МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ ПО ДИСЦИПЛИНЕ «Алгоритмы backend разработки»

для студентов направления подготовки 09.03.04 Программная инженерия
«Разработка и сопровождение программного обеспечения»

Ставрополь, 2026

СОДЕРЖАНИЕ

Введение

Настоящее учебно-методическое пособие содержит комплекс из 31 лабораторной работы, охватывающей ключевые алгоритмы, лежащие в основе современных распределённых систем, балансировщиков нагрузки, кэшей, ограничителей частоты запросов, отказоустойчивых клиентов и инфраструктуры машинного инференса.

Цель практикума — сформировать у студентов практические навыки реализации, анализа производительности и сравнительной оценки алгоритмов, с которыми они будут сталкиваться при проектировании высоконагруженных сервисов. Каждая работа требует программной реализации алгоритма на одном из языков высокого уровня (предпочтительно Python 3), проведения вычислительного эксперимента, оформления отчёта с графиками и формулирования обоснованных выводов.

Пособие сознательно сфокусировано на **алгоритмической реализации** и не дублирует дисциплины, связанные с развёртыванием API, контейнеризацией (Docker), непрерывной интеграцией (CI/CD) или мониторингом в Grafana/Prometheus. Такой подход позволяет изучать внутренние механизмы алгоритмов в контролируемой симуляционной среде, абстрагируясь от инфраструктурных деталей.

Структура каждой лабораторной работы выдержана в едином формате:

- Цель и задачи,
- Теоретическое обоснование,
- Пример практического выполнения с кодом и визуализацией,
- Индивидуальные задания трёх уровней сложности (базовый, средний, повышенный) по вариантам,
- Контрольные вопросы,
- Требования к отчёту,
- Критерии оценивания.

Тематика работ сгруппирована в пять разделов:

1. **Алгоритмы распределения нагрузки** (работы 1–6) — Consistent Hashing, Rendezvous Hashing, Smooth Weighted Round Robin, Power of Two Choices, Phi Accrual Detection, Gossip-протокол.
2. **Алгоритмы кэширования** (работы 7–14) — реализации LRU, LFU, 2Q, ARC, Count-Min Sketch, TinyLFU, оптимальный оффлайн-алгоритм Белади, распределённое кэширование с Consistent Hashing.
3. **Алгоритмы ограничения частоты запросов** (работы 15–18) — Token Bucket, GCRA, Sliding Window Counter в Redis, сравнительный анализ rate limiter'ов.

4. **Алгоритмы согласованности** (работы 19–24) — кворумный протокол, векторные часы, CRDT-счётчики, упрощённый Raft, LWW и сравнение синхронной/асинхронной репликации.
5. **Алгоритмы отказоустойчивости и инференса** (работы 25–31) — Exponential Backoff с джиттером, Circuit Breaker, адаптивный таймаут, комплексный отказоустойчивый клиент, асинхронный dynamic batching, кэширование результатов инференса.

Выполнение лабораторных работ предполагает последовательное наращивание сложности и позволяет студенту самостоятельно выбрать уровень глубины исследования. Отчёт, подкреплённый графиками, таблицами и программным кодом, служит основным средством контроля и развивает компетенции технического письма.

Все работы апробированы в учебном процессе и ориентированы на студентов старших курсов технических направлений, изучающих распределённые вычисления, облачные технологии и архитектуру программных систем.

Лабораторная работа № 1

Реализация Consistent Hashing с виртуальными узлами

---

1. Цель работы

Цель работы – изучить принцип работы алгоритма согласованного хеширования (Consistent Hashing) с виртуальными узлами, реализовать его программно, исследовать равномерность распределения ключей по узлам и провести сравнительный анализ с наивным модульным хешированием.

---

2. Задачи

1. Реализовать хеш-функцию, отображающую идентификаторы узлов и ключей в кольцевое пространство размера $M = 2^{32}$.
2. Разработать эффективную структуру данных для кольца с операциями добавления узла, удаления узла и поиска ответственного узла по ключу.
3. Провести вычислительный эксперимент по распределению нагрузки между узлами при заданном количестве физических узлов и виртуальных точек.
4. Оценить равномерность распределения с помощью коэффициента вариации.
5. Построить гистограмму распределения ключей по узлам.
6. Сравнить согласованное хеширование с алгоритмом hash(key) mod N и сделать выводы.

---

3. Теоретическое обоснование

3.1. Проблема распределения ключей в распределённых системах

В распределённых хранилищах данных (key-value stores, CDN, кеш-серверы) необходимо отображать множество ключей на конечный набор серверов (узлов). Наивный подход – вычисление номера сервера по формуле:

$$\text{server} = H(key) \bmod N,$$

где H – хеш-функция, N – количество серверов. Такой метод критически чувствителен к изменению числа серверов: при добавлении или удалении одного сервера перераспределяется почти весь объём ключей ($\approx \frac{N-1}{N}$ данных), что приводит к лавинообразной ребалансировке и деградации производительности.

3.2. Алгоритм Consistent Hashing

Алгоритм согласованного хеширования, предложенный Дэвидом Каргером в 1997 году, решает эту проблему. Основная идея:

- Пространство выходных значений хеш-функции представляется в виде кольца (например, диапазон $[0, 2^{32} - 1]$).
- Каждый физический сервер (узел) хешируется в одну или несколько точек этого кольца.
- Ключ также хешируется в точку кольца, после чего ответственным за ключ становится сервер, чья точка расположена на кольце **по часовой стрелке** ближе всего к точке ключа (либо непосредственно следующий узел в отсортированном списке точек).
- При добавлении сервера только ключи, попадающие в его диапазон на кольце, переезжают с соседнего сервера; удаление сервера затрагивает только ключи, за которые он отвечал. Это сокращает объём перераспределяемых данных с $\frac{N-1}{N}$ до примерно $\frac{1}{N}$.

3.3. Виртуальные узлы (virtual nodes)

Равномерность распределения нагрузки в базовом Consistent Hashing сильно зависит от случайного распределения точек узлов. На практике для сглаживания дисбаланса каждому физическому узлу назначается несколько виртуальных узлов (vnode). Каждый виртуальный узел имеет собственное положение на кольце. Ответственность физического узла определяется объединением диапазонов всех его виртуальных точек. С увеличением числа виртуальных узлов нагрузка стремится к равномерной, а дисперсия уменьшается. Обычно используют 100–200 виртуальных узлов на физический сервер.

3.4. Метрики равномерности распределения

Для количественной оценки равномерности используется **коэффициент вариации**:

$$c_v = \frac{\sigma}{\mu},$$

где σ – стандартное отклонение числа ключей, назначенных узлу, μ – среднее число ключей на узел. Чем ближе c_v к нулю, тем равномернее распределение. Хорошим ориентиром является $c_v < 0.1$ при достаточном количестве виртуальных узлов.

3.5. Структура данных кольца

Кольцо может быть реализовано как отсортированный список точек с бинарным поиском. Каждая запись в кольце содержит:

- позицию на кольце (целое число в диапазоне $[0, 2^{32} - 1]$),
- идентификатор физического узла,
- (опционально) идентификатор виртуального узла.

Для поиска узла по ключу: вычисляем хеш ключа, находим в отсортированном списке первую точку с позицией ≥ хеш ключа (например, с помощью bisect_left). Если такая точка не найдена (хеш больше всех точек), то ответственным является первый узел в списке (замыкание кольца).

---

4. Пример практического выполнения

4.1. Выбор хеш-функции

В качестве хеш-функции воспользуемся SHA-1 из стандартной библиотеки Python (hashlib.sha1). Она возвращает 160-битное значение; для кольца размера 2^{32} будем брать младшие 4 байта, преобразуя их в целое число.

python

```
import hashlib


RING_SIZE = 2**32


def hash_to_ring(key: str) -> int:
    """Отобразить строку key в точку кольца [0, RING_SIZE-1]."""
    h = hashlib.sha1(key.encode('utf-8')).hexdigest()
    # Возьмём последние 8 символов (4 байта) для 32-битного числа
    return int(h[-8:], 16) % RING_SIZE
```

Примечание: в реальном коде можно брать любые 8 символов (4 байта) и не применять % RING_SIZE, так как значение и так 32-битное. Здесь % оставлен для перестраховки.

4.2. Реализация кольца с виртуальными узлами

python

```
import bisect


class ConsistentHashRing:
    def __init__(self):
        self.ring = []          # отсортированный список позиций
        self.pos_to_node = {}   # позиция -> физический узел
        self.nodes = set()      # множество физических узлов
```

```
    def add_node(self, node_id: str, virtual_nodes: int = 100):
        """Добавить физический узел с указанным числом виртуальных узлов."""
        if node_id in self.nodes:
            raise ValueError(f"Узел {node_id} уже существует")
        self.nodes.add(node_id)
        for i in range(virtual_nodes):
            vnode_key = f"{node_id}::vnode{i}"
            pos = hash_to_ring(vnode_key)
            bisect.insort(self.ring, pos)
            self.pos_to_node[pos] = node_id

    def remove_node(self, node_id: str):
        """Удалить физический узел и все его виртуальные точки."""
        if node_id not in self.nodes:
            raise ValueError(f"Узел {node_id} не найден")
        self.nodes.remove(node_id)
        # Определим позиции для удаления
        to_remove = [pos for pos, n in self.pos_to_node.items() if n == node_id]
        for pos in to_remove:
            self.ring.remove(pos)        # удаление из списка (O(n) – допустимо для наглядности)
            del self.pos_to_node[pos]
        # Пересобираем список, чтобы сохранить сортировку (в рабочем коде можно удалять с учётом индексов)
        self.ring.sort()

    def get_node(self, key: str) -> str:
        """Определить физический узел, ответственный за ключ."""
        if not self.ring:
```

```
            raise RuntimeError("Кольцо пусто")
        key_hash = hash_to_ring(key)
        idx = bisect.bisect_left(self.ring, key_hash)
        if idx == len(self.ring):
            idx = 0  # закольцовка
        node_pos = self.ring[idx]
        return self.pos_to_node[node_pos]
```

