Принципы построения высоконагруженных систем
Институт прикладных компьютерных наук ИТМО

Домашнее задание 2. Масштабируемость и балансировка нагрузки

Георгий Семенов
georgii.v.semenov@mail.ru
Мягкий дедлайн: Сб, 06.12.2025, 23:59 МСК
Жесткий дедлайн: Сб, 13.12.2025, 23:59 МСК

Данила Красов
DDIA25-HW2-DanilaKrasov.pdf

December 2, 2025

---

1 Проектирование высоконагруженных систем

В ходе выполнения упражнений в этом домашнем задании вам предстоит помочь корпорации *«Репликон»* разобраться с вопросами масштабирования и балансировки.

1.1 Масштабирование

Корпорация «Репликон» запускает свой видеохостинг и решила подготовить серверное оборудование для зон облака, в которых планируется разместить его сервисы CDN. В рамках этого процесса Cloud-команде требуется подобрать характеристики для «стойки мечты».

Стажер предложил использовать для всех новых стоек одинаковые машины с максимально возможными характеристиками CPU, RAM, Disk и Network.

Эта идея не очень понравилась опытным инженерам *Репликона*, которые на это возразили, что дорогие вертикальные улучшения серверов не всегда приводят к пропорциональному росту производительности из-за аппаратных ограничений, но совершенно не пояснили, о каких именно ограничениях идет речь.

Также, инженеры предложили не использовать отдельные серверные стойки для видеохостинга, а разместить CDN-сервисы, просто горизонтально масштабируя уже существующие сервера облака *Репликона*.

Задание:

1. (0.75 балл) Помогите стажеру понять – какие аппаратные ограничения имеют в виду опытные инженеры? *(Указание: рассмотрите, как процессор, память, диск и сеть с точки зрения материнской платы взаимодействуют друг с другом; приведите* ≥ 3 *ограничения и поясните их)*

2. (0.75 балл) Почему при горизонтальном масштабировании на всех машинах стараются использовать одни и те же hardware-характеристики (что такое vCPU)?

Решение:

1. С точки зрения аппаратных ограничений при вертикальном масштабировании (стажер хочет использовать наиболее мощные серверы) могут возникнуть следующие проблемы:

   - Ограничение по пропускной способности шины данных. Даже если процессор и память очень мощные, шина данных может стать "бутылочным горлышком", ограничивая скорость обмена данными между компонентами.
   - Иногда само приложение не может эффективно использовать все доступные ресурсы (например, старые игры не утилизируют многоядерные процессоры). Так, например, даже ваше веб-приложение с котиками будет упираться в разделяемый между потоками мьютекс, что не позволит эффективно использовать все ядра мощного процессора.

- Гипотетически, если нагрузить материнскую плату подключив наиболее мощные компоненты, то они могут начать конкурировать за ресурсы шины, что приведет к деградации производительности.

2. Мне кажется, что причина использования машин с похожими характеристиками при горизонтальном масштабировании заключается в том, что это упрощает управление ресурсами. Инженерам будет проще прогнозировать производительность и поведение системы. Балансировка нагрузки с помощью round-robin будет работать эффективнее. vCPU - это виртуальный процессор, который представляет собой логическое ядро процессора. Такие платформы оркестрации контейнеров, как Kubernetes, позволяют назначать подам определенное количество vCPU, измеряемое в долях физических ядер. С помощью такого подхода можно реплицировать поды с одинаковыми ресурсами.

□

1.2 Тонкости CAP-теоремы

Ведущий разработчик корпорации *Репликон* является огромным поклонником CAP-теоремы[1] и исследователя в области распределенных систем Мартина Клеппмана. В очередной раз перелистывая любимого «кабанчика»[2], ведущий разработчик неожиданно для себя обнаружил в книге своего кумира «махровый популизм» – Мартин назвал CAP-теорему «бесполезной»!

Ведущий разработчик был в ярости! Такого предательства он не мог простить. Он решил написать гневное письмо Мартину Клеппману, в котором изложил бы все свои аргументы в пользу важности CAP-теоремы. Но перед этим он решил подробнее изучить вопрос и изучить, почему же исследователь так считает.

