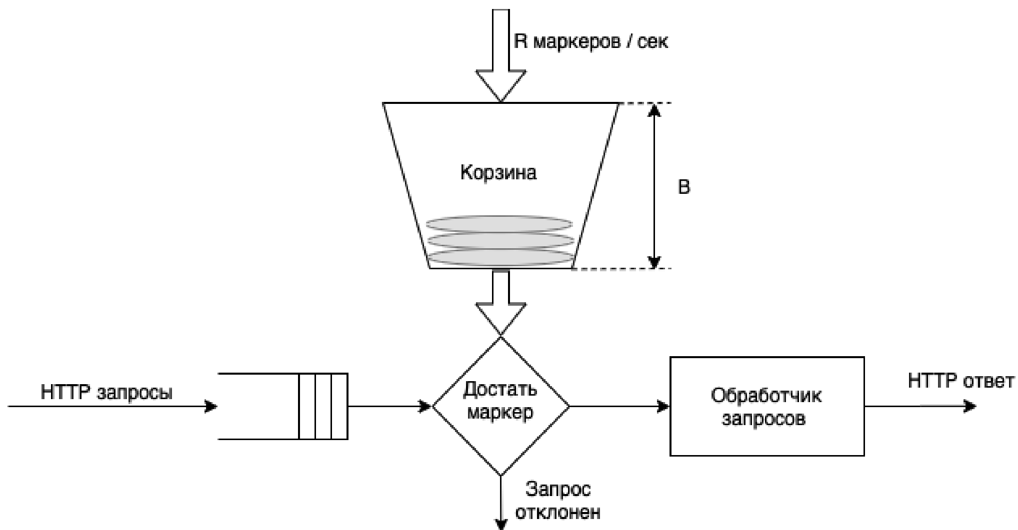

Рис. 6. Алгоритм маркерной корзины

Построим график зависимости $\bar{t}_{\text{сист}}$ от λ (рисунок 4).

На графике видно, что система укладывается в 200мс до 183 запросов в секунду, далее при незначительном увеличении нагрузки на 5% время обработки запросов уже превышает 1с, а при увеличении на 7,5% превышает 10 секунд, что является недопустимым для большинства веб-сервисов. В итоге сервис перестает полностью функционировать, так как не успевает обработать ни одного запроса.

Но, если система начинает отклонять запросы после превышения порога в 183 запросов в секунду, при увеличении нагрузки на 7,5% до 197 запросов секунду 92,5% запросов продолжит успешно выполняться. Допустим сервер тратит 1мс на отклонение запроса, рассчитаем повторно зависимость $\bar{t}_{\text{сист}}$ от λ, учитывая отклонение запросов.

На рисунке 5 видно, что, если сервер отклоняет запросы после превышения порога частоты в 183 запроса в секунду, среднее время на обработку всех запросов

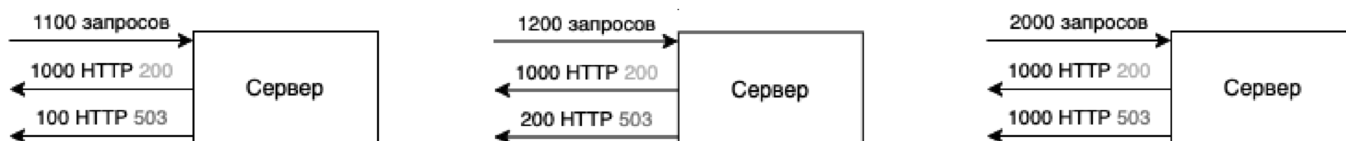

Рис. 7. Мультиплицирование нагрузки

перестает возрастать так стремительно, и большая часть запросов выполняется успешно.

Таким образом, зная предельное значение частоты входящих запросов, которое можно определить эмпирически, используя методы нагрузочного тестирования, система может отслеживать в реальном времени текущую интенсивность входящих запросов и отклонять все запросы, которые приводят к превышению допустимого порога интенсивности. Данный метод позволяет системе выдержать возможную перегрузку, не отказывая целиком в обслуживании.

Рассмотрим наиболее популярные алгоритмы, которые позволяют ограничивать входящую интенсивность запросов в систему.

