Семинар 2. Масштабирование, балансировка и кэширование

Принципы построения высоконагруженных систем

Георгий Семенов

Институт прикладных компьютерных наук
Университет ИТМО

осень 2025

Содержание

Вспоминаем лекцию

- Масштабирование = вертикальное (железом) и горизонтальное (количеством инстансов сервиса)
- Хотим stateless-сервисы за счет stateful хранилищ
- Все еще хотим observability + автомасштабирование
- Хотим единообразно управлять конфигурациями и состоянием (например, 60 подов x 3 зоны = 180 черных ящиков)

Облака – белогривые лошадки

- Абстракция в виде «очередей», «сервисов», «хранилищ», «подов»
- Есть внешний гипервизор/набор автоматических агентов, обслуживающих инфраструктуру
- Связано с docker registry – сервисы загружают образы из единого хранилища
- Манифесты описывают требуемое железо и количество инстансов
- Умное облако управляет ресурсами и автоматически распределяет их между контейнерами
- Избыточность зон

Kubernetes — структура

- Control Plane:
  - **API Server** — единственная точка входа для команд и автоматов
  - **etcd** — хранилище состояния (истина в последней инстанции)
  - **Scheduler** — решает, куда посадить поды
  - **Controller Manager** — следит за соответствием желаемого и фактического состояния
- Worker Node:
  - **Kubelet** — управляет контейнерами
  - **Kube-proxy** — настройка сетевых правил
  - **Container runtime** (containerd / CRI-O)
- Можно переиспользовать конфигурации с помощью Helm Charts

Kubernetes — автомасштабирование

- **Horizontal Pod Autoscaler (HPA)**
  - Масштабирует число подов
  - Метрики: CPU, RAM, кастомные Prometheus-метрики
  - Работает через Metrics Server или адаптеры
- **Vertical Pod Autoscaler (VPA)**
  - Подходит для stateful-сервисов
- **Cluster Autoscaler (CA)**
  - Добавляет или удаляет ноды в кластере
  - Работает с облаками: GCP, AWS, Yandex Cloud

Флаппинг как фактор эксплуатации

- Мгновенные статистики могут «скакать» на небольших интервалах времени
- Ложные срабатывания *алертов* и лишние *операции масштабирования*
- Неразрешимый вопрос – «как отличить флап от реальной проблемы?»

Интеграции сервисов управления конфигурацией

Tool	**Notable Integrations**
Consul	Terraform, Vault, Nomad, Kubernetes
etcd	Kubernetes, CoreDNS, Vitess, Rook, Prometheus
ZooKeeper	Kafka, Hadoop, HBase, Solr, Druid, Apache Curator

Типичные паттерны управления

- Leader election (выбор лидера)
- Distributed locks (распределённые блокировки - ZooKeeper)
- Coordination (координация действий – Raft, Zab)
- Service discovery
- Observe / Watch / Notify
- Dynamic configuration (горячая замена)

Содержание

Зачем нужна балансировка нагрузки?

- Равномерное распределение запросов между серверами
- Устранение single point of failure
- Поддержка горизонтального масштабирования (нужен service discovery)

L4 балансировка

- Работает на уровне транспортного протокола (TCP/UDP)
- Быстрая, не анализирует HTTP или полезную нагрузку
- Идеальна для gRPC, баз данных, бинарных протоколов
- Методы:
  - Round Robin
  - Least Connections
  - Source IP Hash
- Примеры: LVS, HAProxy (L4), AWS NLB, Google GLB

L7 балансировка

- Работает на уровне HTTP/HTTPS/HTTP2/gRPC
- Понимает URL, cookie, headers, методы
- Позволяет делать content-based routing:
  - по пути URL (/api, /static)
  - по домену (example.com)
  - по пользовательскому агенту
  - по версии API
- Поддерживает TLS termination
- Примеры: Nginx, Envoy, Traefik, AWS ALB, GCP HTTPS LB

DNS балансировка

- Самый простой способ распределить трафик
- Один домен $\rightarrow$ несколько IP-адресов
- Распределение зависит от DNS resolver'а
- Ограничения:
  - нет быстрого failover — зависит от TTL
  - нет health checks
  - нет учёта нагрузки
- Используется для глобальной балансировки (GSLB)