4.3. Проведение эксперимента

Параметры: 100 000 случайных ключей, 5 физических узлов, 100 виртуальных узлов на каждый. Для генерации ключей используем UUID.

```
python
import uuid
import random
import statistics
import matplotlib.pyplot as plt
from collections import Counter

# Инициализация кольца
ring = ConsistentHashRing()
node_ids = [f"node-{i}" for i in range(5)]
for n in node_ids:
    ring.add_node(n, virtual_nodes=100)

# Генерация 100 000 случайных ключей
random.seed(42)
keys = [str(uuid.uuid4()) for _ in range(100_000)]

# Распределение ключей
distribution = Counter()
```

```
for key in keys:
    responsible = ring.get_node(key)
    distribution[responsible] += 1

# Метрики
counts = [distribution[n] for n in node_ids]
mean_load = statistics.mean(counts)
std_load = statistics.stdev(counts)
cv = std_load / mean_load

print(f"Средняя нагрузка: {mean_load:.2f}")
print(f"Стандартное отклонение: {std_load:.2f}")
print(f"Коэффициент вариации: {cv:.4f}")
```

4.4. Сравнение с модульным хешированием

Наивный метод: hash(key) mod 5. Для честного сравнения используем ту же хеш-функцию и приведём значение к 32-битному.

```
python
def mod_n_node(key: str, N: int = 5) -> str:
    idx = hash_to_ring(key) % N
    return node_ids[idx]

mod_dist = Counter()
for key in keys:
    mod_dist[mod_n_node(key)] += 1

mod_counts = [mod_dist[n] for n in node_ids]
mod_cv = statistics.stdev(mod_counts) / statistics.mean(mod_counts)
print(f"Коэффициент вариации (mod N): {mod_cv:.4f}")
```

4.5. Визуализация распределения

Построим гистограмму для сравнительного анализа.

python

```
labels = node_ids

x = range(len(labels))

width = 0.35


fig, ax = plt.subplots(figsize=(10, 6))

bar1 = ax.bar([i - width/2 for i in x], counts, width, label='Consistent Hashing', color='steelblue')

bar2 = ax.bar([i + width/2 for i in x], mod_counts, width, label='mod N', color='salmon')


ax.set_xlabel('Узлы')

ax.set_ylabel('Число ключей')

ax.set_title('Распределение ключей по узлам')

ax.set_xticks(x)

ax.set_xticklabels(labels)

ax.legend()

ax.grid(axis='y', linestyle='--', alpha=0.7)


for bar in bar1:

   yval = bar.get_height()

   ax.text(bar.get_x() + bar.get_width()/2, yval + 200, f'{int(yval)}', ha='center', va='bottom', fontsize=8)

for bar in bar2:

   yval = bar.get_height()

   ax.text(bar.get_x() + bar.get_width()/2, yval + 200, f'{int(yval)}', ha='center', va='bottom', fontsize=8)


plt.tight_layout()

plt.show()
```

Ожидаемый результат: гистограмма покажет более равномерное распределение у согласованного хеширования с виртуальными узлами ($c_v < 0.05$), в то время как модульное хеширование также будет достаточно равномерным, но при изменении числа узлов оно окажется катастрофически нестабильным. Убедитесь, что при удалении одного узла из кольца мигрирует лишь часть ключей (примерно $\frac{1}{5}$ от общего числа), тогда как при mod N мигрировали бы почти все.

---

5. Задания для индивидуального выполнения

Номер варианта определяется преподавателем или рассчитывается как (номер_студента_по_списку mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать алгоритм Consistent Hashing с виртуальными узлами и провести эксперимент согласно варианту. Построить гистограмму и вычислить коэффициент вариации.

Вариант	Число физических узлов	Число виртуальных узлов на каждый	Количество ключей	Размер кольца (бит)	Хеш-функция
1	3	50	50 000	32	SHA-1
2	4	80	80 000	32	MD5
3	5	100	100 000	32	SHA-1
4	6	120	120 000	32	SHA-256 (младшие 4 байта)
5	7	150	150 000	32	MD5
6	8	200	200 000	32	SHA-1
7	5	10	100 000	32	SHA-1
8	5	25	100 000	32	SHA-1

Вариант	Число физических узлов	Число виртуальных узлов на каждый	Количество ключей	Размер кольца (бит)	Хеш-функция
9	5	50	100 000	32	MD5
10	5	50	100 000	16 (кольцо 0–65535)	SHA-1
11	3	200	100 000	32	SHA-256
12	10	50	200 000	32	SHA-1
13	10	100	200 000	32	MD5
14	10	20	100 000	32	SHA-1
15	20	50	500 000	32	SHA-1

Требования для оценки «удовлетворительно»: корректный код, вывод гистограммы и рассчитанного коэффициента вариации, сравнение с mod N.

5.2. Средний уровень (на оценку «хорошо»)

Кроме базовой реализации, необходимо:

- Реализовать имитацию динамического изменения состава узлов: добавить 1 новый узел, затем удалить 2 случайных узла.
- Замерить долю перераспределённых ключей после каждой операции и сравнить с теоретической оценкой $\approx 1/N$.
- Провести сравнение с алгоритмом Rendezvous Hashing (он же Highest Random Weight) для статического набора узлов (без изменений) и оценить равномерность распределения.
- Построить график зависимости коэффициента вариации от числа виртуальных узлов (10, 25, 50, 75, 100, 150, 200 при 5 физических узлах).

Вариант	Базовые параметры (из табл. базового уровня)	Дополнительно
1	Вариант 1 (3 узла, 50 vnode, 50k ключей)	См. выше
2	Вариант 2	--
3	Вариант 3	--
4	Вариант 4	--
5	Вариант 5	--
6	Вариант 6	--
7	Вариант 7	--
8	Вариант 8	--
9	Вариант 9	--
10	Вариант 10	--
11	Вариант 11	--
12	Вариант 12	--
13	Вариант 13	--
14	Вариант 14	--
15	Вариант 15	--

Требования для оценки «хорошо»: выполнены все пункты базового уровня, дополнительно реализованы удаление/добавление с замером миграции, сравнение с Rendezvous Hashing, график c_v от числа vnode.

5.3. Повышенный уровень (на оценку «отлично»)

Помимо требований среднего уровня:

- Реализовать собственную хеш-функцию (например, на базе xxHash, murmur3 или упрощённый вариант на основе полиномиального хеширования) и доказать её корректность.
- Провести статистический анализ равномерности с помощью критерия хи-квадрат (χ^2) для распределения ключей по узлам; сделать вывод о согласии с равномерным распределением.
- Реализовать взвешенный Consistent Hashing, когда узлы имеют разную ёмкость (вес). Виртуальные узлы распределяются пропорционально весу.
- Исследовать, как неравномерность хеш-функции (например, намеренно плохая) влияет на распределение нагрузки, сравнить с идеальным случаем.
- Построить тепловую карту перераспределения ключей при последовательном добавлении узлов с 1 до 10.

Вариант	Параметры базовой конфигурации
1	5 физ. узлов, 100 vnode, 200k ключей
2	6 физ. узлов, 120 vnode, 150k ключей
3	7 физ. узлов, 150 vnode, 200k ключей
4	8 физ. узлов, 200 vnode, 300k ключей
5	10 физ. узлов, 100 vnode, 500k ключей
6	10 физ. узлов, 50 vnode, 500k ключей
7	5 физ. узлов, 200 vnode, 100k ключей
8	4 физ. узлов, 300 vnode, 120k ключей
9	3 физ. узлов, 500 vnode, 90k ключей
10	12 физ. узлов, 80 vnode, 400k ключей

Вариант	Параметры базовой конфигурации
11	15 физ. узлов, 100 vnode, 600k ключей
12	20 физ. узлов, 100 vnode, 800k ключей
13	5 физ. узлов, 150 vnode, 100k ключей
14	5 физ. узлов, 150 vnode, 200k ключей
15	5 физ. узлов, 150 vnode, 300k ключей

Требования для оценки «отлично»: полный объём исследований, включая собственную хеш-функцию, статистический тест, взвешенное распределение, анализ качества хеша, тепловая карта миграции.

---

6. Контрольные вопросы

1. В чём заключается проблема наивного модульного хеширования при изменении числа серверов?
2. Как работает алгоритм Consistent Hashing? Что такое хеш-кольцо?
3. Почему в реальных системах используются виртуальные узлы? Как их количество влияет на равномерность и накладные расходы?
4. Какая структура данных оптимальна для представления кольца? Оцените сложность операций поиска, добавления и удаления.
5. Какие существуют альтернативы Consistent Hashing (например, Rendezvous Hashing)? В чём их различия?
6. Как вычислить объём данных, который необходимо переместить при добавлении/удалении одного узла в Consistent Hashing?
7. Что такое коэффициент вариации и как он интерпретируется в контексте распределения ключей?
8. Какие хеш-функции предпочтительнее для Consistent Hashing и почему?
9. Как изменится распределение, если хеш-функция даёт неравномерный выход?
10. Можно ли применять Consistent Hashing без виртуальных узлов? В каких случаях это допустимо?

---

7. Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с названием работы, ФИО, группой.
2. **Цель и задачи** (кратко).
3. **Теоретическая часть** – описание алгоритма, обоснование выбора структур данных и хеш-функции.
4. **Описание реализации** – листинг ключевых фрагментов кода с комментариями, скриншоты или текстовый вывод.
5. **Экспериментальная часть**:
    - Параметры эксперимента.
    - Результаты распределения ключей.
    - Гистограмма распределения ключей по узлам (для Consistent Hashing и mod N).
    - Сравнительная таблица с коэффициентами вариации.
    - (Для среднего и повышенного уровня) дополнительные графики и метрики.
6. **Анализ результатов** – интерпретация полученных данных, связь с теорией.
7. **Выводы** — обобщение, достоинства и недостатки исследованного подхода.
8. **Код программы** (можно вынести в приложение).

---

8. Критерии оценивания

8.1. Базовый уровень – оценка «удовлетворительно»

- Программа реализует хеш-кольцо с виртуальными узлами, присутствуют методы add_node, remove_node, get_node.
- Проведён эксперимент согласно варианту, вычислен коэффициент вариации.
- Построена гистограмма распределения, проведено сравнение с mod N.
- Отчёт содержит минимально требуемые разделы, выводы корректны, но поверхностны.

