

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

«Высоконагруженные и распределённые системы»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного
обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
Лабораторная работа №1. Сравнение блокирующего и неблокирующего сервера (нагрузочное тестирование).....	4
Лабораторная работа №2. Consistent Hashing с виртуальными узлами (Memcached-подобный кэш).....	12
Лабораторная работа №3. Репликация «master-slave» с синхронной/асинхронной записью	20
Лабораторная работа №4. Шардирование по range и hash (симуляция)	28
Лабораторная работа №5. Saga (хореография) через очереди сообщений	37
Лабораторная работа №6. Raft: лидер и репликация лога (упрощённая реализация)	45
Лабораторная работа №7. Распределённый rate limiter через Redis (Token Bucket + Lua)..	54
Лабораторная работа №8. Circuit Breaker с мониторингом (Resilience4j / сторонняя реализация).....	64
Лабораторная работа №9. Distributed tracing с OpenTelemetry и Jaeger	74
Лабораторная работа №10. Backpressure через Reactive Streams (на примере Project Reactor)	85
Лабораторная работа №11. Асинхронная обработка через Kafka (consumer groups, offset)	95
Лабораторная работа №12. Load balancing симуляция: взвешенный RR + health checks..	105
Лабораторная работа №13. Кэш с write-through и TTL (Redis + приложение)	117
Лабораторная работа №14. Симуляция масштабирования read replicas	127
Лабораторная работа №15. CQRS с проекциями через очередь (Event Sourcing light)	136
Лабораторная работа №16. Chaos-тестирование: отказ узла и автоматическое восстановление	146
Лабораторная работа №17. Полный highload-стек (Docker Compose + мониторинг).....	154
Лабораторная работа №18. Итоговый проект: масштабируемое приложение с обоснованием архитектуры	167
СПИСОК ЛИТЕРАТУРЫ.....	176

ВВЕДЕНИЕ

Настоящие методические указания предназначены для выполнения лабораторных работ по дисциплине «Высоконагруженные и распределённые системы».

Цель дисциплины: формирование у студентов системных компетенций по проектированию, реализации, тестированию и сопровождению высоконагруженных распределенных программных систем с применением современных архитектурных паттернов (Saga, CQRS, Circuit Breaker, Backpressure), методов согласованности (CAP/PACELC, Raft), масштабирования (шардирование, репликация), отказоустойчивости (retry, chaos engineering), наблюдаемости (tracing, метрики) и автоматизации (CI/CD, контейнеризация).

Задачи: освоение теоретических основ highload, формирование практических навыков разработки и тестирования распределенных систем, обеспечение интеграции с ранее изученными курсами, развитие проектных и документационных компетенций.

Данная методические указания включают 18 лабораторных работ, которые постепенно проводят студента от базовых концепций асинхронной обработки до создания полноценного высоконагруженного стека с мониторингом и обоснованием архитектурных решений. Каждая работа содержит чётко сформулированные цель и задачи, теоретическую часть, детальную методику выполнения, примеры, индивидуальные варианты заданий, требования к отчёту и контрольные вопросы.

Лабораторная работа №1.

Сравнение блокирующего и неблокирующего сервера (нагрузочное тестирование)

Цель работы: измерить разницу производительности (RPS, p99 latency) между синхронным (thread-per-request) и асинхронным (event loop) HTTP-серверами при возрастающей нагрузке.

Задачи:

- разработать два простых HTTP-сервера – блокирующий (на потоках) и неблокирующий (на асинхронном цикле событий);
- выполнить нагрузочное тестирование с помощью инструментов wrk или bombardier, изменяя число одновременных соединений;
- сравнить полученные метрики: RPS (запросов в секунду), p99 латентность, потребление оперативной памяти;
- сделать вывод о применимости каждого подхода в разных сценариях.

Теоретическая часть

В высоконагруженных распределённых системах способ обработки входящих запросов напрямую влияет на пропускную способность и задержки. Традиционная модель **blocking I/O** (синхронный ввод-вывод) выделяет отдельный поток (или процесс) на каждое соединение. Пока поток ожидает завершения операции ввода-вывода (чтение диска, вызов внешнего API и т.д.), он находится в состоянии ожидания и не может обрабатывать другие запросы. При большом количестве одновременных соединений это приводит к резкому росту потребления памяти (стек потока) и накладным расходам на переключение контекста.

Неблокирующий (асинхронный) подход использует один или несколько потоков, управляемых циклом событий (event loop). Все операции ввода-вывода запускаются в неблокирующем режиме; когда операция

завершается, цикл событий уведомляет обработчик. Один поток может обслуживать тысячи соединений, поскольку не тратит время на простое ожидание. Типичные реализации: asyncio в Python, Netty в Java, Node.js.

Ключевые характеристики для сравнения:

- **RPS** (requests per second) – количество обработанных запросов за секунду;
- **p99 latency** – 99-й перцентиль времени ответа (99% запросов выполняются быстрее этого значения);
- **потребление памяти** (RSS, VIRT) при разных уровнях параллелизма.

Факторы, влияющие на результаты:

- тип нагрузки: CPU-bound (вычисления) или I/O-bound (ожидание данных);
- количество одновременных соединений;
- настройки операционной системы (лимиты открытых файлов, размер сокетных буферов).

Методика выполнения работы

1. Подготовка окружения

1.1. Установите инструменты нагрузочного тестирования:

- Wrk (рекомендуется) – `sudo apt install wrk` (Linux) или `brew install wrk` (macOS);
- альтернативно bombardier – `go install github.com/codesenberg/bombardier@latest`.

1.2. Убедитесь, что установлен Python 3.8+ (или Java 11+, если выбран Java-стек).

1.3. Для контроля памяти установите утилиту psrecord или используйте htop, top.

2. Разработка блокирующего (синхронного) сервера

2.1. Создайте файл `blocking_server.py`.

2.2. Реализуйте простой HTTP-сервер, который на каждый запрос отвечает "OK" после небольшой имитации I/O (например, `time.sleep(0.01)`). Каждый запрос обрабатывается в отдельном потоке.

```
from http.server import HTTPServer, BaseHTTPRequestHandler
from socketserver import ThreadingMixIn
import time
```

```
class Handler(BaseHTTPRequestHandler):
```

```
    def do_GET(self):
        time.sleep(0.01) # имитация I/O
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b'OK')
```

```
class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
```

```
    pass
```

```
if __name__ == '__main__':
```

```
    server = ThreadedHTTPServer(('0.0.0.0', 8080), Handler)
    server.serve_forever()
```

3. Разработка неблокирующего (асинхронного) сервера

3.1. Создайте файл `async_server.py`.

3.2. Используйте `aiohttp` (`pip install aiohttp`).

3.3. Напишите сервер с асинхронной задержкой `asyncio.sleep(0.01)`.

```
from aiohttp import web
import asyncio
```

```
async def handle(request):  
    await asyncio.sleep(0.01) # имитация I/O  
    return web.Response(text='OK')  
  
app = web.Application()  
app.router.add_get('/', handle)  
  
if __name__ == '__main__':  
    web.run_app(app, host='0.0.0.0', port=8080)
```

4. Нагрузочное тестирование

4.1. Запустите каждый сервер по очереди (в отдельных терминалах).

4.2. Для каждого сервера выполните тесты с разным числом соединений, например:

```
wrk -t 2 -c 10 -d 30s http://localhost:8080/  
wrk -t 2 -c 100 -d 30s http://localhost:8080/  
wrk -t 2 -c 500 -d 30s http://localhost:8080/
```

Параметры: -t число потоков нагрузки, -c число одновременных соединений, -d длительность.

4.3. Запишите показатели RPS, задержки (включая p99) и потребление памяти сервером (через psrecord --plot memory.png --interval 1 <PID>).

5. Анализ результатов

5.1. Постройте графики RPS и p99 латентности от числа соединений для обоих серверов.

5.2. Сравните, как растёт память при увеличении параллелизма.

5.3. Объясните, почему один сервер может показать падение производительности при высоком -с, а другой – нет.

Пример выполнения задания

Блокирующий сервер (thread-per-request) – при -с 10 достигает ~800 RPS, p99 = 45 мс, память ~15 МБ.

При -с 500 RPS падает до 200, p99 растёт до 3 с, память увеличивается до 1.2 ГБ (потoki).

Асинхронный сервер – при -с 10 показывает ~950 RPS, p99 = 35 мс, память ~25 МБ.

При -с 500 RPS стабильно держится на уровне 900, p99 = 50 мс, память ~35 МБ.

Вывод: асинхронный сервер почти не теряет производительность при росте соединений и требует существенно меньше памяти, что делает его предпочтительным для I/O-связанных задач с большим количеством клиентов.

Индивидуальные задания

1. Реализовать блокирующий сервер на Java (ServerSocket с новым потоком на соединение) и неблокирующий на Netty. Сравнить при имитации I/O задержкой 20 мс.

2. В обоих серверах вместо sleep выполнить вычисление числа Фибоначчи (n=30) – CPU-bound нагрузка. Проверить, изменится ли соотношение производительности.

3. Добавить третий тип – смешанный (гибридный) сервер с пулом потоков фиксированного размера (например, 50 потоков) и сравнить с двумя исходными.

4. Заменить имитацию I/O на чтение файла (10 МБ) с диска для каждого запроса. Оценить влияние кэширования ОС.

5. Использовать bombardier вместо wtk и сравнить точность измерений при длительности теста 10, 30, 60 секунд.

6. Реализовать серверы на Node.js (блокирующий – без async, просто createServer с fs.readFileSync; неблокирующий – стандартный event loop).

7. Провести тестирование с 1000, 5000 и 10000 соединений (увеличить лимиты ОС: ulimit -n 20000). Оценить, достигнет ли блокирующий сервер предела по количеству потоков.

8. Для асинхронного сервера искусственно ограничить количество рабочих потоков (например, через asyncio.run с разными параметрами) и измерить влияние на RPS.

9. Добавить в оба сервера обработку POST-запросов с телом до 1 КБ и записью в память (имитация логирования). Сравнить результаты.

10. Изменить имитационную задержку на случайную величину от 1 до 50 мс (равномерное распределение). Исследовать, как меняется p99.

11. Запустить серверы в контейнерах Docker с ограничением CPU (1 ядро, 2 ядра) и повторить тесты.

12. Реализовать неблокирующий сервер с использованием uvicorn (FastAPI) и сравнить с aiohttp.

13. Добавить эндпоинт /health без задержки и /data с задержкой. Выполнить смешанную нагрузку (30% health, 70% data).

14. Измерять не только RPS и латентность, но и потребление CPU (user, system). Построить совместные графики.

15. Написать собственный скрипт нагрузочного тестирования на asyncio (без внешних утилит) и сравнить его результаты с wrk.

Требования к отчёту

Отчёт должен быть оформлен в электронном виде (PDF или DOCX) и содержать:

- титульный лист с названием работы, ФИО студента, группой;
- цель работы и задачи (скопировать из методички);

- теоретическая часть – краткое описание блокирующего и неблокирующего подходов;
- раздел «Ход выполнения работы» с кодом серверов и командами для запуска тестов;
- результаты измерений в виде таблиц (RPS, p99 latency, память для каждого уровня -с);
- графики (RPS vs connections, p99 vs connections, память vs connections);
- сравнительный анализ – почему один сервер оказался лучше/хуже;
- выводы по каждому из пунктов цели работы;
- ответы на контрольные вопросы (письменно).

Код должен быть представлен в листингах с комментариями. Все графики и таблицы подписаны.

Контрольные вопросы

1. В чём принципиальное отличие блокирующего сокета от неблокирующего?
2. Как цикл событий (event loop) управляет несколькими соединениями в одном потоке?
3. Почему при CPU-нагруженных задачах асинхронный сервер может не дать выигрыша?
4. Что такое «стек потока» и как он влияет на потребление памяти блокирующего сервера?
5. Какие системные ограничения (ulimit, файловые дескрипторы) могут снизить максимальное число соединений?
6. Почему в тесте с 500 соединениями блокирующий сервер показывает высокую p99 латентность?
7. В чём разница между `wrk -t 2 -c 100` и `wrk -t 10 -c 100`?
8. Какие метрики наблюдаемости (tracing, метрики) вы бы добавили для глубокого анализа производительности?

9. Какую роль играют планировщик потоков ОС и переключение контекста в синхронной модели?

10. Может ли асинхронный сервер уступать по RPS синхронному при очень малом количестве соединений (например, -с 1)? Почему?

Лабораторная работа №2.

Consistent Hashing с виртуальными узлами (Memcached-подобный кэш)

Цель работы: реализовать распределённый кэш на основе consistent hashing с виртуальными узлами и оценить равномерность распределения ключей, а также количество перемещаемых ключей при изменении состава узлов по сравнению с классическим хешированием по модулю.

Задачи:

- реализовать кольцо хешей (пространство 2^{32}) с поддержкой добавления и удаления узлов;
- добавить виртуальные узлы (100–200 на физический узел) для улучшения балансировки;
- смоделировать добавление и удаление узлов, подсчитать долю ключей, изменяющих своё место;
- сравнить поведение consistent hashing с простым $\text{hash}(\text{key}) \% N$;
- визуализировать распределение ключей и сделать вывод о качестве балансировки.

Теоретическая часть

В распределённых системах хранения (например, Memcached, Redis Cluster, Dynamo) необходимо решить, на каком узле хранить данный ключ. Простейший способ – **модульное хеширование**: $\text{node} = \text{hash}(\text{key}) \% N$, где N – число узлов. При добавлении или удалении узла N изменяется, и почти все ключи (около $(N-1)/N$) должны быть перераспределены. В высоконагруженной системе это вызывает лавину перемещений данных и временную деградацию.

Consistent hashing (Когерентное хеширование) решает эту проблему. Пространство хешей представляется в виде кольца (например, от 0 до $2^{32} - 1$). Каждый узел (сервер) отображается на одну или несколько точек кольца (его

хеш). Каждый ключ также хешируется, и он назначается первому узлу, идущему по часовой стрелке от его позиции.

Ключевое свойство: при добавлении/удалении узла перемещаются только ключи, которые лежали на отрезке между старым положением и новым – в среднем $1/N$ часть всех ключей. Однако на практике без виртуальных узлов равномерность распределения нагрузки может быть плохой (особенно при малом числе узлов). **Виртуальные узлы** – это несколько копий одного физического узла на кольце (каждая с собственным хешем). Чем больше виртуальных узлов, тем равномернее распределение.

Метрики сравнения:

- **Коэффициент вариации (CV)** количества ключей на узел – чем меньше, тем равномернее.
- **Доля перемещённых ключей** при изменении состава узлов (чем меньше, тем лучше).

Формулы:

- $\text{hash}(\text{key}) \% N$ – перемещается $(N-1)/N$ ключей.
- Consistent hashing – примерно $(V / (V+1))$ на интервале одного виртуального узла, но суммарно около $1/N$ физических узлов.

Методика выполнения работы

1. Подготовка проекта

- 1.1. Создайте каталог lab2_consistent_hashing.
- 1.2. Напишите код на Python (рекомендуется) или Java/C++.
- 1.3. Установите необходимые библиотеки: matplotlib для графиков, numpy, scipy (по желанию).

2. Реализация кольца consistent hashing

- 2.1. Реализуйте класс ConsistentHashRing с методами add_node(node_name, virtual_count), remove_node(node_name), get_node(key).

2.2. Пространство хешей: $0 \dots 2^{32} - 1$. Используйте `hashlib.md5` или `hashlib.sha1` для хеширования строк, затем возьмите первые 32 бита (`int(hashlib.md5(...).hexdigest()[:8], 16)`).

2.3. Для каждого физического узла создайте `virtual_count` виртуальных узлов. Уникальный идентификатор – `f"{node_name}:{i}"`. Хеш этого идентификатора – позиция на кольце.

2.4. Храните отсортированный список (или `SortedList` из `bisect`) всех позиций виртуальных узлов и словарь `position→node_name`.

2.5. Метод `get_node`: вычислить хеш ключа, найти первый больший элемент в отсортированном списке (циклически – если не найден, взять минимальный).

3. Реализация модульного хеширования для сравнения

3.1. Создайте класс `ModuloHash` с методами `add_node`, `remove_node`, `get_node`.

3.2. Внутри храните список узлов. `get_node(key)` возвращает `hash(key) % len(nodes)`.

3.3. При удалении или добавлении узла всё пересчитывается заново (имитация реального поведения).

4. Генерация данных для тестирования

4.1. Сгенерируйте `N_KEYS = 10000` случайных ключей (например, `f"key_{i}"`).

4.2. Предварительно вычислите хеши ключей (для единообразия).

5. Оценка равномерности распределения

5.1. Добавьте `num_nodes = 4` физических узла. Для `consistent hashing` установите `virtual_count = 150`.

5.2. Для каждого ключа определите узел (оба метода). Подсчитайте, сколько ключей попало на каждый узел.

5.3. Вычислите среднее количество ключей на узел и коэффициент вариации ($CV = \text{стандартное отклонение} / \text{среднее}$).

5.4. Постройте столбчатую диаграмму распределения.

6. Оценка перемещения ключей при изменении узлов

6.1. Исходное состояние: 4 узла (A, B, C, D). Запомните для каждого ключа, какой узел возвращается.

6.2. Добавьте новый узел E. Повторно определите узел для каждого ключа (оба метода). Подсчитайте долю ключей, у которых узел изменился.

6.3. Удалите узел B. Снова подсчитайте долю перемещённых ключей.

6.4. Постройте график: доля перемещений для consistent hashing и для модульного хеширования в зависимости от номера операции.

7. Влияние числа виртуальных узлов

7.1. Повторите п.5 для virtual_count = 1, 50, 150, 500 (при фиксированном числе физических узлов = 4).

7.2. Постройте график зависимости коэффициента вариации от числа виртуальных узлов.

7.3. Сделайте вывод, какое количество виртуальных узлов даёт приемлемую равномерность.

8. Финальное сравнение и отчёт

8.1. Сведите результаты в таблицу.

8.2. Объясните, почему consistent hashing лучше подходит для динамических кластеров.

8.3. Укажите недостатки метода (сложность, память на хранение виртуальных узлов).

Пример выполнения задания

Код реализации кольца (фрагмент на Python):

```
import hashlib  
import bisect
```

```
class ConsistentHashRing:  
    def __init__(self, virtual_count=150):
```

```

self.virtual_count = virtual_count

self.ring = []      # отсортированные позиции
self.nodes = {}     # позиция -> имя узла

def _hash(self, s):
    return int(hashlib.md5(s.encode()).hexdigest()[:8], 16)

def add_node(self, node_name):
    for i in range(self.virtual_count):
        vnode_key = f'{node_name}:{i}'
        pos = self._hash(vnode_key)
        bisect.insort(self.ring, pos)
        self.nodes[pos] = node_name

def remove_node(self, node_name):
    to_remove = []
    for pos, node in self.nodes.items():
        if node == node_name:
            to_remove.append(pos)
    for pos in to_remove:
        del self.nodes[pos]
        self.ring.remove(pos)

def get_node(self, key):
    if not self.ring:
        return None
    hash_key = self._hash(key)
    idx = bisect.bisect_left(self.ring, hash_key)
    if idx == len(self.ring):
        idx = 0

```

```
return self.nodes[self.ring[idx]]
```

Результат теста с 4 узлами, virtual=150:

- Распределение ключей: узел А – 2478, В – 2532, С – 2499, D – 2491.
- Коэффициент вариации $CV = 0.012$ (очень хорошо).
- При добавлении узла Е переместилось только 23% ключей (вместо 80% для модульного хеширования).

Индивидуальные задания

1. Реализовать consistent hashing с использованием CRC32 вместо MD5. Сравнить равномерность распределения для 10000 ключей.
2. Добавить в кольцо возможность взвешенных узлов (разная ёмкость). Например, узел А имеет вес 2, узел В – вес 1. Адаптировать количество виртуальных узлов пропорционально весу.
3. Для consistent hashing измерить время выполнения `get_node` для 1 млн. ключей при 100 и 500 виртуальных узлах. Построить график зависимости времени от числа виртуальных узлов.
4. Реализовать алгоритм “Jump Consistent Hash” (Google, без кольца) и сравнить долю перемещений с классическим consistent hashing.
5. Использовать реальный датасет из 1 млн. ключей (строки UUID). Повторить эксперимент с добавлением/удалением узлов.
6. Написать тест, при котором узлы добавляются и удаляются в случайном порядке (10 операций). Построить накопленную долю перемещений.
7. Добавить репликацию: каждый ключ хранится на следующих R узлах по часовой стрелке. Оценить, как при этом меняется распределение нагрузки.
8. Реализовать визуализацию кольца (matplotlib) с точками виртуальных узлов и ключами (выбрать 100 ключей для наглядности).

9. Сравнить две стратегии выбора виртуальных узлов: хеширование имя:индекс и хеширование имя с добавлением случайного соли. Какая даёт более равномерное распределение?

10. Исследовать, как меняется равномерность при малом количестве узлов (2, 3) и разном числе виртуальных узлов (10, 50, 200).

11. Сымитировать отказ узла: один узел внезапно удаляется, на его нагрузка перераспределяется. Измерить время перестроения и количество перехешированных ключей.

12. Реализовать клиент-серверное приложение: серверы (узлы) запускаются как отдельные процессы, клиент использует consistent hashing для маршрутизации запросов. Провести интеграционное тестирование.

13. Сравнить производительность (операций в секунду) consistent hashing с модульным хешированием при 6 узлах и 50 тыс. запросов в секунду (имитация).

14. Добавить поддержку диапазонных запросов (по префиксу) поверх consistent hashing – как это влияет на равномерность?

15. Реализовать consistent hashing на языке Go с использованием каналов и горутин. Сравнить потребление памяти и скорость с Python-версией.

Требования к отчёту

Отчёт должен быть оформлен в текстовом редакторе (PDF) и содержать:

- титульный лист (название работы, дисциплина, ФИО, группа);
- цель работы и задачи;
- теоретическая часть – описание consistent hashing, виртуальных узлов, сравнение с модульным хешированием;
- раздел «Реализация» – листинги основных классов с комментариями;
- раздел «Эксперименты» – описание методики тестирования (количество ключей, узлов, виртуальных узлов);
- таблицы и графики:
 - распределение ключей по узлам для разных методов;

- коэффициент вариации в зависимости от числа виртуальных узлов;
- доля перемещённых ключей при добавлении/удалении узлов;
- сравнительный анализ – какие результаты лучше и почему;
- выводы по каждому пункту цели работы;
- ответы на контрольные вопросы.

Графики должны быть подписаны, таблицы пронумерованы. Код должен быть отформатирован и читаем.

Контрольные вопросы

1. В чём проблема модульного хеширования $\text{hash}(\text{key}) \% N$ в динамических кластерах?
2. Как расположены ключи и узлы на кольце consistent hashing?
3. Зачем нужны виртуальные узлы и как они влияют на равномерность распределения?
4. Как гарантировать, что при добавлении нового узла переместится минимум ключей?
5. Какой алгоритм используется для поиска узла по ключу в отсортированном списке позиций?
6. Что произойдёт, если два виртуальных узла имеют одинаковый хеш (коллизия)? Как это можно обработать?
7. Почему в consistent hashing используется пространство 2^{32} ? Как выбрать размерность?
8. Как с помощью consistent hashing реализовать репликацию (хранение копий на нескольких узлах)?
9. Какая метрика равномерности распределения считается хорошей для production-систем?
10. Сравните consistent hashing с Rendezvous (Highest Random Weight) hashing – преимущества и недостатки каждого.

Лабораторная работа №3.

Репликация «master-slave» с синхронной/асинхронной записью

Цель работы: реализовать упрощённый key-value storage с master-slave репликацией и сравнить влияние синхронного и асинхронного режимов записи на задержки (latency) и долговечность (durability) данных.

Задачи:

- разработать key-value хранилище на главном узле (master) с поддержкой репликации на один или несколько подчинённых узлов (slave);
- реализовать синхронный режим: мастер записывает данные у себя и ждёт подтверждения от всех реплик перед ответом клиенту;
- реализовать асинхронный режим: мастер отвечает клиенту сразу после локальной записи, а репликация происходит в фоновом потоке (или после некоторой задержки);
- измерить задержки записи (p50, p99) и провести эксперимент по потере данных при внезапном падении мастера;
- сравнить два режима по метрикам согласованности (CAP-теорема: выбор между консистентностью и доступностью/латентностью).

Теоретическая часть

В распределённых системах репликация данных на несколько узлов повышает отказоустойчивость и доступность. Простейшая архитектура – **master-slave (ведущий-ведомый)**: один узел (master) принимает все запросы на запись, а один или несколько slave-узлов копируют данные с мастера. Чтение может выполняться как с мастера, так и с реплик (для балансировки нагрузки, но возможны проблемы согласованности).

Синхронная репликация – мастер отправляет данные репликам и ожидает подтверждения от всех (или от кворума) перед фиксацией операции и ответом клиенту.

- *Достоинство*: данные считаются надёжно сохранёнными (durability) – при падении мастера данные уже есть на реплике.
- *Недостаток*: увеличивается задержка ответа, так как она зависит от скорости самой медленной реплики и сети.

Асинхронная репликация – мастер локально применяет запись, сразу отвечает клиенту, а репликация происходит в фоне (например, периодически или по журналу).

- *Достоинство*: низкая задержка, высокая доступность на запись.
- *Недостаток*: возможна потеря ещё не реплицированных данных при падении мастера.

Связь с CAP-теоремой: в синхронном режиме система жертвует доступностью (если реплика недоступна, запись не удастся) в пользу согласованности (Consistency) и устойчивости к сетевым разделением (Partition-tolerance). В асинхронном режиме выбирается доступность (Availability) и устойчивость к разделениям, но с возможной несогласованностью (eventual consistency).

Метрики эксперимента:

- *Latency записи* – время от отправки запроса клиентом до получения ответа.
- *Durability* – способность данных пережить падение мастера (процент восстановленных записей).
- *Throughput* (операций в секунду) – опционально.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог lab3_replication.
- 1.2. Выберите язык реализации: Python (с socket или asyncio) – для простоты, или Java/Go.
- 1.3. Установите matplotlib и numpy для построения графиков (если Python).

- 1.4. Для эмуляции сетевых задержек используйте `time.sleep(delay)` на стороне `slave`.
2. **Реализация базового key-value хранилища**
 - 2.1. Создайте класс `KVStore`, хранящий данные в словаре: `{key: value}`.
 - 2.2. Добавьте методы `put(key, value)`, `get(key)`.
 - 2.3. Реализуйте журнал операций (`write-ahead log`, `WAL`) для восстановления после перезапуска: каждая операция `put` записывается в файл `wal.log`.
3. **Архитектура master и slave**
 - 3.1. Реализуйте два скрипта: `master.py` и `slave.py`.
 - 3.2. Master запускает TCP-сервер на порту 8080, принимает команды `PUT|key|value` и `GET|key`.
 - 3.3. Slave подключается к master по TCP и ожидает команд репликации (например, `REPLICATE|key|value`).
 - 3.4. Master также запускает фоновый поток/сокет для отправки реплик.
4. **Реализация синхронного режима**
 - 4.1. При получении `PUT` мастер сначала применяет операцию локально.
 - 4.2. Затем мастер отправляет команду репликации каждому подписанному slave.
 - 4.3. Мастер ожидает подтверждение (ACK) от каждого slave с таймаутом (например, 0.5 секунды).
 - 4.4. Если все подтверждения получены в течение таймаута, мастер отправляет клиенту ОК. Если хотя бы один slave не ответил – возвращает ERROR (или TIMEOUT).
 - 4.5. В случае ошибки клиент может повторить запрос (`idempotency` не обязательна).
5. **Реализация асинхронного режима**
 - 5.1. При получении `PUT` мастер сразу применяет операцию локально и отвечает клиенту ОК.

5.2. В фоновом потоке мастер отправляет неприменённые операции репликам (например, каждые 0.1 секунды или по накоплении пачки).

5.3. Для упрощения можно использовать очередь (queue.Queue) для отложенной отправки.

5.4. Slave получает и применяет операции без подтверждения (fire-and-forget).

6. Измерение задержек записи

6.1. Запустите master и один slave на одной машине (разные порты). Для эмуляции сетевой задержки на slave добавьте `time.sleep(0.02)` перед отправкой АСК.

6.2. Напишите клиентский скрипт, который выполняет $N = 1000$ операций PUT с случайными ключами и значениями.

6.3. Измеряйте время каждого запроса на стороне клиента. Вычислите среднее, p50, p90, p99.

6.4. Повторите для синхронного и асинхронного режимов.

6.5. Постройте гистограммы и коробочные диаграммы задержек.

7. Тест на потерю данных (durability)

7.1. Запустите master и slave в синхронном режиме. Выполните 500 операций PUT. Внезапно остановите master (например, `kill -9` или `Ctrl+C`). Убедитесь, что все записи присутствуют на slave (через прямое чтение или отдельный тестовый клиент).

7.2. Повторите то же самое для асинхронного режима. Остановите master сразу после записи последних 100 операций, не дожидаясь фоновой репликации. Затем проверьте, сколько операций сохранено на slave. Вычислите процент потерянных данных.