Anycast балансировка

- Несколько дата-центров объявляют один и тот же IP
- BGP направляет пользователя к ближайшей по сети точке
- Используется крупными CDN и DNS-провайдерами
- Преимущества:
  - низкая задержка (пользователь $\rightarrow$ ближайший POP)
  - автоматический failover на уровне маршрутизации
- Примеры: Cloudflare, Google DNS 8.8.8.8, AWS Global Accelerator

Client-side балансировка

- Клиент (или SDK) сам выбирает сервер для запроса
- Часто используется в сервисах с сервис-мешем
- Требует списка доступных узлов (через Consul/Eureka)
- Примеры:
  - gRPC internal load balancing
  - Service Mesh (Envoy Sidecar)
  - Netflix Ribbon / Spring Cloud
- Можно учитывать локальные метрики (RTT, latency, errors)

Reverse proxy как балансировщик

- Проксирует HTTP-трафик к backend-сервисам
- Делает:
  - TLS termination
  - rate limiting
  - routing по пути, хедерам, метаданным
  - кэширование ответов
- Примеры: Nginx, Envoy, HAProxy, Traefik
- Фактически стандарт для L7 балансировки

Алгоритмы распределения нагрузки

- **Round Robin** — по очереди
- **Weighted Round Robin** — учитывает разные мощности узлов
- **Least Connections** — выбирает узел с наименьшим числом активных соединений
- **Least Response Time** — выбор по задержке
- **IP Hash** — полезно для session stickiness
- **Consistent Hashing** — шардирование и sticky routing без перебалансировки

Health Checks

- LB регулярно проверяет состояние серверов
  - HTTP/HTTPS: статус-коды 200–399
  - TCP: успешное установление соединения
  - gRPC: проверка служебного метода (gRPC Health API)
- Отключает «больные» узлы от трафика
- Позволяет выполнять rolling updates без даунтайма

Паттерны балансировки

- Blue–Green Deployment
- Окружения: testing + shadow + canary (1%) + production (99%)
- Hedging, Circuit Breakers, Rate Limiters, Throttlers...
- Sticky Sessions / Session Affinity

Антипаттерны

- Sticky sessions без необходимости
- Один балансировщик $\rightarrow$ SPOF (single point of failure)
- Игнорирование health checks (service discovery)
- DNS балансировка без контроля TTL
- L7 балансировка для тяжёлого бинарного трафика (неэффективно)

Содержание

Мотивация

- Снижение нагрузки на базу данных и серверы
- Ускорение отклика пользователю
- Позволяет масштабировать систему без дорогих ресурсов

Основные стратегии кэширования

Database Caching Strategies

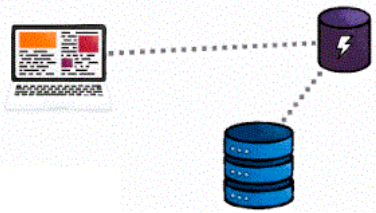
by levelupcoding.com

Write-Through

Data is written and updated simultaneously on both the cache and the underlying datastore.

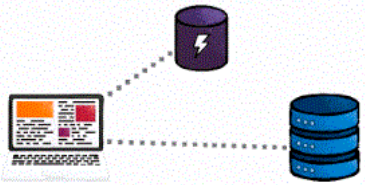
Write-Behind

Data is first written to the cache and then asynchronously to the database.



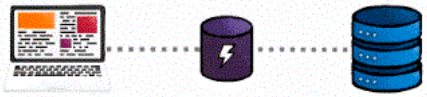
Cache-Aside

Data is explicitly fetched and stored in cache by the application when necessary.



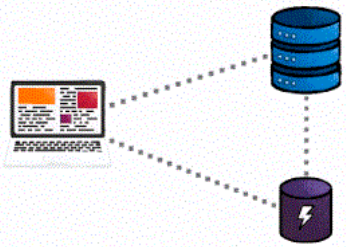
Read-Through

The cache sits between the application and the database.



Write-Around

Data is written to the database, and populated in the cache upon a subsequent read request.



Как выбирать стратегию кэширования

- Преобладают чтения $\rightarrow$ Cache-Aside или Read-Through
- Преобладают записи $\rightarrow$ Write-Around или Write-Behind
- Нужна строгая консистентность $\rightarrow$ Write-Through
- Данные редко меняются $\rightarrow$ долгий TTL + Cache-Aside
- Высокая цена ошибки $\rightarrow$ избегать Write-Behind

Фреймворк для размышления о компоненте системы

- Внешний входной запрос (external input)
- Внутренний входной запрос (internal input)
- Внутренний выходной запрос (internal output)
- Внешний выходной запрос (external output)
- **Вопрос:** что из этого ключ кэша, а что значение кэша?

