Разработка → Exponential Backoff или как «не завалить сервер»

Разработка веб-сайтов*

При любом взаимодействии клиента и сервера мы сталкиваемся с необходимостью повторять запросы. Сетевое соединение может быть ненадежно, могут быть проблемы на сервере или любые другие причины, из-за которых необходимо повторить запрос. То же самое касается и взаимодействия backend-сервера с базой данных или любым другим хранилищем данных (другим сервисом).

Мы сегодня поговорим об интервале повторов запроса. Через какой период времени после неудачного запроса можно его повторить? Давайте рассмотрим две стратегии: повтор через фиксированный интервал времени и экспоненциальное откладывание (exponential backoff). Мы увидим на симуляции, что при условии наличия большого числа клиентов повтор через фиксированный интервал может не дать серверу «подняться», а использование exponential backoff позволяет избежать этой проблемы.

Вопрос интервала повторов становится важным при проблемах на сервере. Очень часто сервер способен выдержать нагрузку от клиентов, которые отправляют запросы в некотором «текущем» режиме, распределяя свои запросы во времени случайным образом. Если на сервере происходит отказ, все клиенты обнаруживают его и начинают повторять запросы через некоторый интервал. Может оказаться, что частота таких запросов превышает тот предел, который сервер может обрабатывать.

Еще одним важным моментом является то, что клиент часто не может отличить проблемы на сервере от проблем с сетевым соединением на стороне клиента: если ответ на запрос не приходит в заданный интервал времени, клиент не может сделать заключение о том, в чем именно проблема. И поведение клиента (повтор запроса, интервал повтора) будут одинаковыми в обоих ситуациях.

Повтор через фиксированный интервал времени

Это самая простая стратегия (и самая часто используемая). В случае неуспешного выполнения запроса клиент повторяет выполнение запроса через интервал T. Здесь мы ищем компромисс между тем, чтобы не «завалить» сервер слишком частыми запросами в случае отказа, и между тем, чтобы не ухудшить user experience, слишком долго не повторяя запросы в случае единичных проблем в сети.

Типичный интервал повтора может составлять от сотен миллисекунд до нескольких секунд.

Экспоненциальное откладывание (exponential backoff)

Алгоритм экспоненциального откладывания можно записать на псевдокоде следующим образом:

```
delay = MinDelay

while True:
    error = perform_request()
    if not error:
        # ok, got response
        break

    # error happened, pause between requests
    sleep(delay)

    # calculate next delay
    delay = min(delay * Factor, MaxDelay)
    delay = delay + norm_variate(delay * Jitter)
```

В этом псевдокоде `delay` — интервал между запросами, `perform_request` выполняет сам запрос, а `norm_variate` вычисляет нормальное распределение.

Есть следующие параметры алгоритма:

- `MinDelay` — минимальный (начальный) интервал повтора (например, 100 мс)
- `MaxDelay` — ограничение на длительность интервала повтора (например, 15 мин)
- `Factor` — коэффициент экспоненциального нарастания задержки (например, 2)
- `Jitter` — «дрожание», определяет случайную составляющую (например, 0.1)

Вычисление интервала задержки работает достаточно просто: первый интервал равняется `MinDelay`, затем в случае ошибки выполнения запроса интервал увеличивается в `Factor` раз, при этом добавляется случайная величина, пропорциональная `Jitter`. Интервал повтора ограничен `MaxDelay` (без учета `Jitter`).

Чем данный способ вычисления повтора лучше повтора через фиксированный промежуток времени:

- интервал увеличивается с каждой попыткой, если сервер не может справиться с потоком запросов, поток запросов от клиентов будет уменьшаться со временем;
- интервалы рандомизированы, т.е. не будет «шквала» запросов в такты, пропорциональные фиксированному интервалу.