Снижение оценки: ошибки в реализации, отсутствие сравнения, неверно построенный график, нет вывода коэффициента вариации.

8.2. Средний уровень – оценка «хорошо»

- Все требования базового уровня выполнены полностью и без ошибок.
- Реализовано динамическое добавление/удаление узлов, посчитана и проанализирована доля перемещённых ключей.
- Реализован прототип Rendezvous Hashing, проведено сравнение равномерности с Consistent Hashing.
- Построен график зависимости c_v от числа виртуальных узлов.
- Выводы содержат анализ влияния виртуальных узлов, сравнение алгоритмов.

Снижение оценки: не все дополнительные пункты выполнены, недостаточный анализ миграции, отсутствует сравнение с Rendezvous.

8.3. Повышенный уровень – оценка «отлично»

- Все требования среднего уровня выполнены.
- Реализована собственная хеш-функция, доказана её корректность (хотя бы эмпирически).
- Проведён статистический анализ распределения (χ^2-тест), дано заключение.
- Реализован взвешенный Consistent Hashing, проверена пропорциональность нагрузки.
- Построена тепловая карта миграции, сделаны соответствующие выводы.
- Отчёт логически структурирован, содержит глубокий анализ, все графики подписаны, код чистый и снабжён комментариями.

Снижение оценки: неполное исследование, нет статистического теста или его интерпретация ошибочна, весовая схема работает некорректно.

Рекомендация: при реализации на Python используйте библиотеки hashlib, bisect, matplotlib, numpy (для расчётов), collections. Для Rendezvous Hashing полезно ознакомиться с оригинальной статьёй или использовать готовые адаптации. Помните о производительности: при большом числе виртуальных узлов (свыше 10000) бинарный поиск всё ещё эффективен, но удаление элементов из списка может стать медленным – в учебных целях это допустимо, но в отчёте обсудите возможные оптимизации (красно-чёрное дерево, упорядоченный массив с отметками об удалении).

Лабораторная работа № 2

Сравнительный анализ Consistent Hashing и Rendezvous Hashing

---

1. Цель работы

Цель работы – изучить альтернативный метод распределения ключей по серверам – **Rendezvous Hashing** (Highest Random Weight, HRW), реализовать его программно и провести сравнительный анализ с **Consistent Hashing** по ключевым критериям: равномерность распределения, объём миграции ключей при изменении числа узлов и скорость вычисления ответственного сервера.

---

2. Задачи

1. Реализовать алгоритм Rendezvous Hashing с возможностью добавления и удаления узлов.
2. Реализовать оптимизированную версию с skeleton-структурой (кэширование K лучших кандидатов).
3. Провести эксперимент с динамическим изменением количества узлов (3 → 4 → 5) для обоих алгоритмов.
4. Измерить:
    - долю ключей, сменивших владельца при добавлении узла (коэффициент миграции);
    - равномерность распределения (коэффициент вариации);
    - среднее время выполнения операции get_node(key) для разного числа узлов.
5. Построить сравнительные графики времени и равномерности для N ∈ {10, 50, 100, 500}.
6. Сформировать таблицу сравнения и дать рекомендации по выбору алгоритма в зависимости от сценария использования.

---

3. Теоретическое обоснование

3.1. Постановка задачи

Задача отображения ключа на один из N серверов возникает в распределённых кэшах, CDN и базах данных. Два наиболее известных алгоритма – **Consistent Hashing** (рассмотренный в лабораторной работе №1) и **Rendezvous Hashing** (также называемый Highest Random Weight). Они обеспечивают минимальную перестройку распределения при изменении числа узлов, но имеют разную алгоритмическую сложность и свойства.

3.2. Алгоритм Rendezvous Hashing

В основе Rendezvous Hashing лежит следующая идея: для каждого ключа k вычисляется «вес» (псевдослучайное число) для каждого сервера s_i с помощью хеш-функции, принимающей на вход конкатенацию ключа и идентификатора сервера:

$$w_i = H(k, s_i)$$

Сервер, давший **наибольший** вес, назначается ответственным за ключ:

$$\text{server}(k) = \arg\max_{i=1..N} H(k, s_i)$$

При добавлении нового сервера s_{new} ключ перемещается на него **только в том случае**, если для него вес нового сервера окажется выше, чем вес текущего назначенного сервера. При удалении сервера ключ переходит к тому из оставшихся, который даёт наивысший вес. Теоретически объём мигрирующих ключей составляет примерно 1/N как при добавлении, так и при удалении.

3.3. Сравнение с Consistent Hashing

Критерий	Consistent Hashing	Rendezvous Hashing
Сложность поиска	$O(\log(N \cdot V))$[1]	$O(N)$ (полный перебор)
Использование памяти	$O(N \cdot V)$ (виртуальные точки)	$O(N)$ (только метаданные серверов)
Равномерность распределения	Требует виртуальных узлов для хорошей равномерности	Изначально равномерное без настройки
Миграция ключей	$\approx 1/N$, зависит от вирт. узлов	$\approx 1/N$, не зависит от истории
Зависимость от кольца	Да, структура кольца	Нет, только список серверов
Простота реализации	Средняя (нужен бинарный поиск)	Очень простая

[1] V – число виртуальных узлов на физический сервер.

Ключевой недостаток Rendezvous Hashing – линейная сложность поиска по числу серверов. Однако при небольшом N (десятки – первые сотни) полный перебор может быть быстрее, чем операции с кольцом и бинарным поиском, особенно если хеш-функция быстрая.

3.4. Оптимизация «skeleton» (Top-K кэширование)

Для ускорения Rendezvous Hashing применяют эвристику: для каждого ключа кэшируются K лучших серверов (с наивысшими весами) и при добавлении новых узлов вычисляется вес только нового сервера; если он превышает K-е значение – кэш обновляется. Поскольку добавление одного сервера ломает распределение редко и затрагивает только ключи, для которых он стал top-K, такая оптимизация значительно снижает амортизированную стоимость поиска. При K=1 (skeleton размером 1) при добавлении сервера достаточно сравнить вес нового сервера с весом текущего назначенного сервера. В честном сравнении мы также можем учитывать стоимость поддержки кэша при удалении серверов.

3.5. Метрики сравнения

- **Коэффициент миграции** – доля ключей, сменивших сервер после изменения топологии.
- **Коэффициент вариации (CV)** – стандартное отклонение числа ключей на узле, делённое на среднее:

$$c_v = \frac{\sigma}{\mu}$$

- **Среднее время** get_node – замер процессорного времени выполнения метода определения сервера.

---

4. Пример практического выполнения

4.1. Реализация базового Rendezvous Hashing

Используем SHA-1 и получение 32-битного целого из первых 8 символов дайджеста (как в ЛР1). Класс RendezvousHash без оптимизации.

python

```
import hashlib


def hrw_hash(key: str, node_id: str) -> int:
    """Вес пары (key, node_id) как 32-битное целое."""
    combined = f"{key}::{node_id}"
```

```
    h = hashlib.sha1(combined.encode('utf-8')).hexdigest()
    return int(h[:8], 16)  # старшие 4 байта для разнообразия


class RendezvousHash:
    def __init__(self):
        self.nodes = set()

    def add_node(self, node_id: str):
        self.nodes.add(node_id)

    def remove_node(self, node_id: str):
        self.nodes.discard(node_id)

    def get_node(self, key: str) -> str:
        if not self.nodes:
            raise RuntimeError("Нет доступных узлов")
        best_node = None
        best_weight = -1
        for node in self.nodes:
            w = hrw_hash(key, node)
            if w > best_weight:
                best_weight = w
                best_node = node
        return best_node
```

4.2. Реализация Consistent Hashing (для сравнения)

Возьмём реализацию из ЛР1 с параметром virtual_nodes=100. Можно оформить в виде класса ConsistentHashRing.

4.3. Эксперимент с миграцией ключей

Рассмотрим сценарий: начальные 3 узла, затем добавляем 4-й, потом 5-й. Для каждого этапа фиксируем распределение 100 000 случайных ключей.

```
python
import uuid, random, time
from collections import Counter

random.seed(123)
keys = [str(uuid.uuid4()) for _ in range(100_000)]

# Инициализация
nodes = ["node-1", "node-2", "node-3"]
rh = RendezvousHash()
for n in nodes:
    rh.add_node(n)

# Первичное распределение
dist_rh_before = Counter()
for key in keys:
    dist_rh_before[rh.get_node(key)] += 1

# Добавление узла 4
rh.add_node("node-4")
dist_rh_after_4 = Counter()
migrated_4 = 0
for key in keys:
    assigned = rh.get_node(key)
    dist_rh_after_4[assigned] += 1
    if assigned != dist_rh_before.get(key, None):
        # dist_rh_before имеет владельца для каждого ключа
        migrated_4 += 1
```

```
print(f"Мигрировало ключей после добавления 4-го узла: {migrated_4} ({100 * migrated_4 / len(keys):.2f}%)")


# Аналогично для 5-го...
```

Аналогичный тест проведём для Consistent Hashing с 100 виртуальными узлами.

4.4. Замер времени get_node для разного N

Создадим функции, измеряющие среднее время на 10 000 вызовов для N = 10, 50, 100, 500.

```
python
import timeit


def benchmark_get_node(algo, keys_subset):
    start = time.perf_counter()
    for key in keys_subset:
        algo.get_node(key)
    end = time.perf_counter()
    return (end - start) / len(keys_subset)


test_keys = keys[:10000]
Ns = [10, 50, 100, 500]
times_rh = []
times_ch = []


for N in Ns:
    # Создаём экземпляры с N узлами
    rh_test = RendezvousHash()
    ch_test = ConsistentHashRing()
    for i in range(N):
        node_id = f"node-{i}"
        rh_test.add_node(node_id)
```

```
        ch_test.add_node(node_id, virtual_nodes=100)
    t_rh = benchmark_get_node(rh_test, test_keys)
    t_ch = benchmark_get_node(ch_test, test_keys)
    times_rh.append(t_rh * 1e6)  # в микросекундах
    times_ch.append(t_ch * 1e6)
```

4.5. Визуализация результатов

```
python
import matplotlib.pyplot as plt

# График времени
plt.figure(figsize=(10,5))
plt.plot(Ns, times_rh, 'o-', label='Rendezvous (O(N))')
plt.plot(Ns, times_ch, 's--', label='Consistent Hashing (100 vnode)')
plt.xlabel('Число узлов')
plt.ylabel('Время get_node, мкс')
plt.title('Сравнение времени выполнения get_node')
plt.legend()
plt.grid(True)
plt.show()

# Гистограмма миграции
algo_names = ['Consistent Hashing', 'Rendezvous']
migration = [migrated_ch_4, migrated_rh_4]  # надо вычислить
# ...
```

Ожидаемый результат: Rendezvous быстрее при малом N, но с ростом N линейный рост становится заметным, в то время как Consistent Hashing демонстрирует логарифмический рост. Коэффициент миграции для обоих алгоритмов около 1/N.