7.3. В асинхронном режиме измените интервал репликации (например, 0.5 сек, 2 сек) и запишите изменение доли потерь.

8. Сравнительный анализ

8.1. Сведите результаты в таблицу: режим, p99 latency, процент потерь при падении master.

8.2. Объясните, как выбор режима влияет на компромисс между производительностью и надёжностью.

8.3. Постройте график зависимости доли потерь от задержки репликации.

Пример выполнения задания

Фрагмент master.py (синхронный режим, упрощённо):

```
import socket, threading, time, json

class Master:
    def __init__(self, slaves):
        self.data = {}
        self.slaves = slaves # список TCP-сокетов реплик

    def _replicate_sync(self, key, value):
        cmd = f'REPLICATE|{key}|{value}'
        for sock in self.slaves:
            sock.send(cmd.encode())
            ack = sock.recv(64) # ожидание "ACK"
            if ack != b'ACK':
                raise Exception("slave ack failed")

    def handle_put(self, key, value, client_sock):
        self.data[key] = value
        try:
            self._replicate_sync(key, value)
            client_sock.send(b'OK')
        except Exception:
            client_sock.send(b'REPLICA_ERROR')
```

Результаты тестов:

- Синхронный режим: p99 latency = 45 мс (при искусственной задержке на slave 20 мс), потерь данных 0%.
- Асинхронный режим: p99 latency = 8 мс, при падении master потеряно 12% последних записей (те, которые не успели реплицироваться).

Индивидуальные задания

1. Реализовать репликацию с двумя slave-узлами. В синхронном режиме ждать подтверждения только от одного из двух (квитум 1 из 2). Сравнить с ожиданием от обоих.

2. Добавить возможность чтения из slave (read-replica). Реализовать стратегию балансировки чтения (round-robin по всем репликам). Измерить задержки чтения.

3. В асинхронном режиме реализовать пакетную репликацию (batch): отправлять сразу 10 операций или по таймеру. Измерить, как меняются задержка записи и процент потерь.

4. Эмулировать задержку сети на уровне 50 мс и 100 мс. Построить график p99 latency в синхронном режиме от величины задержки.

5. Реализовать восстановление мастера из журнала (WAL) и синхронизацию с slave после перезапуска.

6. Добавить идентификаторы операций (sequence number) и проверять, не пропустил ли slave какие-либо операции.

7. Сравнить асинхронную репликацию с использованием отдельного потока против использования очереди и пула потоков. Измерить максимальный throughput.

8. Реализовать автоматический выбор синхронного / асинхронного режима в зависимости от текущей загрузки (например, если задержка превышает порог, переключиться на асинхронный).

9. Добавить шифрование канала между master и slave (например, TLS-подобное вручную). Оценить накладные расходы на задержку.

10. Запустить master и slave на разных хостах (в виртуальных машинах или Docker) и повторить измерения реальной сетевой задержки.

11. Реализовать репликацию на основе push (master отправляет) и pull (slave запрашивает). Сравнить по сложности и надёжности.

12. Сымитировать отказ одной из реплик во время синхронной записи (обрыв соединения). Реализовать стратегию повторной отправки (retry) и таймауты.

13. Добавить метрики (prometheus endpoints) на master для сбора количества успешных/неуспешных репликаций, задержек.

14. Реализовать поддержку нескольких таблиц (ключ-значение с префиксами) и репликацию с фильтрацией (только определённые префиксы).

15. Написать юнит-тесты для модулей (используя unittest/pytest) и измерить покрытие тестами.

Требования к отчёту

Отчёт должен содержать следующие разделы:

- титульный лист с названием работы, дисциплиной, ФИО, группой;
- цель работы и задачи;
- теоретическая часть – описание master-slave репликации, синхронного и асинхронного режимов, связь с CAP-теоремой;
- раздел «Проектирование и реализация» – описание архитектуры, классов, протокола взаимодействия, листинги ключевых участков кода;
- раздел «Эксперименты» – описание условий (количество операций, задержки, используемое оборудование), таблицы с результатами замеров, гистограммы и графики;
- раздел «Анализ потери данных» – результаты тестов на durability, объяснение потерь;
- сравнительная таблица синхронного и асинхронного режимов по 3-4 параметрам;

- выводы – какие режимы когда применять, как найденный компромисс соответствует требованиям highload;
- ответы на контрольные вопросы.

Отчёт оформляется в PDF, все графики подписаны, код отформатирован (с использованием моноширинного шрифта).

Контрольные вопросы

1. Что такое репликация master-slave? В чём её преимущества и недостатки?
2. В чём разница между синхронной и асинхронной репликацией с точки зрения клиентской задержки?
3. Как синхронная репликация влияет на доступность системы?
4. Какие данные могут быть потеряны при падении master в асинхронном режиме?
5. Что такое write-ahead log (WAL) и зачем он нужен?
6. Может ли система с синхронной репликацией гарантировать нулевую потерю данных? Какие условия для этого нужны?
7. Как связаны между собой CAP-теорема, синхронная и асинхронная репликация?
8. Что произойдёт при синхронной репликации, если один из slave временно недоступен? Как это правильно обработать?
9. Каким образом можно улучшить производительность синхронной репликации без потери гарантий?
10. Почему говорят, что асинхронная репликация обеспечивает «eventual consistency»?
11. Как выбрать оптимальный интервал пакетной отправки в асинхронном режиме?
12. Какие метрики наблюдаемости (latency, error rate, replication lag) вы бы внедрили для мониторинга такого кластера?

Лабораторная работа №4.

Шардирование по range и hash (симуляция)

Цель работы: реализовать две стратегии шардирования (range-based и hash-based) и проанализировать их поведение при добавлении новых шардов, а также исследовать проблему «горячего ключа» (skew) и объём перемещаемых данных при ребалансировке.

Задачи:

- реализовать симулятор key-value хранилища с поддержкой range-шардирования и hash-шардирования;
- реализовать операцию добавления нового шарда (ребалансировку) для обеих стратегий;
- измерить долю ключей, перемещённых между шардами при добавлении шарда;
- создать сценарий с «горячим ключом» (один ключ получает 50% запросов) и оценить влияние на распределение нагрузки;
- сравнить стратегии по равномерности распределения, объёму перемещений и устойчивости к skew.

Теоретическая часть

Шардирование (горизонтальное партиционирование) – это метод масштабирования базы данных или хранилища, при котором данные распределяются по нескольким узлам (шардам). Каждый шард хранит только часть данных. Существуют две основные стратегии определения того, какой шард отвечает за данный ключ.

Range-шардирование – диапазоны значений ключей распределяются между шардами. Например, шард 1: ключи [A–F), шард 2: [F–K) и т.д.

- *Достоинства:* эффективно для диапазонных запросов (SELECT * WHERE key BETWEEN ...); простота реализации.

- *Недостатки:* неравномерность распределения (если данные не равномерны); проблема «горячих» диапазонов; сложная перебалансировка (нужно резать диапазоны).
- *Перебалансировка:* при добавлении шарда «разрезается» один из существующих диапазонов, данные перемещаются только из одного шарда.

Hash-шардирование – вычисляется хеш от ключа, и шард определяется как $\text{hash}(\text{key}) \% N$ (или consistent hashing).

- *Достоинства:* отличная равномерность при хорошей хеш-функции; простота маршрутизации.
- *Недостатки:* неэффективно для диапазонных запросов; при изменении количества шардов почти все ключи перехешируются (если не используется consistent hashing).
- *Перебалансировка:* при добавлении шарда меняется $\% N$, большинство ключей перемещается. Проблема решается consistent hashing (см. ЛР 2).

Метрики для сравнения:

- *Равномерность:* коэффициент вариации числа ключей на шард.
- *Объём перемещений:* доля ключей, меняющих шард при добавлении нового.
- *Skew (перекос):* влияние одного популярного ключа на нагрузку шарда.

Проблема «горячего ключа»: если один ключ запрашивается очень часто (например, профиль известного пользователя), то весь трафик идёт в один шард. Hash-шардирование не решает эту проблему (один ключ даёт один хеш), range-шардирование также не спасает. Решения: репликация горячего ключа, кэширование, специальные алгоритмы.

Методика выполнения работы

1. Подготовка проекта

- 1.1. Создайте каталог lab4_sharding.
- 1.2. Язык – Python (рекомендуется) или Java.

- 1.3. Установите matplotlib, numpy, pandas для анализа и графиков.
2. **Базовые классы для шардов и хранилища**
- 2.1. Реализуйте класс Shard, который содержит:
- словарь data: dict (key → value);
 - методы put(key, value), get(key), size(), keys().
- 2.2. Реализуйте абстрактный класс ShardingStrategy с методами get_shard_index(key), add_shard(), rebalance(), get_shard_data().
3. **Реализация range-шардирования**
- 3.1. Создайте класс RangeSharding. В конструкторе передаётся список границ диапазонов: boundaries = [("A", "F"), ("F", "K"), ("K", "Z")].
- 3.2. Метод get_shard_index(key) сравнивает ключ (строковый) с границами и возвращает индекс.
- 3.3. При добавлении нового шарда: выберите самый большой шард (по количеству ключей или по размеру диапазона), разделите его диапазон пополам (по медиане ключей или по середине алфавита). Переместите половину ключей в новый шард.
- 3.4. Реализуйте rebalance() для выполнения перебалансировки (после добавления шарда). Важно: перемещаются ключи только из одного старого шарда.
4. **Реализация hash-шардирования**
- 4.1. Создайте класс HashSharding (без consistent hashing – для наглядности проблемы).
- 4.2. Используйте hash(key) % num_shards (хеш-функция – встроенная hash() или hashlib.md5).
- 4.3. При добавлении нового шарда (увеличении num_shards на 1) все ключи перераспределяются. Реализуйте rebalance(), который создаёт новые шарды и полностью перекладывает все данные.
- 4.4. В методе add_shard() обновите num_shards и вызовите rebalance().
5. **Реализация симулятора**

- 5.1. Напишите класс ShardedKV, который принимает стратегию шардирования.
- 5.2. Методы: put(key, value), get(key), add_shard() (добавляет новый пустой шард и вызывает перебалансировку стратегии).
- 5.3. Для измерения статистики предоставьте методы get_shard_sizes() и get_all_keys().
6. **Генерация тестовых данных**
 - 6.1. Сгенерируйте $N = 5000$ ключей вида `f"user_{i}"` (равномерное распределение).
 - 6.2. Для skew-теста: 80% ключей – обычные, 20% – один «горячий» ключ `"hot_key"`, который будет повторяться многие разы.
 - 6.3. Для распределённого теста: 5000 уникальных ключей.
7. **Тестирование равномерности распределения**
 - 7.1. Инициализируйте 3 шарда. Заполните 5000 ключей.
 - 7.2. Для range-шардирования вычислите количество ключей на шард. Выведите таблицу. Нанесите на столбчатую диаграмму. Вычислите коэффициент вариации.
 - 7.3. Для hash-шардирования сделайте то же самое.
 - 7.4. Сравните CV для обеих стратегий.
8. **Измерение объёма перемещаемых данных при добавлении шарда**
 - 8.1. Начальное состояние: 3 шарда, 5000 уникальных ключей. Запомните, какой шард хранит каждый ключ.
 - 8.2. Добавьте четвёртый шард (выполните перебалансировку).
 - 8.3. Для каждого ключа проверьте, изменился ли шард. Подсчитайте долю перемещённых ключей.
 - 8.4. Повторите для обеих стратегий. Постройте сравнительную диаграмму.
9. **Моделирование горячего ключа (skew)**

9.1. Создайте набор данных: 1000 уникальных ключей + один ключ "hot_key", повторённый 2000 раз (т.е. всего 3000 записей, но 2000 из них – один и тот же ключ).

9.2. Заполните хранилище (3 шарда). Определите, на каком шарде оказался горячий ключ.

9.3. Этот шард будет содержать непропорционально большое количество записей (2000 + некоторое количество уникальных).

9.4. Зафиксируйте разницу в размерах шардов. Вычислите максимальный размер шарда / средний размер.

9.5. Для hash-шардирования проверьте, будет ли hot_key всегда попадать в тот же шард. Для range-шардирования – как выбор диапазонов влияет.

9.6. Предложите и реализуйте одно из решений: выделить горячий ключ в отдельный шард или создать его копии на всех шардах с последующей агрегацией.

10. Анализ результатов

10.1. Сведите в таблицу все метрики (равномерность, перемещения, skew-коэффициент).

10.2. Постройте графики зависимости доли перемещений от числа добавляемых шардов (добавить 1, 2, 3 шарда последовательно).

10.3. Напишите вывод о том, в каких сценариях какая стратегия предпочтительнее.

Пример выполнения задания

Фрагмент range-шардирования с разделением диапазона:

```
class RangeSharding:
```

```
    def __init__(self, boundaries):
```

```
        # boundaries: list of (start, end) inclusive start, exclusive end
```

```
        self.boundaries = boundaries
```

```
        self.num_shards = len(boundaries)
```

```

def get_shard_index(self, key):
    for i, (start, end) in enumerate(self.boundaries):
        if start <= key < end:
            return i
    return self.num_shards - 1 #fallback

def add_shard(self, shard_idx_to_split):
    #split the range of shard_idx_to_split into two halves
    old_start, old_end = self.boundaries[shard_idx_to_split]
    mid = self._find_midpoint(old_start, old_end)
    new_boundary1 = (old_start, mid)
    new_boundary2 = (mid, old_end)
    self.boundaries[shard_idx_to_split] = new_boundary1
    self.boundaries.insert(shard_idx_to_split + 1, new_boundary2)
    self.num_shards += 1
    return shard_idx_to_split, shard_idx_to_split + 1 #indexes of affected
shards

```

Результаты экспериментов:

- *Равномерность (CV):* hash = 0.08, range = 0.35 (при алфавитном распределении ключей).
- **Перемещения при +1 шард*:* hash = 75% ключей, range = 25% ключей.
- *Горячий ключ:* hash – один шард перегружен в 3 раза; range – можно разместить горячий ключ на отдельной границе.

Индивидуальные задания

1. Реализовать range-шардирование с динамическим выбором границ на основе распределения ключей (по медиане). Сравнить с фиксированными границами.
2. Добавить поддержку взвешенных шардов (разные узлы имеют разную ёмкость). Адаптировать перебалансировку.

3. Реализовать hash-шардирование с использованием **consistent hashing** (как в ЛР 2) и сравнить объём перемещений с простым % N.

4. Сымитировать последовательное добавление 5 шардов (от 3 до 8) и построить график накопленного объёма перемещённых данных для каждой стратегии.

5. Реализовать «горячий ключ» с частотой доступа 90% (т.е. 90% запросов к одному ключу). Предложить и реализовать стратегию кэширования на клиенте.

6. Добавить возможность диапазонных запросов (например, `get_range(start, end)`). Оценить эффективность range и hash стратегий для такой операции.

7. Реализовать мониторинг и экспорт метрик (Prometheus-стиль) – количество запросов на шард, размер шарда.

8. Сымитировать отказ одного шарда: перераспределить его данные между оставшимися и затем восстановить. Сравнить усилия для range и hash.

9. Использовать реальные данные (например, имена городов или логи пользователей из датасета). Оценить skew на реальных данных.

10. Реализовать алгоритм **consistent hash with virtual nodes** и сравнить с обычным hash-шардированием в условиях добавления/удаления шардов.

11. Исследовать производительность (время перебалансировки) при перемещении 10 000, 100 000 ключей.

12. Для range-шардирования реализовать автоматическое разделение «горячего» диапазона (если один диапазон получает > 40% трафика).

13. Реализовать стратегию «shard-selection» на основе двухуровневого хеша (сначала определить hot-key group).

14. Написать интеграционный тест с использованием Docker: поднять 3 контейнера с хранилищами, симулировать шардирование через прокси.

15. Добавить сжатие данных перед перемещением при перебалансировке и измерить экономию времени.

Требования к отчёту

Отчёт должен быть оформлен в электронном виде (PDF) и содержать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание range и hash шардирования, проблемы горячего ключа, метрики равномерности;
- раздел «Архитектура и реализация» – диаграмма классов, описание ключевых алгоритмов, листинги кода;
- раздел «Эксперименты»:
 - результаты равномерности (таблицы, CV);
 - объёмы перемещений (гистограммы);
 - моделирование горячего ключа (распределение нагрузки);
 - графики зависимости перемещений от числа шардов;
- сравнительный анализ – таблица «характеристики стратегий»;
- выводы – какая стратегия для каких условий подходит;
- ответы на контрольные вопросы.

Все визуализации подписаны, код должен быть читаем, использовано моноширинное форматирование. Отчёт сдаётся в виде файла lab4_report.pdf и исходных кодов в архиве.

Контрольные вопросы

1. Чем принципиально отличается range-шардирование от hash-шардирования?
2. Как hash-шардирование влияет на возможность выполнения диапазонных запросов?
3. Почему при добавлении нового шарда в простом hash-шардировании (% N) перемещается почти все данные?
4. Как можно уменьшить объём перемещений в hash-шардировании? (Подсказка: consistent hashing)

5. Что такое «горячий ключ» и почему он является проблемой для обеих стратегий?
6. Какие методы борьбы с горячим ключом вы можете предложить?
7. Как вычислить коэффициент вариации и что он показывает?
8. Приведите пример, когда range-шардирование лучше hash-шардирования, и наоборот.
9. Как бы вы реализовали автоматическую перебалансировку в range-шардировании без остановки системы?
10. Какое отношение имеет шардирование к CAP-теореме?
11. Какие метрики нужно мониторить в реальной системе, чтобы обнаружить неравномерность шардирования?
12. Как повлияет выбор хеш-функции на равномерность hash-шардирования? Можно ли использовать $\text{hash}(\text{key}) \% N$ на строковых ключах?

Лабораторная работа №5.

Saga (хореография) через очереди сообщений

Цель работы: реализовать распределённую транзакцию по паттерну Saga в стиле хореографии на двух взаимодействующих сервисах (заказ и оплата) с использованием очереди сообщений, а также продемонстрировать механизм компенсирующих транзакций при сбое.

Задачи:

- создать два микросервиса: Service A (обработка заказов) и Service B (обработка платежей);
- организовать асинхронное взаимодействие через очередь сообщений (RabbitMQ, Kafka или in-memory реализация);
- реализовать основную транзакцию: создание заказа → запрос на оплату;
- реализовать компенсирующую транзакцию: отмена заказа при неудачной оплате;
- симитировать сбой в сервисе оплаты и продемонстрировать автоматический откат через систему событий;
- проанализировать гарантии согласованности и сложность реализации по сравнению с распределённой двухфазной фиксацией (2PC).

Теоретическая часть

В распределённых системах, особенно в микросервисной архитектуре, классические ACID-транзакции с двухфазной фиксацией (2PC) часто неприменимы из-за требования высокой доступности, низкой задержки и устойчивости к сетевым разделениям (CAP-теорема). Паттерн **Saga** (Сага) предлагает способ поддержания согласованности данных между несколькими сервисами с помощью последовательности локальных транзакций, каждая из которых может быть скомпенсирована в случае сбоя.

Выделяют два основных способа реализации Saga:

- **Хореография (Choreography):** каждый сервис публикует события, на которые подписываются другие сервисы. Координация распределённая, без центрального оркестратора.
- **Оркестрация (Orchestration):** центральный координатор (оркестратор) явно вызывает сервисы и управляет ходом транзакции.

В данной работе используется **хореография**, так как она в большей степени соответствует принципам асинхронных высоконагруженных систем, снижает связанность и лучше масштабируется.

Сценарий Saga (заказ-оплата):

1. Сервис заказов получает запрос на создание заказа.
2. Он создаёт заказ в статусе «в ожидании оплаты» и публикует событие OrderCreated.
3. Сервис платежей подписан на OrderCreated, обрабатывает его, пытается списать средства.
4. Если списание успешно – публикуется событие PaymentSuccess, сервис заказов переводит заказ в статус «подтверждён».
5. Если списание не удалось – публикуется событие PaymentFailed, сервис заказов запускает компенсирующую транзакцию: отменяет заказ (статус «отменён»).

Компенсирующая транзакция – операция, логически отменяющая эффект предыдущей локальной транзакции. В отличие от отката БД, она не возвращает состояние «как было», а выполняет бизнес-компенсацию (например, возврат средств, отмена заказа, освобождение ресурсов).

Важные свойства: **идемпотентность** – одно и то же событие не должно привести к повторным нежелательным эффектам; **eventual consistency** – система придёт к согласованному состоянию за конечное время.

Для целей лабораторной работы можно использовать упрощённую in-memory очередь (на основе queue.Queue и потоков) или лёгкую реализацию с Redis Pub/Sub, но рекомендуется RabbitMQ для приобретения практического опыта.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог lab5_saga.
- 1.2. Установите и запустите RabbitMQ (через Docker):
`docker run -d --name rabbitmq -p 5672:5672 rabbitmq:3-management`
(или используйте in-memory очередь для упрощения).
- 1.3. Установите библиотеку для работы с RabbitMQ (Python: pika, Java: amqp-client, Go: amqp091-go).
- 1.4. Установите pytest и requests (если будете добавлять HTTP-интерфейсы).

2. Реализация сервиса заказов (Service A)

- 2.1. Создайте класс OrderService. Он будет содержать:
 - базу данных (в памяти – словарь) для хранения заказов: order_id → {"status": str, "amount": float}.
 - методы: create_order(order_id, amount), confirm_order(order_id), cancel_order(order_id).
- 2.2. Организуйте соединение с очередью. Создайте обменник (exchange) saga типа topic или direct.
- 2.3. Подпишитесь на очереди:
 - payment_success – при получении события изменить статус заказа на confirmed.
 - payment_failed – при получении события изменить статус на cancelled (компенсация).
- 2.4. При вызове HTTP-эндпоинта /orders (POST) сервис создаёт заказ со статусом pending и публикует событие order.created в очередь с параметрами order_id, amount.

3. Реализация сервиса платежей (Service B)

- 3.1. Создайте класс PaymentService. Содержит:
 - имитацию счёта пользователя или просто логику успеха/неудачи.

- метод `process_payment(order_id, amount)`, возвращает `True` (успех) или `False` (ошибка).

3.2. Подпишитесь на очередь `order.created`. При получении события:

- попытаться выполнить платёж (с возможностью случайного сбоя).
- при успехе: опубликовать событие `payment.success`.
- при неудаче: опубликовать событие `payment.failed`.

3.3. Для симуляции сбоя заложите вероятность ошибки 30% (или флаг вручную).

3.4. Убедитесь, что обработка сообщений идемпотентна (по `order_id` можно игнорировать повторные запросы).

4. Интеграция очереди сообщений (пример на Python + Pika)

4.1. Настройте соединение и каналы.

4.2. Функция публикации:

```
channel.basic_publish(exchange='saga', routing_key='order.created',  
body=json.dumps(data))
```

4.3. Функция обработки:

```
def callback(ch, method, properties, body):
```

```
    event = json.loads(body)
```

```
    if method.routing_key == 'order.created':
```

```
        # обработать создание заказа (платёж)
```

4.4. Запустите сервисы в отдельных процессах (или потоках), чтобы они могли одновременно получать сообщения.

5. Написание клиента и запуск сценария

5.1. Создайте скрипт `client.py`, который отправляет POST-запрос на `http://localhost:8081/orders` (сервис заказов) с JSON `{"order_id": "123", "amount": 100}`.

5.2. После получения ответа (синхронного) от сервиса заказов (например, `{"status": "pending"}`) подождите 2 секунды и проверьте окончательный статус заказа через `GET /orders/123`.

5.3. Запустите сначала оба сервиса, затем клиент. Пронаблюдайте поток событий в логах.

6. Симуляция сбоя и отката

6.1. Намеренно заставьте сервис платежей вернуть ошибку (например, `amount > 1000` или флаг `force_fail=True`).

6.2. Иницируйте создание заказа с суммой, вызывающей ошибку.

6.3. Убедитесь, что сервис заказов получает `payment.failed` и выполняет компенсацию: переводит заказ в статус `cancelled`.

6.4. Проверьте, что компенсация не пытается повторно выполнить платёж.

7. Измерение и логирование

7.1. Добавьте логирование времени обработки событий.

7.2. Постройте диаграмму последовательности событий (sequence diagram) по логам.

7.3. Оцените время, через которое система приходит к согласованному состоянию после сбоя (eventual consistency lag).

8. Расширение (опционально) – оркестрация

8.1. Для сравнения можно реализовать вариант с оркестратором, который явно вызывает сервисы.

8.2. Сравните сложность, связанность и отслеживаемость двух подходов.

Пример выполнения задания

Логи успешного сценария (хореография):

[OrderService] Received POST /orders: order=456, amount=50

[OrderService] Created order 456 with status pending

[OrderService] Published event order.created

[PaymentService] Received order.created: order=456, amount=50

[PaymentService] Payment success for order 456

[PaymentService] Published event payment.success

[OrderService] Received payment.success for order 456

[OrderService] Confirmed order 456 -> status confirmed

Логи сценария с откатом (компенсация):

[OrderService] Created order 789 with status pending

[PaymentService] Payment FAILED for order 789 (insufficient funds)

[PaymentService] Published event payment.failed

[OrderService] Received payment.failed for order 789

[OrderService] Cancelled order 789 -> status cancelled

Клиент, проверив статус через 2 секунды, получит cancelled.

Индивидуальные задания

1. Добавить третий сервис – «Сервис инвентаризации» (резервирование товара). Реализовать Saga из трёх шагов с компенсацией: заказ → оплата → резерв.
2. Реализовать версию с оркестратором (центральный координатор) и сравнить с хореографией по количеству сообщений и сложности отладки.
3. В сервисе платежей добавить имитацию частичного сбоя (платёж прошёл, но сервис упал перед публикацией payment.success). Использовать подтверждения (ack) очереди, чтобы гарантировать доставку.
4. Реализовать идемпотентность для событий: если сервис заказов получит два одинаковых payment.success, не должно быть двойного подтверждения.
5. Добавить таймауты и механизм повторной отправки (retry) для событий, которые не были обработаны.
6. Использовать Kafka вместо RabbitMQ (с партициями и группами потребителей). Показать преимущества для высокой нагрузки.
7. Реализовать хранение состояния каждого шага Saga в отдельной таблице (журнал саги) для возможности восстановления после перезапуска сервиса.

8. Добавить компенсирующую транзакцию для случая, когда после успешной оплаты не удаётся подтвердить заказ (редкий сценарий).
9. Написать интеграционный тест с использованием testcontainers (RabbitMQ) и pytest: проверить успешный и неуспешный сценарии.
10. Измерить пропускную способность (событий/сек) при 100 параллельных заказах и задержку до конечной согласованности.
11. Реализовать UI на Flask/FastAPI для мониторинга заказов и событий (дашборд).
12. В сервисе заказов добавить возможность ручного отката заказа (компенсация) через административный запрос.
13. Заменить очередь на in-memory шину (например, asyncio.Queue для двух сервисов в одном процессе) – оценить упрощение, но потерю масштабируемости.
14. Реализовать версию с долгоживущими Saga: заказ создаётся, оплата может идти несколько дней (сохранять контекст в БД) – запланированная отложенная обработка.
15. Добавить метрики (количество успешных/неуспешных саг, среднее время завершения) и экспорт в Prometheus.

Требования к отчёту

Отчёт должен быть оформлен в формате PDF и содержать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание паттерна Saga, хореографии vs оркестрации, компенсирующих транзакций, eventual consistency;
- раздел «Архитектура решения» – диаграмма компонентов, взаимодействие через очередь, описание очередей и событий;
- раздел «Реализация» – листинги кода с комментариями (основные части: публикация событий, обработка, компенсация);

- раздел «Тестирование» – описание тестовых сценариев, логи успешного и аварийного выполнения, скриншоты или выдержки из логов;
- раздел «Анализ» – плюсы и минусы подхода, наблюдения о доставке сообщений, задержках;
- выводы – достигнуты ли цели, как паттерн Saga помогает в высоконагруженных системах;
- ответы на контрольные вопросы.

Код должен быть чистым, логгирование информативным. Прилагается архив с исходными кодами.

Контрольные вопросы

1. В чём разница между ACID и BASE транзакциями?
2. Что такое Saga и для решения каких проблем она используется?
3. Чем отличается хореография от оркестрации в контексте Saga?
4. Какое свойство транзакций (A, C, I, D) в классической БД отсутствует в Saga? Чем это компенсируется?
5. Что такое компенсирующая транзакция? Приведите пример.
6. Почему в Saga не используется двухфазная фиксация (2PC) в микросервисах?
7. Как гарантировать идемпотентность обработки событий в сервисах?
8. Как очередь сообщений влияет на общую доступность системы?
9. Что может пойти не так при публикации события payment.success (например, сетевой сбой после списания денег)? Как спроектировать решение для такого случая?
10. Какие метрики нужны для мониторинга состояния Saga в production?
11. Как увеличить масштабируемость системы на основе хореографии?
12. Когда паттерн Saga не подходит и лучше использовать другие подходы?