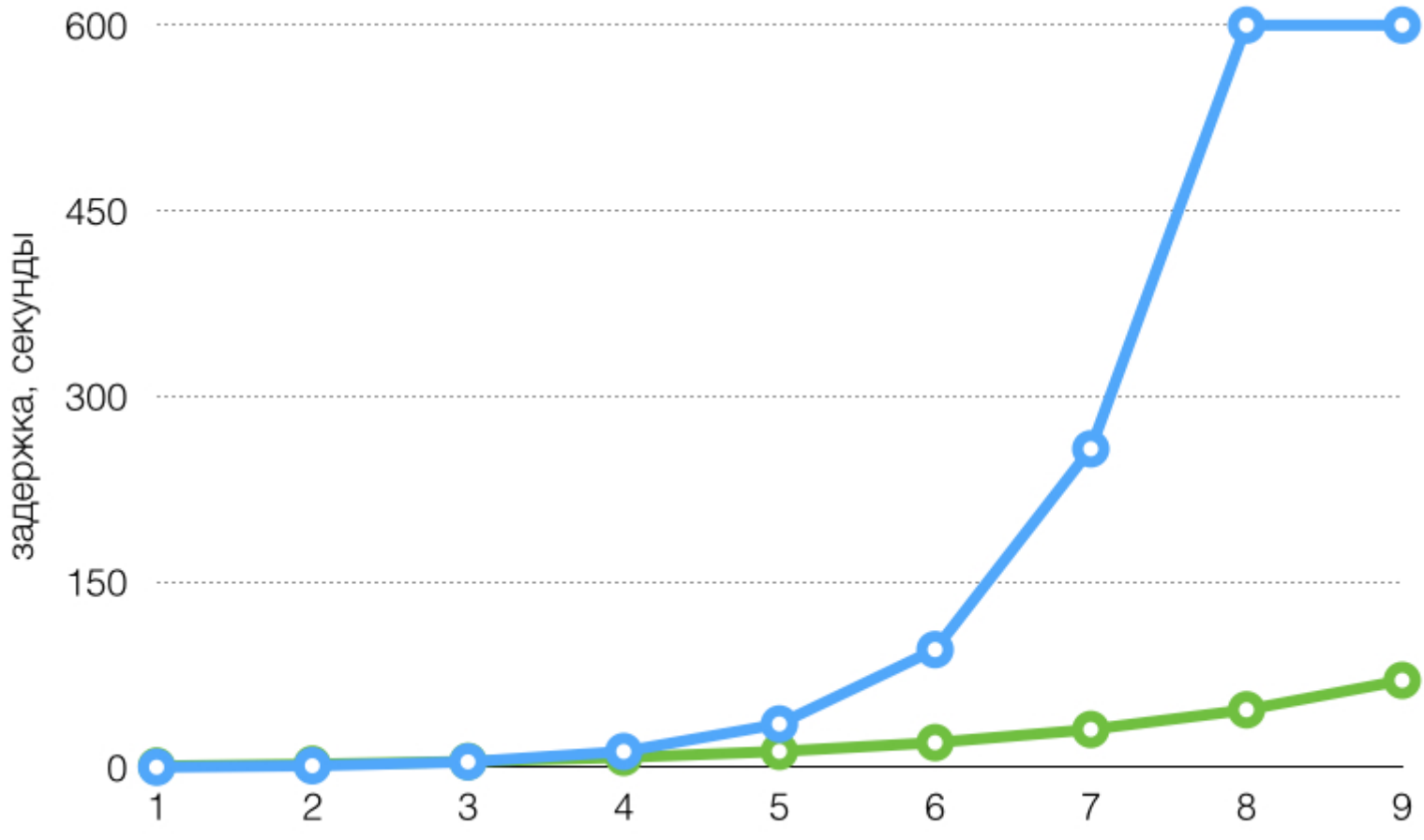
Можно задать резонный вопрос: а что если у клиента просто плохое сетевое соединение? Ведь алгоритм экспоненциального откладывания будет все время увеличивать интервал запроса, а пользователь не будет ждать бесконечно. Здесь можно привести три соображения:

- если соединение действительно плохое (неудачны 2, 3, 4 попытки), может и не стоит повторять запросы так часто;
- клиент не знает в чем проблема (см. выше), и ухудшить на какое-то время поведение для пользователя может не так уж и плохо, по сравнению с ситуацией, когда сервер будет завален запросами и не сможет вернуться к нормальной жизнедеятельности;
- параметры `Factor`, `MaxDelay` и `MinDelay` можно изменить, чтобы замедлить нарастание интервала повтора или ограничить его.