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента_по_списку mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать алгоритмы Consistent Hashing (с фиксированным числом виртуальных узлов $V = 100$) и Rendezvous Hashing. Провести эксперимент по варианту: задано начальное количество узлов N_start, последовательно добавляются узлы до N_end. Измерить коэффициент миграции при каждом добавлении и коэффициент вариации финального распределения. Построить гистограмму распределения ключей по узлам до и после последнего добавления.

Вариант	N_start – N_end	Количество ключей	Примечание
1	3 → 5	10 000	
2	3 → 5	50 000	
3	4 → 6	20 000	
4	4 → 6	60 000	
5	5 → 7	100 000	
6	5 → 7	150 000	
7	6 → 8	30 000	
8	7 → 9	40 000	
9	8 → 10	50 000	
10	10 → 12	80 000	
11	10 → 12	120 000	
12	12 → 15	200 000	
13	15 → 18	250 000	

Вариант	N_start – N_end	Количество ключей	Примечание
14	3 → 6	5 000	
15	5 → 8	75 000	

Требования к отчёту базового уровня: корректная реализация обоих алгоритмов, измерение процента миграции и CV, гистограммы, вывод о сравнении.

5.2. Средний уровень (на оценку «хорошо»)

В дополнение к базовому уровню:

- Реализовать оптимизацию «skeleton» для Rendezvous Hashing с размером кэша K (задаётся вариантом). При добавлении/удалении узлов показать, как обновляется кэш и как это влияет на время.
- Провести замеры среднего времени get_node для обоих алгоритмов (Consistent Hashing с $V = 100$) при числе узлов $N \in \{10,50,100,500\}$. Построить график времени.
- Сравнить равномерность с Consistent Hashing при разном количестве виртуальных узлов (50, 100, 200) и для Rendezvous — при разных K.
- Проанализировать зависимость миграции от шага добавления (например, добавление по одному узлу, потом удаление одного).

Вариант	Параметры базового уровня	K (размер skeleton)	Дополнительно
1	Вариант 1	1	
2	Вариант 2	2	
3	Вариант 3	3	
4	Вариант 4	1	
5	Вариант 5	2	
6	Вариант 6	3	

Вариант	Параметры базового уровня	K (размер skeleton)	Дополнительно
7	Вариант 7	1	
8	Вариант 8	2	
9	Вариант 9	3	
10	Вариант 10	1	
11	Вариант 11	2	
12	Вариант 12	3	
13	Вариант 13	1	
14	Вариант 14	2	
15	Вариант 15	3	

Требования: все пункты базового уровня выполнены, дополнительно реализован skeleton, замеры времени с графиком, анализ равномерности для разных V у Consistent Hashing, выводы о влиянии K и V.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **взвешенный Rendezvous Hashing** (веса узлов различаются: вес влияет на вероятность выбора). Классический подход – использовать $w_i = H(k, s_i)^{1/weight_i}$ или сравнение $weight_i \cdot H(k, s_i)$. Необходимо доказать корректность и измерить соответствие распределения заданным весам.
- Провести анализ потребления памяти обоими алгоритмами при $N = 1000$ (Consistent Hashing с разным числом виртуальных узлов).
- Реализовать сравнение с различными хеш-функциями (MD5, SHA-1, SHA-256, xxHash из библиотеки xxhash) и оценить их влияние на время get_node и равномерность.
- Построить трёхмерный график (например, heatmap) зависимости процента миграции от числа узлов и количества виртуальных точек/свойств скелетона.

- Изучить «эффект холодного старта»: распределение при очень малом количестве узлов и ключей.

Вариант	Параметры базового уровня	Особое задание
1	5 узлов, 100k ключей	Взвешенный HRW: веса [1,2,3,4,5]
2	6 узлов, 120k ключей	Взвешенный HRW: веса [2,2,3,3,4,4]
3	7 узлов, 150k ключей	Сравнение хеш-функций (MD5, SHA-1, xxHash)
4	8 узлов, 200k ключей	Анализ памяти, N=1000
5	10 узлов, 250k ключей	Heatmap миграции
6	3 узла, 80k ключей	Взвешенный HRW: [1,2,4]
7	10 узлов, 300k ключей	Сравнение хеш-функций + быстродействие
8	12 узлов, 400k ключей	Анализ холодного старта (N от 1 до 12)
9	15 узлов, 500k ключей	Взвешенный HRW: веса пропорциональны i
10	20 узлов, 600k ключей	Heatmap миграции
11	5 узлов, 200k ключей	Взвешенный HRW + хеш-функции
12	8 узлов, 350k ключей	Анализ памяти и времени
13	4 узла, 90k ключей	Взвешенный HRW: веса [3,1,1,1]
14	6 узлов, 180k ключей	Heatmap миграции
15	9 узлов, 270k ключей	Все дополнительные пункты (на выбор преподавателя)

Требования: обширный отчёт с глубоким анализом, сравнение по всем метрикам, нестандартные визуализации, обоснованные выводы.

---

6. Контрольные вопросы

1. В чём состоит принцип Highest Random Weight? Почему он обеспечивает минимальную миграцию?
2. Какова алгоритмическая сложность определения сервера в Rendezvous Hashing без кэширования? В каких пределах числа узлов это допустимо?
3. Что такое «skeleton» в контексте Rendezvous Hashing? Как размер кэша K влияет на компромисс между памятью и скоростью?
4. Какие преимущества и недостатки имеет Rendezvous Hashing по сравнению с Consistent Hashing?
5. При каком количестве серверов Consistent Hashing с виртуальными узлами становится эффективнее Rendezvous Hashing с точки зрения быстродействия?
6. Можно ли построить взвешенное распределение в Rendezvous Hashing? Какие математические приёмы для этого используются?
7. Как изменятся показатели миграции при удалении узла вместо добавления?
8. Почему при использовании Rendezvous Hashing не требуется предварительное построение кольца и виртуальных узлов?
9. Какие хеш-функции наиболее предпочтительны для Rendezvous Hashing и почему?
10. В каких практических системах (базы данных, CDN) применяется Rendezvous Hashing?

---

7. Требования к отчёту

Отчёт оформляется в соответствии с общими требованиями кафедры и должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическое введение (Rendezvous Hashing, сравнение, скелетон).
4. Программная реализация: описание структур данных, ключевые фрагменты кода.
5. Экспериментальная часть:
    - Параметры экспериментов для своего варианта.
    - Таблицы и графики миграции, равномерности, времени.

- Гистограммы распределения ключей.
- Сравнительный анализ двух алгоритмов по всем метрикам.

6. Выводы по работе, обоснование выбора алгоритма для разных сценариев.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Оба алгоритма реализованы верно.
- Проведён эксперимент добавления узлов согласно варианту, вычислены коэффициент миграции и коэффициент вариации.
- Построены требуемые гистограммы.
- Отчёт содержит основные разделы и выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализован skeleton для Rendezvous Hashing, продемонстрирована его работа.
- Построен график времени get_node для N = 10, 50, 100, 500.
- Проведён анализ влияния числа виртуальных узлов и размера скелетона.
- Выводы аргументированы, содержат сравнение алгоритмов.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован взвешенный Rendezvous Hashing с проверкой соответствия весам.
- Проведён анализ памяти и/или влияния хеш-функций.
- Построены дополнительные графики (heatmap, 3D и т.п.).
- Исследование носит завершённый характер, отчёт отлично структурирован, код оптимизирован и документирован.

Примечание: несоблюдение требований к реализации, отсутствие графиков или ошибки в логике алгоритма ведут к снижению оценки на любом уровне.

Лабораторная работа № 3

Реализация Smooth Weighted Round Robin (SWRR)

---

1. Цель работы

Цель работы — изучить алгоритм **Smooth Weighted Round Robin** (SWRR), применяемый в высоконагруженных балансировщиках (например, Nginx), реализовать его программно, сравнить с классическим взвешенным круговым обслуживанием по критериям равномерности и отсутствию «всплесков», а также оценить количественные метрики гладкости диспетчеризации запросов.

---

2. Задачи

1. Изучить принцип работы классического Weighted Round Robin (WRR) и алгоритма гладкого взвешенного раунд-робина (SWRR).
2. Реализовать на Python два класса диспетчеров: WeightedRoundRobin и SmoothWeightedRoundRobin с методами add_server(server_id, weight) и next_server().
3. Провести вычислительный эксперимент на заданной конфигурации весов (например, [5, 3, 1]) при числе запросов $10\,000$.
4. Для каждого алгоритма измерить:
    - отклонение фактической доли запросов от идеальной (заданной весами);
    - коэффициент всплесков (burstiness) — максимальное число последовательных запросов, направленных одному серверу.
5. Визуализировать последовательность выбора серверов по шагам.
6. Сравнить результаты и сделать выводы о преимуществах SWRR с точки зрения «гладкости» диспетчеризации.

---

3. Теоретическое обоснование

3.1. Классический взвешенный Round Robin (WRR)

Пусть имеется N серверов с весами $w_1, w_2, \dots, w_N$. Классический WRR формирует циклическую очередь, в которой каждый сервер s_i повторяется w_i раз подряд, и последовательно выбирает серверы из этой очереди. Например, для весов [5, 3, 1] очередь выглядит как [s0, s0, s0, s0, s0, s1, s1, s1, s2], затем повторяется.