Алгоритм маркерной корзины

Алгоритм заключается в том, что в «ведро» помещаются «маркеры» с постоянной скоростью R маркеров в секунду, ведро ограниченно объемом B маркеров.

Скорость R является максимальной средней скоростью поступления входящих запросов, когда объем ведра B соответствует максимальному размеру пульсации потока входящих запросов. Сервер перед обработкой запроса должен проверить, есть ли в ведре маркеры, если маркеры в ведре имеются, то из ведра удаляется маркер и запрос обрабатывается, если маркеры в ведре отсутствуют, то запрос отклоняется.

Если запросы в систему имеют разный вес и требуют разного количества ресурсов для обработки, то для каждого типа запросов можно доставать из корзины разное количество маркеров, пропорционально объему запроса.

Алгоритм фиксированного окна

Данный алгоритм разбивает временную шкалу на окна фиксированного размера, например 1 секунду, каждое окно содержит счетчик. Для каждого запроса, основываясь на времени его поступления, мы находим соответствующее окно, если счетчик в данном окне превысил максимального значения, то запрос отклоняется, в противном случае запрос обрабатывается и увеличивается счетчик в соответствующем окне.

Очевидно, что данный алгоритм обеспечивает необходимую среднюю частоту запросов только внутри интервального окна, но не между окнами, так как на границах окон могут возникать всплески, превышающие среднюю частоту запросов вдвое.

Алгоритм скользящего окна

Следующий похож на алгоритм фиксированного окна, но данный алгоритм сглаживает всплеск, которые происходят на границах окон. Это осуществляется за счет того, что к текущему окну добавляется взвешенное значение счетчика предыдущего окна.

Рассмотрим пример. Предположим, размер окна равный 1 секунде и ограничение на 10 запросов в секунду. Допустим, что в полуинтервале интервале [0; 1) с пришло 10 запросов в систему, а в текущем полуинтервале [1.0; 2.0) с пришел 1 запрос. В момент времени 1.150с приходит запрос, чтобы рассчитать текущую среднюю частоту запросов, воспользуемся формулой:

$$r = c_n + \left(1 - \frac{t \cdot modw}{w}\right) c_{n-1}$$

Где t — текущее значение времени, w — размер окна, c_n — счетчик запросов в текущем окне, c_{n-1} — счетчик запросов в предыдущем окне.

Подставим значение в формулу, получим $r = 1 + 85\% * 10 = 9{,}5 < 10$, запрос может быть обработан.

Данный алгоритм по-прежнему является не точным, так как сглаживание происходит из предположения, что запросы внутри предыдущего окна распределены равномерно. Но данный алгоритм достаточно прост в реализации и несет небольшие накладные расходы из-за чего достаточно часто применяется на практике и в среднем показывает достаточную точность.

Мультиплицирование нагрузки

Ограничив частоту запросов, мы защитились от полного отказа системы в случае перегрузок. Но если в си-