Стратегии инвалидации

- TTL (time-to-live) — автоматическое устаревание данных
- Write-through invalidation — обновление кэша при записи
- Delete-on-write — удаление из кэша при изменении данных
- Periodic cache refresh — регулярное обновление по cron

Подходы к реализации кэша

- В памяти приложения (in-process cache): LRU (least recently used), ARC (adaptive replacement cache)
- Внешний кэш: Redis, Memcached
- Распределённый кэш: Redis Cluster, Hazelcast, Ignite, YTsaurus
- CDN-кэширование для статического контента
- Database-level caching: buffer pool, query cache

Антипаттерны кэширования

- Кэширование всего подряд (cache pollution)
- Слишком большой TTL $\rightarrow$ stale data
- Слишком маленький TTL $\rightarrow$ низкая эффективность
- Отсутствие политики инвалидации
- Сложная логика кэширования внутри бизнес-кода

Содержание

NGINX как балансировщик нагрузки

- Легковесный и быстрый reverse-proxy
- Поддерживает L4 и L7 балансировку
- Позволяет гибко управлять трафиком: round-robin, least connections (fair-подход), IP hash
- Идеален для демонстрации распределения нагрузки

Базовая конфигурация NGINX

- Конфигурационный файл: `/etc/nginx/nginx.conf` или `/etc/nginx/conf.d/*.conf`
- Основные блоки конфигурации:
  - `upstream` — пул backend-серверов
  - `server` — настройки виртуального хоста
  - `location` — маршрутизация
- Проверка конфигурации:

```
nginx -t
```

- Перезапуск:

```
nginx -s reload
```

Round-robin балансировка в NGINX

- Простой и наиболее распространённый метод
- Запросы равномерно распределяются по backend-серверам

```
upstream backend {
    server web1:5000;
    server web2:5000;
    server web3:5000;
}

server {
    listen 80;
    location / {
        proxy_pass http://backend;
    }
}
```

Least connections & IP Hash

```
upstream backend1 {
    least_conn;
    server web1:5000;
    server web2:5000;
    server web3:5000;
}

upstream backend2 {
    ip_hash;
    server web1:5000;
    server web2:5000;
    server web3:5000;
}
```

Примеры конфигураций

```
error_page  404  /404.html;

location = /404.html {
  root  /spool/www;
}

location /old_stuff/ {
  rewrite   ^/old_stuff/(.*)$  /new_stuff/$1  permanent;
}
```

Примеры конфигураций

```
location / {
  proxy_pass          http://127.0.0.1/;
  proxy_redirect      off;

  proxy_set_header    Host              $host;
  proxy_set_header    X-Real-IP         $remote_addr;
  #proxy_set_header   X-Forwarded-For   $proxy_add_x_forward

  client_max_body_size       10m;
  client_body_buffer_size    128k;

  proxy_connect_timeout      70;
  proxy_send_timeout         90;
  proxy_read_timeout         90;

  charset  koi8-r;
}
```

Содержание

Redis: основы

- In-memory key-value хранилище
- Идеален для кэшей, очередей, счётчиков
- Поддерживает множество структур данных: strings, hash, list, set, sorted set
- Master/slave – это значит, что Master - точка отказа!

Основные команды Redis

- Установка ключа:

```
SET user:1 "Alice"
```

- Чтение:

```
GET user:1
```

- Истечение (TTL):

```
SET session:123 ABC EX 60
TTL session:123
```

- Удаление:

```
DEL user:1
```

- Проверка наличия:

```
EXISTS user:1
```

Cache-Aside (Lazy Loading)

- Приложение проверяет кэш → если miss → читает из БД и кладёт в Redis
- Самый популярный паттерн

```
value = redis.get(key)
if value is None:
    value = db.query(key)
    redis.set(key, value)
return value
```

Команды Redis для демонстрации

- Подключиться:

  `redis-cli`
- Протестировать скорость GET:

  `redis-benchmark -t get -n 10000`
- Получить все ключи:

  `KEYS *`
- Посмотреть статистику:

  `INFO`
- Удалить кэш:

  `FLUSHALL`

Содержание

Итоги

- Рассмотрели практические аспекты масштабирования, балансировки и кэширования
- Посмотрели демо с NGINX и Redis
- Выдано второе домашнее задание