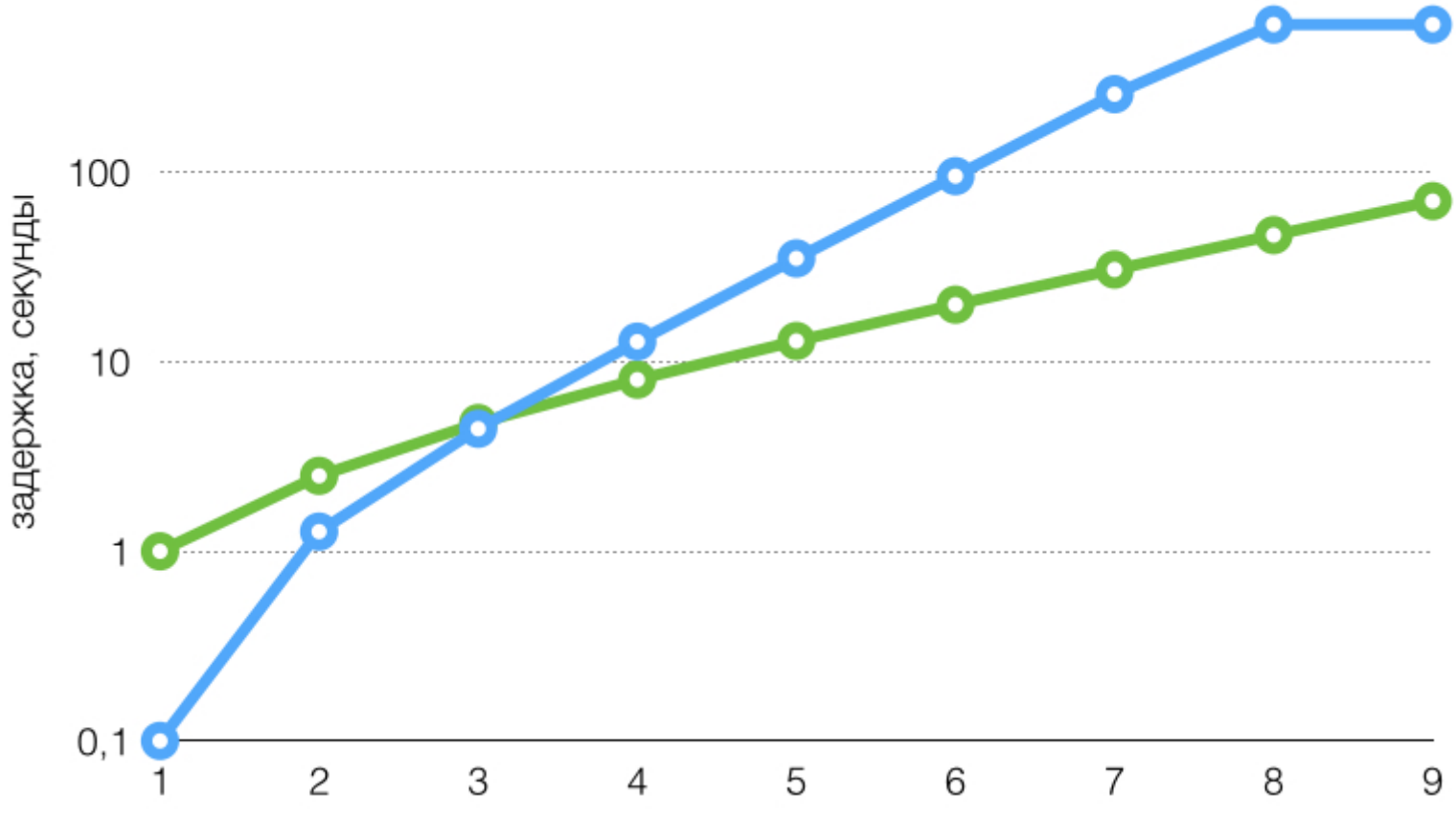
Для иллюстрации графики интервала задержки при таком алгоритме с параметрами:

- голубой график: `MinDelay` 100 мс, `Factor` 2.7, `Jitter` 0.1, `MaxDelay` 10 мин;
- зеленый график: `MinDelay` 1 сек, `Factor` 1.5, `Jitter` 0.1, `MaxDelay` 5 мин.

По горизонтальной оси — номер попытки.



Те же графики, шкала логарифмическая:



Симуляция

Так ли легко поверить, что сложность экспоненциального откладывания оправдана? Давайте попробуем провести симуляцию и выяснить эффект от разных алгоритмов. Симуляция состоит из двух независимых частей: клиента и сервера, я их опишу по очереди.

Сервер эмулирует типичную для базы данных задержку ответа на запрос: до некоторого предела количества одновременных запросов сервер отвечает с фиксированной задержкой, а затем время отклика начинает нарастать экспоненциально в зависимости от количества одновременных запросов. Если мы говорим о backend-сервере, чаще всего его время отклика также ограничено временем отклика базы данных.

Сервер отвечает за `minDelay` (100 мс), пока количество одновременных соединений не превысит `concurrencyLimit` (30), после этого время отклика нарастает экспоненциально: `minDelay * factor ^ ((concurrency - concurrencyLimit) / K)`, где `factor`, `K` — константы, а `concurrency` — текущее количество одновременных соединений. В действительности сервер не выполняет никакой работы, он отвечает на запрос фиксированным ответом, но задержка вычисляется по выше указанной формуле.

Клиент моделирует поток запросов от `N` клиентов, каждый из которых выполняет запросы независимо. Каждый поток клиента выполняет запросы с интервалами, соответствующим экспоненциальному распределению с параметром `lambda` (мат. ожидание интервала запросов равно `lambda`). Экспоненциальное распределение хорошо описывает случайные процессы, происходящие в реальном мире (например, активность пользователей, которая приводит к запросам на сервер). При ошибке выполнения запроса (или при превышении времени ожидания ответа) клиент начинает повторять запрос либо с простым фиксированным интервалом, либо по алгоритму экспоненциального откладывания.