Такой подход обеспечивает точное соблюдение долгосрочных пропорций, однако приводит к **неравномерности на коротких интервалах**: один сервер может получать

множество запросов подряд, создавая пиковую нагрузку. Максимальная длина непрерывной серии (burst) для сервера s_i равна w_i (в классической реализации без перемешивания).

3.2. Алгоритм Smooth Weighted Round Robin (SWRR)

Алгоритм SWRR, известный по реализации в Nginx, лишён этого недостатка. Он поддерживает для каждого сервера «текущий вес» (dynamic current weight) и на каждом шаге выбирает сервер с наибольшим значением этого параметра, имитируя равномерное прореживание. Псевдокод алгоритма:

text

```
Инициализация:
 для каждого сервера i:
    current_weight[i] = 0
    effective_weight[i] = weight[i]
 total_weight = sum(weight[i])

Функция next_server():
 best = -1
 для каждого сервера i:
    current_weight[i] += effective_weight[i]
    если best == -1 ИЛИ current_weight[i] > current_weight[best]:
       best = i
 current_weight[best] -= total_weight
 вернуть best
```

Интуитивно: на каждом шаге все серверы «накапливают» вес, а тот, кто накопил больше всех, получает запрос и «тратит» накопленное преимущество. Это приводит к частому чередованию серверов с большими весами, предотвращая длинные серии.

3.3. Метрики качества балансировки

- **Идеальная доля** сервера i: $p_i = \frac{w_i}{\sum_{j=1}^{N} w_j}$.
- **Фактическая доля** за R запросов: $f_i = \frac{\text{count}_i}{R}$.
- **Отклонение распределения** – может оцениваться как максимальное абсолютное отклонение:

$$\Delta = \max_i | f_i - p_i |$$

или среднее абсолютное отклонение $\frac{1}{N}\sum_i | f_i - p_i |$.

- **Коэффициент всплесков (burstiness)** – максимальная длина непрерывной последовательности запросов, приходящейся на один сервер:

$$B = \max_i \text{run\_length}_i.$$

Для классического WRR $B = \max_i w_i$ (при целых весах); для SWRR B значительно меньше, обычно не превышает $\max_i w_i$, но в хорошо разбавленной последовательности может быть 1–2 при большом числе серверов.

---

4. Пример практического выполнения

4.1. Реализация классов на Python

Классический Weighted Round Robin

python

```
from collections import deque
import itertools

class WeightedRoundRobin:
    def __init__(self):
        self.servers = {}         # server_id -> weight
        self.queue = deque()      # очередь серверов

    def add_server(self, server_id, weight):
        self.servers[server_id] = weight

    def build_queue(self):
        # Создаём очередь: повторяем каждый сервер weight раз подряд
        self.queue = deque()
        for sid, w in self.servers.items():
```

```
            self.queue.extend([sid] * w)

    def next_server(self):
        if not self.queue:
            self.build_queue()
        return self.queue.popleft()
```

Smooth Weighted Round Robin (SWRR)

python

```
class SmoothWeightedRoundRobin:
    def __init__(self):
        self.servers = []          # list of dicts: {id, weight, current_weight}
        self.total_weight = 0

    def add_server(self, server_id, weight):
        self.servers.append({
            'id': server_id,
            'weight': weight,
            'current_weight': 0
        })
        self.total_weight += weight

    def next_server(self):
        best = None
        for server in self.servers:
            server['current_weight'] += server['weight']
            if best is None or server['current_weight'] > best['current_weight']:
                best = server
        best['current_weight'] -= self.total_weight
        return best['id']
```

4.2. Эксперимент с весами [5, 3, 1] и 10 000 запросов

python

```
# Инициализация
swrr = SmoothWeightedRoundRobin()
wrr = WeightedRoundRobin()
for sid, w in zip(['s0','s1','s2'], [5,3,1]):
    swrr.add_server(sid, w)
    wrr.add_server(sid, w)
wrr.build_queue()


# Генерация последовательности запросов
TOTAL = 10000
seq_swrr = [swrr.next_server() for _ in range(TOTAL)]
seq_wrr  = [wrr.next_server() for _ in range(TOTAL)]
```

4.3. Расчёт метрик

python

```
from collections import Counter, defaultdict
import statistics


def calculate_metrics(seq, weights_dict, total_req):
    ideal = {sid: w/sum(weights_dict.values()) for sid, w in weights_dict.items()}
    counts = Counter(seq)
    actual = {sid: counts.get(sid, 0)/total_req for sid in weights_dict}
    # Максимальное абсолютное отклонение
    deviation = max(abs(actual[sid] - ideal[sid]) for sid in weights_dict)
    # Коэффициент всплесков: максимальная длина серии
    burst = 0
    current_sid = None
    current_len = 0
```

```
    for sid in seq:
        if sid == current_sid:
            current_len += 1
        else:
            if current_len > burst:
                burst = current_len
            current_sid = sid
            current_len = 1
    burst = max(burst, current_len)
    return deviation, burst

weights = {'s0':5, 's1':3, 's2':1}
dev_swrr, burst_swrr = calculate_metrics(seq_swrr, weights, TOTAL)
dev_wrr, burst_wrr = calculate_metrics(seq_wrr, weights, TOTAL)

print(f"SWRR: отклонение={dev_swrr:.4f}, burstiness={burst_swrr}")
print(f"WRR : отклонение={dev_wrr:.4f}, burstiness={burst_wrr}")
```

Ожидаемый результат:

- SWRR: отклонение < 0.002, burstiness не более 2–3.
- WRR: отклонение ≈ 0 (почти идеальное из-за цикличности), burstiness = 5 (для сервера s0).

4.4. Визуализация последовательности

```
python
import matplotlib.pyplot as plt
import numpy as np

def plot_sequence(seq, title, stride=100):
    # Отображаем первые stride запросов
    ids = {'s0':0, 's1':1, 's2':2}
```

```
    y = [ids[s] for s in seq[:stride]]
    x = np.arange(stride)
    plt.figure(figsize=(12,3))
    plt.scatter(x, y, s=10, c=y, cmap='Set1', marker='s')
    plt.yticks([0,1,2], ['s0 (5)','s1 (3)','s2 (1)'])
    plt.xlabel('Номер запроса')
    plt.ylabel('Сервер')
    plt.title(title)
    plt.grid(axis='x', linestyle='--', alpha=0.5)
    plt.show()


plot_sequence(seq_swrr, 'Smooth Weighted Round Robin (первые 100 запросов)')
plot_sequence(seq_wrr, 'Weighted Round Robin (первые 100 запросов)')
```

График наглядно демонстрирует «гладкость» SWRR — серверы чередуются часто, тогда как классический WRR выдаёт длинные монотонные блоки.

---

5. Задания для индивидуального выполнения

Номер варианта определяется по формуле (номер_студента mod 15) + 1.

5.1. Базовый уровень (оценка «удовлетворительно»)

Реализовать оба алгоритма, провести эксперимент с указанными в таблице весами и числом запросов, вычислить отклонение и burstiness, построить график последовательности (первые 100 запросов для наглядности).

Вариант	Веса серверов	Число запросов	Примечание
1	[5, 3, 1]	10 000	Пример из описания
2	[5, 2, 2]	12 000	
3	[10, 5, 2]	15 000	
4	[4, 3, 2, 1]	10 000	4 сервера

Вариант	Веса серверов	Число запросов	Примечание
5	[8, 4, 2, 1]	20 000	
6	[3, 3, 3]	9 000	Равные веса
7	[7, 5, 3, 2]	14 000	
8	[6, 4, 4, 2]	16 000	
9	[9, 6, 3, 1]	18 000	
10	[2, 1, 1, 1]	8 000	
11	[12, 6, 3, 2, 1]	25 000	5 серверов
12	[10, 8, 6, 4, 2]	30 000	
13	[5, 5, 5, 5]	20 000	Все веса равны
14	[1, 1, 1]	3 000	
15	[15, 10, 5, 3, 2]	35 000	

Требования к отчёту: рабочий код двух алгоритмов, таблица с метриками, графики последовательности, вывод о преимуществах SWRR по гладкости.

5.2. Средний уровень (оценка «хорошо»)

В дополнение к базовому уровню:

- Исследовать зависимость коэффициента всплесков (burstiness) от распределения весов: для фиксированного числа серверов $N = 3$ и суммарного веса $W_\Sigma = 10$ построить график burstiness для разных комбинаций весов (например, [8,1,1], [5,3,2], [4,3,3] и т.д.).
- Реализовать возможность **динамического изменения весов** в процессе работы (метод update_weight(server_id, new_weight)) без прерывания последовательности. Продемонстрировать перераспределение запросов при изменении веса на лету.

- Провести замер времени выполнения next_server() для обоих алгоритмов при числе серверов $N \in \{10,50,100\}$ и 1 000 000 вызовах; построить сравнительный график.
- Визуализировать **скользящее среднее распределения** запросов по времени (например, окно 100 запросов), подтверждающее плавность SWRR.

Вариант	Базовые веса (из табл.)	Дополнительное исследование
1	По варианту 1	Построить скользящее распределение и график burst vs weights
2	По варианту 2	Динамическое изменение весов
3	По варианту 3	График burstiness от комбинаций весов
4	По варианту 4	Сравнение времени next_server() для N=10,50,100
5	По варианту 5	Скользящее распределение + динамика
6	По варианту 6	Время выполнения и burstiness
7	По варианту 7	Динамическое изменение весов
8	По варианту 8	График burstiness от комбинаций весов
9	По варианту 9	Сравнение времени next_server()
10	По варианту 10	Скользящее распределение
11	По варианту 11	Динамическое изменение весов
12	По варианту 12	График burstiness vs веса, время выполнения
13	По варианту 13	Скользящее распределение
14	По варианту 14	Динамическое изменение весов

Вариант	Базовые веса (из табл.)	Дополнительное исследование
15	По варианту 15	Все дополнительные пункты

Требования: выполнен базовый уровень, добавлены указанные исследования, в отчёте представлены дополнительные графики и выводы.