Задание:

1. (1.5 балл) «Докажите» (или покажите) CAP-теорему, показывая несовместность каждой двойки свойств с оставшимся третьим свойством. Приведите по примеру соответствующих систем и соответствующий тип consistency.

2. (0.75 балл) Приведите ≥ 3 критических аргумента Мартина Клеппмана о CAP-теореме из книги (раздел «Теорема CAP»).

3. (0.75 балл) Наряду с CAP-теоремой существует также PACELC-теорема. Сформулируйте PACELC-теорему и объясните, как она расширяет CAP-теорему.

4. (2 балл) (*) Свойство SEC (strong eventual consistency) считается «разрешением» CAP-теоремы. Объясните, что это за свойство и почему рассматривая набор свойств {SEC, A, P} мы избегаем противоречия CAP-теоремы.

5. (3 балла) (*) В статье «A Critique of the CAP Theorem» Мартин Клеппман более подробно критикует классическую формулировку CAP-теоремы, а также неявно предлагает альтернативную формулировку на основе «delay-sensitivity framework». Сформулируйте ее.

Решение:

1. Доказательство CAP-теоремы.

    - C + A: При сетевой разделенности (network partition) невозможно одновременно сохранить согласованность и отвечать на запросы: узлу нужно либо ждать синхронизации, либо отбрасывать запросы.

[1] CAP-теорема (Brewer, 2000): в распределённых системах невозможно одновременно обеспечить все три свойства: согласованность (consistency), доступность (availability) и устойчивость к разделению (partition tolerance).

[2] «Кабанчик» – это книга Мартина Клеппмана «Высоконагруженные приложения: программирование, масштабирование, поддержка». Для работы над заданием можно воспользоваться текстом на русском или английском языке

- C + P: Чтобы сохранить согласованность во время сетевой разделенности, узлы должны перестать обслуживать часть запросов, что приводит к снижению доступности.
- A + P: Если оба узла продолжают отвечать во время сетевой разделенности, они могут принять противоречивые записи. Это приводит к потере согласованности.

2. Критические аргументы Мартина Клеппмана о CAP-теореме:
   - Оригинальная версия теоремы рассматривает только одну модель согласованности - линейризуемость (в оригинале linearizability, так автор называет strong consistency) и один тип отказа - сетевую недоступность/разделенность (network partition). Модель, по словам автора, не учитывает реальные проблемы распределённых систем: задержки сети, падение узлов, необходимость высокой производительности. Поэтому её практическая ценность для проектирования систем очень маленькая.
   - CAP-теорему подают как выбор между Consistency, Availability и Partition tolerance, но сетевая разделенность не выбор, а неизбежность в распределённых системах. Когда сеть работает нормально, система может быть и согласованной, и доступной. Выбор появляется только во время сетевой разделенности, и звучит так: либо согласованность, либо доступность при сетевой разделенности (что уже не похоже на исходную формулировку "выбери два из трех").
   - Многие системы используют более слабую согласованность не ради устойчивости к проблемам сетевой связностью (network partition), а ради повышения производительности уменьшения задержек, возможности работать быстрее при нормальной сети. Линейризуемость медленная всегда, а не только при сетевой разграниченности. Поэтому многие системы осознанно выбирают более слабую модель и теорема к этому решению отношения почти не имеет.

3. Если происходит сетевая разделенность компонентов, то система должна выбирать между доступностью и согласованностью. Когда же проблем с сетевой связностью нет, система должна выбирать между задержкой и согласованностью. PACELC-теорема расширяет CAP-теорему тем, что указывает на компромиссы, которые возникают не только во время сетевой сетевой разделенности, но и в обычном режиме работы. Даже когда сеть полностью функционирует, система должна балансировать между низкой задержкой и согласованностью.