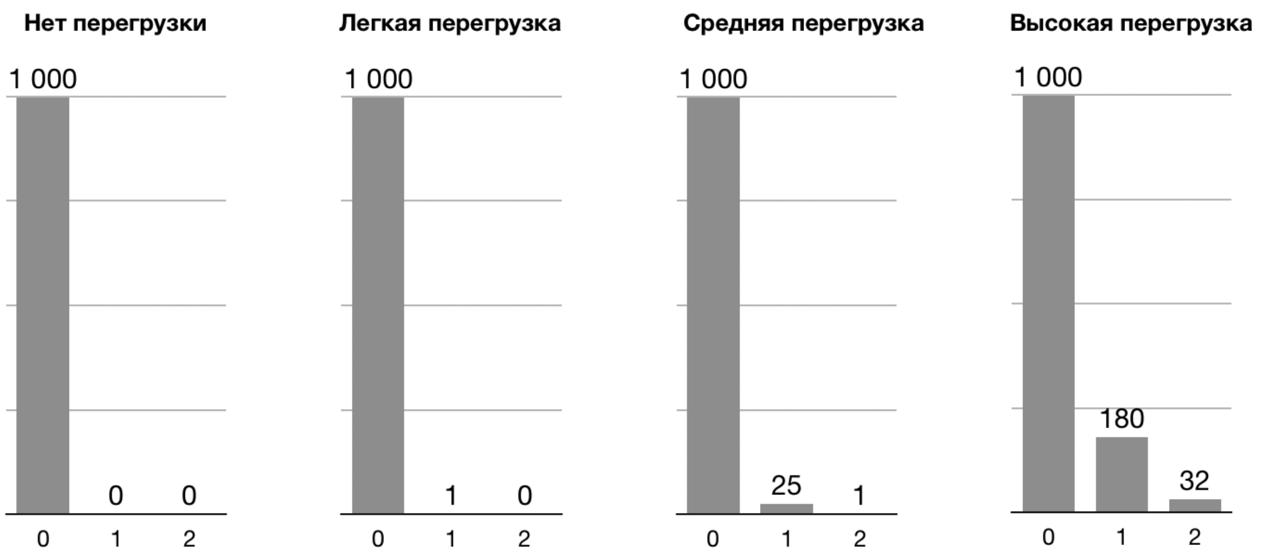

Рис. 8. Гистограмма распределения повторений запросов на сервере

стеме используются перезапросы при перегрузках возникает положительная обратная связь.

Рассмотрим пример, изображенный на рисунке 7. Предположим, что мы ограничили максимальную частоту запросов на уровне 1000 запросов/сек. На сервер поступает поток запросов с частотой 1100 запросов/сек, 1000 запросов (90%) выполняется успешно, а 100 запросов (10%) мы отклоняем. В следующую секунду на сервер уже поступает 1200 запросов, из них 1100 — новые запросы, а 100 это перезапросы по запросам, которые не были обработаны в предыдущую секунду. Таким образом мы отклоняем уже 200 запросов (18%). Если перезапросы на клиенте никак не ограничены, то через 4.2 секунды суммарное кол-во запросов на сервер станет равным 2000 и 50% запросов будут отклоняться с ошибкой.

Ситуация усугубляется, если пользовательский запрос при обработке проходит через несколько серверов и на каждом уровне происходит перезапрос при ошибке, в сложных топологиях это может приводить к экспоненциальному росту нагрузки.

В итоге вся система испытывает так называемый «шторм перезапросов» (retry storm), обрабатывая только перезапросы. Далее мы рассмотрим различные методики борьбы с подобными ситуациями.

Бюджет перезапросов

Чтобы не приводить к ситуации, когда количество перезапросов превышает изначальное число запросов, нужно ограничивать число перезапросов на стороне клиента. Как показано в примере изображенном на рис. 7, при неограниченных перезапросах нагрузка может увеличиться вдове в течение нескольких секунд.

Чтобы ограничить общее число перезапросов на клиенте, необходимо вести счетчик всех запросов и счетчик повторных запросов. Далее вводим ограничение на% повторных запросов, например: если текущее соотношение перезапросов не превышает 10% — делаем повторный запрос, иначе, перестаем делать запросы и возвращаем ошибку. Таким образом купируется положительная обратная связь и максимальный возможный всплеск нагрузки из-за перезапросов ограничивается 10%.

Остановка перезапросов

Делая перезапрос при неудачном запросе на сервер, мы исходим из двух предположений: либо ошибка была случайной, либо сервер, на который мы сейчас попали, перегружен, но при перезапросе балансировщик нагрузки направит запрос на менее загруженный сервер и запрос выполниться успешно. Но бывают ситуации, когда перегружен не один сервер, а вся система целиком. В таком случае делать перезапрос — бессмысленно, мы попадем на такой же перегруженный сервер и запрос не будет выполнен.

Но существует метод, который позволяет на отдельно взятом сервере понять, что вся система испытывает перегрузку.