5.3. Повышенный уровень (оценка «отлично»)

Расширенное исследование:

- Реализовать алгоритм **Least-Weighted Connection** (динамический выбор сервера с наименьшим числом активных соединений, умноженным на вес) и сравнить его качество балансировки с SWRR при симуляции запросов с распределённой длительностью.
- Реализовать SWRR с **нецелыми весами** (вещественные веса), проверить равномерность на длинной серии и сравнить с классическим подходом.
- Построить **тепловую карту** последовательности выборов для 500 запросов при большом числе серверов ($N = 20$) и неравномерных весах, отразив гладкость чередования.
- Провести анализ чувствительности burstiness к округлению весов.

Вариант	Базовая конфигурация (веса, запросы)	Расширенное задание
1	[5,3,1], 10k	Least-Weighted Connection vs SWRR
2	[10,7,3,1], 20k	Нецелые веса [5.5, 2.3, 1.2]
3	[8,4,2,1], 15k	Тепловая карта для N=20
4	[12,6,3,2,1], 25k	Least-Weighted Connection
5	[7,5,3,2], 14k	Нецелые веса
6	[6,4,4,2], 16k	Тепловая карта
7	[9,6,3,1], 18k	Least-Weighted Connection

Вариант	Базовая конфигурация (веса, запросы)	Расширенное задание
8	[15,10,5,3,2], 35k	Нецелые веса и анализ burstiness
9	[2,1,1,1], 8k	Чувствительность к округлению
10	[5,5,5,5], 20k	Тепловая карта
11	[10,8,6,4,2], 30k	Least-Weighted Connection
12	[1,1,1], 3k	Нецелые веса
13	[12,6,3,2,1], 25k	Сравнение трёх алгоритмов
14	[4,3,2,1], 10k	Тепловая карта
15	[3,3,3], 9k	Все дополнительные пункты

Требования к отчёту: полное выполнение среднего уровня, наличие расширенных реализаций, глубокий анализ, развернутые выводы о применимости алгоритмов в реальных системах.

---

6. Контрольные вопросы

1. Каков главный недостаток классического Weighted Round Robin, решаемый алгоритмом Smooth Weighted Round Robin?
2. Объясните по шагам, как работает алгоритм SWRR с «текущим весом». Почему он обеспечивает гладкость?
3. Что произойдёт, если все веса равны? Будет ли разница между SWRR и обычным Round Robin?
4. Как измеряется коэффициент всплесков (burstiness) и почему он важен для балансировщика?
5. Каким образом можно модифицировать SWRR для поддержки динамического изменения весов без перезапуска?
6. Какие структуры данных можно использовать для оптимизации выбора максимального current_weight при большом количестве серверов?

7. В чём преимущество SWRR перед другими алгоритмами балансировки (например, Least Connections) в контексте HTTP-запросов?
8. Как оценить теоретический максимум burstiness для SWRR при заданных весах?
9. Можно ли применять SWRR для распределения запросов, если веса заданы нецелыми числами? Какие могут возникнуть сложности?
10. В каких известных программных продуктах используется алгоритм SWRR?

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: описание алгоритмов, метрик.
4. Программная реализация (листинги с комментариями).
5. Экспериментальная часть:
    - Параметры эксперимента.
    - Таблица с измеренными отклонением и burstiness для обоих алгоритмов.
    - Визуализация последовательности выбора серверов (точечные диаграммы).
    - Для среднего/повышенного уровней: дополнительные графики (зависимость burstiness от весов, скользящее распределение, тепловая карта и т.д.).
6. Анализ результатов и выводы.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Оба алгоритма реализованы корректно.
- Проведён эксперимент согласно варианту, вычислены метрики.
- Построены графики последовательности.
- В отчёте присутствуют основные разделы и выводы о «гладкости».

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.

- Проведены дополнительные исследования согласно варианту (графики зависимости, динамическое изменение, замеры времени).
- Выводы содержат сравнительный анализ метрик и рекомендации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы расширенные алгоритмы (Least-Weighted Connection, нецелые веса, тепловая карта).
- Проведён глубокий анализ, представлены убедительные доказательства преимуществ SWRR.
- Отчёт отлично структурирован, код сопровождается пояснениями.

Примечание. Для реализации на Python рекомендуется использовать стандартные библиотеки (collections, itertools), а для визуализации — matplotlib и numpy. Для замера времени полезна функция time.perf_counter.

Лабораторная работа № 4

Вероятностная балансировка: Power of Two Choices

---

1. Цель работы

Цель работы — изучить вероятностный алгоритм балансировки нагрузки **«Power of Two Choices»**, реализовать симулятор системы массового обслуживания и экспериментально подтвердить его теоретические свойства, сравнив с чисто случайной балансировкой. Исследовать, как выбор из двух случайных серверов вместо одного кардинально улучшает максимальную загрузку.

---

2. Задачи

1. Разработать программный симулятор, моделирующий распределение M запросов по N одинаковым узлам (серверам).
2. Реализовать три стратегии выбора узла:
   - **Random**: случайный выбор одного узла.
   - **Power of Two Choices (P2C)**: выбрать два случайных узла и направить запрос на менее загруженный.
   - **Power of *d* Choices**: обобщение на $d = 3,4$.
3. Провести серию экспериментов при различных N и M, измеряя максимальную нагрузку среди узлов max_load.
4. Построить графики зависимости max_load от числа узлов N для разных стратегий и числа запросов M (например, $M = N$, $M = 10N$, $M = 100N$).
5. Сравнить эмпирические результаты с известной теоретической асимптотикой:

$$\text{max\_load} \approx \frac{\log\log N}{\log d} + O(1).$$

6. Оценить коэффициент использования системы (загрузку) и распределение нагрузки.

---

3. Теоретическое обоснование

3.1. Модель «шары и корзины» (Balls and Bins) как основа балансировки

Задача распределения M запросов (шаров) по N серверам (корзинам) — классическая вероятностная модель балансировки. Если каждый запрос независимо направляется на

случайный сервер, биномиальное распределение приводит к тому, что максимальное число запросов на сервере асимптотически составляет

$$\Theta\left(\frac{\log N}{\log\log N}\right)$$

при $M = N$. Для больших M диспропорция возрастает, создавая значительный перекос.

3.2. Алгоритм Power of Two Choices (d = 2)

Идея, предложенная Михаэлем Митценмахером, заключается в минимальной модификации случайного выбора: для каждого запроса случайно (равномерно, независимо) выбираются **два** сервера-кандидата, и запрос направляется на тот из них, у которого **меньшая текущая нагрузка** (число уже назначенных запросов). При $M = N$ максимальная загрузка резко падает до

$$O(\log\log N),$$

что представляет собой экспоненциальное улучшение по сравнению с чистым Random.

3.3. Обобщение на d > 2

При выборе из d серверов (Power of *d* Choices) максимальная нагрузка уменьшается ещё сильнее и оценивается как

$$\text{max\_load} \approx \frac{\log\log N}{\log d} + O(1).$$

Практически уже при $d = 2$ достигается почти оптимальная равномерность; дальнейшее увеличение d даёт лишь логарифмический выигрыш и быстро теряет смысл из-за роста накладных расходов на опрос дополнительных серверов.

3.4. Метрики для эмпирического анализа

- **Максимальная нагрузка** (**max_load**) — максимальное количество запросов, назначенное одному серверу после обработки M обращений. Основной показатель перекоса.
- **Коэффициент вариации** нагрузки $c_v = \sigma/\mu$, где $\mu = M/N$ — средняя нагрузка, σ — среднеквадратическое отклонение нагрузки по серверам.
- **Загрузка** (utilization) — в гомогенной системе все серверы считаются одинаковыми, поэтому полезно оценивать равномерность через гистограмму и эмпирическую функцию распределения.

---

4. Пример практического выполнения

4.1. Класс-симулятор на Python

Создадим класс LoadBalancerSimulator, который принимает число серверов N и поддерживает различные стратегии диспетчеризации.

python

```
import random
import numpy as np
from collections import Counter
import matplotlib.pyplot as plt


class LoadBalancerSimulator:
    def __init__(self, N):
        self.N = N
        self.loads = np.zeros(N, dtype=int)  # нагрузки каждого сервера

    def reset(self):
        self.loads[:] = 0

    def dispatch_random(self, M):
        """Случайное равномерное распределение."""
        for _ in range(M):
            idx = random.randrange(self.N)
            self.loads[idx] += 1

    def dispatch_power_of_two(self, M):
        """Power of Two Choices (d=2)."""
        for _ in range(M):
            a = random.randrange(self.N)
            b = random.randrange(self.N)
            # гарантируем a != b (можно просто сравнивать, повторная выборка редка)
```

```
        while b == a:
            b = random.randrange(self.N)
        if self.loads[a] < self.loads[b]:
            idx = a
        else:
            idx = b
        self.loads[idx] += 1

    def dispatch_power_of_d(self, M, d):
        """Power of d Choices."""
        for _ in range(M):
            candidates = random.sample(range(self.N), d)
            idx = min(candidates, key=lambda i: self.loads[i])
            self.loads[idx] += 1

    def max_load(self):
        return self.loads.max()

    def stats(self):
        mean = self.loads.mean()
        std = self.loads.std()
        cv = std / mean if mean > 0 else float('inf')
        return mean, std, cv
```

4.2. Эксперимент: зависимость max_load от N

Зафиксируем $M = N$ (т.е. число запросов равно числу серверов). Для ряда значений N (например, от 100 до 10 000) запустим каждую стратегию несколько раз и усредним max_load. Построим график в полулогарифмическом масштабе.

python

```
def run_experiment(N_values, M_factor=1, reps=20):
    results = {'Random': [], 'P2C': [], 'P3C': [], 'P4C': []}
```

```
    for N in N_values:
        M = int(M_factor * N)
        rand_vals, p2c_vals, p3c_vals, p4c_vals = [], [], [], []
        for _ in range(reps):
            sim = LoadBalancerSimulator(N)
            sim.reset()
            sim.dispatch_random(M)
            rand_vals.append(sim.max_load())
            sim.reset()
            sim.dispatch_power_of_two(M)
            p2c_vals.append(sim.max_load())
            sim.reset()
            sim.dispatch_power_of_d(M, 3)
            p3c_vals.append(sim.max_load())
            sim.reset()
            sim.dispatch_power_of_d(M, 4)
            p4c_vals.append(sim.max_load())
        results['Random'].append(np.mean(rand_vals))
        results['P2C'].append(np.mean(p2c_vals))
        results['P3C'].append(np.mean(p3c_vals))
        results['P4C'].append(np.mean(p4c_vals))
    return results

N_vals = [100, 200, 500, 1000, 2000, 5000, 10000]
res = run_experiment(N_vals, M_factor=1)

plt.figure(figsize=(10,6))
for label, vals in res.items():
    plt.plot(N_vals, vals, 'o-', label=label)
```

```
plt.xscale('log')

plt.xlabel('Число серверов N (лог шкала)')

plt.ylabel('Максимальная нагрузка (M=N)')

plt.title('Зависимость max_load от N при различных стратегиях')

plt.legend()

plt.grid(True, which='both', linestyle='--')

plt.show()
```

Для сравнения с теорией можно добавить расчётную линию $\log\log N/\log d$ (со смещением).