4. Strong Eventual Consistency (SEC) - это модель согласованности, в которой реплики могут временно расходиться, но гарантированно сходятся к одному состоянию после обмена обновлениями благодаря разрешению конфликтов. Поскольку SEC не требует сильной согласованности, она не попадает под запрет CAP-теоремы. Поэтому набор {SEC, A, P} не создаёт противоречия - система остаётся доступной и устойчивой к сетевым сбоям, а согласованность достигается не сразу, а в конечном итоге, когда узлы смогут обменяться обновлениями. (дисклеймер - я не до конца уверен в правильности ответа, до этого я слышал о сравнение strong consistency и eventual consistency, но не strong eventual consistency, поэтому ориентировался на статью)

5. После прочтения статьи можно сформулировать идею Мартина Клеппмана так: распределенные системы не должны рассматриваться через призму CAP-теоремы, вместо этого следует использовать delay-sensitivity framework. Операции, требующие глобальной координации, неизбежно становятся чувствительными к сетевым задержкам и в условиях больших или неопределённых задержек будут иметь высокую латентность (в оригинале latency, но стоит разделять network delay и latency) или недоступность. Операции, не требующие координации, являются delay-independent - их латентность не зависит от задержек в сети и сетевой разграниченности. Таким образом, архитектор системы должен оценивать операции на основе их чувствительности к задержкам и выбирать подходящие модели согласованности и доступности в зависимости от требований приложения.

□

1.3 Модели репликации

Для хранения пользовательских данных в «Репликоне» решили воспользоваться платформой хранения больших данных YTsaurus, а точнее – ее механизмом реплицированных динамических таблиц, которые представляют собой key-value хранилища в реляционном стиле.

Если коротко, применение **модифицирующих операций** сводится к добавлению соответствующей записи в очередь операций на master-кластере. Затем каждое изменение из лога последовательно применяется на всех slave-кластерах синхронно и/или асинхронно (это может приводить к рассогласованию данных между master- и slave-кластерами в течение некоторого времени). Пока операция модификации не применена на всех slave-кластерах, она не очищается из очереди на master-кластере. **Операции чтения** производятся на slave-кластерах.

Задание:

1. (1.25 балл)
   - Сформулируйте общее и различия между понятиями *федерация* и *шардирование* в контексте баз данных.
   - Как эти подходы помогают масштабировать системы хранения данных?
   - Что делать, если мы уперлись в пределы вертикального масштабирования внутри одной СУБД (при федерации) / партиции (при шардировании)?
2. (1.25 балла)
   - Опишите с точки зрения CAP-теоремы, к какому классу систем можно отнести реплицированные динамические таблицы в YTsaurus.
   - Предположим, что один из slave-кластеров отказал. Почему в этот момент система все еще может считаться доступной?
   - Какой тип согласованности (consistency) обеспечивается реплицированными динамическими таблицами? Это модель ACID или BASE?
3. (2 балла) (*) Утверждается, что CRDT-типы [3] способны решить задачу master-master репликации. Объясните, что это за типы данных и почему им не требуется механизм консенсуса для обеспечения согласованности.

Решение:

1. Масштабирование БД.
   - Федерация и шардирование - это подходы к горизонтальному масштабированию базы данных. Согласно статье, федерация подразумевает разделение данных на несколько независимых баз данных, каждая из которых отвечает за свою

[3] Хабр: Наивное введение в CRDT-типы

часть данных. Шардирование делит данные внутри одной базы данных на более мелкие части (шарды), которые распределяются по нескольким узлам, но при этом остаются частью единой базы данных. NoSQL базы данных часто шардируют данные из коробки.

- Оба подхода позволяют балансировать нагрузку между несколькими серверами, что увеличивает общую производительность системы.
- Если мы упремся в пределы масштабирования, то всегда можно попробовать решить проблему другими способами: 1. проанализировать запросы (может разрабочик плохо написал запросы к БД) и таблицы (может стоит изменить структуру таблиц, накинуть или убрать индексы); 2. добавить слой с кешированием; 3. переехать на другую СУБД, которая лучше подходит под задачи проекта.