Лабораторная работа №6.

Raft: лидер и репликация лога (упрощённая реализация)

Цель работы: реализовать упрощённую версию алгоритма консенсуса Raft на двух-трёх узлах, демонстрирующую выбор лидера, репликацию лога команд и автоматическое продолжение работы при падении лидера.

Задачи:

- создать несколько узлов (процессов или потоков), каждый из которых реализует конечный автомат Raft-узла (состояния Follower, Candidate, Leader);
- реализовать механизм выборов лидера с использованием таймаутов (election timeout) и голосования (RequestVote RPC);
- реализовать репликацию команд от лидера к последователям (AppendEntries RPC) с ведением лога;
- применить логические часы (term) и номера индексов записей в логе;
- проверить, что при внезапном отказе лидера система выбирает нового лидера, который продолжает принимать новые команды;
- проанализировать гарантии консенсуса (безопасность, живучесть) на упрощённой модели.

Теоретическая часть

В распределённых системах, особенно в кластерах, управляющих реплицированным конечным автоматом (например, etcd, Consul, ZooKeeper), требуется единое мнение узлов о порядке команд. Алгоритм **Raft** (Diego Ongaro, 2014) спроектирован как более понятная альтернатива Paxos. Он разделяет проблему консенсуса на три независимые подзадачи: выбор лидера, репликация лога, безопасность.

Основные понятия Raft:

- **Term (срок)** – монотонно возрастающее целое число; каждый термин начинается с выборов.

- **Состояния узла:**

- *Follower* – подчиняется лидеру, пассивен;
- *Candidate* – инициирует выборы;
- *Leader* – принимает все клиентские запросы, управляет репликацией.

- **Таймауты выборов** – случайные величины (например, 150–300 мс), запускающие переход в *Candidate* при отсутствии сообщений от лидера.
- **RequestVote RPC** – запрос голоса, отправляемый кандидатом другим узлам.
- **AppendEntries RPC** – сообщение от лидера для репликации записей лога (или heartbeat).

Репликация лога (упрощённо):

- Лидер получает команду от клиента, добавляет её в свой лог как новую запись (с индексом и текущим term).
- Лидер параллельно отправляет **AppendEntries** всем последователям.
- Когда запись получает подтверждение от большинства узлов (включая лидера), она считается *commit* и применяется к конечному автомату.
- В нашей упрощённой реализации конечный автомат – просто применение команды к состоянию (например, key-value хранилище).

Гарантии (безопасность):

- В каждом термине может быть только один лидер.
- Узлы с отстающим логом не могут стать лидером (правило выборов: кандидат должен иметь лог не менее актуальный, чем у большинства).

Для простоты работы мы опустим сложные аспекты (установка соответствия логов, оптимизация heartbeat и перезапись), но сохраним ключевую механику – таймауты, голосование, репликация и смену лидера.

Методика выполнения работы

1. Подготовка проекта

- 1.1. Создайте каталог lab6_raft.

- 1.2. Язык – Python (с `threading`, `socket` или `asyncio`) или Go (горутины).
- 1.3. Для взаимодействия используйте TCP-сокеты (можно `localhost` с разными портами) или HTTP/REST (упростит отладку).
- 1.4. Для генерации случайных таймаутов используйте `random.uniform(0.15, 0.3)`.

2. Реализация базовой структуры узла

2.1. Создайте класс `RaftNode` с атрибутами:

- `id`: уникальный идентификатор (0, 1, 2)
- `peers`: список адресов (портов) других узлов
- `current_term`: целое, начинается с 0
- `voted_for`: кому отдан голос в этом термине (или `None`)
- `log`: список записей, каждая запись – `{"term": term, "command": cmd}`
- `commit_index`: индекс последней подтверждённой (применённой) записи
- `state`: "follower", "candidate", "leader"
- `election_timeout`: таймер (рандомное значение); при истечении – переход в `Candidate`
- `heartbeat_interval`: для лидера – 0.05 сек (50 мс)

2.2. Добавьте поток (или `asyncio.Task`) для прослушивания входящих RPC.

2.3. Добавьте таймер (можно через `threading.Timer` или `time.sleep` в отдельном потоке), который при истечении пробуждает узел и запускает выборы.

3. Реализация RPC-вызовов (на сокетах/HTTP)

3.1. Определите два типа сообщений:

- `RequestVote` (аргументы: `term`, `candidateId`, `lastLogIndex`, `lastLogTerm`)
- `AppendEntries` (аргументы: `term`, `leaderId`, `prevLogIndex`, `prevLogTerm`, `entries[]`, `leaderCommit`)

3.2. Напишите обработчики на узле для каждого типа RPC.

3.3. Для упрощения можно использовать JSON и HTTP (библиотека Flask/FastAPI), либо слушать UDP.

Пример HTTP-обработчика (FastAPI):

```
@app.post("/request_vote")
```

```
async def request_vote(req: RequestVoteRequest):
```

```
    term_granted = node.handle_request_vote(req)
```

```
    return {"term": node.current_term, "vote_granted": term_granted}
```

4. Логика выборов лидера

4.1. Когда таймаут истекает и узел не получал heartbeat от лидера, он переходит в состояние candidate:

- увеличивает current_term на 1
- голосует за себя (voted_for = self.id)
- отправляет RequestVote всем остальным узлам
- ожидает ответы; если получает голоса от большинства ($> N/2$) -> становится лидером

4.2. При получении RequestVote от кандидата с большим term: follower обновляет свой term и, если ещё не голосовал в этом terme и лог кандидата не менее актуален, отдаёт голос.

4.3. Если пришёл AppendEntries или RequestVote с termом выше текущего, узел переходит в состояние follower.

5. Репликация лога (лидер → последователи)

5.1. После становления лидером, он немедленно отправляет AppendEntries (heartbeat) всем последователям с пустым entries[].

5.2. При получении клиентской команды (например, SET key value), лидер:

- добавляет запись в свой лог (index = len(log) + 1, term = current_term)

- отправляет AppendEntries всем последователям, содержащий эту запись.

5.3. Последователь принимает запись, если prevLogIndex и prevLogTerm соответствуют последней записи его лога. Применяет запись и отвечает успехом.

5.4. Лидер подсчитывает число узлов, сохранивших запись. Как только успешный ответ получен от большинства (включая лидера), лидер увеличивает commit_index и в следующем heartbeat сообщает последователям leaderCommit.

5.5. Узлы применяют закоммиченные записи к конечному автомату (простой словарь).

6. Тестирование:

6.1. Запустите 3 узла на разных портах (5001, 5002, 5003). Настройте таймауты выборов 200–400 мс.

6.2. В течение нескольких секунд должен избраться лидер (наблюдайте логи).

6.3. Отправьте через API лидера несколько команд SET. Проверьте, что все узлы получили и применили одинаковые команды.

6.4. Остановите лидера (завершите процесс). Оставшиеся узлы через таймаут должны переизбрать нового лидера. Отправьте ещё команду – она должна успешно реплицироваться.

6.5. Восстановите остановленный узел – он должен догнать лог от текущего лидера (синхронизация).

7. Логирование и метрики

7.1. Добавьте подробное логирование каждого шага (переходы состояний, получение RPC, добавление записей).

7.2. Постройте временную диаграмму работы кластера (например, в виде таблицы событий).

Пример выполнения задания

Вывод логов при старте 3 узлов:

Node 0: state FOLLOWER, term 0, timeout 0.23s

Node 1: state FOLLOWER, term 0, timeout 0.18s

Node 2: state FOLLOWER, term 0, timeout 0.31s

Node 1: election timeout -> become CANDIDATE, term 1

Node 1: votes received: 1 (self), requesting from node 0,2

Node 0: vote granted for candidate 1 (term 1)

Node 2: vote granted for candidate 1 (term 1)

Node 1: became LEADER for term 1

Node 1: send heartbeat to all

Node 0: received AppendEntries, leader is 1

Node 2: received AppendEntries, leader is 1

[Client] SET x=10 via leader 1

Leader 1: log append [index1 term1 SET x=10]; replicate to followers

Leader 1: commit index=1, apply to state machine

All nodes: state now {x:10}

При падении лидера 1:

Node 0: no heartbeat from leader > timeout, term 1 -> becomes CANDIDATE,
term 2

Node 0: wins election with votes from node2, becomes LEADER term2

Node 2: follows new leader 0

Client: SET y=20 -> accepted by leader0, replicated.

Индивидуальные задания

1. Реализовать Raft для 5 узлов. Показать, что требуется 3 голоса для выборов и подтверждения записи.

2. Добавить сохранение состояния (persistent state) на диск (current_term, voted_for, лог) и продемонстрировать восстановление после перезапуска.
3. Реализовать безопасность лога: если лидер отправляет AppendEntries, а последователь обнаруживает несоответствие prevLogIndex, последователь должен возвращать неудачу и лидер должен уменьшать индекс (реализовать nextIndex).
4. Вместо HTTP использовать gRPC (Protocol Buffers) и измерить задержки RPC.
5. Добавить механизм heartbeat с переменной частотой: при высокой нагрузке – чаще, при отсутствии команд – реже.
6. Реализовать клиентскую библиотеку, которая перенаправляет запросы текущему лидеру (клиент узнаёт лидера через попытки или редиректы).
7. Добавить симуляцию сетевых задержек и потерь пакетов (через случайную задержку ответа или дроп). Как это влияет на выборы?
8. Создать визуализацию состояний узлов и логов в реальном времени (через WebSocket + простой веб-интерфейс).
9. Расширить конечный автомат до поддержки нескольких операций (CAS, удаление) и реализовать идемпотентность чтения.
10. Реализовать снапшоты (snapshot) – когда лог становится слишком длинным, узел создаёт дамп состояния и усекает лог.
11. Измерить время до восстановления после отказа лидера при разных таймаутах (100, 300, 500 мс). Построить график.
12. Реализовать Raft на языке Rust с использованием async/await и каналов tokio.
13. Добавить шифрование и аутентификацию между узлами (например, предварительно разделённый ключ).
14. Реализовать неблокирующую репликацию: лидер может продолжать приём команд, даже если некоторые последователи отстают (сохраняя кворум).
15. Сравнить упрощённую реализацию с эталонной (например, библиотека raft на Python) по функциональности и производительности.

Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание алгоритма Raft: термины, состояния, RPC, выборы, репликация лога;
- раздел «Архитектура и реализация» – общая схема взаимодействия узлов, описание классов, протокол RPC (форматы сообщений), многопоточная модель;
- раздел «Ход выполнения» – шаги запуска и тестирования, листинги ключевых методов (`handle_request_vote`, `handle_append_entries`, алгоритм выборов);
- раздел «Тестирование и результаты»:
 - доказательство избрания лидера (логи, скриншоты);
 - демонстрация репликации команд на все узлы;
 - сценарий отказа лидера и перевыборов;
 - таблица с замерами времени восстановления;
- раздел «Анализ» – соответствие принципам Raft, ограничения упрощённой реализации;
- выводы (достигнут ли консенсус, как работает отказоустойчивость);
- ответы на контрольные вопросы.

Исходные коды должны быть приложены отдельным архивом или ссылкой на репозиторий.

Контрольные вопросы

1. Почему распределённым системам нужен консенсус?
2. Какие три ключевых компонента выделяет алгоритм Raft?
3. Что такое `term` и как он монотонно увеличивается?

4. Как происходит обнаружение таймаута выборов? Что делает узел при его истечении?
5. Почему в Raft используются случайные таймауты?
6. Как гарантируется, что в одном термине не будет двух лидеров?
7. Что такое реплицированный лог? Как достигается согласованность логов на всех узлах?
8. Какое правило определяет, может ли узел стать лидером?
9. Что происходит, если лидер падает до того, как запись была закоммичена?
10. Почему Raft считается более понятным, чем Paxos?
11. Какие гарантии безопасности, предоставляемые Raft, были реализованы в работе?
12. Как бы вы добавили поддержку динамического изменения состава узлов (добавление/удаление) в Raft?

Лабораторная работа №7.

Распределённый rate limiter через Redis (Token Bucket + Lua)

Цель работы: реализовать распределённый ограничитель скорости (rate limiter), работающий на нескольких серверах-воркерах, с использованием алгоритма Token Bucket и атомарного Lua-скрипта в Redis, а также проверить соблюдение суммарного лимита запросов к ресурсу.

Задачи:

- изучить алгоритм Token Bucket для ограничения частоты запросов;
- реализовать Lua-скрипт в Redis, который атомарно обновляет состояние ведра токенов;
- создать несколько клиентов (воркеров), которые конкурентно обращаются к общему ресурсу;
- убедиться, что суммарное число успешных запросов всех воркеров не превышает заданного лимита (например, RPS 10, burst 2);
- сравнить поведение распределённого rate limiter с локальным (каждый воркер имеет свой счётчик);
- проанализировать влияние задержек при обращении к Redis и предложить оптимизации.

Теоретическая часть

Rate limiting – механизм контроля частоты запросов к сервису, API или ресурсу. Необходим для предотвращения злоупотреблений, защиты от DDoS, обеспечения равномерной нагрузки и соблюдения SLA.

Алгоритм Token Bucket («ведро токенов») – один из наиболее популярных. Параметры:

- capacity (max_burst) – максимальное количество токенов, которое может накопиться;
- refill_rate (токенов в секунду) – скорость добавления токенов;
- начальное состояние – ведро заполнено до capacity.

При каждом запросе: если текущее количество токенов ≥ 1 , запрос разрешается, один токен удаляется. Если токенов нет – запрос отклоняется (или ставится в очередь, что реже). Токены добавляются с фиксированной скоростью, но не превышая capacity.

Распределённый rate limiter должен работать согласованно для многих экземпляров приложения (воркеров), чтобы общий лимит на ресурс соблюдался. Для этого общее состояние ведра обычно хранят во внешнем хранилище с атомарными операциями. **Redis** идеально подходит благодаря:

- высокой производительности (in-memory);
- атомарным Lua-скриптам, которые выполняются без прерываний;
- поддержке структур данных (строки, хеши).

Lua-скрипт в Redis обеспечивает атомарность чтения и записи нескольких ключей. Скрипт принимает ключ (идентификатор лимитируемого ресурса/пользователя),

параметры: capacity, refill_rate, current_time_microseconds. На основе предыдущего состояния вычисляет, сколько токенов добавилось, разрешён ли запрос, обновляет состояние.

Атомарность и согласованность критичны: если бы мы сначала читали состояние, потом записывали, два параллельных воркера могли бы нарушить лимит.

Методика выполнения работы

1. Подготовка окружения

1.1. Установите и запустите Redis (локально или через Docker):

```
docker run -d --name redis -p 6379:6379 redis
```

1.2. Установите клиент Redis для вашего языка: для Python – redis-py (pip install redis), для Node.js – ioredis, для Java – Jedis.

1.3. Установите библиотеку для создания нескольких параллельных воркеров – threading (Python) или asyncio, concurrent.futures.

2. Проектирование Lua-скрипта Token Bucket

2.1. Создайте скрипт token_bucket.lua. Он должен:

- принимать
аргументы: key, capacity, refill_rate, now_us, cost (обычно cost=1).
- читать из Redis хеш по ключу key с полями: last_update, tokens.
- вычислить прошедшее время, добавить токены ($added = (now_us - last_update) / 1e6 * refill_rate$), ограничить $tokens = \min(capacity, tokens + added)$.
- если $tokens \geq cost$, уменьшить tokens на cost, обновить last_update, вернуть {allowed=1, remaining_tokens}.
- иначе вернуть {allowed=0, remaining_tokens}.

2.2. Загрузите скрипт в Redis через SCRIPT LOAD или передавайте его при каждом вызове.

Пример скрипта (упрощённый для микросекунд):

lua

local key = KEYS[1]

local capacity = tonumber(ARGV[1])

local refill_rate = tonumber(ARGV[2]) -- *tokens per second*

local now = tonumber(ARGV[3]) -- *microseconds*

local cost = tonumber(ARGV[4]) or 1

local data = redis.call('HMGET', key, 'last_update', 'tokens')

local last = data[1] and tonumber(data[1]) or now

local tokens = data[2] and tonumber(data[2]) or capacity

local delta = (now - last) / 1000000 -- *to seconds*

tokens = math.min(capacity, tokens + delta * refill_rate)

local allowed = 0

if tokens >= cost then

tokens = tokens - cost

```
        allowed = 1
    end
    redis.call('HMSET', key, 'last_update', now, 'tokens', tokens)
    redis.call('EXPIRE', key, 60) -- очистка старых ключей
    return {allowed, tokens}
```

3. Реализация распределённого rate limiter на Python/Java/Node.js

3.1. Создайте класс DistributedRateLimiter.

3.2. В конструкторе инициализируйте клиент Redis и сохраните SHA загруженного скрипта.

3.3. Метод allow(key) вызывает evalsha (или eval) с параметрами.

3.4. Ключом может быть идентификатор пользователя, IP, или глобальный ресурс.

```
def allow(self, resource_id):
    now_us = int(time.time() * 1_000_000)
    res = self.redis.evalsha(self.script_sha, 1, resource_id,
                             self.capacity, self.refill_rate, now_us, 1)
    return res[0] == 1
```

4. Создание нескольких воркеров (клиентов)

4.1. Напишите функцию worker_loop(limiter, resource_id, duration_sec, worker_id):

- в цикле в течение duration_sec пытается выполнить запрос (вызвать allow).
- подсчитывает количество успешных и неудачных попыток.
- можно добавить случайную задержку (например, 0...100 мс) между попытками.

4.2. Запустите num_workers = 5 параллельных потоков/процессов.

4.3. Соберите общую статистику: сколько раз каждый воркер получил разрешение.

5. Тестирование соблюдения суммарного лимита

- 5.1. Задайте `capacity = 5`, `refill_rate = 10` (10 токенов в секунду, `burst 5`).
- 5.2. Симулируйте нагрузку, при которой общее количество запросов от всех воркеров ~ 50 в секунду.
- 5.3. Запустите тест на 10 секунд. Ожидаемый общий лимит ≈ 100 разрешений ($10/\text{сек} * 10 \text{ сек}$). Максимальный `burst` может дать немного больше в первые секунды.
- 5.4. Сравните фактическое общее число `allowed` с теоретическим.
- 5.5. Повторите тест с локальным `rate limiter` (каждый воркер имеет своё ведро) – должно получиться превышение общего лимита примерно в `num_workers` раз.

6. Измерение задержек и нагрузки на Redis

- 6.1. Замерьте p99 задержку вызова `allow()` для 50 воркеров (каждый делает 1000 запросов).
- 6.2. Используйте `redis-benchmark` или метрики в коде.
- 6.3. Оцените, сколько RPS выдерживает ваш `rate limiter`.
- 6.4. Постройте график зависимости RPS лимитера от числа воркеров (2, 5, 10, 20).

7. Анализ и оптимизации

- 7.1. Обратите внимание на то, что `now_us` генерируется клиентом – может быть рассинхрон часов. Альтернатива: получать время внутри Lua через `redis.call('TIME')`.
- 7.2. Добавьте поддержку сценария с несколькими ключами (например, лимит для каждого клиента и глобальный лимит).
- 7.3. Если лимитер используется для `high-load`, объединяйте несколько проверок в один Lua-скрипт.

Пример выполнения задания

Упрощённый скрипт тестирования (Python):

```
import threading, time, redis
```

```
class RedisRateLimiter:
```

```
    def __init__(self, host='localhost', capacity=5, rate=10):
        self.r = redis.Redis(host=host, decode_responses=True)
        self.capacity = capacity
        self.rate = rate
        script = """
        local key = KEYS[1]
        local capacity = tonumber(ARGV[1])
        local rate = tonumber(ARGV[2])
        local now = tonumber(ARGV[3])
        local data = redis.call('HMGET', key, 'tokens', 'last')
        local tokens = tonumber(data[1]) or capacity
        local last = tonumber(data[2]) or now
        delta = math.max(0, now - last)
        tokens = math.min(capacity, tokens + delta * rate / 1000000)
        allowed = 0
        if tokens >= 1 then
            tokens = tokens - 1
            allowed = 1
        end
        redis.call('HMSET', key, 'tokens', tokens, 'last', now)
        redis.call('EXPIRE', key, 5)
        return {allowed, tokens}
        """

        self.sha = self.r.script_load(script)

    def allow(self, key):
        now_us = int(time.time() * 1_000_000)
        res = self.r.evalsha(self.sha, 1, key, self.capacity, self.rate, now_us)
        return res[0] == 1
```

```

def worker(limiter, duration, stats, wid):
    end = time.time() + duration
    allowed = 0
    while time.time() < end:
        if limiter.allow("global_resource"):
            allowed += 1
            # небольшая пауза, чтобы не перегружать redis
            time.sleep(0.01)
    stats[wid] = allowed

if __name__ == "__main__":
    limiter = RedisRateLimiter(capacity=5, rate=10)
    workers = 5
    threads = []
    stats = {}
    start = time.time()
    for i in range(workers):
        t = threading.Thread(target=worker, args=(limiter, 10, stats, i))
        t.start()
        threads.append(t)
    for t in threads:
        t.join()
    total = sum(stats.values())
    print(f'Total allowed: {total}, expected ~ {10 * 10} (capacity burst extra)')

```

Результат:

- При локальном лимите (каждый воркер своё ведро) `total_allowed` ≈ 500 (5 воркеров $\times$ 100).
- При распределённом лимите через Redis `total_allowed` ≈ 105 + небольшие вариации (burst) – лимит соблюдается.

Индивидуальные задания

1. Реализовать алгоритм **Leaky Bucket** (вытекающее ведро) через Lua-скрипт и сравнить с Token Bucket по поведению при burst.
2. Добавить поддержку лимитов по нескольким измерениям (ip + user-agent) через составной ключ.
3. Заменить встроенный `time.time()` на получение времени непосредственно внутри Lua-скрипта (`redis.call('TIME')`), чтобы устранить погрешность синхронизации.
4. Реализовать распределённый **sliding window** (скользящее окно) с помощью отсортированных множеств Redis (ZSET) и сравнить точность с Token Bucket.
5. Добавить механизм «очереди ожидания» (throttling): запросы, превышающие лимит, не отклоняются, а откладываются на некоторое время.
6. Создать HTTP-мидлвару на FastAPI/Express, которая использует ваш rate limiter для ограничения API. Протестировать с ab (ApacheBench).
7. Измерить, какое максимальное количество RPS способен обработать Redis на одной машине с вашим скриптом (построить график).
8. Реализовать иерархический rate limiter: глобальный лимит (100 gps) и на каждого пользователя (10 gps) – оба в одном Lua-скрипте.
9. Добавить возможность изменения лимитов "на лету" (обновление параметров в Redis без перезагрузки приложения).
10. Вместо блокирующих вызовов Redis использовать асинхронный драйвер (`redis.asyncio` в Python) и измерить повышение пропускной способности при 1000 конкурентных клиентах.
11. Сымитировать отказ Redis (например, отключить сеть) и реализовать fallback: лимитер работает в локальном режиме с предупреждением.
12. Реализовать лимитер для распределённой системы, где токены начисляются пачками раз в фиксированный интервал (например, раз в секунду). Сравнить с непрерывным пополнением.

13. Добавить логирование метрик наблюдения: счётчик разрешённых и отклонённых запросов, текущий уровень токенов, экспорт в Prometheus.

14. Реализовать rate limiter на основе очередей Redis Streams + Consumer Group (аналог токенов).

15. Сравнить производительность и точность вашего решения с популярной библиотекой redis-cell (модуль для Redis с готовым rate limiter).

Требования к отчёту

Отчёт должен быть оформлен в виде PDF и содержать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание алгоритмов Token Bucket, распределённых ограничителей, атомарных Lua-скриптов в Redis;
- раздел «Проектирование» – схема работы, выбор ключей, структура данных в Redis, текст Lua-скрипта с комментариями;
- раздел «Реализация» – листинги кода на выбранном языке (класс limiter, воркеры, клиентская часть);
- раздел «Эксперименты» – таблицы и графики:
 - Общее количество разрешённых запросов (распределённый vs локальный).
 - Зависимость допустимого RPS от числа воркеров.
 - p99 задержка при разной нагрузке.
- раздел «Анализ» – объяснение, почему распределённый лимитер точен, а локальный – нет, как влияет burst, узкие места;
- выводы – достигнута ли цель, применимость подхода, возможные улучшения;
- ответы на контрольные вопросы.

Код должен быть читаем, скриншоты логов или выводов консоли приветствуются.

Контрольные вопросы

1. Для чего нужен rate limiter в высоконагруженных системах?
2. Как работает алгоритм Token Bucket? Чем он отличается от Leaky Bucket?
3. Почему для распределённого rate limiter необходимо атомарное обновление состояния?
4. Как Redis обеспечивает атомарность выполнения Lua-скриптов?
5. Что произойдёт, если два воркера одновременно вызовут Redis с неатомарными командами GET + SET?
6. Какая структура данных в Redis используется в данной работе для хранения состояния ведра?
7. Как учесть разницу во времени на разных клиентах при вычислении now?
8. Можно ли обойтись без Lua-скрипта, используя Redis-команду SET NX и счётчики? Почему это приведёт к ошибкам?
9. Как влияет задержка сети между воркером и Redis на точность лимита?
10. Как реализовать «смягчённый» rate limiter, который позволяет небольшие превышения, но длительные превышения блокирует?
11. Какие метрики нужно собирать для мониторинга работы распределённого rate limiter в production?
12. Сравните подход с Lua-скриптом в Redis и подход с использованием INCR и истечения срока (sliding window через EXPIRE). Какой точнее?

Лабораторная работа №8.

Circuit Breaker с мониторингом (Resilience4j / сторонняя реализация)

Цель работы: реализовать паттерн Circuit Breaker с тремя состояниями (Closed, Open, Half-Open) и интеграцией с системой метрик (Prometheus) для защиты распределённого приложения от каскадных отказов при вызове нестабильного внешнего сервиса.

Задачи:

- создать клиентское приложение, которое вызывает нестабильный сервис (имитация падений, длительных ответов, таймаутов);
- реализовать собственный Circuit Breaker или использовать готовую библиотеку (Resilience4j на Java, pybreaker на Python, opossum на Node.js) с настраиваемыми порогами ошибок и таймаутами;
- реализовать три состояния: Closed (замкнут) – пропуск запросов, Open (разомкнут) – быстрый возврат ошибки без вызова сервиса, Half-Open (полуоткрыт) – ограниченное количество проверочных запросов для восстановления;
- экспортировать метрики (количество вызовов, число ошибок, состояние переключателя, время в каждом состоянии) в формате Prometheus;
- визуализировать метрики в Grafana или просто вывести статистику;
- проверить, что Circuit Breaker предотвращает лавину отказов при падении зависимого сервиса.

Теоретическая часть

Circuit Breaker («выключатель цепи») – паттерн отказоустойчивости, который предотвращает повторные вызовы заведомо неработающего или перегруженного внешнего сервиса. Он работает как электрический автомат: при превышении порога ошибок «размыкает цепь» и все последующие вызовы

мгновенно завершаются ошибкой (без реального запроса), давая сервису время на восстановление.

Состояния Circuit Breaker:

- **Closed (замкнут):** запросы проходят, статистика ошибок накапливается. Если за определённое время (окно) доля или абсолютное количество ошибок превышает порог, переходим в Open.
- **Open (разомкнут):** запросы блокируются, мгновенно возвращается fallback или ошибка. Через заданный таймаут (timeout восстановления) переходим в Half-Open.
- **Half-Open (полуоткрыт):** пускается ограниченное количество запросов (например, 1 или 5). Если они успешны – переходим в Closed; если хотя бы один ошибочен – возвращаемся в Open.

Параметры конфигурации:

- `failureThreshold` – количество (или процент) ошибок для перехода в Open.
- `waitDurationInOpenState` – время ожидания перед переходом в Half-Open.
- `ringBufferSizeInClosedState` – размер кольцевого буфера для подсчёта ошибок в Closed.
- `permittedNumberOfCallsInHalfOpenState` – сколько пробных вызовов разрешено в Half-Open.

Интеграция с метриками (Prometheus):

Мониторинг Circuit Breaker необходим для отслеживания состояния системы.

Типовые метрики:

- `circuit_breaker_state` (gauge) – 0=Closed, 1=Open, 2=Half-Open.
- `circuit_breaker_calls_total` (counter) – общее количество вызовов.
- `circuit_breaker_failures_total` (counter) – ошибки.
- `circuit_breaker_successes_total` – успешные вызовы.
- `circuit_breaker_rejected_calls_total` – вызовы, отклонённые в Open.

Зачем использовать:

- Предотвращает «эффект снежного кома» (потoki ждут таймауты и исчерпывают ресурсы).
- Позволяет быстрее восстановиться, разгружая проблемный сервис.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог `lab8_circuit_breaker`.
- 1.2. Установите библиотеку Circuit Breaker:
 - Python: `pip install pybreaker` (или реализовать вручную).
 - Java: подключите Resilience4j (`io.github.resilience4j:resilience4j-circuitbreaker`).
 - Node.js: `npm install opossum`.
- 1.3. Установите библиотеку для метрик:
 - Python: `prometheus_client`.
 - Java: `micrometer-registry-prometheus` + Spring Boot.
 - Node.js: `prom-client`.
- 1.4. При необходимости установите `requests`, `aiohttp` или встроенный HTTP-клиент.