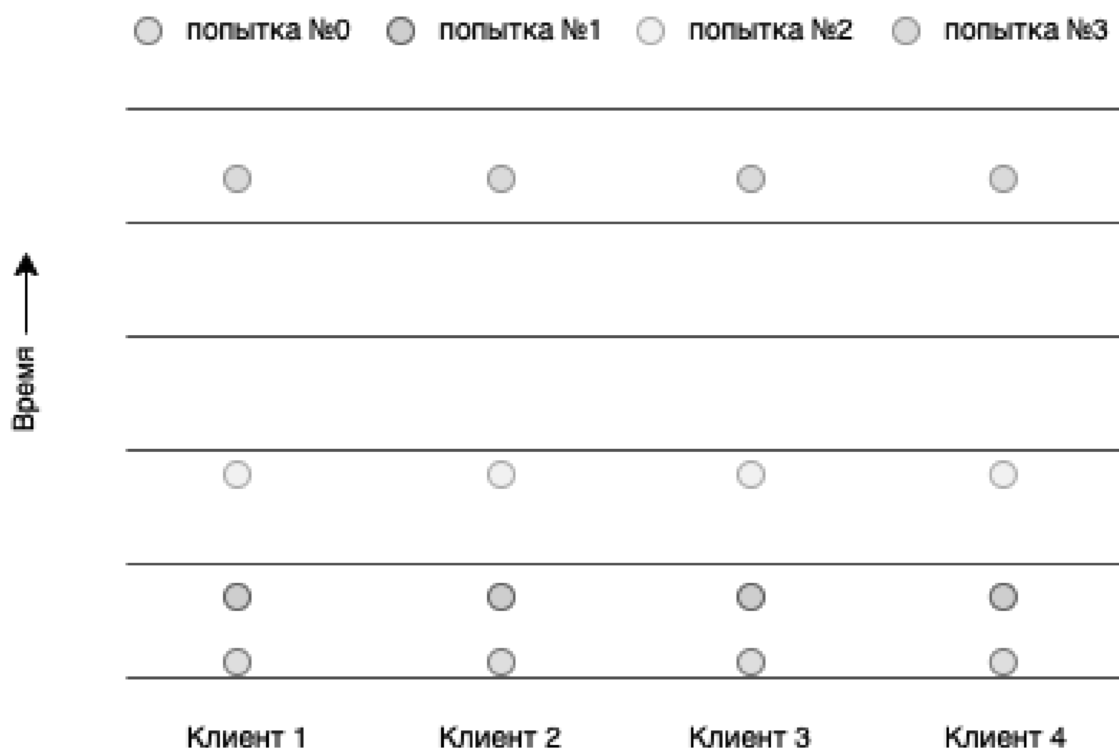

Рис. 9. Перезапросы с экспоненциальным отставанием

Для каждого запроса на клиенте введем счетчик попыток (по умолчанию 0), и будем передавать значение счетчика в каждом запросе на сервер. Если клиент-серверное взаимодействие в системе происходит по протоколу HTTP, мы можем добавить к запросу заголовок *X-Request-Attempt*, значение которого — текущая попытка получить ответ от сервера. С помощью алгоритма скользящего окна на сервере мы можем постоянно рассчитывать текущее распределение «попыток».

Так как балансировщик нагрузки равномерно распределяет запросы, то перезапросы, как мы уже заметили ранее, с большой вероятностью попадает на другой сервер и, если на сервере фиксируется большое количество запросов со значением счетчика попыток больше 0, это значит, что соседние сервера испытывают перегрузку. На рисунке 8 наглядно показано, что, построив гистограмму текущего распределения количества попыток, можно сделать вывод об общем состоянии системы [1, с. 253].

Таким образом, если на сервере сработало ограничение на количество запросов, нужно не просто отдать ошибку на клиент, нужно проверить не испытывает ли перегрузку вся система. Если текущая гистограмма показывает, что количество перезапросов высокое — в ответе сервера нужно отдать ошибку, которая прекратит перезапросы на стороне клиента.

Важным аспектом в остановке перезапросов является то, что в сложных топологиях, когда обработка 1 запроса пользователя проходит через множество серверов, если мы на каком либо из уровней получили ошибку перегрузки системы, ее нужно прокинуть вверх по всему стеку, чтобы предотвратить повторы запросов на уровнях выше.