2. YTsaurus

   - Реплицированные динамические таблицы в YTsaurus можно отнести к AP-системам (Availability и Partition tolerance). В упомянутой статье в главе "Гарантии" говорится "для читателя реплики нет гарантий атомарности — он может застать реплику в промежуточном состоянии, когда к ней применилась лишь часть изменений транзакции". Из этого следует, что согласованность не гарантируется. Однако, судя по статье, система приходит к согласованности не сразу, а в конечном итоге. Partition tolerance обеспечивается за счет репликации между мета-кластером и кластерами-репликами. Availability обеспечивается тем, что если один из slave-кластеров отказал, система продолжает обслуживать запросы на чтение с других slave-кластеров.
   - Если один из slave-кластеров отказал, система все еще может считаться доступной, потому что другие slave-кластеры продолжают обслуживать запросы на чтение, а запросы на запись продолжают приниматься работающим master-кластером и реплицироваться на оставшиеся slave-кластеры.
   - Реплицированные динамические таблицы обеспечивают eventual consistency, что ближе к модели BASE, чем к ACID.

3. CRDT (Conflict-free Replicated Data Type) - это такие структуры данных, которые спроектированы для работы в распределённых системах, Их главная особенность в том, что они гарантируют eventual consistency. CRDT обходят необходимость консенсуса, потому что операции над ними могут быть применены в любом порядке на разных узлах без возникновения конфликтов (коммутативность операций). Для примера можно привести GCounter - распределенный счетчик, который может только увеличиваться. Каждый узел хранит не одно число, а массив чисел, где каждое число - это инкремент с этого узла. При слиянии двух счетчиков берется максимум для каждого элемента массива и суммируются эти максимумы. Таким образом, независимо от порядка применения операций, все узлы придут к одному и тому же значению счетчика.

□

1.4 Балансировка нагрузки

«Дореплицировались!» – подумал лид Конвергентий Смоуктуновский одной из команд «Репликона», когда его поисковая система картинок с котиками «Котоморфизм» сложилась карточным домиком под напором пользователей.

СТО «Репликона» не остался в долгу и поручил Конвергентию разобраться с проблемой, хотя и не очень обрадовался тому, что лид в его подчинении не догадался заранее позаботиться о масштабировании и балансировке нагрузки в критически важном сервисе.

Чтобы быстро исправить техдолг перед внедрением полноценного кэша на основе Redis, Конвергентий решил выполнить простые шаги:

1. Переехать из одной зоны в три георазпределенные зоны с балансировкой нагрузки между ними с помощью **nginx**, чтобы «Котоморфизм» спокойно переживал отказ одной из зон.

2. Добавить **fallback** на уровне балансировщика, чтобы отображать типовые картинки с котиками, если «Котоморфизм» снова начнет «мяукать» под нагрузкой.

Помогите Конвергентию выполнить поставленные задачи, чтобы «Котоморфизм» снова начал радовать пользователей. В папке `homeworks/cotomorphism` находится тестовый стенд из нескольких узлов «Котоморфизма», которые возвращают различные картинки с котиками. С помощью эндпоинтов `/degrade/on` и `/degrade/off` можно включать и выключать «режим деградации», при котором сервер начинает возвращать ошибку 503 вместо картинки.

В вашем распоряжении baseline-конфигурация `default.conf`, которая балансирует нагрузку между тремя узлами «в разных зонах» «Котоморфизма» с помощью round-robin.

```
upstream app_backend {
    server cat1:80;
    server cat2:80;
    server cat3:80;
}

upstream app_fallback {
    server cat_cached:80;
}

server {
    listen 80;
    server_name _;
```

```
    location / {
        proxy_pass http://app_backend;
    }

    location @fallback {
        proxy_pass http://app_fallback;
    }
}
```

Задание:

1. (0.75 балл) Сделайте так, чтобы при возникновении ошибки 503 от одного из узлов без каких-либо дальнейших попыток балансировщик переадресовывал запрос на upstream `app_fallback`.
2. (0.75 балл) Настройте балансировщик так, чтобы он пытался повторить запрос на другой узел при возникновении ошибок 503. Если `cat1`, `cat2`, `cat3` лежат, то необходимо спроксировать запрос на upstream `app_fallback`.
3. (0.75 балл) Добавьте кэширование запроса картинок на уровне балансировщика с временем жизни кэша 5 секунд.
4. (0.75 балл) Добавьте rate limiting на уровне балансировщика: не более 1 запроса в секунду от одного клиента. Убедитесь, что при превышении лимита вы падаете на `app_fallback`.