2. Реализация нестабильного сервиса (заглушка)

- 2.1. Напишите простой HTTP-сервер (можно локально на порту 8081), который:
 - имеет вероятность ошибки `error_rate` (например, 40%).
 - иногда отвечает очень долго (например, `sleep(2)`).
 - имеет эндпоинт `/health` для проверки.
- 2.2. Реализуйте метод `call_service()` (в клиенте), который отправляет запрос с таймаутом (например, 1 секунда). При таймауте, сетевой ошибке или HTTP 5xx считать ошибкой.

`def unstable_request():`

`if random.random() < 0.4:`

`time.sleep(2) # таймаут`

```
        raise TimeoutError
    else:
        return "OK"
```

3. Реализация (или подключение) Circuit Breaker

3.1. Настройте экземпляр Circuit Breaker с параметрами:

- порог ошибок: 3 ошибки в течение 5 последних вызовов.
- время в Open: 10 секунд.
- в Half-Open разрешить 2 пробных вызова.

3.2. Если используется готовая библиотека – изучите её API. В противном случае реализуйте свой класс CircuitBreaker с переменными состояния, счётчиками и таймерами.

3.3. Напишите декоратор или функцию call_with_circuit_breaker(cb, func), которая:

- в Closed вызывает func(), обрабатывает исключения и обновляет статистику.
- в Open сразу возвращает исключение CircuitBreakerOpenError.
- в Half-Open вызывает не более permitted попыток; результат влияет на переход.

4. Интеграция с Prometheus

4.1. Создайте метрики:

- cb_state_gauge – отслеживает текущее состояние (0,1,2).
- cb_calls_counter – увеличивается при каждом вызове (с меткой result="success" или "failure").
- cb_rejected_counter – увеличивается, когда вызов отклонён из-за Open.

4.2. Обновляйте метрики в ключевых точках (перед вызовом, после успеха/ошибки, при изменении состояния).

4.3. Запустите HTTP-сервер для метрик (например, start_http_server(8000)).

4.4. Проверьте, что эндпоинт `http://localhost:8000/metrics` отдаёт данные в формате Prometheus.

5. Симуляция нагрузки и анализ поведения

5.1. Напишите цикл, который в течение 60 секунд каждые 0.2 секунды вызывает защищённый вызов.

5.2. В отдельном потоке или процессе каждые 10 секунд меняйте состояние нестабильного сервиса (через флаг или эндпоинт `/set_error_rate`). Например:

- первые 20 секунд: 60% ошибок (переход в Open).
- следующие 20 секунд: сервис восстановлен (0% ошибок) – должно перейти в Half-Open затем в Closed.
- пауза 20 секунд: снова 100% ошибок.

5.3. Запишите логи переходов состояния и ошибок. Постройте график состояния во времени (по данным метрик).

5.4. Оцените: сколько вызовов было реально отправлено на сервис в периоды Open по сравнению с Closed.

6. Дополнительно: реализация ручного сброса и fallback

6.1. Добавьте возможность принудительно сбросить Circuit Breaker в Closed (например, через HTTP-эндпоинт `/reset`).

6.2. Реализуйте fallback-значение (кэшированный ответ, сообщение «сервис временно недоступен»), которое возвращается вместо выброса исключения в состоянии Open.

6.3. Продемонстрируйте работу fallback.

7. Сравнительный анализ

7.1. Запустите аналогичную нагрузку без Circuit Breaker. Зафиксируйте, как падение сервиса приводит к зависаниям и накоплению таймаутов в клиенте.

7.2. Сравните метрики времени ответа (p99) и количество успешных вызовов с СВ и без него.

Пример выполнения задания

Пример реализации простого Circuit Breaker на Python (без внешних библиотек):

```
import time
import threading
from enum import Enum

class State(Enum):
    CLOSED = 0
    OPEN = 1
    HALF_OPEN = 2

class CircuitBreaker:
    def __init__(self, failure_threshold=3, recovery_timeout=10,
half_open_max_calls=2):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout # секунд
        self.half_open_max_calls = half_open_max_calls
        self.state = State.CLOSED
        self.failure_count = 0
        self.last_failure_time = 0
        self.half_open_calls_made = 0
        self.lock = threading.Lock()

    def call(self, func, *args, **kwargs):
        with self.lock:
            if self.state == State.OPEN:
                if time.time() - self.last_failure_time >= self.recovery_timeout:
                    self.state = State.HALF_OPEN
```

```

        self.half_open_calls_made = 0
    else:
        raise Exception("CircuitBreakerOpen")
    if self.state == State.HALF_OPEN:
        if self.half_open_calls_made >= self.half_open_max_calls:
            raise Exception("CircuitBreakerHalfOpenLimited")

    try:
        result = func(*args, **kwargs)
        #ycnex
    with self.lock:
        if self.state == State.HALF_OPEN:
            self.half_open_calls_made += 1
            if self.half_open_calls_made >= self.half_open_max_calls:
                self.state = State.CLOSED
                self.failure_count = 0
                self.half_open_calls_made = 0
            elif self.state == State.CLOSED:
                self.failure_count = 0
        return result
    except Exception as e:
        with self.lock:
            if self.state == State.CLOSED:
                self.failure_count += 1
                if self.failure_count >= self.failure_threshold:
                    self.state = State.OPEN
                    self.last_failure_time = time.time()
            elif self.state == State.HALF_OPEN:
                self.state = State.OPEN
                self.last_failure_time = time.time()

```

```
self.half_open_calls_made = 0  
raise e
```

Результаты теста:

- Без СВ: при 60% ошибок все вызовы ждут таймаута 2 с, p99 = 2000 мс.
- С СВ: после 3 ошибок переходит в Open, следующие вызовы мгновенно падают (задержка 1 мс), через 10 с Half-Open – восстанавливается.

Индивидуальные задания

1. Реализовать Circuit Breaker с учётом процента ошибок (например, если 50% последних 10 запросов ошибочны) вместо абсолютного счётчика.
2. Добавить поддержку нескольких различных таймаутов: таймаут вызова сервиса, таймаут восстановления, таймаут сброса.
3. Использовать Resilience4j в Spring Boot приложении, подключить Micrometer + Prometheus, вывести дашборд в Grafana.
4. Расширить реализацию на несколько сервисов (разные Circuit Breaker для разных эндпоинтов).
5. Внедрить retry-политику совместно с Circuit Breaker: при ошибках делать повторы, и только после исчерпания повторов учитывать ошибку для СВ.
6. Реализовать неблокирующий (асинхронный) Circuit Breaker с asyncio (Python) или CompletableFuture (Java).
7. Добавить эндпоинт управления для ручного переключения состояний (принудительно Open/Closed) в целях тестирования.
8. Интегрировать СВ в существующее приложение из ЛР 5 (Saga), защитив вызовы сервиса оплаты.
9. Сымитировать частичный отказ сервиса (только медленные ответы без ошибок). Настроить порог на основе задержки (slow call threshold).
10. Реализовать кэширование fallback-ответов при размыкании СВ (например, вернуть последний успешный ответ).

11. Экспортировать метрики в формате JSON через REST, а не только Prometheus.

12. Написать юнит-тесты на конечный автомат Circuit Breaker (переходы, граничные условия).

13. Реализовать распределённый Circuit Breaker через Redis (состояние синхронизируется между несколькими экземплярами клиента).

14. Сравнить поведение при разных стратегиях сброса счётчика ошибок (скользящее окно, фиксированное окно, экспоненциальное убывание).

15. Добавить логирование каждого перехода состояния в structured-формате (JSON) и отправить в ELK.

Требования к отчёту

Отчёт должен быть оформлен как PDF и включать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание паттерна Circuit Breaker, состояний, параметров, связь с отказоустойчивостью;
- раздел «Проектирование» – выбор библиотеки или собственная реализация, схема конечного автомата, диаграмма переходов;
- раздел «Реализация» – листинги кода (СВ, интеграция с метриками, клиент вызова нестабильного сервиса);
- раздел «Эксперименты и результаты» – описание сценариев нагрузки, таблицы и графики:
 - количество реальных вызовов сервиса со СВ и без;
 - временная диаграмма состояний;
 - метрики из Prometheus (снимки экрана дашборда или консольный вывод);
 - сравнение p99 latency до и после применения.
- раздел «Анализ» – как СВ предотвращает каскадные отказы, ограничения подхода;

- выводы;
- ответы на контрольные вопросы.

Все графики подписаны. Код должен быть читаемым, листинги отформатированы.

Контрольные вопросы

1. Какие три основных состояния имеет Circuit Breaker? Опишите переходы между ними.
2. Как паттерн Circuit Breaker помогает в высоконагруженных системах?
3. Что произойдёт, если в состоянии Closed происходит 4 ошибки подряд, а порог установлен в 3?
4. Зачем нужно состояние Half-Open? Почему нельзя сразу перейти из Open в Closed?
5. Какие метрики необходимы для мониторинга Circuit Breaker в реальном времени?
6. В чём разница между таймаутом вызова сервиса и таймаутом восстановления СВ?
7. Можно ли использовать Circuit Breaker без fallback? Каковы последствия?
8. Как сочетается Circuit Breaker с паттерном Retry? В каком порядке их применять?
9. Как бы вы настроили порог срабатывания для сервиса с высокой вариабельностью ошибок?
10. Как реализовать распределённый Circuit Breaker на несколько экземпляров приложения?
11. Почему в состоянии Half-Open обычно разрешают лишь небольшое число вызовов?
12. Как интеграция с Prometheus помогает при эксплуатации системы с Circuit Breaker?

Лабораторная работа №9.

Distributed tracing с OpenTelemetry и Jaeger

Цель работы: внедрить распределённую трассировку (distributed tracing) в приложение, состоящее минимум из двух взаимодействующих сервисов и базы данных, с использованием OpenTelemetry, и проанализировать собранные трейсы в Jaeger для выявления узких мест и задержек.

Задачи:

- создать два микросервиса (Service A и Service B) и подключить к ним простую БД (PostgreSQL или in-memory);
- инструментировать код с помощью OpenTelemetry API/SDK для генерации спанов и контекста трассировки;
- настроить экспорт телеметрии в Jaeger (через OTLP или Jaeger exporter);
- запустить Jaeger (в Docker) и просмотреть сгенерированные трейсы;
- симитировать задержки (например, медленный запрос к БД или внешнему API) и найти узкое место по данным трассировки;
- понять, как distributed tracing помогает в диагностике проблем в высоконагруженных распределённых системах.

Теоретическая часть

Distributed tracing — метод мониторинга запросов, проходящих через несколько сервисов в распределённой системе. Позволяет восстановить полный путь запроса, измерить задержки на каждом этапе и выявить узкие места.

Ключевые понятия:

- **Trace (трейс)** — полное представление запроса, проходящего через систему. Состоит из множества спанов.
- **Span (спан)** — единица работы в одном сервисе (например, обработка HTTP-запроса, вызов БД, RPC). Содержит имя, время начала, длительность, атрибуты (метаданные) и ссылки на родительский спан.

- **Trace ID** – уникальный идентификатор трейса. Передаётся между сервисами через заголовки (например, `tracereparent`).
- **Context propagation** – механизм передачи контекста трассировки (Trace ID, Span ID) между сервисами через HTTP/gRPC/очереди.

OpenTelemetry (OTel) – единый стандарт для сбора телеметрии (трейсов, метрик, логов). Предоставляет API, SDK и инструментарий для автоматического и ручного инструментирования приложений.

Jaeger – распределённая система трассировки, разработанная Uber. Включает: агент, коллектор, хранилище (Elasticsearch/Cassandra), UI для поиска и анализа трейсов. Популярен благодаря интеграции с OpenTelemetry.

Почему это важно для highload:

- Поиск узких мест (медленные запросы к БД, блокировки, сетевые задержки).
- Понимание зависимостей между сервисами.
- Обнаружение «выбросов» латентности (p99).
- Помощь при отладке в распределённой среде, где логи разрознены.

В лабораторной работе мы вручную инструментируем два сервиса (например, на Python FastAPI или Java Spring) и подключаем их к Jaeger через OTLP gRPC.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог `lab9_tracing`.
- 1.2. Установите Docker и Docker Compose.
- 1.3. Создайте файл `docker-compose.yml` для запуска Jaeger (all-in-one) и БД (например, PostgreSQL):

```
version: '3'
```

```
services:
```

```
  jaeger:
```

```
    image: jaegertracing/all-in-one:latest
```

ports:

- "16686:16686" # *UI*
- "4317:4317" # *OTLP gRPC*
- "4318:4318" # *OTLP HTTP*

environment:

COLLECTOR_OTLP_ENABLED: "true"

postgres:

image: postgres:13

environment:

POSTGRES_USER: test

POSTGRES_PASSWORD: test

POSTGRES_DB: tracing

ports:

- "5432:5432"

1.4. Запустите: `docker-compose up -d`.

1.5. Установите библиотеки OpenTelemetry для выбранного языка.

- Python: `pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-otlp opentelemetry-instrumentation-requests opentelemetry-instrumentation-fastapi`.
- Node.js: `npm install @opentelemetry/sdk-node @opentelemetry/exporter-trace-otlp-grpc @opentelemetry/instrumentation-http @opentelemetry/instrumentation-express`.
- Java: `io.opentelemetry:opentelemetry-exporter-otlp + io.opentelemetry.instrumentation:opentelemetry-spring-boot-starter`.

2. Создание сервисов

2.1. **Service A (порт 8000)** – принимает HTTP-запрос GET `/process?data=<text>`.

- Логика: получить data, сделать вызов к Service B, сформировать ответ {"result": ...}.

2.2. Service B (порт 8001) – принимает GET /enrich?value=....

- Логика: записать полученное значение в PostgreSQL (таблица logs с полями id, value, created_at), затем выполнить простую имитацию вычисления (time.sleep(0.05) или sum(range(10000))).

2.3. Напишите код без трассировки сначала, убедитесь, что связка работает.

3. Инструментирование OpenTelemetry

3.1. Инициализируйте TracerProvider в каждом сервисе.

3.2. Настройте экспортёр OTLP (gRPC или HTTP) на адрес localhost:4317.

3.3. В Service A: создайте спан на обработку запроса (process_request).

- При вызове Service B используйте HTTP headers propagation: добавьте заголовки traceparent, tracestate (или используйте автоматическую инструментацию requests).

3.4. В Service B: создайте спан enrich_request. Внутри:

- спан на операцию с БД (выполнение INSERT),
- спан на вычислительную задачу.

3.5. Добавьте атрибуты к спанам: http.method, http.url, db.statement, custom.data_size.

3.6. Обязательно добавьте обработку ошибок: при исключении записывайте событие exception в спан.

Пример (Python + Flask/FastAPI + OTel):

```
from opentelemetry import trace

from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter

from opentelemetry.sdk.trace import TracerProvider
```

```

from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.instrumentation.requests import RequestsInstrumentor

trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)
exporter = OTLPSpanExporter(endpoint="localhost:4317", insecure=True)
span_processor = BatchSpanProcessor(exporter)
trace.get_tracer_provider().add_span_processor(span_processor)

RequestsInstrumentor().instrument()      # автоматический пропуск

```

заголовков

При вызове `requests.get(...)` контекст передастся автоматически.

4. Добавление симулированной задержки для выявления узкого места

4.1. В Service B добавьте возможность случайной задержки (например, `time.sleep(random.uniform(0.01, 0.3))`).

4.2. Или имитируйте медленный запрос к БД: вместо простого INSERT выполните `pg_sleep(0.2)` или загрузите CPU циклом.

4.3. Сделайте так, чтобы примерно 10% запросов к БД были «тяжёлыми» (задержка 500 мс).

4.4. В Service A также добавьте небольшую `time.sleep(0.01)` для имитации обработки.

5. Генерация нагрузки

5.1. Напишите скрипт `load.py`, который в течение 2 минут отправляет 200 запросов к Service A (параллельно, например, 10 воркеров).

5.2. После завершения нагрузки откройте Jaeger UI: <http://localhost:16686>.

5.3. Найдите сервис `service-a` и выполните поиск трейсов за последние 10 минут.

5.4. Выберите несколько трейсов (особенно с большой длительностью). Проанализируйте временную диаграмму – какой спан самый долгий.

6. Анализ узкого места

6.1. По данным трейсов определите, где возникают основные задержки:

- Задержка в Service A (до вызова Service B).
- Сетевой вызов между A и B.
- База данных в Service B.
- Вычисления в Service B.

6.2. Сравните трейсы «медленных» и «быстрых» запросов.

6.3. Постройте вывод (например, «80% общего времени запроса приходится на вызов Service B, из них 70% – на запись в БД»).

6.4. Сформулируйте рекомендацию по оптимизации: добавить индекс, кэшировать, асинхронно писать в БД.

7. Дополнительно: логирование и метрики

7.1. Интегрируйте метрики (например, количество запросов, длительность) через OpenTelemetry Metrics и экспортируйте в Prometheus (опционально).

7.2. Добавьте корреляцию трейсов с логами (через `trace_id` в вывод логов).

8. Сравнение с мониторингом без трассировки

8.1. Продемонстрируйте, что без distributed tracing невозможно определить, какой именно компонент вызывает задержку, имея только среднюю латентность из метрик.

Пример выполнения задания

Фрагмент Service A (FastAPI + OTel):

```
from fastapi import FastAPI
import requests
from opentelemetry import trace
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
```

```

app = FastAPI()
tracer = trace.get_tracer(__name__)

@app.get("/process")
async def process(data: str):
    with tracer.start_as_current_span("service-a-process") as span:
        span.set_attribute("request.data", data)
        # вызов Service B
        resp = requests.get(f"http://localhost:8001/enrich?value={data}")
        result = resp.json()
        span.set_attribute("response.length", len(result["result"]))
        return {"result": result["result"]}

```

```
FastAPIInstrumentor.instrument_app(app)
```

Спан Service B (Pyramid/Flask):

```

@routes.route('/enrich')
def enrich(request):
    value = request.query.get('value')
    with tracer.start_as_current_span("service-b-enrich") as span:
        span.set_attribute("db.value", value)
        # запись в БД
        with tracer.start_as_current_span("db-insert") as db_span:
            cur.execute("INSERT INTO logs (value) VALUES (%s)", (value,))
            conn.commit()
            db_span.set_attribute("db.rows_affected", 1)
        # имитация вычислений
        with tracer.start_as_current_span("compute") as comp_span:
            heavy_compute() # например, range(100000)

```

```
return {"result": f'enriched_{value}'}
```

Скриншот Jaeger UI: отображается трейс:

- service-a-process (длительность 280 мс)
 - http-call-to-service-b (220 мс)
 - service-b-enrich (215 мс)
 - db-insert (200 мс)
 - compute (15 мс)

Вывод: узкое место — запись в БД (200 мс). Рекомендация: партиционирование, асинхронная запись, подключение Connection Pooler.

Индивидуальные задания

1. Добавить третий сервис (Service C — «аутентификация») и проследить полный трейс через три звена.
2. Вместо синхронного HTTP использовать асинхронное взаимодействие через очередь сообщений (RabbitMQ) и передавать контекст трассировки через заголовки сообщений.
3. Внедрить трассировку с использованием AWS X-Ray вместо Jaeger (или вместе, экспортируя OTLP в X-Ray через Adapter).
4. Сгенерировать нагрузку с распределением латентности по модели «длинный хвост» (например, 99 перцентиль). Показать, как в трейсах находятся выбросы.
5. Добавить логирование с корреляцией: в каждый лог добавлять trace_id (через MDC в Java, logging filter в Python) и сопоставить с трейсами.
6. Инструментировать не HTTP, а gRPC сервисы (с использованием gRPC interceptors для OpenTelemetry).

7. Автоматически инструментировать PostgreSQL-драйвер (например, opentelemetry-instrumentation-psycopg2 в Python) и сравнить ручное инструментирование с автоматическим.

8. Реализовать кастомную выборку трейсов – найти все запросы, у которых задержка > 300 мс, и их общую структуру (например, через API Jaeger).

9. Добавить трассировку клиентской части (браузер) с помощью OpenTelemetry Web и связать её с серверными трейсами.

10. Измерить overhead внедрения трассировки (время выполнения запросов без трейсинга и с включённым экспортом) при 1000 RPS.

11. Настроить выборку (sampling) – всегда сохранять 10% трейсов (probabilistic sampler). Оценить, как уменьшается нагрузка на систему.

12. Использовать OpenTelemetry Collector как агент между сервисами и Jaeger, добавив преобразование атрибутов и фильтрацию.

13. Реализовать алерт на основе трейсов (например, если latency БД > 200 мс в течение 2 минут – отправить уведомление).

14. Внедрить трассировку в существующее приложение из ЛР 5 (Saga) – трассировать события через очередь.

15. Сравнить Jaeger с другим бэкендом трейсинга (например, Tempo от Grafana) по простоте настройки и возможностям UI.

Требования к отчёту

Отчёт оформляется в PDF и включает:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание distributed tracing, OpenTelemetry, Jaeger, структуры трейсов и спанов;
- раздел «Архитектура приложения» – схема взаимодействия сервисов, указание портов, таблицы БД;

- раздел «Реализация трассировки» – листинги инициализации OTel, кода с ручным/автоматическим инструментированием, настройки экспорта;
- раздел «Эксперименты»:
 - описание сценария нагрузки (количество запросов, параллельность);
 - скриншоты Jaeger UI (список трейсов, детальный вид одного трейса с временной диаграммой);
 - таблица с задержками (minimum, average, p99) для каждого сервиса и БД, извлечённая из Jaeger;
 - выявленное узкое место (какой компонент самый медленный).
- раздел «Анализ» – обсуждение, почему без трассировки было бы сложно локализовать проблему, рекомендации по оптимизации;
- выводы;
- ответы на контрольные вопросы.

Все графики и скриншоты должны быть подписаны. Приложен архив с кодом и docker-compose.yml.

Контрольные вопросы

1. Что такое distributed tracing и для каких задач он используется в highload?
2. В чём разница между трассировкой, метриками и логированием?
3. Что такое span? Что содержится в span?
4. Как передаётся контекст трассировки между разными сервисами через HTTP?
5. Какую роль выполняет Jaeger в системе?
6. Что означает термин «propagation» (распространение контекста)?
7. Как асинхронные системы (очереди) могут поддерживать трассировку?

8. Что такое «выборочное семплирование» (sampling) и зачем оно нужно?

9. Каким образом OpenTelemetry упрощает внедрение трассировки в разных языках?

10. Как по данным трейса определить, что виновник задержки – база данных, а не код приложения?

11. Может ли трассировка сама стать причиной падения производительности? Как этого избежать?

12. Какие ещё форматы заголовков для трейсинга существуют кроме W3C TraceContext (например, B3 от Zipkin)?

13. Как связать трассировку с бизнес-транзакцией (например, ID заказа)?

14. Какие узкие места вы обнаружили в своём приложении и как их можно исправить?

Лабораторная работа №10.

Backpressure через Reactive Streams (на примере Project Reactor)

Цель работы: реализовать асинхронную систему producer-consumer с явным управлением backpressure с использованием Reactive Streams (Project Reactor) и сравнить поведение системы без backpressure (приводящее к переполнению буфера или исчерпанию памяти) и с корректной стратегией ограничения запросов.

Задачи:

- создать Publisher (Flux), который генерирует поток данных с высокой скоростью (например, 1000 событий/сек);
- реализовать медленного Consumer (Subscriber), который обрабатывает данные с задержкой (например, 50 мс на элемент);
- настроить backpressure через механизм `request(n)`, ограничивающий количество элементов, передаваемых от Publisher к Subscriber;
- продемонстрировать последствия отсутствия backpressure: накопление элементов в буферах, рост потребления памяти, возможное исчерпание памяти (OOM);
- сравнить метрики (пропускная способность, задержка, использование памяти) для двух сценариев: без backpressure и с использованием `request(n)`;
- сделать выводы о важности backpressure в высоконагруженных асинхронных системах.

Теоретическая часть

Backpressure (обратное давление) – механизм в потоковых системах, позволяющий потребителю сигнализировать производителю о своей готовности обрабатывать новые данные. Без backpressure быстрый производитель может переполнить буферы потребителя (или промежуточные

очереди), что приведёт к росту задержек, исчерпанию памяти и падению приложения.

Reactive Streams – спецификация для асинхронной обработки потоков с неблокирующим `backpressure`. Определяет четыре основных интерфейса: `Publisher<T>`, `Subscriber<T>`, `Subscription` и `Processor<T,R>`.

Основная идея: подписчик через `Subscription.request(n)` указывает, сколько элементов он готов принять, а издатель не отправляет больше запрошенного.

Project Reactor – реализация Reactive Streams для JVM (Java/Kotlin). Основные типы: `Flux<T>` (поток 0..N элементов) и `Mono<T>` (0..1 элемент). `Backpressure` управляется автоматически при использовании стандартных операторов, но его можно явно контролировать через `limitRate()`, `onBackpressureBuffer()`, `onBackpressureDrop()`, а также реализуя собственный `Subscriber`.

Проблемы без `backpressure`:

- Производитель генерирует элементы быстрее, чем потребитель успевает обрабатывать.
- В традиционных блокирующих очередях (например, `LinkedBlockingQueue`) очередь растёт бесконечно, что может вызвать OOM.
- В реактивном потоке без управления `backpressure` `Publisher` по умолчанию будет буферизовать элементы (размер буфера по умолчанию 256), но при превышении может выбросить исключение или переключиться на стратегию `drop/error`.

Стратегии `backpressure` в Reactor:

- `onBackpressureBuffer(capacity)` – буферизация с ограничением (при переполнении – ошибка или политика переполнения).
- `onBackpressureDrop()` – сброс элементов, если потребитель не успевает.
- `onBackpressureLatest()` – хранить только последний элемент.
- `limitRate(n)` – автоматически запрашивать элементы пачками по `n`.

В данной работе мы реализуем кастомного Subscriber, явно управляющего request(n), чтобы увидеть механизм изнутри. Затем сравним с типичными ошибками, когда backpressure не настраивается.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог lab10_backpressure.
- 1.2. Язык – Java с Maven/Gradle (рекомендуется) или Kotlin. Добавьте зависимость io.projectreactor:reactor-core:3.5.x.
- 1.3. Для Python можно использовать reactivex (RxPY) с поддержкой backpressure, но акцент – на Project Reactor.
- 1.4. Установите JDK 11+ и инструмент для замера памяти (например, jconsole или VisualVM).

2. Создание быстрого Producer (Flux)

- 2.1. Создайте класс FastProducer. Генерируйте бесконечный поток целых чисел с фиксированной задержкой между элементами (например, 1 мс):

```
java
Flux.interval(Duration.ofMillis(1))
    .map(i -> "Item-" + i)
```

- 2.2. Можно также использовать Flux.generate() для эмуляции высокой нагрузки.

- 2.3. Producer не должен иметь внутренних ограничений по скорости – он будет выдавать элементы с максимально возможной скоростью.

3. Создание медленного Consumer (Subscriber)

- 3.1. Реализуйте собственный класс SlowSubscriber<T> implements Subscriber<T>.
- 3.2. В методе onSubscribe(Subscription s) сохраните subscription и запросите request(1) (начальный запрос).
- 3.3. В onNext(T item) имитируйте долгую обработку: Thread.sleep(50). Затем снова вызовите subscription.request(1).

3.4. В `onError` и `onComplete` добавьте логирование.

java

```
public class SlowSubscriber<T> implements Subscriber<T> {  
    private Subscription subscription;  
    private final long delayMs;  
    public SlowSubscriber(long delayMs) { this.delayMs = delayMs; }  
    @Override public void onSubscribe(Subscription s) {  
        this.subscription = s;  
        s.request(1);  
    }  
    @Override public void onNext(T item) {  
        try { Thread.sleep(delayMs); } catch (InterruptedException e) {}  
        System.out.println("Processed: " + item);  
        subscription.request(1);  
    }  
    @Override public void onError(Throwable t) { t.printStackTrace(); }  
    @Override public void onComplete() { System.out.println("Done"); }  
}
```

3.5. Подпишите `FastProducer` на `SlowSubscriber`.

4. Демонстрация проблем без `backpressure` (с буферизацией по умолчанию)

4.1. Запустите пример, где вы подписываете медленного подписчика на быстрый `Flux` без явной настройки `backpressure`.

4.2. По умолчанию `Reactor` буферизирует до 256 элементов (стратегия `onBackpressureBuffer(256)`). Наблюдайте за выводом – через некоторое время (когда буфер заполнится) произойдёт ошибка `OverflowException`.

4.3. Если увеличить буфер до очень большого числа (`onBackpressureBuffer(1000000)`), потребление памяти резко возрастёт – можно измерить через `jconsole`.

4.4. Зафиксируйте, через сколько элементов возникает исключение (или насколько вырастает память).