Задержки между перезапросами

Если перезапрос на клиенте происходит сразу после получение ошибки от сервера, то с большой долей вероятности сервер также ответит ошибкой, так как, если сервер страдает от перегрузки, то отсутствие задержки при перезапросе не приводит к снижению нагрузки в целом и сервер остается перегруженным. Поэтому рекомендуется использовать метод экспоненциального отставания, который позволяет постепенно снижать нагрузку на сервер с каждым перезапросом. Метод заключается в том, что с каждым перезапросом мы экспоненциально увеличиваем задержку между следующим запросом по формуле:

$$delay = base * f^n ,$$

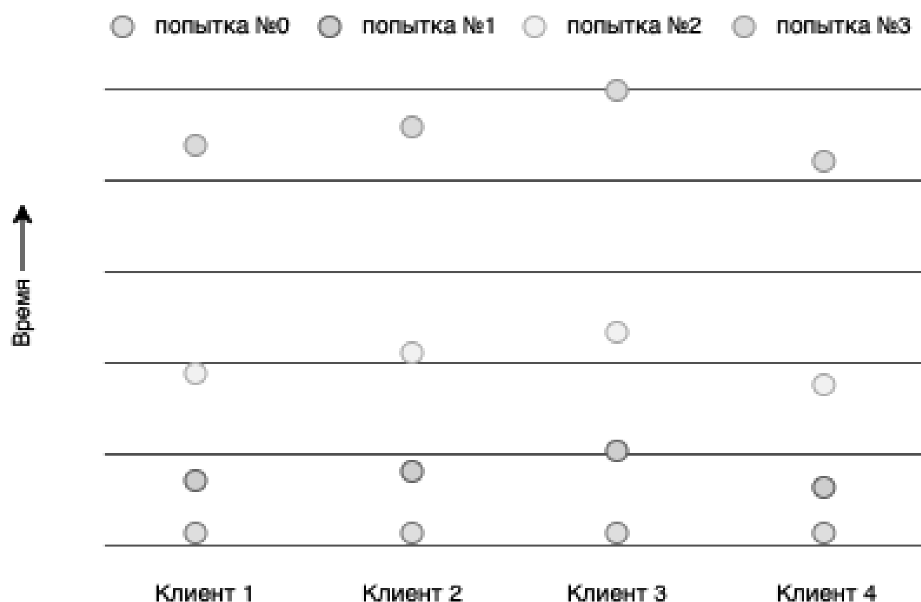

Рис. 10. Перезапросы с экспоненциальным отставанием и джиттером

где $base$ — начальный интервал задержки, f — коэффициент экспоненциального нарастания задержки (обычно равен 2), n — количество предыдущих неудачных запросов.

На рисунке 9 изображен график запросов нескольких клиентов при использовании экспоненциального отставания. Видно, что со временем задержка между запросами увеличивается, давая системе восстановиться, но все запросы происходят в одно и то же время из-за чего ресурсы сервера начинают расходоваться неравномерно. Для решения данной проблемы к задержке добавляют джиттер — случайную задержку, таким образом мы имеем формулу:

$$delay = base * f^n + jitter$$

Добавляя случайную задержку, мы снижаем вероятность коллизии, что два запроса, которые получили ошибку в одно и то же время повторяться в один и тот же промежуток времени. Пример распределения задержек с использованием джиттером показан на рисунке 10.

Заключение

В данной статье были рассмотренные различные методики, которые позволяют сделать распределенную систему более надежной и устойчивой к перегрузкам. Использование перезапросов позволяет снизить количество ошибок, которые возникают из-за передачи по ненадежным каналам связи. Ограничение частоты запросов на сервере с использованием алгоритма скользящего окна позволяет переживать перегрузки в системе и позволяет избегать полного отказа системы. Бюджетирование запросов на клиенте в сочетании с экспоненциальным отставанием позволяет избежать «шторма перезапросов» при замедлении работы системы и позволяют системе восстановиться после перегрузок. Построение гистограмма распределения попыток на сервере позволяет серверу определить, что вся система перегружена, чтобы остановить бессмысленные перезапросы со стороны клиента, что дополнительно ускоряет восстановление всей системы.

ЛИТЕРАТУРА

1. Бейер Б., Джоунс К. Site Reliability Engineering. Надежность и безотказность как в Google. Спб.: Питер, 2019. 596 с.
2. Авсиевич А.В., Авсиевич Е. Н. Теория массового обслуживания. Самара: Самарская государственная академия путей сообщения, 2004. 24 с.
3. Ньюмен С. Создание микросервисов. Спб.: Питер, 2016. 304 с.

© Рубцов Дмитрий Викторович (rubtsov.dmv@gmail.com).

Журнал «Современная наука: актуальные проблемы теории и практики»