Указание: оформите ваше решение как минимальный набор команд, которые надо дописать в baseline-конфигурацию. Воспользуйтесь блоком minted..

Решение:

```
proxy_cache_path /var/cache/nginx keys_zone=app_cache:10m;
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

upstream app_backend {
        # fail_timeout позволяет серверу "восстанавливаться" после того,
        # как nginx пометит его как недоступный
        server cat1:80 max_fails=1 fail_timeout=5s;
        server cat2:80 max_fails=1 fail_timeout=5s;
        server cat3:80 max_fails=1 fail_timeout=5s;
}

upstream app_fallback {
        server cat_cached:80;
}

server {
```

```
        listen 80;
        server_name _;

        location / {
                proxy_pass http://app_backend;

                # тротлинг
                limit_req zone=one burst=1 nodelay;
                limit_req_status 503;

                # кеширование
                proxy_cache app_cache;
                proxy_cache_valid 200 5s;

                # обработка деградированных инстансов
                proxy_next_upstream error timeout invalid_header http_503;
                proxy_next_upstream_tries 3;

                # фолбэк на инстанс приложения с "кешированным" котэ
                proxy_intercept_errors on;
                error_page 503 = @fallback;
        }

        location @fallback {
                proxy_pass http://app_fallback;
        }
}
```

□

1.5 Кэширование

Уверенный мидл разработчик Шардимир в свободное время от работы в «Репликоне» играет в Minecraft и больше всего на свете любит мод ComputerCraft, который добавляет в игру компьютеры, которые работают на операционной системе и поддерживают программы в виде скриптов на языке Lua. Единственное, что огорчает Шардимира в этом моде, – это необходимость писать на совершенно бесполезном в жизни языке Lua.

Шардимир получил задание от своего тимлида Конвергентия – реализовать кэш для сервиса «Котоморфизм» на основе Redis. Какое же было удивление Шардимира, когда он осознал, что для этого ему придется написать пару Lua-скриптов для Redis!

```
return 'Hello World'
```

```
EVAL "return 'Hello World'" 0
```

Помогите Шардимиру реализовать **cache-aside** кэширование запросов к сервису «Котоморфизм» с помощью Lua-скриптов для Redis. Поднять консоль можно с помощью папки `homeworks/redis`.

Задание:

1. (1 балл) Напишите два скрипта: один принимает в качестве аргументов поисковый запрос (строка), TTL (в секундах) и путь к файлу (строка); скрипт должен сохранить в Redis значение из файла по ключу поискового запроса. Второй должен принимать в качестве аргумента поисковый запрос (строка) и возвращать значение по этому ключу из Redis. Вам могут пригодиться команды `EVAL`, `GET`, `SETEX`.
2. (1 балл) Реализуйте rate limiter для пользователя как соответствующие Lua скрипты для Redis. Вам могут пригодиться команды `INCR`, `EXPIRE`.

Решение:

```
-- Сохранение пути файла в кеш с TTL
local key = ARGV[1]
local ttl = ARGV[2]
local filePath = ARGV[3]

redis.call("SET", key, filePath)
redis.call("EXPIRE", key, ttl)

return "OK"
```

```
-- Получение пути файла из кеша
local key = ARGV[1]

local value = redis.call("GET", key)
return value
```

```
-- Rate limiter
-- Использование: EVAL {...script} 0 user_id limit time_window
local rateLimiterKey = "rate:" .. ARGV[1]
local limit = tonumber(ARGV[2])
local timeWindow = tonumber(ARGV[3])

local requestsAmount = redis.call("INCR", rateLimiterKey)

if requestsAmount == 1 then
        redis.call("EXPIRE", rateLimiterKey, timeWindow)
end

local shouldDeny = requestsAmount > limit
if shouldDeny then
        return "DENY"
else
        return "OK"
end
```

□