5. Реализация с явным управлением **backpressure** через **request(n)**

5.1. Используйте созданного выше `SlowSubscriber`, который запрашивает следующий элемент только после обработки предыдущего. Это гарантирует, что `Publisher` не будет генерировать новые элементы до востребования.

5.2. Скорость генерации будет автоматически замедляться до скорости обработки потребителя (50 мс на элемент → 20 элементов/сек).

5.3. Потребление памяти останется стабильным (обычно 1-2 элемента в буфере).

5.4. Запустите тест на 200 элементов и сравните время выполнения (должно быть $\approx 200 * 50 \text{ мс} = 10 \text{ сек}$). Без `backpressure` время не увеличивается, но память растёт.

6. Сравнение различных стратегий **backpressure**

6.1. Реализуйте подписчика, который запрашивает элементы пачками: например, при `onSubscribe` вызывает `request(10)`, а в `onNext` на каждые 10 обработанных элементов запрашивает новые 10.

6.2. Используйте `limitRate(10)` на стороне Flux:

```
java
```

```
flux.limitRate(10).subscribe(slowSubscriber);
```

6.3. Используйте стратегию `onBackpressureDrop()` – лишние элементы отбрасываются.

6.4. Для каждой стратегии измерьте: количество обработанных элементов за фиксированное время (5 сек), потребление памяти (RSS), количество потерянных элементов (для `drop`).

6.5. Постройте таблицу сравнения.

7. Сценарий переполнения с неограниченным буфером (симуляция OOM)

7.1. Создайте `Flux.range(0, 10_000_000).map(i -> new byte[1024])` (каждый элемент – 1 КБ).

7.2. Подпишите на него медленный подписчик (задержка 10 мс).

7.3. Без `backpressure` буфер будет накапливать миллионы элементов → очень быстро наступит `OutOfMemoryError`.

7.4. С включённым `backpressure` (запрос 1 за раз) память остаётся в пределах нескольких МБ.

7.5. Зафиксируйте разницу (можно запускать с ограничением памяти - `Xmx128m`).

8. Анализ результатов и выводы

8.1. Объясните, почему в асинхронных `highload`-системах `backpressure` обязателен.

8.2. Приведите примеры реальных систем, где используется `backpressure` (Apache Kafka, RSocket, gRPC flow control).

8.3. Отметьте, что выбор стратегии (`buffer`, `drop`, `latest`) зависит от требований к надёжности и задержкам.

Пример выполнения задания

Пример реализации Flux с подпиской (Java):

```
Flux<String> fastFlux = Flux.interval(Duration.ofMillis(1))  
    .map(i -> "Event-" + i);
```

```
SlowSubscriber<String> slow = new SlowSubscriber<>(50);  
fastFlux.subscribe(slow);
```

Результаты:

- Без управления `backpressure`: через ~256 элементов выбрасывается `Exceptions.OverflowException`.

- С `slowSubscriber.request(1)`: система стабильна, обрабатывается ~20 элементов/сек, память не растёт.
- С `flux.limitRate(10)`: пакетная обработка, при задержке 50 мс обрабатывается 20 элементов/сек, но сетевые/системные накладные расходы ниже.
- С `onBackpressureDrop()`: обработано около 1000 элементов за 5 секунд (остальные отброшены).

Индивидуальные задания

1. Реализовать `Producer`, который генерирует огромные объекты (например, `byte[8192]`). Показать разницу в использовании памяти между `request(1)` и `request(Integer.MAX_VALUE)`.

2. Использовать `onBackpressureBuffer` с ограничением ёмкости и политикой `BufferOverflowStrategy.ERROR / DROP_OLDEST`. Провести эксперимент.

3. Создать систему из двух потребителей с разной скоростью (один быстрый, один медленный) на одном `Publisher` и использовать `Flux.share()`. Показать, как `backpressure` влияет на обоих.

4. Реализовать не на Java, а на Kotlin с `coroutines` + `Flow` (`kotlinx.coroutines.flow`). Сравнить синтаксис и механизмы `backpressure`.

5. Эмулировать сетевую задержку между `Producer` и `Consumer` (через `delayElements()`). Как это взаимодействует с `backpressure`?

6. Использовать `Flux.push()` для ручной генерации элементов и управления `demand` (через `onRequest`).

7. Добавить мониторинг потребления памяти через `MemoryMXBean` и построить график зависимости памяти от времени для различных стратегий.

8. Сравнить `Project Reactor` с `Akka Streams (Scala)` по удобству работы с `backpressure`.

9. Реализовать протокол `flow control` через TCP (кастомный), где клиент посылает «window size» серверу. Сопоставить с `Reactive Streams`.

10. Создать приложение, обрабатывающее события из Kafka с использованием Reactor Kafka (библиотека поддерживает backpressure).

11. Сымитировать всплеск нагрузки: Producer внезапно начинает генерировать 10000 событий/сек, Consumer остаётся медленным. Показать работу onBackpressureLatest().

12. Реализовать кастомную стратегию backpressure, которая обрабатывает элементы в пакетах по времени (например, каждые 100 мс обрабатывает до 10 элементов).

13. Измерить пропускную способность (блок/сек) для разных стратегий при Thread.sleep (блокирующий воркер) и асинхронном Mono.delay (неблокирующая задержка). Объяснить разницу.

14. Использовать Scheduler.newParallel для Consumer, чтобы обработка велась в отдельном пуле потоков. Как это влияет на backpressure?

15. Интегрировать пример из ЛР 10 в существующее приложение из ЛР 1 (неблокирующий сервер) и добавить backpressure при отправке ответов клиенту через Flux<DataBuffer>.

Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – определение backpressure, спецификация Reactive Streams, реализации (Project Reactor), стратегии управления;
- раздел «Архитектура эксперимента» – описание Producer, Consumer, схемы передачи данных, выбранных параметров;
- раздел «Реализация» – листинги кода (FastProducer, SlowSubscriber, настройка Flux для разных стратегий);
- раздел «Эксперименты и результаты» – таблицы и графики:
 - время выполнения и память без backpressure (падение с ошибкой или OOM);

- время выполнения и память с `request(1)`;
- сравнение `limitRate`, `buffer`, `drop`, `latest` по обработанным и потерянным элементам;
- демонстрация OOM-сценария.
- раздел «Анализ» – объяснение, почему `backpressure` критичен для асинхронных `highload`-систем, как выбрать стратегию;
- выводы;
- ответы на контрольные вопросы.

Обязательны скриншоты падения (`StackOverflowError` или `OOM`) и стабильной работы. Все графики и таблицы подписаны. Код должен быть выложен в репозиторий или приложен архивом.

Контрольные вопросы

1. Что такое `backpressure`? Почему он возникает в потоковых системах?
2. Как механизм `backpressure` в `Reactive Streams` отличается от простого увеличения очереди?
3. Какой метод `Subscriber` отвечает за передачу `demand` издателю?
4. Что произойдёт, если `Subscriber` не вызывает `request(n)` после подписки?
5. Как стратегии `onBackpressureBuffer`, `onBackpressureDrop`, `onBackpressureLatest` отличаются по поведению?
6. Может ли `backpressure` привести к `deadlock`? Приведите пример.
7. Как в `Reactor` работают операторы `limitRate` и `limitRequest`?
8. Почему в традиционных многопоточных очередях (например, `LinkedBlockingQueue`) `backpressure` не встроен и требует ручного контроля?
9. Как асинхронные драйверы БД и HTTP-клиенты (например, `R2DBC`, `WebClient`) используют `backpressure`?

10. Какие проблемы могут возникнуть при использовании `backpressure` в распределённой системе с сетью?
11. Как измерять уровень `backpressure` (например, текущий `demand`) в реальном приложении?
12. Для каких сценариев подходит `onBackpressureBuffer`, а для каких —

Лабораторная работа №11.

Асинхронная обработка через Kafka (consumer groups, offset)

Цель работы: настроить асинхронное взаимодействие между производителем и потребителями Apache Kafka с использованием партиций, групп потребителей и ручного управления смещениями (offset), а также смоделировать отказ consumer и проверить гарантию доставки at-least-once.

Задачи:

- развернуть Kafka (одну ноду) с использованием Docker;
- создать топик с несколькими партициями (3–6);
- реализовать производителя, который отправляет сообщения с ключами для распределения по партициям;
- реализовать группу потребителей (2–3 экземпляра), обрабатывающих сообщения с ручным подтверждением offset;
- симитировать падение одного из потребителей и продемонстрировать перебалансировку (rebalance) и повторное чтение необработанных сообщений;
- убедиться в соблюдении семантики at-least-once (сообщение не потеряется, но может быть обработано более одного раза);
- измерить пропускную способность и оценить влияние размера партиций и размера группы на скорость обработки.

Теоретическая часть

Apache Kafka – распределённая платформа потоковой передачи событий с высокой пропускной способностью, устойчивостью и низкой задержкой. Основные абстракции:

- **Топик (topic)** – логический канал для сообщений. Разделяется на **партиции (partitions)** – упорядоченные, неизменяемые последовательности сообщений.

- **Producer** – публикует сообщения в топик. Может указывать ключ; сообщения с одинаковым ключом попадают в одну партицию.
- **Consumer** – читает сообщения из одной или нескольких партиций. Входит в **группу потребителей (consumer group)**. Каждая партиция топика обслуживается только одним consumer из группы, что позволяет масштабировать обработку.
- **Offset (смещение)** – уникальный идентификатор сообщения внутри партиции. Consumer сохраняет текущий offset (где остановился).
- **Commit offset** – процесс сохранения смещения. Может быть автоматическим (периодически) или ручным (после успешной обработки).

Гарантии доставки:

- *At-most-once* – сообщение может быть потеряно, но не обработано повторно.
- *At-least-once* – сообщение не будет потеряно, но может быть обработано несколько раз (стандарт для Kafka с ручным коммитом после обработки).
- *Exactly-once* – достигается с использованием транзакций Kafka или идемпотентного продюсера, сложнее.

Перебалансировка (rebalance) – процесс перераспределения партиций между consumers в группе при добавлении или удалении consumer (или отказе). Во время перебалансировки группа временно останавливается. Для минимизации простоев используют стратегии Cooperative (инкрементальная перебалансировка, начиная с Kafka 2.4+).

Ручное управление offset позволяет гарантировать at-least-once: consumer обрабатывает сообщение, сохраняет результат, *только затем* фиксирует offset. Если consumer падает после обработки, но до коммита, тот же offset будет прочитан снова при перебалансировке.

В работе используется Kafka, развёрнутая в Docker (с Zookeeper) или отдельно. Клиент – на Java (библиотека kafka-clients) или Python (kafka-

python, confluent-kafka). Основной упор: ручное управление offset, симуляция сбоя и наблюдение за поведением.

Методика выполнения работы

1. Подготовка окружения

1.1. Создайте каталог lab11_kafka.

1.2. Создайте файл docker-compose.yml для запуска Kafka и Zookeeper:

```
version: '3'
```

```
services:
```

```
  zookeeper:
```

```
    image: confluentinc/cp-zookeeper:latest
```

```
    environment:
```

```
      ZOOKEEPER_CLIENT_PORT: 2181
```

```
  kafka:
```

```
    image: confluentinc/cp-kafka:latest
```

```
    depends_on:
```

```
      - zookeeper
```

```
    ports:
```

```
      - "9092:9092"
```

```
    environment:
```

```
      KAFKA_BROKER_ID: 1
```

```
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

```
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
```

```
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

1.3. Запустите: `docker-compose up -d`.

1.4. Установите клиентскую библиотеку:

- Python: `pip install kafka-python` или `confluent-kafka`.

- Java: добавьте org.apache.kafka:kafka-clients:3.5.x.

1.5. Создайте топик с 3 партициями (через команду или код):

```
docker exec -it <kafka_container> kafka-topics --create --topic test-topic --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1
```

2. Реализация производителя (Producer)

2.1. Напишите скрипт producer.py, который отправляет 1000 сообщений.

2.2. Каждое сообщение – JSON: {"order_id": i, "timestamp": ...}.

2.3. Ключ сообщения – order_id % num_partitions или строка, чтобы распределить по партициям.

2.4. Настройте acks=all для подтверждения записи на брокер.

```
from kafka import KafkaProducer
```

```
import json, time
```

```
producer = KafkaProducer(bootstrap_servers='localhost:9092',  
                           value_serializer=lambda v: json.dumps(v).encode('utf-8'),  
                           key_serializer=str.encode)
```

```
for i in range(1000):
```

```
    key = f"key_{i % 3}" #равномерно по 3 партициям
```

```
    value = {"order_id": i, "msg": f"msg_{i}"}
```

```
    producer.send('test-topic', key=key, value=value)
```

```
    time.sleep(0.01) #задержка для управляемого потока
```

```
producer.flush()
```

3. Реализация потребителя (Consumer) с ручным commit offset

3.1. Создайте потребителя, входящего в группу test-group.

3.2. Отключите автоматический коммит: enable_auto_commit=False.

3.3. Подпишитесь на топик test-topic.

3.4. В цикле чтения сообщений:

- обработайте сообщение (имитация: time.sleep(0.05), вывод на печать).

- после успешной обработки вызовите consumer.commit() (синхронно) или commit_async().

3.5. Для симуляции ошибки в обработке: при определённых значениях (например, `order_id % 20 == 0`) вызывайте исключение, НЕ делайте КОММИТ.

```
from kafka import KafkaConsumer

consumer = KafkaConsumer('test-topic', group_id='test-group',
                          bootstrap_servers='localhost:9092',
                          enable_auto_commit=False,
                          auto_offset_reset='earliest')

for msg in consumer:
    data = json.loads(msg.value)
    print(f'Partition {msg.partition}, offset {msg.offset}: {data}')
    if data['order_id'] % 20 == 0:
        print("Simulating processing error, not committing")
        continue # сообщение не будет закоммичено
    # успешная обработка
    consumer.commit()
```

4. Симуляция отказа и перебалансировки

- 4.1. Запустите 2 экземпляра потребителя (разные терминалы).
- 4.2. Запустите производителя, дождитесь отправки всех сообщений.
- 4.3. В процессе работы остановите один из потребителей (Ctrl+C).
- 4.4. Наблюдайте в логах второго потребителя: он должен получить сообщения из партиций, ранее принадлежавших остановленному (rebalance).
- 4.5. Убедитесь, что сообщения, не закоммиченные упавшим потребителем, будут прочитаны повторно (проверьте дублирование).

5. Проверка гарантии at-least-once

- 5.1. Создайте отдельный тест: производитель отправляет 500 сообщений.

- 5.2. Потребитель обрабатывает, но перед коммитом эмулирует краш (например, `sys.exit(1)`).
- 5.3. Запустите второго потребителя в той же группе.
- 5.4. Подсчитайте, сколько уникальных сообщений было обработано суммарно. Потерянных быть не должно, но часть может обработаться дважды.
- 5.5. В отчёте приведите логи и объясните, почему это at-least-once.
- 6. Измерение пропускной способности**
- 6.1. Напишите скрипт, измеряющий время обработки 10000 сообщений для разного числа партиций (1, 3, 6) и количества consumers в группе (1, 2, 3).
- 6.2. Используйте ручной коммит каждые N сообщений (например, раз в 100) для снижения накладных расходов.
- 6.3. Постройте графики: RPS от числа consumers, RPS от числа партиций.
- 6.4. Сделайте вывод о том, почему количество партиций должно быть $\geq$ количеству consumers в группе.
- 7. Дополнительно: стратегии назначения партиций**
- 7.1. Изучите и примените разные стратегии `partition.assignment.strategy`: `RangeAssignor`, `RoundRobinAssignor`, `CooperativeStickyAssignor`.
- 7.2. Сравните перебалансировку при падении consumer: с `Cooperative` vs `Range` (последняя приводит к полной остановке группы).

Пример выполнения задания

Логи успешной обработки с ручным коммитом:

[Consumer-1] Started, assigned partitions: [test-topic-0, test-topic-2]

[Consumer-1] Processed: partition 0 offset 10, commit

[Consumer-2] Started, assigned partitions: [test-topic-1]

[Producer] Sent 1000 messages

[Consumer-2] Processed: partition 1 offset 25, commit

... (Consumer-1 падает) ...

[Consumer-2] Rebalance triggered, new assignment: [test-topic-0, test-topic-1, test-topic-2]

[Consumer-2] Re-processing partition 0 from offset 11 (not committed by dead consumer)

Вывод: Сообщения с offset 11 и выше в partition 0 были обработаны повторно, но ни одно не потеряно.

Индивидуальные задания

1. Реализовать идемпотентного потребителя: сохранять обработанные ID сообщений в Redis и при повторном получении игнорировать их.

2. Вместо ручного коммита на каждое сообщение использовать `commit` раз в секунду или каждые 100 сообщений. Измерить влияние на пропускную способность.

3. Использовать `commit_async(callback)` с обработкой ошибок коммита (например, при перебалансировке).

4. Добавить паузу (throttling) в потребителе, если обработка слишком быстрая, чтобы ограничить использование CPU.

5. Реализовать производителя с ключом, основанным на `user_id`, и показать, что все сообщения одного пользователя попадают в одну партицию.

6. Расширить эксперимент на кластер Kafka из 3 брокеров (в Docker Compose) и настроить репликацию факторов 2.

7. Заменить стандартную сериализацию на Avro с использованием Schema Registry.

8. Сравнить поведение при автоматическом коммите (`enable_auto_commit=True`) и ручном – при отказе потребителя.

9. Реализовать `exactly-once` с помощью транзакций Kafka: продюсер использует `initTransactions()`, consumer – `isolation.level=read_committed`.

10. Написать тест на конкурентное добавление и удаление consumers (через API администратора) и измерить время перебалансировки.

11. Интегрировать Kafka с базой данных: после обработки сообщения обновлять PostgreSQL, и коммитить offset только после успешного обновления БД.

12. Реализовать Dead Letter Queue для сообщений, которые не удаётся обработать после N попыток.

13. Использовать Kafka Streams для простой агрегации (например, подсчёт сообщений по ключу) и сравнить с ручным consumer.

14. Сымитировать сетевую задержку между consumer и брокером (через tc netem) и измерить влияние на задержки коммита.

15. Перенести реализацию на язык Go (библиотека Sarama) и сравнить производительность с Python/Java.

Требования к отчёту

Отчёт оформляется в электронном виде (PDF) и содержит:

- титульный лист;
- цель работы и задачи;
- теоретическая часть – описание Kafka (топики, партиции, группы, offset), гарантии доставки, ручной commit;
- раздел «Архитектура и настройка» – схема развёртывания (Docker), параметры топика, конфигурация клиентов;
- раздел «Реализация» – листинги producer и consumer с комментариями;
- раздел «Эксперименты»:
 - результаты работы producer (количество сообщений по партициям);
 - логи работы двух consumers, ребалансировка при падении;
 - проверка at-least-once (подсчёт обработанных сообщений, выявление дублей);

- графики пропускной способности в зависимости от партиций и consumers;
 - сравнение стратегий назначения партиций;
- раздел «Анализ» – объяснение, почему ручной commit даёт at-least-once, проблема дублирования и как её решать (идемпотентность), ограничения Kafka;
- выводы;
- ответы на контрольные вопросы.

Код должен быть доступен в репозитории. Скриншоты логов и дашбордов (например, в Kafdrop или UI) приветствуются.

Контрольные вопросы

1. Зачем в Kafka нужны партиции и как они влияют на параллелизм потребления?
2. Что такое группа потребителей и как партиции распределяются внутри группы?
3. Что такое offset? Какие бывают стратегии коммита offset?
4. Почему автоматический коммит (auto commit) не гарантирует at-least-once?
5. Как ручной коммит помогает добиться at-least-once? Какая цена этого подхода?
6. Что происходит во время ребалансировки потребителей в группе?
7. Как можно добиться exactly-once в Kafka?
8. Почему количество партиций должно быть не меньше количества потребителей в группе?
9. Как проверить, что сообщение обработано только одним consumer в группе?
10. Как сериализовать сложные объекты (например, Protobuf) для Kafka?
11. Какие метрики важны для мониторинга consumer lag?
12. В чём разница между acks=0, acks=1 и acks=all у producer?

13. Может ли consumer читать сообщения из конкретной партиции вручную (помимо подписки)?

14. Какие гарантии даёт Kafka для порядка сообщений в одной партиции?

15. Как откатить offset consumer'a для повторного чтения старых сообщений?

Лабораторная работа №12.

Load balancing симуляция: взвешенный RR + health checks

Цель работы: реализовать программный балансировщик нагрузки, поддерживающий взвешенный алгоритм round-robin (SWRR) и динамическое исключение бэкендов на основе проверок их здоровья (health checks), а также проанализировать влияние весов на распределение запросов и отказоустойчивость.

Задачи:

- создать несколько бэкенд-серверов (3 штуки) с симулированной разной производительностью (различные веса);
- реализовать балансировщик, который распределяет входящие запросы по алгоритму взвешенного round-robin (Smooth Weighted Round Robin);
- добавить механизм health checks (периодические запросы на /health эндпоинт бэкендов);
- при обнаружении недоступного бэкенда временно исключить его из ротации, а при восстановлении – вернуть обратно;
- провести нагрузочное тестирование, сравнить распределение запросов с теоретическими весами;
- проанализировать поведение системы при отказе одного из бэкендов.

Теоретическая часть

Балансировка нагрузки – распределение входящих запросов между несколькими серверами с целью повышения пропускной способности, отказоустойчивости и масштабируемости.

Алгоритмы балансировки:

- **Round Robin (RR)** – последовательное распределение запросов по кругу.

- **Weighted Round Robin (WRR)** – каждому серверу назначается вес, пропорционально которому распределяются запросы. Однако классический WRR может создавать сгустки запросов к одному серверу.
- **Smooth Weighted Round Robin (SWRR)** – улучшенная версия, используемая в Nginx. Обеспечивает более плавное распределение, избегая последовательных вызовов одного тяжелого сервера. Алгоритм поддерживает текущие веса (`current_weight`) и обновляет их на каждом шаге.

Алгоритм SWRR (описание):

1. Для каждого сервера заданы `weight` (фиксированный вес) и `effective_weight`, `current_weight` (оба инициализируются нулём).
2. При выборе сервера:
 - суммируются веса всех серверов в `total_weight`;
 - для каждого сервера `current_weight += effective_weight`;
 - выбирается сервер с максимальным `current_weight`;
 - `current_weight` выбранного сервера уменьшается на `total_weight`;
 - возвращается выбранный сервер.
3. Вес `effective_weight` может динамически уменьшаться при ошибках и восстанавливаться при успешных проверках здоровья.

Health Checks – механизм мониторинга состояния бэкендов. Бывают:

- **Активные** – балансировщик периодически отправляет запросы (например, HTTP GET /health) и ожидает ответ 2xx.
- **Пассивные** – анализ ответов на реальные запросы (ошибки 5xx, таймауты).

Стратегии исключения:

- **Fail-fast** – немедленно перестать направлять запросы на недоступный бэкенд.
- **Graceful degradation** – снизить вес до нуля и постепенно его восстанавливать.

В данной работе реализуем активные health checks с периодом 2 секунды. При двух последовательных неудачах бэкенд помечается как down и исключается из списка выбора. После восстановления (успешный health check) он возвращается с исходным весом.

Метрики для оценки:

- Коэффициент соответствия распределения запросов заданным весам (отклонение).
- Время обнаружения отказа (health check latency).
- Количество запросов, отправленных на отказавший бэкенд (в период между отказом и обнаружением).

Методика выполнения работы

1. Подготовка проекта

- 1.1. Создайте каталог lab12_load_balancer.
- 1.2. Язык – Python (Flask или FastAPI для бэкендов, aiohttp для балансировщика) или Go/Java.
- 1.3. Установите requests, flask, threading (встроенные).
- 1.4. Для нагрузочного теста используйте wrk или напишите простой клиент на asyncio.

2. Реализация бэкенд-серверов

- 2.1. Создайте три Flask-приложения на разных портах (5001, 5002, 5003).
- 2.2. Каждый сервер имеет эндпоинт / (основной) и /health (проверка здоровья).
- 2.3. Сервер может иметь параметр weight (например, 1, 2, 5). Имитируйте разную производительность через time.sleep(1 / weight) или случайную задержку.
- 2.4. Добавьте возможность ручного отключения сервера (через флаг или эндпоинт /fail), чтобы симулировать падение.

```
# backend.py
```

```
from flask import Flask
```

```

import time, os

app = Flask(__name__)

weight = int(os.getenv("WEIGHT", 1))

@app.route('/')
def index():
    time.sleep(0.1 / weight) # чем больше вес, тем быстрее ответ
    return f'Backend {os.getenv('PORT')} (weight={weight})'

@app.route('/health')
def health():
    return "OK", 200

```

3. Реализация балансировщика (Smooth Weighted RR)

3.1. Создайте класс SmoothWeightedBalancer.

3.2. Инициализация: список бэкендов (host, port, weight). Для каждого храним `current_weight = 0`, `effective_weight = weight`, `is_healthy = True`.

3.3. Метод `next_server()`:

```

def next_server(self):
    total = 0
    best = None
    for backend in self.backends:
        if not backend.is_healthy:
            continue
        backend.current_weight += backend.effective_weight
        total += backend.effective_weight
        if best is None or backend.current_weight > best.current_weight:
            best = backend
    if best is None:
        raise Exception("No healthy backends")
    best.current_weight -= total
    return best

```

3.4. Запустите балансировщик на порту 8080. При получении запроса GET /proxy?path=... перенаправляйте его на выбранный бэкенд (через requests.get).

3.5. Важно: обрабатывайте таймауты и ошибки подключения; при ошибке временно снижайте effective_weight или помечайте бэкенд как нездоровый.

4. Добавление health checks

4.1. Запустите фоновый поток (или threading.Timer), который каждые 2 секунды вызывает check_health() для каждого бэкенда.

4.2. Отправляйте GET на http://backend:port/health. Если ответ не 2xx или таймаут – увеличивайте счётчик неудач. При достижении failure_threshold=2 установите is_healthy=False.

4.3. Если бэкенд был недоступен, но health check стал успешным – восстановите is_healthy=True, сбросьте счётчик неудач, верните effective_weight к исходному весу.

4.4. При восстановлении бэкенда не сбрасывайте current_weight – алгоритм сам скорректирует распределение.

5. Интеграция и тестирование без отказов

5.1. Запустите три бэкенда с весами (1, 2, 5). Запустите балансировщик.

5.2. Отправьте 1000 запросов через балансировщик (можно с помощью wrk -t1 -c1 -d 30s <http://localhost:8080/proxy>).

5.3. На стороне бэкендов подсчитайте количество полученных запросов (через глобальные счётчики).

5.4. Сравните фактическое распределение с теоретическим (1:2:5). Вычислите отклонение в процентах.

6. Симуляция отказа и восстановления

6.1. Вручную выключите один из бэкендов (например, остановите Flask-процесс).

6.2. Продолжайте отправлять запросы через балансировщик.

6.3. Через некоторое время (не более 2 секунд + время проверки) балансировщик должен перестать направлять трафик на упавший бэкенд. Распределение должно переключиться на оставшиеся здоровые серверы (с сохранением соотношения их весов).

6.4. Логируйте момент обнаружения отказа и перераспределение.

6.5. Восстановите бэкенд – убедитесь, что он снова получает трафик.

7. Измерение метрик

7.1. Добавьте в балансировщик эндпоинт /stats, возвращающий:

- текущее состояние каждого бэкенда (вес, healthy, количество запросов за всё время).
- общее количество запросов, распределение по бэкендам.

7.2. Постройте графики распределения запросов для сценариев:

- все бэкенды здоровы;
- один бэкенд отключён;
- бэкенд восстановлен.

7.3. Рассчитайте точность распределения: $(\text{expected} - \text{actual}) / \text{expected} * 100\%$.

8. Опционально: поддержка нескольких алгоритмов

8.1. Реализуйте также простой Weighted Round Robin (без smooth) и сравните с SWRR на предмет равномерности распределения в краткосрочных сессиях.

8.2. Добавьте алгоритм Least Connections и сравните с SWRR при разном времени ответа бэкендов.