Сценарий симуляции следующий: мы запускаем клиент и сервер с некоторыми параметрами, через 10-15 секунд возникает установившийся режим, в котором запросы выполняются успешно с небольшими задержками. Затем мы останавливаем сервер (эмулируем «падение» сервера или сетевые проблемы) просто сигналом `SIGSTOP`, поток клиентов это обнаруживает и начинает повторять запросы. Затем восстанавливаем работу сервера и смотрим, как быстро работа сервера придет в норму (или не придет).

Код проекта

Код для симуляции написан на Go и выложен на GitHub. Для сборки потребуется Go 1.1+, компилятор можно поставить из пакета ОС или скачать:

```
$ git clone https://github.com/smira/exponential-backoff.git
$ cd exponential-backoff
$ go build -o client client.go
$ go build -o server server.go
```

Простой сценарий: в первой консоли запускаем сервер:

```
$ ./server
Jun 21 13:31:18.708: concurrency:    0, last delay: 0
...
```

Сервер ожидает входящих HTTP-запросов на порту 8070 и каждую секунду печатает на экран текущее количество одновременных соединений и последнюю вычисленную задержку обслуживания клиентов.

В другой консоли запускаем клиент:

```
$ ./client
OK: 98.00 req/sec, errors: 0.00 req/sec, timedout: 0.00 req/sec
OK: 101.00 req/sec, errors: 0.00 req/sec, timedout: 0.00 req/sec
OK: 100.00 req/sec, errors: 0.00 req/sec, timedout: 0.00 req/sec
...
```

Клиент каждые 5 секунд печатает на экран статистику по успешным запросам, запросам, завершившимся с ошибкой, а также запросам, для которых был превышен таймаут на ожидание ответа.

В первой консоли останавливаем сервер:

```
Jun 21 22:11:08.519: concurrency:    8, last delay: 100ms
Jun 21 22:11:09.519: concurrency:   10, last delay: 100ms
^Z
[1]+  Stopped                 ./server
```

Клиент обнаруживает, что сервер остановлен:

```
OK: 36.00 req/sec, errors: 0.00 req/sec, timedout: 22.60 req/sec
OK: 0.00 req/sec, errors: 0.00 req/sec, timedout: 170.60 req/sec
OK: 0.00 req/sec, errors: 0.00 req/sec, timedout: 298.80 req/sec
OK: 0.00 req/sec, errors: 0.00 req/sec, timedout: 371.20 req/sec
```

Пробуем восстановить работу сервера:

```
$ fg
./server
Jun 21 22:13:06.693: concurrency:    0, last delay: 100ms
Jun 21 22:13:07.492: concurrency: 1040, last delay: 2.671444385s
Jun 21 22:13:08.492: concurrency: 1599, last delay: 16.458895305s
Jun 21 22:13:09.492: concurrency: 1925, last delay: 47.524196455s
Jun 21 22:13:10.492: concurrency: 2231, last delay: 2m8.580906589s
...
```

Количество одновременных соединений нарастает, увеличивается задержка ответа сервера, для клиента это будет означать еще больше ситуаций таймаута, еще больше повторов, еще больше одновременных запросов, сервер будет увеличивать задержку еще больше, наш модельный сервис «упал» и больше не поднимется.

Отстановим клиент и перезапустим сервер, чтобы начать сначала с экспоненциальным откладыванием повтора:

```
$ ./client -exponential-backoff
```

Повторим ту же последовательность действий с приостановкой и возобновлением работы сервера — сервер не упал и успешно поднялся, в течение короткого промежутка времени восстановилась работа сервиса.

Вместо заключения

У тестовых клиента и сервера есть большое количество параметров, которыми можно управлять как нагрузкой, так и таймаутами, поведением под нагрузкой и т.п.:

```
$ ./client -h
$ ./server -h
```

Даже в этой простой симуляции легко видно, что поведение группы клиентов (которых эмулирет client) отличается в случае «успешной» работы и в случае ситуации проблем (в сети или на стороне сервера):

- при нормальной работе нагрузка на сервер (количество запросов в секунду) определяется случайным распределением, но будет примерно постоянно (с параметрами по умолчанию около 100 запросов в секунду);
- в случае проблем при простой задержке количество запросов достигает быстро высоких значений (определяется только количеством клиентов и задержкой повтора);
- в случае проблем при экспоненциальном откладывании нагрузка на сервер снижается со временем (пока сервер не сможет справиться с нагрузкой).

Совет: используйте экспоненциальное откладывание при любом повторе запроса — в клиенте при обращении к серверу или в сервере при обращении к базе данных или другому сервису.

Этот пост был написан по материалам мастер-класса про высокие нагрузки и надежность.