4.3. Анализ распределения нагрузки

Построим гистограмму нагрузок для конкретного эксперимента.

```
python

sim = LoadBalancerSimulator(N=1000)

sim.dispatch_power_of_two(M=10000)  # M = 10*N

plt.hist(sim.loads, bins=30, edgecolor='black', alpha=0.7)

plt.xlabel('Нагрузка')

plt.ylabel('Количество серверов')

plt.title('Распределение нагрузки при Power of Two Choices (N=1000, M=10000)')

plt.grid(axis='y')

plt.show()
```

Визуализация должна показать узкое распределение вокруг среднего $\mu = 10$, с резким ограничением сверху, в отличие от Random, где хвост значительно длиннее.

---

5. Задания для индивидуального выполнения

Номер варианта вычисляется по стандартной формуле:

$$\text{вариант} = (\text{номер в списке}\, mod\, 15) + 1$$

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать симулятор с поддержкой стратегий Random и Power of Two Choices. Провести эксперимент согласно таблице, построить график зависимости max_load от N для указанных стратегий, вычислить коэффициент вариации при финальном распределении.

Вариант	Диапазон N	M (запросов)	Число повторений	Особое примечание
1	100, 200, 500, 1000, 2000	M = N	30	
2	100, 200, 500, 1000, 2000, 5000	M = 2N	20	
3	50, 100, 200, 500, 1000	M = N	50	
4	200, 500, 1000, 2000, 5000	M = 5N	15	
5	100, 300, 1000, 3000, 10000	M = N	25	
6	500, 1000, 2000, 5000	M = 10N	20	
7	100, 200, 500, 1000, 2000, 5000, 10000	M = N	20	
8	100, 500, 1000, 5000	M = N	10	Упрощённое, для времени
9	200, 400, 800, 1600	M = 3N	15	
10	50, 100, 200, 400	M = 100N	30	Сильная нагрузка
11	500, 1000, 2000, 5000	M = N	20	

Вариант	Диапазон N	M (запросов)	Число повторений	Особое примечание
12	1000, 2000, 5000, 10000	M = 2N	15	
13	200, 500, 1000, 2000, 5000	M = 0.5N (N четно)	20	Обработать M<N
14	300, 600, 1200, 2400	M = N	25	
15	100, 250, 500, 1000, 2500	M = N	20	

Требования к отчёту: корректный код двух стратегий; график max_load(N) в полулогарифмическом масштабе по N; вычислен коэффициент вариации для одной из конфигураций; сравнение с Random.

5.2. Средний уровень (на оценку «хорошо»)

Помимо заданий базового уровня необходимо:

- Реализовать стратегию Power of *d* Choices для $d = 3$ и $d = 4$. Построить зависимость max_load от числа d (при фиксированном $N = 1000$, $M = N$).
- Сравнить полученные экспериментальные значения с теоретической границей $\frac{\log\log N}{\log d}$, наложив теоретическую кривую на график (с подбором константы $O(1)$).
- Измерить время выполнения симуляций для каждого алгоритма при $N = 10000, M = 100000$ и сравнить накладные расходы.
- Построить гистограмму распределения нагрузки (не только максимум) для Random и P2C при $M = 10N$ и $N = 1000$, обсудить форму.

Вариант	Параметры базового уровня	Дополнительные задания
1	вариант 1	d=3,4, сравнение с теорией, гистограмма

Вариант	Параметры базового уровня	Дополнительные задания
2	вариант 2	d=3,4, временные замеры, теоретическая кривая
3	вариант 3	d=3, теория, гистограмма
4	вариант 4	временные замеры, сравнение d=2 и d=3
5	вариант 5	d=3,4, теория, гистограмма
6	вариант 6	временные замеры, теоретическая линия
7	вариант 7	d=3,4, гистограмма при M=10N
8	вариант 8	d=3,4, сравнение с теорией
9	вариант 9	d=3,4, временные замеры, гистограмма
10	вариант 10	теоретическая кривая, анализ при очень большом M
11	вариант 11	d=3,4, временные замеры, теория
12	вариант 12	d=3,4, гистограмма
13	вариант 13	теоретическая линия, анализ для M < N
14	вариант 14	d=3,4, временные замеры
15	вариант 15	Все дополнительные пункты

Требования: выполнены все пункты базового уровня; представлены дополнительные графики и анализ; отчёт содержит обоснованные выводы о влиянии d и сравнение с теорией.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать асинхронную модель, в которой запросы не только добавляются, но и **завершаются** (удаляются из сервера) через случайное время (распределение Пуассона). Сравнить стратегии в динамике.
- Исследовать **гетерогенные серверы** с разной скоростью обработки: пусть сервер i имеет вес w_i (обрабатывает w_i запросов в единицу времени). Адаптировать алгоритм Power of Two Choices для учёта весов при сравнении загрузки (например, сравнивать load_i / w_i). Провести анализ равномерности.
- Построить **трёхмерный график** зависимости max_load от N и M для стратегии P2C.
- Реализовать вариант с **частичной информацией**: выбор не из двух, а из двух, но с вероятностью p выбирается один случайный сервер (моделирование ошибок). Исследовать деградацию max_load.
- Сравнить с теоретической моделью эмпирические вероятности больших нагрузок (процент серверов с нагрузкой выше порога).

Вариант	Базовые параметры	Расширенное задание
1	N=1000, M=N, d=2	Динамические запросы с завершениями (Пуассон)
2	N=1000, M=10N	Гетерогенные серверы с весами [1,2,3,..]
3	N=500, M=5N	3D-график зависимости max_load от N и M
4	N=2000, M=2N	Частичная информация (p=0.8)
5	N=5000, M=N	Теоретический анализ хвостов распределения
6	N=1000, M=20N	Динамические запросы + гетерогенные серверы
7	N=2000, M=5N	3D-график и сравнение со случайной стратегией
8	N=10000, M=N	Частичная информация (p=0.5 и p=0.9)

Вариант	Базовые параметры	Расширенное задание
9	N=500, M=2N	Гетерогенные серверы с случайными весами
10	N=1500, M=3N	Имитационное моделирование с очередями
11	N=1000, M=N	Сравнение d=2 с d=бесконечность (Join-Idle-Queue)
12	N=3000, M=10N	Теоретическая граница max_load и эмпирические данные
13	N=800, M=8N	Частичная информация и влияние на burstiness
14	N=2000, M=20N	Взвешенные серверы с Power of Two Choices
15	N=500, M=50N	Объединение нескольких расширенных заданий (по выбору)

Требования к отчёту: полный комплект среднего уровня, реализованы расширенные модели, проведён сравнительный анализ с теоретическими предсказаниями, выводы содержат практические рекомендации.

---

6. Контрольные вопросы

1. Почему максимум нагрузки в чисто случайной стратегии растёт как $\log N / \log \log N$, а в Power of Two Choices как $\log \log N$?
2. Как зависит максимальная нагрузка от параметра d? В чём причина логарифмического масштаба улучшения?
3. Объясните, каким образом даже выбор из двух серверов (а не из всех) даёт столь существенное снижение max load.
4. В чём ограничения алгоритма Power of Two Choices? Когда он менее эффективен?
5. Как изменится поведение алгоритма, если опрос серверов (получение загрузки) имеет задержку или вероятность ошибки?
6. Как обобщить метод на случай гетерогенных серверов с разной производительностью?

7. Какие структуры данных или протоколы позволяют реализовать Power of Two Choices в реальных распределённых системах?
8. Почему в большинстве практических систем (например, HAProxy, Envoy) не используют чисто случайный выбор, а применяют Least Connections или его аппроксимации?
9. Что такое «эффект очереди» (queueing effect) и как он влияет на сравнение стратегий в динамике?
10. Можно ли утверждать, что Power of Two Choices всегда лучше, чем Round Robin? Обоснуйте.

---

7. Требования к отчёту

Отчёт оформляется в соответствии с кафедральными требованиями и должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: постановка задачи, описание стратегий, теоретические асимптотики.
4. Описание программной реализации (основные классы и методы, пояснения).
5. Экспериментальная часть:
    - Таблицы и графики max_load(N) для разных стратегий.
    - Гистограммы нагрузки, коэффициенты вариации.
    - (для среднего/повышенного) дополнительные визуализации, сравнение с теоретическими кривыми, анализ времени.
6. Анализ результатов и выводы.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Симулятор реализует Random и Power of Two Choices.
- Проведён эксперимент по варианту, построен график max_load(N), вычислен коэффициент вариации.
- Отчёт содержит корректные результаты и базовые выводы.

8.2. Оценка «хорошо» (средний уровень)

- Все пункты базового уровня выполнены полностью.
- Реализован Power of *d* Choices для $d = 3,4$, построены графики зависимости от d с теоретической кривой.
- Проведены замеры времени и гистограммы распределения.
- Отчёт демонстрирует понимание теории и умение интерпретировать графики.

8.3. Оценка «отлично» (повышенный уровень)

- Выполнены все требования среднего уровня.
- Реализованы расширенные модели (динамическое удаление, гетерогенные серверы, частичная информация).
- Построены трёхмерные/дополнительные графики, исследовано влияние ошибок или весов.
- Выводы содержат глубокий анализ и обоснованные рекомендации.