Пример выполнения задания

Фрагмент реализации балансировщика (Python):

```
import threading, time, requests
from flask import Flask, request, jsonify

class Backend:
```

```
def __init__(self, host, port, weight):
```

```
    self.host = host
```

```
    self.port = port
```

```
    self.weight = weight
```

```
    self.effective_weight = weight
```

```
    self.current_weight = 0
```

```
    self.healthy = True
```

```
    self.failures = 0
```

```
    self.total_requests = 0
```

```
app = Flask(__name__)
```

```
backends = [Backend("localhost", 5001, 1), Backend("localhost", 5002, 2),  
Backend("localhost", 5003, 5)]
```

```
def health_check():
```

```
    while True:
```

```
        for b in backends:
```

```
            try:
```

```
                r = requests.get(f"http://{b.host}:{b.port}/health", timeout=1)
```

```
                if r.status_code == 200:
```

```
                    if not b.healthy:
```

```
                        print(f"Backend {b.port} recovered")
```

```
                        b.healthy = True
```

```
                        b.failures = 0
```

```
                        b.effective_weight = b.weight
```

```
                    else:
```

```
                        b.failures += 1
```

```
            except:
```

```
                b.failures += 1
```

```
            if b.failures >= 2 and b.healthy:
```

```

        print(f"Backend {b.port} marked unhealthy")
        b.healthy = False
    time.sleep(2)

def next_backend():
    total = 0
    best = None
    for b in backends:
        if not b.healthy:
            continue
        b.current_weight += b.effective_weight
        total += b.effective_weight
        if best is None or b.current_weight > best.current_weight:
            best = b
    if best is None:
        return None
    best.current_weight -= total
    return best

@app.route('/proxy')
def proxy():
    backend = next_backend()
    if not backend:
        return "No healthy backends", 503
    backend.total_requests += 1
    try:
        resp = requests.get(f"http://{backend.host}:{backend.port}/",
timeout=3)
        return resp.text, resp.status_code
    except:

```

```

        backend.failures += 1

    if backend.failures >= 2:
        backend.healthy = False
        return "Backend error", 502

if __name__ == '__main__':
    threading.Thread(target=health_check, daemon=True).start()
    app.run(port=8080)

```

Результаты теста (веса 1:2:5, 800 запросов):

- Бэкенд 5001: 98 запросов (ожидалось 100) – отклонение 2%
- Бэкенд 5002: 201 запрос (200) – 0.5%
- Бэкенд 5003: 501 запрос (500) – 0.2%

После отключения 5003: распределение 5001:5002 $\approx$ 1:2.

Индивидуальные задания

1. Реализовать балансировщик с поддержкой алгоритма **Least Connections** (выбор бэкенда с наименьшим числом активных соединений). Сравнить с SWRR.

2. Добавить **пассивные health checks**: анализировать ошибки ответов (таймауты, 5xx) и при трёх последовательных ошибках отключать бэкенд.

3. Реализовать **градуальное восстановление**: при возвращении бэкенда сначала давать ему маленький вес (например, 20% от исходного) и постепенно увеличивать до полного.

4. Использовать **взвешенный round robin с фиксированным распределением** (каждые 100 запросов пропорционально весам) и сравнить с SWRR по плавности.

5. Добавить в балансировщик поддержку **конфигурации через YAML/JSON** (веса, порты, интервалы health check).

6. Реализовать **экспорт метрик в Prometheus** (количество запросов на бэкенд, статусы здоровья, общее число ошибок).
7. Вместо HTTP-бэкендов использовать TCP-echo серверы (простое сокетное приложение). Адаптировать балансировщик.
8. Создать **UI на FastAPI + WebSocket**, показывающий в реальном времени текущее состояние бэкендов и распределение запросов.
9. Добавить **поддержку липкости (sticky sessions)** на основе cookie или IP-хеша и исследовать конфликт с взвешенным RR.
10. Реализовать **динамическое изменение весов** в зависимости от загрузки CPU бэкенда (получать через метрики агента).
11. Сымитировать **замедление (latency spike)** на одном бэкенде (без полного отказа) и реализовать автоматическое снижение его веса.
12. Реализовать балансировщик на языке Go с использованием горутин и каналов. Измерить производительность в сравнении с Python.
13. Добавить поддержку **TLS-termination** на балансировщике и HTTPS-запросы к бэкендам.
14. Написать **нагрузочный тест** (кастомный клиент), который отправляет запросы и записывает задержки. Построить графики p99 latency для разных алгоритмов.
15. Интегрировать балансировщик с Docker и Docker Compose, запуская бэкенды как отдельные контейнеры, а балансировщик как ещё один контейнер.

Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- **цель работы и задачи**;
- **теоретическая часть** – описание алгоритмов RR, WRR, SWRR, механизмов health checks;
- **раздел «Архитектура и реализация»** – схема взаимодействия, описание классов, алгоритм выбора бэкенда, health check loop;

- **раздел «Реализация»** – листинги кода (бэкенды, балансировщик, health check) с комментариями;
- **раздел «Эксперименты»:**
 - результаты распределения запросов при всех здоровых бэкендах (таблица фактических vs ожидаемых количеств, график);
 - логи переключения при отказе бэкенда (с временными метками);
 - распределение после отказа и после восстановления;
 - сравнение с алгоритмом без весов (обычный RR);
- **раздел «Анализ»** – преимущества SWRR перед WRR, время детекции отказа, влияние health check интервала, возможные улучшения;
- **выводы** – достигнута ли цель, применимость подхода в высоконагруженных системах;
- **ответы на контрольные вопросы.**

Отчёт в PDF, код в архиве или репозитории.

Контрольные вопросы

1. В чём разница между round-robin и взвешенным round-robin?
2. Как работает алгоритм Smooth Weighted Round Robin? Почему он лучше классического WRR?
3. Какие существуют виды health checks (активные, пассивные)? Их преимущества и недостатки.
4. Как часто следует выполнять health checks в высоконагруженной системе?
5. Что произойдёт, если все бэкенды будут помечены как unhealthy? Как балансировщик должен это обработать?
6. Как можно избежать отправки запросов на отказавший бэкенд в период между отказом и обнаружением через health check?
7. Влияет ли использование current_weight на долгосрочную точность распределения?

8. Как адаптировать алгоритм SWRR для случая, когда веса изменяются динамически (например, при частичном отказе)?

9. Какие метрики (наблюдаемость) вы бы добавили в production-балансировщик?

10. Как можно реализовать «липкие сессии» вместе с взвешенным RR? Какие проблемы это создаёт?

11. Чем отличается балансировка на уровне L4 (TCP) от L7 (HTTP)? В каком случае какой подход предпочтительнее?

12. Как в вашей реализации гарантируется потокобезопасность при работе нескольких потоков/запросов одновременно?

Лабораторная работа №13.

Кэш с write-through и TTL (Redis + приложение)

Цель работы: реализовать стратегию кэширования write-through (сквозная запись) с использованием Redis в качестве кэша и реляционной базы данных (например, PostgreSQL) в качестве постоянного хранилища, а также применить TTL (время жизни) со случайным разбросом для предотвращения одновременной инвалидации большого количества ключей (эффект «stampeding herd »).

Задачи:

- развернуть Redis и PostgreSQL (в Docker или локально);
- реализовать простое приложение (на Python/Java/Go), которое предоставляет CRUD-операции для сущности (например, «Товар» или «Пользователь»);
- реализовать стратегию write-through: при записи сначала обновить кэш в Redis, затем асинхронно (или синхронно, но с фокусом на производительность) записать в БД;
- при чтении сначала проверять наличие данных в Redis, при промахе – загружать из БД и помещать в кэш;
- установить TTL для ключей кэша с добавлением случайной величины (jitter) к базовому значению;
- измерить процент попаданий в кэш, время ответа и нагрузку на БД;
- продемонстрировать, как случайный TTL предотвращает одновременное истечение срока действия большого числа ключей и лавинную нагрузку на БД.

Теоретическая часть

Кэширование – временное хранение часто используемых данных в быстром хранилище (обычно in-memory) для снижения задержек и нагрузки

на основное хранилище (БД). В высоконагруженных системах применяются различные стратегии работы с кэшем.

Стратегии записи:

- **Write-through (сквозная запись):** при записи данные сначала сохраняются в кэш, затем синхронно (или асинхронно) в БД. Преимущество: кэш всегда согласован с БД. Недостаток: задержка записи увеличивается.
- **Write-behind (write-back):** данные сначала пишутся в кэш, а в БД асинхронно с задержкой (пакетно). Лучшая производительность записи, но возможна потеря данных при падении кэша.
- **Write-around:** запись идёт сразу в БД, минуя кэш. При чтении после записи возможен промах кэша.

В данной работе используется **write-through с асинхронной записью в БД** (компромисс). Приложение синхронно записывает в Redis и сразу возвращает ответ клиенту, а сохранение в БД ставится в очередь или выполняется в фоновом потоке.

TTL (Time To Live) – время жизни ключа в кэше. По истечении TTL ключ автоматически удаляется. При следующем чтении произойдёт промах кэша и загрузка из БД.

Проблема «одновременной инвалидации» (stampeding herd / thundering herd): если множество ключей имеют одинаковый TTL и истекают одновременно, то в момент их удаления множество запросов одновременно обратятся к БД, создавая пиковую нагрузку, которая может привести к отказу БД.

Решение – jitter (случайный разброс):

- К базовому TTL добавляется случайная величина (например, от 0 до 30% от базового).
- Ключи истекают в разные моменты времени, пиковая нагрузка на БД сглаживается.

Метрики оценки:

- **Cache hit ratio** – доля запросов, обслуженных из кэша.
- **Latency (p50, p99)** – задержка чтения и записи.
- **Нагрузка на БД (RPS)** – число запросов к БД.

Методика выполнения работы

1. Подготовка окружения

1.1. Создайте каталог lab13_cache.

1.2. Запустите Redis и PostgreSQL через Docker Compose:

```
version: '3'
```

```
services:
```

```
  redis:
```

```
    image: redis:7
```

```
    ports:
```

```
      - "6379:6379"
```

```
  postgres:
```

```
    image: postgres:13
```

```
    environment:
```

```
      POSTGRES_USER: test
```

```
      POSTGRES_PASSWORD: test
```

```
      POSTGRES_DB: cache_db
```

```
    ports:
```

```
      - "5432:5432"
```

1.3. Выполните `docker-compose up -d`.

1.4. Установите клиентские библиотеки:

Python: `pip install redis pycpg2-binary flask` (или `fastapi`, `uvicorn`).

1.5. Создайте таблицу в PostgreSQL:

```
CREATE TABLE items (
```

```
  id SERIAL PRIMARY KEY,
```

```
  key VARCHAR(255) UNIQUE NOT NULL,
```

```
value TEXT NOT NULL,  
updated_at TIMESTAMP DEFAULT NOW()  
);
```

2. Реализация базового приложения без кэша (для сравнения)

2.1. Напишите простое HTTP-приложение с эндпоинтами:

- GET /item/<key> – чтение из PostgreSQL.
- POST /item – запись (key, value) в PostgreSQL.

2.2. Замерьте среднюю задержку при 1000 запросах (wrk или ab).
Зафиксируйте RPS БД.

3. Реализация кэширования с write-through + асинхронная запись в БД

3.1. Создайте класс CacheStore, инкапсулирующий логику:

- get(key) – попытка чтения из Redis. Если нет – загрузить из PostgreSQL, записать в Redis с TTL и вернуть.
- set(key, value) – записать в Redis синхронно, затем поставить задачу в очередь (или запустить поток) для записи в PostgreSQL.

3.2. Используйте Redis команды: SET, GET, EXPIRE.

3.3. Для асинхронной записи в БД используйте threading.Thread или asyncio.create_task (или простую очередь queue.Queue с воркером).

```
import threading, queue, psycopg2
```

```
write_queue = queue.Queue()
```

```
def db_writer_worker():
```

```
    while True:
```

```
        key, value = write_queue.get()
```

```
        conn = psycopg2.connect(...)
```

```
        cur = conn.cursor()
```

```
        cur.execute("INSERT INTO items (key, value) VALUES (%s, %s) ON  
CONFLICT (key) DO UPDATE SET value = EXCLUDED.value, updated_at =  
NOW()", (key, value))
```

```
conn.commit()
```

```
cur.close()
```

```
conn.close()
```

```
threading.Thread(target=db_writer_worker, daemon=True).start()
```

3.4. При `set(key, value)` после записи в Redis вызывайте `write_queue.put((key, value))`.

4. Добавление TTL со случайным разбросом (jitter)

4.1. Определите базовый TTL, например, `base_ttl = 60` секунд.

4.2. При записи в Redis (и при загрузке после промаха) вычисляйте `ttl = base_ttl + random.uniform(-0.2, 0.2) * base_ttl` (разброс $\pm 20\%$).

4.3. Используйте `redis.setex(key, int(ttl), value)`.

4.4. Для демонстрации эффекта создайте тест, который записывает 1000 ключей в кэш с одинаковым базовым TTL (без jitter) и с jitter. Затем после истечения TTL измерьте пиковую нагрузку на БД (число одновременных запросов к БД).

5. Тестирование hit ratio и производительности

5.1. Напишите скрипт, который выполняет 70% операций чтения и 30% записи (рабочая нагрузка, типичная для многих приложений).

5.2. Прогрейте кэш: выполните 500 чтений различных ключей, чтобы заполнить кэш.

5.3. Затем в течение 2 минут выполняйте случайные операции.

5.4. Собирайте статистику: сколько чтений обслужено из Redis (hit), сколько из БД (miss).

5.5. Вычислите hit ratio. Затем отключите кэш и сравните RPS и латентность.

6. Симуляция эффекта «thundering herd»

6.1. Создайте 1000 ключей с `TTL = 5` секунд (без jitter). В отдельном потоке постоянно делайте чтения всех этих ключей (например, каждые 100 мс).

6.2. Наблюдайте, как каждые 5 секунд одновременно истекает срок действия всех ключей, и БД получает лавину запросов (1000 одновременных SELECT).

6.3. Повторите с jitter (TTL = 5 ± 1 секунда). Убедитесь, что пиковая нагрузка на БД снижается (запросы распределяются во времени).

6.4. Постройте график количества запросов к БД по времени для обоих случаев.

7. Измерение задержек

7.1. Используйте `time.perf_counter()` для замера времени выполнения чтения и записи.

7.2. Сравните:

- Чтение из кэша (попадание) – должно быть < 5 мс.
- Чтение с промахом (БД + запись в кэш) – может быть 20–50 мс.
- Запись (Redis + очередь) – задержка записи в Redis мала, клиент получает ответ быстро; асинхронная запись в БД не влияет на клиентскую задержку.

7.3. Постройте гистограммы.

8. Дополнительно: обработка отказа Redis

8.1. Сымитируйте отказ Redis (остановите контейнер). Приложение должно прозрачно переключаться на прямое чтение/запись в БД.

8.2. При восстановлении Redis – синхронизировать кэш (можно инвалидировать все ключи или использовать версионирование).

Пример выполнения задания

Код чтения с кэшированием (псевдокод):

```
def get_item(key):  
    # попытка чтения из Redis  
    cached = redis_client.get(key)  
    if cached:  
        return cached
```

```

# промах - читаем из БД
conn = get_db_conn()
cur = conn.cursor()
cur.execute("SELECT value FROM items WHERE key=%s", (key,))
row = cur.fetchone()
if row:
    value = row[0]
    # записываем в кэш с TTL + jitter
    ttl = 60 + random.randint(-12, 12) # ±20%
    redis_client.setex(key, ttl, value)
    return value
else:
    return None

```

Результаты теста:

- Hit ratio после прогрева: 94% (при 70% чтений, 30% записей).
- Задержка чтения из кэша: p99 = 3 мс, из БД: p99 = 45 мс.
- Без jitter: после каждого общего TTL БД получает всплеск 1000 запросов за 0.2 сек. С jitter: пиковая нагрузка снижена до 100 запросов/сек, равномерно распределённых.

Индивидуальные задания

1. Реализовать стратегию **write-behind** (асинхронная запись с накоплением) и сравнить write-through и write-behind по задержкам и надёжности.
2. Вместо асинхронного потока использовать **Redis Streams** для очереди записей в БД.
3. Добавить **версионирование** (эффект «блокировка» для предотвращения гонок при одновременной записи одного ключа).

4. Реализовать **read-through** с блокировкой на уровне ключа (чтобы при промахе один клиент загружал данные из БД, а остальные ждали).
5. Использовать **Redis Cluster** вместо одного экземпляра и повторить тесты.
6. Добавить метрики **Prometheus**: число попаданий, промахов, задержки, размер кэша.
7. Реализовать **автоматическую регулировку TTL** на основе частоты обращений (горячие ключи получают больший TTL).
8. Сравнить производительность с использованием библиотеки **Redis-JSON** для хранения сложных объектов.
9. Сымитировать длительное отключение БД и продемонстрировать, что приложение продолжает работать (кэш обслуживает чтение, запись временно буферизуется).
10. Реализовать **пассивную инвалидацию** через Pub/Sub: при обновлении данных в БД (внешнее изменение) отправлять сообщение для удаления ключа из кэша.
11. Вместо PostgreSQL использовать **MongoDB** и сравнить поведение.
12. Добавить **сжатие** данных перед записью в Redis (например, gzip) для экономии памяти, измерить изменение задержек.
13. Реализовать **рандомизированный TTL на основе sliding window**: TTL обновляется при каждом обращении, но с ограничением максимального значения.
14. Написать интеграционный тест на pytest с использованием testcontainers (Redis, PostgreSQL).
15. Внедрить кэш в существующее приложение из ЛР 4 (шардирование), измерить улучшение производительности.

Требования к отчёту

Отчёт оформляется в PDF и содержит:

- титульный лист;

- **цель работы и задачи;**
- **теоретическая часть** – стратегии кэширования (write-through, write-behind, write-around), TTL, проблема thundering herd;
- **раздел «Архитектура решения»** – схема взаимодействия приложения, Redis, БД, очередь асинхронной записи;
- **раздел «Реализация»** – листинги кода (инициализация, методы get/set, worker для БД, генерация jitter);
- **раздел «Эксперименты»:**
 - таблица hit ratio при разной нагрузке;
 - сравнение задержек с кэшем и без кэша;
 - графики нагрузки на БД при одинаковом TTL и при TTL с jitter;
 - гистограммы времени чтения (hit vs miss).
- **раздел «Анализ»** – как write-through улучшает производительность, как jitter спасает от пиков, ограничения подхода;
- **выводы;**
- **ответы на контрольные вопросы.**

Код прилагается архивом или ссылкой на репозиторий.

Контрольные вопросы

1. Какие основные стратегии кэширования записи существуют?
2. В чём преимущество write-through перед write-around?
3. Зачем делать запись в БД асинхронной? Какие риски это несёт?
4. Что такое TTL и почему оно важно для кэша?
5. В чём заключается проблема «thundering herd» при истечении TTL?
6. Как случайный разброс (jitter) помогает решить эту проблему?
7. Каковы последствия выбора слишком маленького или слишком большого TTL?
8. Что произойдёт с приложением, если Redis перестанет отвечать? Как это можно обработать?

9. Как измерить эффективность кэша (hit ratio)? Какое значение считается хорошим?

10. Можно ли использовать write-through для кэширования результатов агрегатных запросов (например, SELECT COUNT(*))?

11. Как обеспечить согласованность кэша при прямых изменениях данных в БД (минуя приложение)?

12. Какие метрики нужно мониторить для обнаружения деградации кэша в production?

Лабораторная работа №14.

Симуляция масштабирования read replicas

Цель работы: имитировать архитектуру «master с несколькими read replica» (одна нода для записи, две или более реплики только для чтения), измерить увеличение пропускной способности чтения при добавлении реплик, а также продемонстрировать проблему eventual consistency (чтение устаревших данных – stale read) в асинхронно реплицируемых системах.

Задачи:

- развернуть один master-узел (принимает запросы на запись и чтение) и две read replica (только чтение, реплицируют данные с master'a асинхронно);
- реализовать простое key-value хранилище с репликацией (имитация через REST API);
- провести нагрузочное тестирование в трёх сценариях: только запись, только чтение, смешанная нагрузка (70% чтения / 30% записи);
- измерить RPS и p99 латентность для разных конфигураций (только master, master + 1 replica, master + 2 replicas);
- симитировать задержку репликации и показать, что чтение с реплики может возвращать старые данные (stale read) сразу после записи;
- проанализировать компромисс между масштабируемостью чтения и согласованностью данных.

Теоретическая часть

В распределённых системах с высокой нагрузкой часто применяется **репликация «ведущий-ведомый» (master-slave)** для **масштабирования чтения**. Мастер обрабатывает все запросы на запись, а также может обслуживать чтение. Одна или несколько реплик (read replicas) асинхронно копируют данные с мастера и обслуживают только запросы на чтение. Это позволяет:

- увеличить общую пропускную способность чтения (реплик может быть несколько);
- разгрузить мастер;
- повысить отказоустойчивость (при падении мастера реплика может временно взять часть нагрузки, хотя запись пропадёт).

Цена масштабирования – eventual consistency (конечная согласованность). Из-за асинхронной репликации между моментом записи на мастер и моментом применения этой записи на реплике существует задержка (replication lag). Клиент, выполнивший запись на мастер, может при последующем чтении с реплики получить устаревшее значение (stale read).

Показатели эффективности:

- **RPS (read/write)** – количество операций в секунду.
- **Replication lag** – задержка распространения изменений (в миллисекундах).
- **Staleness rate** – вероятность прочесть устаревшие данные в заданном временном окне.

Стратегии балансировки чтения:

- Round-robin между репликами.
- На основе близости (ближайшая реплика).
- Приоритет репликам с меньшей задержкой репликации.

В лабораторной работе мы имитируем асинхронную репликацию через фоновый поток с задержкой (например, 100 мс), чтобы явно продемонстрировать stale read и измерить влияние на производительность.

Методика выполнения работы

1. Подготовка окружения

- 1.1. Создайте каталог lab14_read_replicas.
- 1.2. Язык – Python (Flask/FastAPI) для простоты; можно использовать threading и queue для имитации репликации.
- 1.3. Установите flask, requests, numpy, matplotlib для графиков.

- 1.4. Для нагрузочного тестирования используйте wrk или напишите асинхронный клиент на asyncio (с разными типами запросов).
2. **Создание имитации master и read replicas**
- 2.1. Реализуйте один экземпляр приложения, который может работать в режиме master или replica. Параметры передаются через аргументы командной строки или переменные окружения.
- 2.2. Master хранит данные в словаре data. У него есть эндпоинты:
- POST /set?key=...&value=... – запись (возвращает OK).
 - GET /get?key=... – чтение из мастера (для сравнения).
 - GET /replicate – эндпоинт, который реплика вызывает для получения изменений (или мастер отправляет их через очередь).
- 2.3. Replica хранит локальную копию data_replica и периодически (или через push) синхронизируется с мастером. Для упрощения: реплика каждые 0.1 секунды опрашивает мастер по GET /get_updates?since=timestamp (простейший pull).
- 2.4. В эмуляции задержки репликации добавьте time.sleep(replication_delay) перед применением изменений на реплике. По умолчанию replication_delay = 0.1 (100 мс).
3. **Реализация репликации (имитация)**
- 3.1. Master ведёт лог изменений в виде списка: [(key, value, timestamp), ...].
- 3.2. Реплика при старте получает полный дамп (или постепенно синхронизирует).
- 3.3. В фоновом потоке реплика:
- ```
def sync_loop():
 last_ts = 0
 while True:
 resp = requests.get(f'http://master:{port}/get_updates?since={last_ts}')
 updates = resp.json()
 for upd in updates:
```

```
имитация задержки репликации
time.sleep(replication_delay)
self.data_replica[upd['key']] = upd['value']
last_ts = upd['ts']
time.sleep(0.05)
```

3.4. Master возвращает обновления с момента since (можно хранить в глобальном списке до 1000 записей).

#### 4. Запуск узлов

4.1. Запустите master на порту 5000.

4.2. Запустите две реплики на портах 5001 и 5002.

4.3. Убедитесь, что реплики видят данные, записанные на мастер (с учётом задержки).

#### 5. Балансировщик для чтения

5.1. Напишите простой балансировщик (можно встроить в клиент), который направляет запросы на запись всегда на мастер, а запросы на чтение распределяет:

- стратегия round-robin (поочерёдно на мастер и реплики);
- можно добавить вес (мастеру меньше, если он загружен).

5.2. Балансировщик может быть отдельным процессом (прокси) или просто логикой в клиенте нагрузочного теста.

#### 6. Нагрузочное тестирование

##### 6.1. Сценарий 1: только запись (все запросы – POST /set)

- Измерьте RPS и p99 задержку мастера при: только master; master + 2 реплики (но реплики не участвуют в записи). Ожидается, что добавление реплик не ухудшит запись (репликация асинхронна).

##### 6.2. Сценарий 2: только чтение (все запросы – GET /get)

- Измерьте общий RPS (суммарно через балансировщик) для конфигураций:
  - а) только master;

- b) master + 1 replica (50% чтения на master, 50% на replica);
- c) master + 2 replicas (каждый получает  $\approx 33\%$  запросов).

- Постройте график роста RPS при добавлении реплик.

### 6.3. Сценарий 3: смешанная нагрузка (70% чтения, 30% записи)

- Аналогичные замеры. Ожидается, что чтение масштабируется, а запись остаётся на одном мастере.

## 7. Демонстрация stale read

7.1. Установите задержку репликации намеренно большой (например, 2 секунды).

7.2. Напишите клиент, который:

- отправляет запись на мастер SET key=1 value=A.
- сразу после этого (без паузы) читает этот ключ через балансировщик (может попасть на реплику).
- если чтение попало на реплику, возвращается старое значение (или None), если реплика ещё не получила обновление.

7.3. Выполните 100 таких циклов. Запишите долю stale reads (прочитано неактуальное значение).

7.4. Повторите эксперимент с чтением только с мастера – stale reads не должны возникнуть.

7.5. Постройте график зависимости процента stale reads от задержки репликации (0, 0.1, 0.5, 1, 2 секунды).

## 8. Анализ влияния количества реплик

8.1. Для фиксированной задержки репликации (100 мс) постройте график RPS\_чтения от числа реплик (0, 1, 2, 4). Учтите, что балансировщик должен равномерно распределять нагрузку.

8.2. Покажите, что прирост RPS чтения близок к линейному, но не идеален из-за накладных расходов на балансировку и сеть.

## 9. Дополнительно: семантика «read your writes»

- 9.1. Обеспечьте для одного клиента (одной сессии) обязательное чтение с мастера в течение  $N$  секунд после его записи (session stickiness). Реализуйте через сохранение недавно записанных ключей в сессии.
- 9.2. Сравните производительность такого подхода с чтением с реплик без гарантий.

## Пример выполнения задания

### Пример конфигурации master (Flask):

```
python
```

```
app = Flask(__name__)
```

```
data = {} # key: value
```

```
log = [] # list of (key, value, timestamp)
```

```
@app.route('/set', methods=['POST'])
```

```
def set_key():
```

```
 key = request.args['key']
```

```
 value = request.args['value']
```

```
 ts = time.time()
```

```
 data[key] = value
```

```
 log.append((key, value, ts))
```

```
 return 'OK'
```

```
@app.route('/get')
```

```
def get_key():
```

```
 key = request.args['key']
```

```
 return data.get(key, "")
```

```
@app.route('/get_updates')
```

```
def get_updates():
```

```
 since = float(request.args.get('since', 0))
```

```
updates = [(k, v, ts) for (k, v, ts) in log if ts > since]
return jsonify(updates)
```

### **Результаты нагрузки (для примера):**

- Чтение только master: 4000 RPS, p99 = 25 мс.
- Master + 1 replica: чтение распределяется → 7500 RPS, p99 = 22 мс.
- Master + 2 replicas: 10500 RPS, p99 = 23 мс.
- Запись (не зависит от реплик) стабильно 2000 RPS.
- Stale reads при задержке 100 мс: 15% чтений с реплики сразу после записи возвращают старое значение.

## **Индивидуальные задания**

1. Реализовать активную репликацию (push) вместо pull: мастер отправляет изменения по вебсокету или в очередь (Redis Pub/Sub) и измерить разницу в задержке репликации.

2. Добавить механизм «read repair»: если реплика возвращает None (ключ отсутствует), клиент переспрашивает мастер.

3. Внедрить стратегию балансировки на основе задержки репликации: направлять запросы только на те реплики, у которых lag меньше порога (например, < 200 мс).

4. Расширить систему до 4 реплик, измерить насыщение (когда добавление новой реплики перестаёт давать прирост RPS из-за сетевых накладных расходов).

5. Реализовать запись на мастер с синхронной репликацией на одну из реплик (quorum) и сравнить задержки записи.

6. Сымитировать отказ master: переключить чтение только на реплики, запись запретить. Показать деградацию.

7. Добавить метрики Prometheus (количество запросов на каждой реплике, lag) и простой дашборд в Grafana.

8. Использовать реальную СУБД (PostgreSQL с master-slave репликацией) вместо самодельного хранилища.

9. Вместо HTTP использовать gRPC для более эффективной репликации и измерить прирост.

10. Написать тест, демонстрирующий, что при высокой частоте записи (2000 записей/сек) lag растёт, и stale reads увеличиваются.

11. Реализовать клиентскую библиотеку, которая для ключей, записанных в текущей сессии, всегда читает с мастера (гарантия read your writes).

12. Измерить влияние размера пакета обновлений при pull-репликации (10, 100, 1000 записей за раз) на задержки.

13. Добавить сжатие данных при передаче обновлений (zlib) и измерить экономию трафика.

14. Реализовать автоматическое переключение приложения на другой мастер при отказе текущего (failover) с помощью консенсуса (алгоритм на выбор).

15. Перенести реализацию на язык Java с использованием Spring Boot и PostgreSQL replication.

## Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- **цель работы и задачи;**
- **теоретическая часть** – репликация master-slave, масштабирование чтения, eventual consistency, stale read;
- **раздел «Архитектура эксперимента»** – схема узлов (master, replicas), протокол репликации, балансировщик;
- **раздел «Реализация»** – листинги кода (master, replica, синхронизация, балансировка);
- **раздел «Эксперименты и результаты»:**

- таблицы RPS и латентности для разных сценариев и конфигураций;
- графики масштабирования (RPS чтения от числа реплик);
- график доли stale reads от задержки репликации;
- логи демонстрирующие stale read после записи;
- **раздел «Анализ»** – объяснение, почему eventual consistency неизбежна, компромисс между производительностью и согласованностью;
- **выводы;**
- **ответы на контрольные вопросы.**

Приложить исходный код и скрипты нагрузочного тестирования.

## **Контрольные вопросы**

1. Зачем в распределённых системах используются read replicas?
2. Какую проблему решает асинхронная репликация? Какую создаёт?
3. Что такое «stale read»? В каких условиях оно возникает?
4. Как увеличение числа реплик влияет на пропускную способность чтения? Всегда ли рост линейный?
5. Почему запись не масштабируется добавлением реплик? Как можно масштабировать запись?
6. Какие стратегии балансировки чтения вы знаете?
7. Что такое replication lag и как его измерить?
8. Как гарантировать «read your own writes» при наличии реплик?
9. В чём разница между синхронной и асинхронной репликацией с точки зрения согласованности?
10. Какие метрики нужно мониторить, чтобы обнаружить проблемы с репликами?
11. Что произойдёт с системой, если все запросы чтения направлять только на реплики, а мастер использовать только для записи?
12. Как временно отключённая реплика догоняет изменения после восстановления?

## Лабораторная работа №15.

### CQRS с проекциями через очередь (Event Sourcing light)

**Цель работы:** реализовать упрощённую версию паттерна CQRS (Command Query Responsibility Segregation) с разделением на команды (запись) и проекции (чтение) с использованием очереди сообщений, а также построить денормализованную read model на основе событий, генерируемых при изменении агрегата, и протестировать поведение системы в условиях конечной согласованности (eventual consistency).

#### Задачи:

- спроектировать простой домен (например, управление заказами или учёт товаров) с агрегатом, поддерживающим команды;
- реализовать command side (сервис команд), который принимает команды, изменяет состояние агрегата и публикует события в очередь (например, RabbitMQ, Kafka или in-memory queue);
- реализовать отдельный сервис проекций (projector), который потребляет события из очереди и обновляет денормализованную модель для чтения в базе данных (например, PostgreSQL или просто в памяти);
- создать query side (сервис чтения), который извлекает данные из read model без какой-либо сложной логики;
- продемонстрировать, что при написании данных изменения не видны мгновенно в read модели (eventual consistency);
- измерить задержку распространения (replication lag) между записью и обновлением проекции.

### Теоретическая часть

**CQRS (Command Query Responsibility Segregation)** – паттерн, разделяющий модели для чтения и записи. В отличие от классического CRUD, где одна модель используется и для команд, и для запросов, в CQRS command

side отвечает за бизнес-логику, валидацию и изменения состояния, а query side – за оптимизированное чтение (часто через денормализованные проекции).

### **Связь с Event Sourcing (light):**

- Часто CQRS сочетают с Event Sourcing, где каждое изменение агрегата сохраняется как неизменяемое событие. В нашей упрощённой версии мы не будем хранить полную историю событий, а только генерировать их и отправлять в очередь.
- Событие содержит информацию о том, что произошло (например, OrderCreated, OrderStatusChanged).

### **Компоненты системы:**

- **Command Service** – принимает команды (например, CreateOrder), вызывает бизнес-логику агрегата, генерирует события, сохраняет агрегат в command DB (опционально) и публикует события в очередь.
- **Queue (брокер сообщений)** – передаёт события от command service к projector'у.
- **Projector** – потребляет события, обновляет одну или несколько read моделей (таблицы, оптимизированные для запросов).
- **Query Service** – предоставляет API для чтения из read model (без сложной логики, часто простые SELECT).

### **Преимущества CQRS:**

- Независимое масштабирование: можно увеличить количество экземпляров projector или query service.
- Разные модели для разных сценариев чтения.
- Бизнес-логика не загрязняется требованиями представления.

### **Недостатки / цена:**

- Сложность в поддержке (две модели).
- Конечная согласованность: после выполнения команды read model обновляется асинхронно.

### **Пример сценария:**

- Пользователь создаёт заказ (команда).

- Command Service сохраняет заказ в состоянии pending и отправляет событие OrderCreated.
- Projector читает событие, вставляет запись в таблицу orders\_read (денормализованную).
- Пользователь запрашивает список заказов – Query Service читает из orders\_read. Если запрос выполнится сразу после команды, новый заказ может ещё отсутствовать (eventual consistency).

В лабораторной работе используем простую очередь (in-memory queue.Queue или Redis Pub/Sub, либо RabbitMQ). Для упрощения command DB можно не сохранять, а хранить только read model (но тогда теряется часть смысла CQRS). Лучше сохранять агрегаты в отдельном хранилище (например, Redis или PostgreSQL) для возможности восстановления.

## **Методика выполнения работы**

### **1. Подготовка окружения**

- 1.1. Создайте каталог lab15\_cqrs.
- 1.2. Установите Python (или Java) и необходимые библиотеки: flask, requests, pika (для RabbitMQ) или используйте встроенный queue.
- 1.3. Для простоты начните с in-memory очереди (queue.Queue) и потоков. Затем можно перейти на RabbitMQ.
- 1.4. Установите и запустите PostgreSQL (или SQLite) для read model.

### **2. Проектирование домена**

- 2.1. Выберите домен: например, **система управления задачами** (Task Management).
- 2.2. Агрегат Task: поля: id, title, status (pending, in\_progress, done).
- 2.3. Команды: CreateTask(title), UpdateTaskStatus(id, new\_status).
- 2.4. События: TaskCreated(task\_id, title), TaskStatusChanged(task\_id, old\_status, new\_status).

### 3. Реализация Command Service

3.1. Создайте класс `CommandService` с методами `create_task(title)` и `update_status(task_id, new_status)`.

3.2. Для простоты храните агрегаты в словаре `tasks = {}` (имитация command DB).

3.3. При выполнении команды:

- проверьте бизнес-правила (например, нельзя завершить уже завершённую задачу).
- сохраните новое состояние агрегата.
- сформируйте событие и положите его в очередь.

3.4. Событие – это JSON: `{"type": "TaskCreated", "data": {...}, "timestamp": ...}`.

```
import json, threading, queue
```

```
event_queue = queue.Queue()
```

```
def create_task(title):
```

```
 task_id = str(uuid.uuid4())
```

```
 tasks[task_id] = {"title": title, "status": "pending"}
```

```
 event = {"type": "TaskCreated", "data": {"id": task_id, "title": title}}
```

```
 event_queue.put(event)
```

```
 return task_id
```

3.5. Предоставьте HTTP API (Flask) для команд: `POST /tasks` (создать), `PUT /tasks/<id>/status` (обновить).

### 4. Реализация Projector (Consumer)

4.1. Создайте отдельный класс или скрипт, который в бесконечном цикле читает события из очереди.

4.2. Для каждого события обновляйте `read model` (например, таблицу `tasks_read` в PostgreSQL).

4.3. Read model должна быть денормализованной: для системы задач можно хранить ту же структуру, но в реальном CQRS она может быть специальной (например, для поиска).

4.4. Используйте psycorg2 для записи в БД.

```
def projector_loop():
 while True:
 event = event_queue.get()
 if event["type"] == "TaskCreated":
 data = event["data"]
 cur.execute("INSERT INTO tasks_read (id, title, status) VALUES
(%s, %s, %s)",
 (data["id"], data["title"], "pending"))
 conn.commit()
 elif event["type"] == "TaskStatusChanged":
 cur.execute("UPDATE tasks_read SET status=%s WHERE id=%s",
 (event["data"]["new_status"], event["data"]["id"]))
 conn.commit()
```

4.5. Запустите projector в отдельном потоке (или процессе) при старте приложения.

## 5. Реализация Query Service

5.1. Создайте HTTP API для чтения:

- GET /tasks – список всех задач (читает из tasks\_read).
- GET /tasks/<id> – конкретная задача.

5.2. Query Service может быть частью того же веб-приложения, но на отдельном порту, или просто эндпоинтом в command service (что нарушает разделение, но упрощает). Для чистоты разделите на два отдельных приложения: Command API (порт 8000) и Query API (порт 8001).

## 6. Тестирование eventual consistency

6.1. Напишите клиентский скрипт, который:

- отправляет команду POST /tasks на создание задачи.
- сразу (без паузы) запрашивает GET /tasks через Query API.
- повторяет 100 раз и подсчитывает, сколько раз новая задача отсутствовала в ответе (stale).

6.2. Для демонстрации задержки добавьте в projector искусственную задержку: `time.sleep(0.5)` перед обработкой каждого события.

6.3. Измерьте долю успешных мгновенных чтений и время, через которое задача появляется (можно опрашивать Query API раз в 100 мс до появления).

6.4. Постройте гистограмму задержек распространения.

## 7. Масштабирование проектора

7.1. Запустите два экземпляра projector (в разных потоках), читающих из одной очереди (используйте потокобезопасную очередь или RabbitMQ).

7.2. Сравните пропускную способность обработки событий (событий/сек) при одном и двух projector'ах.

7.3. Обратите внимание на возможную неупорядоченную обработку (если события одного агрегата обрабатываются параллельно) – можно использовать ключ партиционирования.

## 8. Расширение: идемпотентность проектора

8.1. Добавьте проверку, что одно событие не приводит к двойной вставке (например, сохраняйте уже обработанные `event_id` в таблице).

8.2. Продемонстрируйте, что при перезапуске projector'a события не обрабатываются повторно (или обрабатываются идемпотентно).

## 9. Сравнение с монолитной архитектурой

9.1. Напишите простую (традиционную) версию того же приложения, где при записи сразу обновляется та же таблица, которая используется для чтения.

9.2. Сравните задержки записи и чтения (в CQRS запись быстрее, потому что не обновляет тяжёлую read модель).

9.3. Обсудите, в каких сценариях CQRS оправдан.

### **Пример выполнения задания**

#### **Событие в очереди (JSON):**

```
{
 "event_id": "evt-123",
 "type": "TaskCreated",
 "timestamp": 1699123456.789,
 "data": {
 "id": "task-1",
 "title": "Write CQRS lab"
 }
}
```

#### **Логирование projector'a:**

```
[Projector] Received event TaskCreated: task-1
[Projector] Updated read model
[Projector] Event processed in 5 ms
```

#### **Тест согласованности (с задержкой projector 500 мс):**

- Выполнено 100 созданий задач.
- Сразу после команды: только 3% запросов к Query API вернули новую задачу (попали на projector до задержки).
- Среднее время до появления задачи: 520 мс (std 50 мс).

**График:** гистограмма задержек распространения с пиком около 500 мс.

## Индивидуальные задания

1. Реализовать проекцию для другого домена (например, «Блоги – комментарии»: команды создают пост, проекция строит готовый HTML для отображения).
2. Использовать RabbitMQ вместо in-memory очереди, реализовать подтверждения (ack) и обработку ошибок.
3. Добавить несколько различных read моделей (например, для отчётов и для поиска), обновляемых одним projector'ом.
4. Реализовать механизм «снапшота» проекции: периодически сохранять состояние и начинать обработку с последнего смещения.
5. Добавить команды, изменяющие несколько агрегатов (saga) и порождающие несколько событий, которые проектор обрабатывает транзакционно.
6. Внедрить метрики (Prometheus): количество обработанных событий, задержка проекции, количество ошибок.
7. Реализовать repeatable чтение (симуляция) – обеспечить, чтобы клиент после своей команды мог прочитать результат, если подождать определённое время.
8. Показать, как при отказе projector'a очередь накапливает события, а после восстановления всё обрабатывается.
9. Заменить command DB на реальную PostgreSQL (для агрегатов), а read model – на Redis (JSON).
10. Добавить функциональность переигрывания событий (replay) для восстановления read model с нуля.
11. Реализовать versioning событий и миграцию проекции при изменении формата события.
12. Использовать Kafka с группами потребителей и продемонстрировать распределённую обработку событий.
13. Реализовать проверку идемпотентности проектора через хранение обработанных идентификаторов событий.

14. Построить дашборд в Grafana, показывающий лаг между последним событием в очереди и последним обработанным событием (consumer lag).

15. Интегрировать с аутентификацией: разные пользователи имеют разные проекции (например, только свои задачи).

## Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- **цель работы и задачи;**
- **теоретическая часть** – описание CQRS, Event Sourcing, eventual consistency, role of projector;
- **раздел «Архитектура решения»** – схема компонентов (Command Service, очередь, Projector, Query Service, read model);
- **раздел «Реализация»** – листинги кода (команды, генерация событий, обработчик очереди, обновление read model, API для чтения);
- **раздел «Эксперименты»:**
  - таблица с задержками распространения при разных нагрузках;
  - график доли мгновенных чтений (success rate) от задержки проектора;
  - демонстрация того, что система остаётся работоспособной при восстановлении projector'a;
  - (опционально) сравнение производительности с монолитом.
- **раздел «Анализ»** – преимущества и недостатки подхода, как eventual consistency влияет на UX, когда CQRS оправдан;
- **выводы;**
- **ответы на контрольные вопросы.**

Код должен быть предоставлен в архиве или репозитории.

## Контрольные вопросы

1. Что такое CQRS? Как он отличается от CRUD?
2. Зачем отделять command side от query side?
3. Какую роль выполняет очередь сообщений в реализации CQRS?
4. Что такое проекция (read model) и кто её строит?
5. Почему в CQRS возникает eventual consistency? Как долго может длиться задержка?
6. Как projector гарантирует, что событие не будет обработано дважды?
7. Как вы обеспечиваете идемпотентность обновления read model?
8. Какие преимущества даёт CQRS при масштабировании?
9. В каких случаях CQRS избыточен и лучше использовать традиционный подход?
10. Что такое event sourcing и чем «light» версия отличается от полной?
11. Как бы вы реализовали повторную обработку событий после сбоя projector'a?
12. Может ли query service читать напрямую из command DB? Почему это плохая практика в CQRS?
13. Какие метрики позволят вам узнать, что проектор не успевает за потоком событий?

## **Лабораторная работа №16.**

### **Chaos-тестирование: отказ узла и автоматическое восстановление**

**Цель работы:** провести хаос-тест в распределённом кластере из трёх узлов с репликацией или шардированием: отключить один узел, проанализировать поведение системы (наличие отказов запросов, изменение пропускной способности) и убедиться, что после восстановления узла кластер автоматически приходит к сбалансированному состоянию.

#### **Задачи:**

- развернуть небольшой кластер (3 узла) с реализованной репликацией данных (например, на основе consistent hashing, master-slave или шардирования из предыдущих работ);
- обеспечить автоматическое обнаружение отказа узла и перераспределение нагрузки на оставшиеся узлы;
- с помощью нагрузочного тестирования измерить влияние отказа на процент успешных запросов и задержки;
- восстановить упавший узел и наблюдать за процессом ребалансировки (синхронизации данных и восстановления копий);
- сделать выводы об отказоустойчивости системы и обосновать необходимость chaos engineering в высоконагруженных распределённых приложениях.

### **Теоретическая часть**

**Chaos engineering (инжиниринг хаоса)** – практика экспериментального внесения отказов в распределённую систему для проверки её способности выдерживать нештатные ситуации. Цель – выявить слабые места до того, как они приведут к реальному инциденту. Один из классических экспериментов – отказ узла (node failure) в кластере.

#### **Требования к отказоустойчивому кластеру:**

- **Избыточность данных** – репликация (каждый кусок данных хранится на нескольких узлах) или кодирование (erasure coding).
- **Автоматическое обнаружение отказа** – система должна быстро узнать, что узел перестал отвечать (через heartbeats, health checks, таймауты).
- **Перераспределение нагрузки** – после отказа запросы, которые шли на упавший узел, должны быть перенаправлены на другие узлы (с сохранением целостности данных).
- **Автоматическое восстановление** – когда узел возвращается, он должен получить актуальные данные от других узлов (репликация catch-up) и снова начать обслуживать запросы.

В зависимости от архитектуры (репликация master-slave, consistent hashing, Raft consensus) поведение при отказе может различаться. В этой работе мы можем взять за основу реализацию из ЛР 2 (consistent hashing) или ЛР 3 (master-slave репликация) и добавить к ней механизм обнаружения сбоев и перераспределения. Для упрощения будем эмулировать кластер на одной машине (разные порты / процессы).

#### **Метрики для анализа:**

- **Error rate** – доля неуспешных запросов во время отказа и после.
- **Latency (p50, p99)** – изменение задержек.
- **Recovery time** – время, за которое система возвращается к полной работоспособности (все запросы успешны, данные синхронизированы).
- **Data consistency** – отсутствие потери данных (все подтверждённые записи доступны после восстановления).

### **Методика выполнения работы**

#### **1. Подготовка основы – отказоустойчивый кластер**

- 1.1. Используйте одну из предыдущих лабораторных работ как базовую:
  - ЛР 2 (consistent hashing) – добавьте поддержку репликации (каждый ключ хранится на K узлах).

- ЛР 3 (master-slave) – поднимите 1 master и 2 slave, настройте асинхронную репликацию.
  - ЛР 6 (Raft) – готовый консенсусный кластер, устойчивый к отказу лидера.
- 1.2. Убедитесь, что кластер может обрабатывать запросы на чтение и запись, и что выход одного узла не приводит к полной недоступности.
  - 1.3. Для простоты в данной методике опишем подход на основе consistent hashing с репликацией (фактор репликации = 2). Но вы можете выбрать любую реализацию.
2. **Реализация обнаружения отказа (failure detection)**
    - 2.1. Каждый узел периодически отправляет heartbeat (например, раз в 1 сек) на выделенный эндпоинт /health всех остальных узлов.
    - 2.2. Ведите в каждом узле список «живых» узлов. Если узел не отвечает на 3 последовательных heartbeat, объявите его мёртвым и обновите локальное представление о кольце.
    - 2.3. Используйте gossip-подобный протокол или просто централизованное хранение состояния (например, Redis или etcd), но для учебных целей можно обойтись взаимным мониторингом.
  3. **Перераспределение нагрузки при отказе**
    - 3.1. При обнаружении отказа узла (например, node2), измените consistent hashing: удалите виртуальные узлы этого физического узла из кольца.
    - 3.2. Запросы, которые ранее направлялись на node2, будут автоматически перенаправлены на следующий узел по кольцу. Благодаря репликации, данные не потеряны, если фактор репликации  $\geq 2$ .
    - 3.3. Клиент (балансировщик) должен получать обновлённый список узлов. Реализуйте простой механизм: клиент периодически (раз в 5 сек) запрашивает у любого узла список живых узлов.
  4. **Восстановление узла и ребалансировка**

4.1. После того как узел возвращается (вы снова запускаете его процесс), он должен:

- начать отправлять heartbeat;
- другие узлы обнаруживают его появление и добавляют его виртуальные узлы в кольцо;
- узел должен синхронизировать недостающие данные (те записи, которые были сделаны во время его отсутствия). Для этого каждый узел хранит журнал последних операций или использует антиэнтропийный протокол (например, сверка хешей).

4.2. После синхронизации узел снова допускается к обслуживанию запросов.

4.3. Продемонстрируйте, что данные, записанные во время простоя узла, после восстановления доступны на нём (проверьте через прямой запрос к узлу).

## 5. Симуляция отказа и сбор метрик

5.1. Запустите кластер из 3 узлов.

5.2. Запустите нагрузочный генератор, который постоянно выполняет PUT и GET случайных ключей (например, 100 операций/сек).

5.3. Убейте один процесс узла (например, kill -9 на Linux или taskkill на Windows).

5.4. Продолжайте нагрузку ещё 30 секунд, затем восстановите узел.

5.5. Собирайте следующие данные:

- доля ошибок (ответы 5xx или таймауты) с момента отказа до восстановления;
- время, в течение которого наблюдались ошибки (период перестроения);
- p99 латентность до, во время и после отказа;
- количество потерянных записей (если запись была подтверждена клиенту, она должна быть прочитана после восстановления).

## 6. Сравнение разных стратегий репликации

6.1. Проведите тот же эксперимент для фактора репликации = 1 (нет избыточности) – ожидается потеря данных и недоступность части ключей.

6.2. Для фактора репликации = 2 (каждый ключ на двух узлах) – при отказе одного узла данные доступны на другом, ошибок записи быть не должно (если кворум записи  $\leq$  количества живых узлов).

6.3. Постройте графики error rate и latency для двух вариантов.

## 7. Визуализация процесса восстановления

7.1. Создайте простой веб-интерфейс или выводите в консоль статусы узлов (жив/мёртв) и лог переключений.

7.2. Запишите видео или сделайте серию скриншотов, показывающих:

- нормальная работа;
- момент отказа (появление ошибок на части запросов);
- стабилизация (ошибки исчезают);
- восстановление узла и ребалансировка.

## 8. Анализ и выводы

8.1. Посчитайте **Recovery Point Objective (RPO)** – сколько данных потеряно (в идеале 0).

8.2. Оцените **Recovery Time Objective (RTO)** – время, за которое система вернулась к нормальной работе после отказа.

8.3. Сформулируйте, какие архитектурные решения (replication factor, quorum, gossip) повлияли на живучесть.

## Пример выполнения задания

**Конфигурация кластера:** Consistent hashing с 150 виртуальными узлами на физический узел, репликация = 2 (каждый ключ хранится на 2 узлах).

**Отказ:** Узел 2 убит на 10-й секунде теста.

**Результаты:**

- Первые 2 секунды после отказа: часть запросов (около 30%) возвращает ошибку – это запросы, которые использовали узел 2 как primary, но secondary ещё не взял на себя нагрузку (перестроение кольца).
- С 3-й по 10-ю секунду: ошибок нет, latency выросла на 20% из-за перегрузки оставшихся узлов.
- Восстановление узла 2 на 20-й секунде: через 5 секунд он синхронизирует данные и начинает принимать запросы, нагрузка распределяется, latency возвращается к норме.
- Потеря данных: 0 (все подтверждённые записи пережили отказ).

## Индивидуальные задания

1. Реализовать кластер на основе Raft (ЛР 6) и провести отказ лидера. Измерить время перевыборов и задержки.
2. В кластере с consistent hashing добавить quorum записи (запись считается успешной только если подтвердили  $W=2$  узла). Измерить влияние на latency и надёжность.
3. Внести случайную задержку обнаружения отказа (разные таймауты heartbeat) и показать, как это влияет на RTO.
4. Использовать toxiproxy для введения сетевых задержек и потерь пакетов вместо физического убийства узла. Сравнить поведение.
5. Реализовать «предсказание отказа» – если узел начинает отвечать медленно, снизить его вес в балансировке, а затем удалить.
6. Подключить к кластеру Prometheus и настроить алерты на отказ узла.
7. Сравнить отказ узла в архитектуре master-slave (без авто-выбора нового мастера) и в Raft. Какой быстрее восстанавливается?
8. Добавить постоянное хранилище (LevelDB/RocksDB) для каждого узла, чтобы данные не терялись при рестарте узла.
9. Реализовать автоматическую ребалансировку данных после добавления узла (перенос части ключей).

10. Создать сценарий, в котором отказывают сразу два узла. При факторе репликации 2 – что произойдёт?

11. Вместо готового решения написать эмулятор кластера на asyncio, где каждый узел – корутина.

12. Внедрить логирование в формате JSON и отправить в Elasticsearch для анализа временной линии отказа.

13. Сравнить поведение при отказе узла для целостности на основе сильной согласованности (синхронная репликация) и eventual consistency.

14. Реализовать API для ручного введения хаоса (эндпоинт /chaos/kill/{node}) и провести автоматизированный тест.

15. Показать, как репликация данных на трёх узлах (фактор 3) позволяет выдержать отказ двух узлов.

## Требования к отчёту

Отчёт должен содержать:

- титульный лист;
- **цель работы и задачи;**
- **теоретическая часть** – описание chaos engineering, отказоустойчивые архитектуры, понятия RPO, RTO;
- **раздел «Архитектура кластера»** – схема узлов, алгоритм распределения данных, механизм репликации, детектор отказов;
- **раздел «Реализация»** – основные фрагменты кода: heartbeat, обнаружение отказа, перестроение кольца, синхронизация после восстановления;
- **раздел «Хаос-тест»** – описание сценария (время отказа, нагрузка), команды для запуска и убийства узла;
- **раздел «Результаты»** – графики error rate, latency, пропускной способности; таблица с RPO, RTO; скриншоты логов;
- **раздел «Анализ»** – объяснение поведения системы, выявленные слабые места (если есть), рекомендации по улучшению;

- **ВЫВОДЫ** – достигнута ли цель, как chaos testing помогает повысить надёжность;
- **ответы на контрольные вопросы.**

Приложить исходный код и скрипты для запуска тестов.

## **Контрольные вопросы**

1. Что такое chaos engineering и зачем он нужен в highload?
2. Какие метрики характеризуют отказоустойчивость системы?
3. Чем RPO отличается от RTO?
4. Как часто нужно проводить chaos-тесты в production?
5. Почему при отказе узла в consistent hashing с репликацией можно потерять данные, если репликация настроена неправильно?
6. Как быстро должен срабатывать детектор отказов? Что будет, если сделать таймаут слишком маленьким или слишком большим?
7. Как перераспределение нагрузки при отказе влияет на оставшиеся узлы (эффект «холостого выстрела»)?
8. Что произойдёт, если в Raft-кластере упадут два узла из трёх?
9. Какие инструменты для chaos engineering существуют (Chaos Monkey, Gremlin, Litmus)?
10. Как проверить, что данные не повреждены при восстановлении узла?
11. Может ли автоматическое восстановление привести к вторичному отказу (например, из-за перегрузки)?
12. Как вы обеспечиваете идемпотентность повторной синхронизации данных?

## Лабораторная работа №17.

### Полный highload-стек (Docker Compose + мониторинг)

**Цель работы:** собрать многокомпонентную высоконагруженную систему в контейнерах Docker под управлением Docker Compose, включив в неё балансировщик Nginx, два экземпляра приложения, кэш Redis, реплицируемую базу данных PostgreSQL (master + slave), а также настроить систему мониторинга на базе Prometheus и Grafana, после чего провести нагрузочное тестирование и проанализировать полученные дашборды.

#### Задачи:

- разработать простое веб-приложение (например, на Python/Flask или Node.js), которое использует Redis для кэширования и PostgreSQL для постоянного хранения;
- описать инфраструктуру в docker-compose.yml: Nginx (reverse proxy / load balancer), два и более экземпляра приложения (масштабирование через replicas), Redis, PostgreSQL master + PostgreSQL slave (с потоковой репликацией);
- настроить экспорт метрик: приложение должно отдавать метрики в формате Prometheus (через клиентскую библиотеку), PostgreSQL exporter, Redis exporter, Nginx exporter (или использовать cAdvisor для сбора метрик контейнеров);
- запустить Prometheus (сбор метрик со всех компонентов) и Grafana (визуализация, готовые дашборды);
- провести нагрузочное тестирование с помощью wrk, bombardier или k6, наблюдая в реальном времени за графиками в Grafana;
- по результатам теста проанализировать узкие места, загруженность компонентов, эффективность кэширования и репликации.

## Теоретическая часть

Современные высоконагруженные системы редко состоят из одного сервиса. Типичный стек включает:

- **Балансировщик нагрузки** (Nginx, HAProxy) для распределения трафика между несколькими экземплярами приложения и обеспечения отказоустойчивости.
- **Несколько экземпляров приложения** (горизонтальное масштабирование). Приложение должно быть stateless, чтобы любой экземпляр мог обработать любой запрос.
- **Кэш** (Redis, Memcached) для снижения нагрузки на базу данных и ускорения ответов.
- **Реплицируемая база данных** (PostgreSQL master-slave) для надёжности и масштабирования чтения.
- **Мониторинг** (Prometheus + Grafana) для наблюдения за состоянием всех компонентов, метриками производительности, ошибками.

**Docker Compose** позволяет описать всю инфраструктуру в одном YAML-файле, поднять её одной командой и легко масштабировать сервисы. Это идеальный инструмент для локального тестирования интеграционных сценариев, которые максимально приближены к production.

### Метрики для анализа:

- **Nginx:** количество запросов в секунду, коды ответов (2xx, 4xx, 5xx), время ответа.
- **Приложение:** количество вызовов на эндпоинты, ошибки, количество обращений к Redis и PostgreSQL, время выполнения.
- **Redis:** hit/miss ratio, количество активных ключей, использование памяти.
- **PostgreSQL (master/slave):** количество транзакций, lag репликации, размер БД, количество активных соединений.
- **Системные метрики:** CPU, память, сетевой трафик контейнеров (через cAdvisor или node exporter).

Нагрузочное тестирование в такой среде позволяет:

- проверить, как система масштабируется под нагрузкой;
- оценить, насколько эффективно кэширование снижает число запросов к БД;
- убедиться, что репликация БД не создаёт узкое место;
- выявить компонент, который первым достигает предела (база данных, приложение, сеть).

## **Методика выполнения работы**

### **1. Подготовка проекта**

- 1.1. Создайте каталог `lab17_full_stack`.
- 1.2. Внутри создайте подкаталоги:
  - `app/` – код приложения;
  - `nginx/` – конфигурация Nginx;
  - `prometheus/` – конфигурация Prometheus;
  - `grafana/` – provisioning для дашбордов;
  - `postgres/` – скрипты инициализации (опционально).
- 1.3. Установите Docker и Docker Compose на вашу машину.

### **2. Разработка приложения**

- 2.1. Напишите простое веб-приложение на Python (Flask) или Node.js (Express).
- 2.2. Эндпоинты:
  - GET / – приветствие (проверка работы).
  - POST /data – принимает JSON `{"key": "some_key", "value": "some_value"}`, сохраняет в PostgreSQL, а также записывает в Redis (write-through).
  - GET /data/<key> – сначала читает из Redis (если ключ есть), при промахе – из PostgreSQL, затем кладёт в Redis с TTL.

2.3. Приложение должно экспортировать метрики для Prometheus: используйте `prometheus_flask_exporter` (Python) или `prom-client` (Node.js).

2.4. Конфигурация подключения к БД и Redis должна читаться из переменных окружения (`DB_HOST`, `REDIS_HOST`, `DB_PASSWORD` и т.д.), чтобы при запуске подставить разные значения для разных экземпляров.

### 3. Создание Docker-образов

3.1. Напишите Dockerfile для приложения:

```
FROM python:3.11-slim
```

```
WORKDIR /app
```

```
COPY requirements.txt .
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY . .
```

```
CMD ["python", "app.py"]
```

3.2. Соберите образ (`docker build -t myapp .`) или укажите сборку в `docker-compose.yml`.

### 4. Настройка компонентов в docker-compose.yml

4.1. Файл `docker-compose.yml` должен содержать следующие сервисы:

- `postgres_master` (порт 5432)
- `postgres_slave` (порт 5433) – настроить потоковую репликацию.
- `redis` (порт 6379)
- `app` – с масштабированием `replicas: 2`
- `nginx` (порт 80) – балансировщик между двумя экземплярами `app`
- `prometheus` (порт 9090)
- `grafana` (порт 3000)

4.2. Для PostgreSQL настройте репликацию:

- Мастер: инициализация, создание пользователя репликации, `wal_level = replica`.

- Слейв: использует pg\_basebackup и запускается в режиме реплики.

4.3. Для каждого сервиса добавьте healthcheck (через curl или специальные эндпоинты).

4.4. Настройте сеть highload-net, чтобы сервисам были доступны имена друг друга.

## 5. Настройка Nginx как балансировщика

5.1. Создайте nginx/nginx.conf с upstream для двух экземпляров приложения:

```
upstream backend {
 server app:5000;
 # второй экземпляр будет добавлен автоматически при
масштабировании
}

server {
 listen 80;
 location / {
 proxy_pass http://backend;
 proxy_set_header Host $host;
 }
 location /metrics {
 proxy_pass http://prometheus:9090/metrics;
 # или отдельно метрики nginx exporter
 }
}
```

5.2. Добавьте в docker-compose сервис nginx с пробросом порта 80 на хост.

## 6. Настройка сбора метрик

6.1. Добавьте в каждый сервис экспортёры метрик:

- Для приложения — уже есть в коде.

- Для PostgreSQL – используйте prometheuscommunity/postgres-exporter.
- Для Redis – oliver006/redis\_exporter.
- Для Nginx – nginxinc/nginx-prometheus-exporter.
- Для контейнеров (CPU/память) – google/cadvisor.

6.2. В файле prometheus/prometheus.yml настройте статические цели для всех экспортёров (используйте имена сервисов из docker-compose и их порты).

scrape\_configs:

- job\_name: 'app'

static\_configs:

- targets: ['app:5000']

- job\_name: 'postgres\_master'

static\_configs:

- targets: ['postgres-exporter-master:9187']

...

6.3. Запустите Prometheus и проверьте, что все цели отображаются в UI (/targets).

## 7. Настройка Grafana

7.1. Добавьте в docker-compose.yml сервис Grafana с переменными окружения (GF\_SECURITY\_ADMIN\_PASSWORD=admin).

7.2. Подключите Grafana к Prometheus как источнику данных (можно автоматически через provisioning).

7.3. Импортируйте готовые дашборды:

- PostgreSQL dashboard (ID 9628)
- Redis dashboard (ID 11835)
- Nginx dashboard (ID 12708)
- Node (cAdvisor) dashboard (ID 14282)

7.4. Либо создайте свой дашборд с основными метриками: RPS приложения, hit ratio Redis, lag репликации PostgreSQL, использование CPU/памяти.

## 8. Нагрузочное тестирование

8.1. Запустите весь стек: `docker-compose up -d --scale app=2`.

8.2. Прогрейте систему: сделайте 1000 запросов POST /data с разными ключами (через скрипт).

8.3. Проведите нагрузочный тест с помощью wrk или k6:

- Сценарий: 30 секунд, 10 параллельных потоков, смесь 70% чтений (GET /data/<random\_key>) и 30% записей (POST /data).
- Запустите тест, одновременно наблюдая за Grafana (в реальном времени).

8.4. Увеличьте масштабирование до 4 экземпляров приложения (`--scale app=4`) и повторите тест. Сравните RPS.

8.5. Отключите один из экземпляров БД (слейв) и повторите тест – как изменится latency чтения?

## 9. Анализ дашбордов

9.1. Сделайте скриншоты дашбордов во время пиковой нагрузки.

9.2. Запишите:

- Максимальный RPS, достигнутый системой (при разных масштабах).
- Hit ratio Redis (доля чтений из кэша).
- Lag репликации PostgreSQL (в секундах).
- Какой компонент был узким местом (CPU памяти приложения, сеть, база данных).

9.3. Сформулируйте рекомендации по дальнейшей оптимизации (добавление индексов, увеличение TTL, настройка пулов соединений).

## 10. Дополнительное задание: автоматическое масштабирование

10.1. Реализуйте простой эмулятор горизонтального автомасштабирования: при превышении  $RPS > 1000$  увеличивать число реплик приложения (через Docker API) – по желанию.

### Пример выполнения задания

**docker-compose.yml (сокращённый фрагмент):**

```
version: '3.8'
```

```
services:
```

```
 postgres_master:
```

```
 image: postgres:13
```

```
 environment:
```

```
 POSTGRES_USER: test
```

```
 POSTGRES_PASSWORD: test
```

```
 POSTGRES_DB: highload
```

```
 volumes:
```

```
 - ./postgres/master_init:/docker-entrypoint-initdb.d
```

```
 networks:
```

```
 - highload
```

```
 postgres_slave:
```

```
 image: postgres:13
```

```
 depends_on:
```

```
 - postgres_master
```

```
 environment:
```

```
 POSTGRES_USER: test
```

```
 POSTGRES_PASSWORD: test
```

```
 command: >
```

```
 sh -c "pg_basebackup -h postgres_master -U test -D
/var/lib/postgresql/data -R --slot=replica_slot && exec postgres"
```

```
 networks:
```

- highload

redis:

image: redis:7

networks:

- highload

app:

build: ./app

deploy:

replicas: 2

environment:

DB\_HOST: postgres\_master

REDIS\_HOST: redis

networks:

- highload

nginx:

image: nginx:alpine

volumes:

- ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf

ports:

- "80:80"

networks:

- highload

prometheus:

image: prom/prometheus

volumes:

- ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

ports:

- "9090:9090"

networks:

- highload

grafana:

image: grafana/grafana

ports:

- "3000:3000"

networks:

- highload

networks:

highload:

### **Результаты нагрузочного теста (при 2 репликах app):**

- RPS чтения: ~2500, записи: ~800 (перегрузка БД).
- Redis hit ratio: 85% (после прогрева).
- Replication lag: стабильно 0.2–0.5 сек.
- Узкое место: CPU PostgreSQL master достигает 90%. Увеличение числа реплик app не помогает. Решение: включить read replica для запросов чтения (настроить балансировщик или изменить логику приложения).

## **Индивидуальные задания**

1. Заменить Flask на FastAPI + async и сравнить RPS при том же сценарии.
2. Реализовать кэширование через Redis Cluster (3 ноды) вместо одного экземпляра.
3. Добавить в систему очередь сообщений (RabbitMQ или Kafka) для асинхронной обработки записей, освобождая приложение.

4. Вместо ручного масштабирования `--scale` использовать `docker-compose` с `replicas` и автоматический перезапуск.
5. Внести в приложение преднамеренную ошибку (например, долгий запрос к БД) и показать, как она отображается на дашборде (латенси, ошибки 5xx).
6. Заменить Nginx на HAProxy и сравнить производительность балансировки.
7. Добавить в стек мониторинг логов (ELK или Loki + Promtail) и коррелировать метрики с логами ошибок.
8. Настроить алерты в Prometheus (например, если PostgreSQL lag > 10 сек, или RPS падает ниже 100).
9. Провести тест с постепенным увеличением нагрузки от 10 до 5000 RPS (ramp-up) и построить график зависимости latency от нагрузки.
10. Реализовать эндпоинт `/health` в приложении и настроить в Nginx активные health checks.
11. Использовать реплику PostgreSQL для чтения (`balance_reads`) – изменить приложение так, чтобы SELECT шли на slave, а INSERT/UPDATE на master.
12. Заменить Prometheus + Grafana на VictoriaMetrics + Grafana и измерить потребление ресурсов.
13. Добавить в систему кэш второго уровня (например, локальный кэш в приложении на `functools.lru_cache`) и оценить улучшение.
14. Написать юнит-тесты и интеграционные тесты для приложения с использованием `pytest` и `testcontainers`.
15. Развернуть весь стек в облаке (например, Yandex Cloud или AWS) с помощью Terraform и повторить нагрузочное тестирование уже в реальной сети.

## Требования к отчёту

Отчёт должен быть оформлен в виде PDF и содержать:

- титульный лист;
- **цель работы и задачи;**
- **теоретическая часть** – описание компонентов highload-стека, Docker Compose, мониторинга;
- **раздел «Архитектура системы»** – схема взаимодействия сервисов с указанием портов и протоколов;
- **раздел «Реализация»** – файлы конфигурации (docker-compose.yml, nginx.conf, prometheus.yml) и листинги ключевых частей приложения (особенно кэширование и метрики);
- **раздел «Запуск и тестирование»** – команды, скриншоты логов запуска, результатов docker-compose ps;
- **раздел «Нагрузочное тестирование и мониторинг»:**
  - описание сценариев нагрузки;
  - скриншоты дашбордов Grafana до, во время и после нагрузки;
  - таблицы с RPS, latency, hit ratio, lag репликации;
  - сравнение при разном масштабировании app;
- **раздел «Анализ узких мест»** – какой компонент первым достиг предела и почему;
- **выводы** – достигнута ли цель, какие практические навыки приобретены;
- **ответы на контрольные вопросы.**

Все исходные коды и конфигурационные файлы должны быть приложены архивом или предоставлены ссылкой на репозиторий.

## Контрольные вопросы

1. Преимущества использования Docker Compose для развёртывания highload-стеков.
2. Как Nginx балансирует запросы между несколькими экземплярами приложения? Какие алгоритмы поддерживает?
3. Зачем в архитектуре нужен Redis, если есть PostgreSQL?
4. Каким образом настраивается репликация PostgreSQL master-slave в Docker?
5. Как экспортируются метрики из приложения в Prometheus?
6. Что показывает метрика hit ratio для Redis? Как её интерпретировать?
7. Как определить, что узким местом является база данных, а не приложение?
8. Как влияет увеличение числа реплик приложения на RPS чтения? На RPS записи?
9. Какую роль играет cAdvisor в сборе метрик?
10. Какие метрики в Grafana вы использовали для выявления проблемы с репликацией PostgreSQL?
11. Что произойдёт, если все экземпляры приложения будут пытаться одновременно записать один и тот же ключ в Redis?
12. Как можно автоматически масштабировать количество контейнеров приложения на основе метрик (CPU, RPS)?
13. Каким образом нагрузочное тестирование помогает прогнозировать поведение системы в production?
14. Почему в производственных системах используют отдельные экспортёры метрик для каждого сервиса (а не только cAdvisor)?

## **Лабораторная работа №18.**

### **Итоговый проект: масштабируемое приложение с обоснованием архитектуры**

**Цель работы:** разработать небольшую распределённую систему с признаками highload (например, чат, сервис коротких ссылок, голосовалка), применить на практике изученные архитектурные паттерны (шардирование, репликация, кэширование, очереди, rate limiting), провести нагрузочное тестирование, задокументировать принятые архитектурные решения с обоснованием выбора компромиссов согласно CAP/PACELC, а также защитить проект в форме презентации.

#### **Задачи:**

- выбрать предметную область и сформулировать функциональные и нефункциональные требования (целевая нагрузка, RPS, latency, доступность);
- спроектировать архитектуру системы, включая выбор стратегий шардирования, репликации, кэширования, асинхронной обработки, ограничения частоты запросов;
- реализовать работающий прототип системы (минимально жизнеспособный продукт) с использованием изученных технологий и паттернов;
- провести нагрузочное тестирование, измерить ключевые метрики (RPS, p99 latency, hit ratio, использование ресурсов);
- подготовить документ, в котором для каждого архитектурного решения обосновать компромисс с точки зрения CAP-теоремы и PACELC;
- представить результаты в виде презентации и ответить на вопросы.

### **Теоретическая часть**

Итоговый проект является синтезом всех предыдущих лабораторных работ. Студент должен продемонстрировать умение выбирать и комбинировать

архитектурные паттерны для достижения требуемого уровня производительности, отказоустойчивости и масштабируемости. Ключевая часть – **архитектурное обоснование**, то есть явное указание, какие компромиссы сделаны и почему.

**CAP-теорема** гласит, что распределённая система может предоставлять только два из трёх свойств: Consistency (согласованность), Availability (доступность), Partition tolerance (устойчивость к разделениям). На практике выбирают AP (доступность + устойчивость к разделениям) с eventual consistency или CP (согласованность + устойчивость) с возможной недоступностью во время разделения.

**PACELC** расширяет CAP: If Partition (разделение), then trade-off between A and C; Else (без разделения), trade-off between L (latency) and C (consistency). Эта модель помогает выбирать стратегии репликации: синхронная (лучше C, хуже L) vs асинхронная (лучше L, хуже C).

#### **Примеры предметных областей:**

- **Сервис коротких ссылок** (как [bit.ly](https://bit.ly)): запись – создание короткой ссылки, чтение – редирект. Нагрузка на чтение намного выше (100:1). Можно использовать кэш, шардирование по хешу короткого кода.
- **Чат** (асинхронный обмен сообщениями): требуется доставка сообщений, история, комнаты. Применимы очереди сообщений, шардирование по комнатам, чтение из read replica.
- **Голосовалка / опросы** (как в реальном времени): высокая интенсивность записи голосов, агрегация результатов, кэширование результатов. Можно использовать CQRS, асинхронное обновление проекций.

#### **Архитектурные элементы, которые должны быть обоснованы:**

- Стратегия шардирования (range, hash, consistent hashing) и фактор репликации.
- Типы репликации (синхронная/асинхронная) для разных данных.
- Использование кэша (write-through, write-behind, TTL с jitter).

- Применение очередей для асинхронной обработки (Saga, CQRS).
- Rate limiting – глобальный или распределённый (Redis + Lua).
- Circuit breaker для защиты от сбоев внешних сервисов.
- Мониторинг и observability (метрики, трейсинг).

## **Методика выполнения работы**

### **1. Выбор темы и сбор требований**

- 1.1. Выберите одну из предложенных предметных областей (или предложите свою).
- 1.2. 1.2. Задокументируйте нефункциональные требования:
  - Ожидаемая нагрузка (RPS чтения и записи).
  - Допустимая задержка (p99 чтения  $\leq 50$  мс, записи  $\leq 200$  мс).
  - Доступность (99.9% или 99.99%).
  - Сценарии отказов (падение узла, потеря связи).
- 1.3. Опишите функциональные требования в виде user stories.

### **2. Проектирование архитектуры**

- 2.1. Нарисуйте диаграмму компонентов (балансировщик, приложение, кэш, БД, очереди, мониторинг).
- 2.2. Для каждого компонента выберите технологию и обоснуйте:
  - Балансировщик: Nginx / HAProxy (почему).
  - Кэш: Redis / Memcached (почему).
  - База данных: PostgreSQL / Cassandra / MongoDB.
  - Очереди: RabbitMQ / Kafka / Redis Streams.
- 2.3. Примените CAP/PACELC: если выбрана асинхронная репликация, укажите, что жертвуете C в пользу L; если выбрана синхронная – жертвуете L/A в пользу C.
- 2.4. Определите, как будете шардировать данные (по какому ключу, как переназначать).
- 2.5. Опишите стратегии ограничения запросов (rate limit) и защиту от перегрузок (circuit breaker).

### **3. Реализация прототипа**

- 3.1. Реализуйте MVP системы (можно на базе кода из предыдущих ЛР).
- 3.2. Используйте Docker Compose для локального развёртывания всех компонентов.
- 3.3. Обеспечьте сбор метрик (Prometheus) и визуализацию (Grafana).
- 3.4. В коде явно отметьте архитектурные решения комментариями (например, `async replication – chose availability over consistency`).

### **4. Нагрузочное тестирование**

- 4.1. Напишите сценарий нагрузки, имитирующий реальное поведение пользователей (например, 70% чтений, 30% записей).
- 4.2. Используйте wrk, k6 или locust. Начинайте с низкой нагрузки и увеличивайте до появления ошибок или превышения latency.
- 4.3. Запишите ключевые метрики:
  - Максимальный устойчивый RPS.
  - p50, p90, p99 latency для чтения и записи.
  - Hit ratio кэша.
  - Загрузка CPU/памяти на контейнерах.
  - Replication lag (если использована репликация).
- 4.4. Сымитируйте отказ одного узла и замерьте время восстановления (RTO) и потерю данных (RPO).

### **5. Оформление документа с обоснованием**

- 5.1. Создайте документ (PDF) следующей структуры:
  - Введение (цели, требования).
  - Архитектурная диаграмма и описание компонентов.
  - Таблица решений с колонками: «Решение», «Альтернативы», «Выбор», «Обоснование с точки зрения CAP/PACELC».
  - Результаты нагрузочного тестирования (графики, таблицы).
  - Анализ узких мест и возможные улучшения.
  - Заключение.
- 5.2. Пример строки таблицы:

| Решение               | Альтернативы             | Выбор                      | Обоснование                                                                                                                                                                   |
|-----------------------|--------------------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Репликация PostgreSQL | синхронная / асинхронная | асинхронная (master-slave) | PACELC: без сетевого разделения мы выбираем латентность (L) вместо строгой согласованности (C), так как для нашего сценария (чтение новостей) допустима eventual consistency. |

## 6. Подготовка презентации

6.1. Презентация должна занимать 5–7 минут. Содержать:

- слайд с архитектурой;
- слайд с ключевыми компромиссами (CAP/PACELC);
- слайд с результатами тестирования;
- слайд с выводами и демонстрацией (запись экрана или скриншоты Grafana).

6.2. Будьте готовы ответить на вопросы: почему выбран конкретный шардинг, как система ведёт себя при отказе, каковы плюсы/минусы выбранных технологий.

## 7. Защита проекта

7.1. Представьте презентацию и демонстрацию работающей системы.

7.2. Ответьте на вопросы преподавателя и группы.

7.3. Сдайте отчёт и исходный код (ссылка на репозиторий).

## Пример выполнения задания

**Тема:** Сервис коротких ссылок (аналог [bit.ly](https://bit.ly)).

**Требования:**

- 10 000 новых ссылок в день ( $\approx 0.1$  RPS write).
- 1 000 000 переходов в день ( $\approx 11$  RPS read).
- p99 latency чтения  $< 50$  мс, записи  $< 100$  мс.

- Доступность 99.95%.

### **Архитектура:**

- Балансировщик Nginx (распределяет чтение между несколькими экземплярами).
- Приложение на Flask + Gunicorn (stateless).
- Redis (кэш для коротких URL) – TTL 24 часа, write-through.
- PostgreSQL (master + 2 read replica) для постоянного хранения: шардирование по хешу от короткого ключа (consistent hashing, 4 шарда).
- Асинхронная репликация (Master → Replicas).
- Очереди не используются (можно добавить для аналитики переходов).
- Rate limiting через Redis (100 запросов в минуту на IP).

### **Обоснование CAP/PACELC:**

- Чтения – eventual consistency приветствуется, поэтому асинхронная репликация ( $L > C$ ).
- Записи – синхронное сохранение в master (C, но latency выше, что допустимо из-за низкой частоты записи).
- При сетевом разделении (partition) выбираем доступность (A) – система продолжает работать с eventual consistency.

### **Результаты тестирования (кб):**

- При 1000 RPS чтения: p99 = 42 мс, hit ratio Redis = 89%.
- При 50 RPS записи: p99 = 78 мс.
- Отказ одного из read replica – RPS чтения упал на 15%, но без ошибок (запросы перенаправлены на master).
- Восстановление replica: репликация отставания догнала через 10 сек.

## Индивидуальные задания

1. **Сервис коротких ссылок с аналитикой** – добавить очередь (Kafka) для асинхронного сбора статистики переходов (гео, браузер).
2. **Чат с комнатами** – реализовать шардирование по ID комнаты, хранение истории в PostgreSQL, кэш последних сообщений в Redis.
3. **Голосование в реальном времени** – использовать WebSocket, агрегацию голосов в Redis, периодическую запись в БД.
4. **Система очередей задач** – асинхронное API (как Celery): хранилище задач в PostgreSQL, статусы в Redis, воркеры.
5. **Витрина товаров (магазин)** – поиск по каталогу, шардирование по категориям, кэш страниц, CQRS с проекцией для поиска.
6. **Сокращатель ссылок с кастомными алиасами** – добавить проверку уникальности через Redis cluster, блокировки при создании.
7. **Аукцион** (ставки в реальном времени) – синхронная репликация для ставок (C важнее L), rate limiting, circuit breaker для внешнего платёжного шлюза.
8. **Сервис уведомлений** (email/push) – очереди с разными приоритетами, шардирование по user\_id, репликация для высокой доступности.
9. **Система лайков** (как у Instagram) – использование Redis (counter) для мгновенных лайков и асинхронная запись в БД.
10. **Сервис комментариев** – древовидные комментарии, денормализация в read model, eventual consistency при обновлении.
11. **Рекомендательный сервис** (товары, которые часто покупают вместе) – предрасчёт через offline jobs, хранение в Redis и реплики PostgreSQL.
12. **Система бронирования билетов** – транзакции (с блокировками) на мастере, read replica для просмотра доступных билетов, Saga для отмены.
13. **Короткие ссылки с использованием consistent hashing** для шардирования – реализовать поверх ЛР 2.

14. **Доска объявлений** – полнотекстовый поиск через Elasticsearch, синхронизация через CDC (Debezium) из PostgreSQL.

15. **Любая собственная идея** – с обоснованием highload-характеристик.

## Требования к отчёту

Итоговый отчёт должен быть оформлен как PDF и содержать:

- титульный лист с темой проекта, ФИО, группой;
- **цель работы и задачи;**
- **теоретическая часть** (сжатое описание CAP/PACELC применительно к проекту);
- **раздел «Анализ требований»** – функциональные и нефункциональные требования, профиль нагрузки;
- **раздел «Архитектура»** – диаграмма компонентов, описание взаимодействия, выбор технологий с обоснованием;
- **раздел «Обоснование компромиссов»** – таблица решений с ссылками на CAP/PACELC (пример выше);
- **раздел «Реализация»** – структура проекта, ключевые фрагменты кода (приводятся только важные для архитектуры), Docker-файлы;
- **раздел «Тестирование»** – сценарии, результаты нагрузочных тестов (графики RPS, latency, hit ratio, CPU), результаты chaos-тестов;
- **раздел «Анализ узких мест и пути улучшения»** – что можно оптимизировать при росте нагрузки;
- **раздел «Заключение»** – соответствие исходным требованиям, приобретённые компетенции;
- **список литературы** (опционально);
- **ответы на контрольные вопросы.**

Приложить ссылку на репозиторий с кодом и конфигурациями.  
Презентация сдаётся отдельным файлом (PDF или PPT).

## Контрольные вопросы

1. Почему для вашего проекта выбрана именно такая стратегия шардирования?
2. Как ваша система ведёт себя в соответствии с CAP-теоремой? Приведите конкретный сценарий.
3. Чем вы руководствовались при выборе между синхронной и асинхронной репликацией?
4. Как бы изменилась архитектура, если бы требования к согласованности стали строже (например, финансовая система)?
5. Какие метрики из Prometheus вы использовали для выявления узкого места при нагрузочном тестировании?
6. Как в вашей системе реализован graceful degradation при отказе кэша?
7. Каким образом вы обеспечили идемпотентность при повторной обработке сообщений в очереди?
8. Какие компромиссы (trade-offs) вашей архитектуры вы считаете самыми критичными?
9. Как масштабировать систему, если RPS записи вырастет в 10 раз?
10. Почему выбрана конкретная очередь сообщений (Kafka / RabbitMQ / in-memory)?
11. Как вы проверяли, что система не теряет данные при аварийном отключении питания?
12. Опишите сценарий, при котором eventual consistency может привести к проблеме, и как вы её смягчили.
13. Какие инструменты мониторинга и tracing вы применили и почему?
14. Что было самым сложным при проектировании и реализации?
15. Как ваш проект демонстрирует применение не менее 5 паттернов из списка дисциплины (Saga, CQRS, Circuit Breaker, Backpressure, Consistent Hashing, etc.)?

## СПСНОК ЛТЕРАТУРЫ

1. Климов А.В., Семёнов А.А. Высоконагруженные системы: архитектура, проектирование и эксплуатация. – М.: ДМК Пресс, 2021. – 412 с.
2. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. – СПб.: Питер, 2019. – 640 с. (Оригинал: Kleppmann M. Designing Data-Intensive Applications, 2017).
3. Бернштейн Ф., Ньюкомер Э. Принципы построения распределённых систем. – М.: Вильямс, 2016. – 320 с.
4. Онгоро Д., Остерхаут Дж. In Search of an Understandable Consensus Algorithm (Raft). – Proceedings of USENIX ATC, 2014. – URL: <https://raft.github.io/raft.pdf>.
5. Гилберт С., Линч Н. CAP Theorem: Revisiting the Fundamentals. – ACM SIGACT News, 2012. – DOI: 10.1145/2185376.2185385.
6. Abadi D.J. Consistency Tradeoffs in Modern Distributed Database Systems (PACELC). – IEEE Data Engineering Bulletin, 2012.
7. Richardson C. Microservices Patterns: With examples in Java. – Manning Publications, 2018. – 520 p.
8. Newman S. Building Microservices: Designing Fine-Grained Systems. – 2nd ed., O'Reilly Media, 2021.
9. Apache Kafka Documentation. – URL: <https://kafka.apache.org/documentation/>.
10. Redis Documentation. – URL: <https://redis.io/docs/>.
11. Prometheus Documentation. – URL: <https://prometheus.io/docs/>.
12. Grafana Documentation. – URL: <https://grafana.com/docs/>.
13. OpenTelemetry Documentation. – URL: <https://opentelemetry.io/docs/>.
14. Jaeger Documentation. – URL: <https://www.jaegertracing.io/docs/>.
15. Docker Documentation. – URL: <https://docs.docker.com/>.
16. PostgreSQL Documentation. High Availability, Load Balancing, and Replication. – URL: <https://www.postgresql.org/docs/current/high-availability.html>.

17. Nginx Documentation. HTTP Load Balancing. – URL: [https://nginx.org/en/docs/http/load\\_balancing.html](https://nginx.org/en/docs/http/load_balancing.html)

18. Андреев Р.Н., Бухаров И.А. Инжиниринг хаоса: практика внесения отказов в распределённые системы. – Открытые системы, №4, 2020. – С. 24–29.

19. Confluent Team. Exactly-once Semantics in Kafka. – URL: <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>

20. Google SRE Book. Site Reliability Engineering: How Google Runs Production Systems. – O'Reilly Media, 2016

21. Stolz S., Kienzler R. Designing Event-Driven Systems. – O'Reilly Media, 2021.

22. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. – Addison-Wesley, 2003 (Русский перевод: Шаблоны интеграции корпоративных приложений, 2007).

23. Nginx Inc. Rate Limiting with NGINX and NGINX Plus. – URL: <https://www.nginx.com/blog/rate-limiting-nginx/>

24. RabbitMQ Documentation. – URL: <https://www.rabbitmq.com/documentation.html>

25. gRPC Documentation. Flow Control. – URL: <https://grpc.io/docs/guides/flow-control/>

26. Petrov A. Database Internals: A Deep Dive into How Distributed Data Systems Work. – O'Reilly Media, 2019.

27. Baier D., Biermann F. High-Performance Web Applications with FastAPI and Redis. – Packt Publishing, 2023.