Снижение оценки: неполный эксперимент, отсутствие сравнения с теорией, неверная реализация алгоритма, некачественные графики.

Рекомендация: при написании кода используйте библиотеки numpy для массивов нагрузки, random для выбора, matplotlib для графиков. Для ускорения экспериментов можно использовать векторизованные операции, однако для наглядности лучше сохранить пошаговый цикл. Для построения теоретической линии $\log\log N / \log d$ примените np.log(np.log(N)).

Лабораторная работа № 5

Алгоритм обнаружения отказов: Phi Accrual Detection

---

1. Цель работы

Цель работы — изучить вероятностный подход к обнаружению отказов в распределённых системах на основе алгоритма **Phi Accrual Failure Detection**, реализовать его программно, провести симуляцию различных профилей задержек и оценить чувствительность, время обнаружения и уровень ложных срабатываний.

---

2. Задачи

1. Реализовать мониторинг времени ответа (heartbeat latency) с хранением истории в скользящем окне фиксированной длины.
2. Оценивать среднее μ и стандартное отклонение σ задержек на основе хранящейся выборки.
3. Реализовать функцию phi(latency), вычисляющую уровень подозрительности текущей задержки как -log10(вероятность) при нормальном распределении.
4. Реализовать логику принятия решений: узел считается недоступным при phi > threshold.
5. Создать симулятор, генерирующий последовательности задержек с различными профилями: нормальный режим, экстремальные выбросы, ступенчатое увеличение задержки, полный обрыв связи.
6. Измерить время обнаружения отказа и количество ложных срабатываний для заданного порога.
7. Построить график изменения phi во времени, проанализировать результаты.

---

3. Теоретическое обоснование

3.1. Проблема обнаружения отказов

В распределённых системах (P2P-сети, кластеры, базы данных) участники обмениваются периодическими heartbeat-сообщениями. Задача мониторинга — своевременно выявить узел, который перестал отвечать, и при этом минимизировать ложные тревоги из-за спорадических сетевых задержек. Простейшее решение — фиксированный таймаут: если время ответа превысило порог, узел помечается отказавшим. Такой подход не адаптируется к изменению сетевых условий и часто даёт либо запоздалую реакцию, либо много ложных срабатываний.

3.2. Алгоритм Phi Accrual Detection

Алгоритм, предложенный Наохиро Хаяшибара и соавторами, поддерживает **степень подозрительности** (ϕ), непрерывно обновляемую на основе статистики последних задержек. Основная идея: вычислить вероятность того, что текущая задержка (время с момента последнего успешного heartbeat) соответствует нормальному поведению, и преобразовать её в логарифмическую шкалу. Если вероятность чрезвычайно мала, значение ϕ становится большим, и при превышении заданного порога генерируется событие отказа.

В учебной реализации используется скользящее окно последних W измеренных задержек {t_1, t_2, ..., t_W}. Предполагается, что задержки распределены по нормальному закону с параметрами, оценёнными по окну:

$$\mu = \frac{1}{W}\sum_{i=1}^{W} t_i \, , \sigma = \sqrt{\frac{1}{W-1}\sum_{i=1}^{W} (\, t_i - \mu)^2}$$

Для нового измерения t вычисляется стандартизованное значение:

$$z = \frac{t-\mu}{\sigma}$$

Вероятность получить задержку больше t (функция выживания) равна $P(X > t) = 1 - \Phi(z)$, где Φ — функция стандартного нормального распределения. Уровень подозрительности в шкале ϕ определяется как:

$$\phi(t) = -\log_{10}(1 - \Phi(z))$$

Если σ = 0 (все задержки одинаковы), то ϕ = 0 при t ≤ μ и +∞ при t > μ.

Решение об отказе принимается при ϕ ≥ threshold. Типичные значения порога — от 5 до 12, что соответствует вероятности ложного срабатывания от 10^{-5} до 10^{-12} на одно измерение.

3.3. Преимущества и особенности

- **Адаптивность**: с изменением сетевых условий параметры μ и σ обновляются, поэтому детектор сохраняет чувствительность без перенастройки.
- **Вероятностная интерпретация**: значение ϕ напрямую связано с вероятностью; порог может быть выбран, исходя из допустимой частоты ложных тревог.
- **Непрерывная величина**: приложения могут использовать ϕ для динамического принятия решений (например, исключить узел из пула, но не считать мёртвым окончательно).

3.4. Метрики качества детектора

- **Время обнаружения отказа** (Detection Time) — число измерений (или секунд) от момента реального отказа до превышения порога ϕ.
- **Частота ложных срабатываний** (False Positive Rate) — доля измерений при нормальной работе, в которых ϕ превышает порог.
- **Устойчивость к выбросам** — способность не поднимать ϕ слишком сильно при единичных скачках задержки.

---

4. Пример практического выполнения

4.1. Реализация Phi Accrual Detector с использованием скользящего окна

python

```
import math
from collections import deque
import numpy as np
import matplotlib.pyplot as plt


class PhiAccrualDetector:
    def __init__(self, window_size=1000, threshold=8.0):
        self.window = deque(maxlen=window_size)
        self.threshold = threshold
        self.mu = 0.0
        self.sigma = 0.0

    def add_latency(self, latency):
        self.window.append(latency)
        self.update_stats()

    def update_stats(self):
        if len(self.window) > 1:
            self.mu = np.mean(self.window)
            self.sigma = np.std(self.window, ddof=1)
```

```
        else:
            self.mu = self.window[0] if self.window else 0.0
            self.sigma = 0.0

    def phi(self, latency):
        if self.sigma == 0.0:
            return 0.0 if latency <= self.mu else float('inf')
        z = (latency - self.mu) / self.sigma
        # survival function: 1 - Φ(z)
        # Φ(z) = 0.5 * (1 + erf(z / sqrt(2)))
        cdf = 0.5 * (1.0 + math.erf(z / math.sqrt(2.0)))
        surv = 1.0 - cdf
        if surv <= 0:
            return float('inf')
        return -math.log10(surv)

    def is_suspicious(self, latency):
        return self.phi(latency) >= self.threshold
```

4.2. Профили задержек для симуляции

```
python
import random

def normal_profile(base=50, std=10):
    """Нормальная работа со стабильной задержкой."""
    return max(1, random.gauss(base, std))

def burst_profile(base=50, std=10, burst_every=50, burst_magnitude=300):
    """Нормальные задержки с периодическими всплесками."""
    step = burst_profile.counter = getattr(burst_profile, 'counter', 0) + 1
```

```
    if step % burst_every == 0:
        return max(1, random.gauss(base + burst_magnitude, std))
    return max(1, random.gauss(base, std))

def gradual_increase_profile(start=50, slope=0.5, step=0):
    """Плавное возрастание задержки (деградация канала)."""
    return max(1, random.gauss(start + slope * step, 5))

def failure_profile(after_step=200, base=50, std=10):
    """Нормальная работа, затем обрыв (очень большая задержка)."""
    step = failure_profile.counter = getattr(failure_profile, 'counter', 0) + 1
    if step > after_step:
        return 5000  # имитация обрыва
    return max(1, random.gauss(base, std))
```

4.3. Симуляция и визуализация

```
python
def run_simulation(detector, profile_func, steps=1000):
    phi_values = []
    suspicions = []
    latencies = []
    for i in range(steps):
        lat = profile_func(i) if 'i' in profile_func.__code__.co_varnames else profile_func()
        detector.add_latency(lat)
        phi_val = detector.phi(lat)
        phi_values.append(phi_val)
        suspicions.append(detector.is_suspicious(lat))
        latencies.append(lat)
    return latencies, phi_values, suspicions
```

```
# Пример: нормальная работа с последующим отказом

detector = PhiAccrualDetector(window_size=500, threshold=8)

lats, phis, susp = run_simulation(detector, lambda i: failure_profile(base=50, std=10, after_step=600, counter_reset=True))


plt.figure(figsize=(12, 4))

plt.subplot(2,1,1)

plt.plot(lats, label='Задержка (мс)', color='blue', alpha=0.7)

plt.axvline(600, color='red', linestyle='--', label='Момент отказа')

plt.ylabel('мс')

plt.legend()

plt.grid()


plt.subplot(2,1,2)

plt.plot(phis, label='ф', color='purple')

plt.axhline(8, color='black', linestyle='--', label='Порог ф=8')

plt.xlabel('Шаг')

plt.ylabel('ф')

plt.legend()

plt.grid()

plt.tight_layout()

plt.show()
```

График покажет, что до отказа ф близок к нулю, а после резкого скачка задержки ф быстро пересекает порог, регистрируя отказ.

4.4. Измерение метрик

Вычисляем время до обнаружения (шаги от момента отказа до первого is_suspicious == True) и количество ложных срабатываний до отказа.

```
python

first_suspicious = None

false_positives = 0
```

```
for i, (lat, phi, susp) in enumerate(zip(lats, phis, susp)):
    if i < 600:
        if susp:
            false_positives += 1
    else:
        if susp and first_suspicious is None:
            first_suspicious = i - 600
print(f"Ложных срабатываний до отказа: {false_positives}")
print(f"Обнаружение отказа через {first_suspicious} шагов")
```

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать детектор Phi Accrual с заданными параметрами окна и порога. Провести симуляцию на одном из профилей задержек, построить график ф(t), измерить время обнаружения и число ложных срабатываний.

Вариант	Размер окна	Порог ф	Профиль задержки	Количество шагов
1	500	8	Нормальный + отказ после 800 шагов	1500
2	1000	8	Нормальный + отказ после 1000 шагов	1800
3	1000	10	Нормальный + отказ после 600 шагов	1200
4	500	6	Нормальный + периодические выбросы, отказ на 700	1400

Вариант	Размер окна	Порог ф	Профиль задержки	Количество шагов
5	2000	8	Нормальный + отказ после 1200 шагов	2000
6	1000	5	Плавное увеличение задержки (slope=0.2)	2000
7	500	12	Нормальный + отказ после 500 шагов	1000
8	1000	8	Выбросы каждые 30 шагов, затем отказ на 900	1600
9	800	7	Нормальный + отказ после 1000 шагов	1800
10	1500	9	Нормальный + отказ после 1500 шагов	2500
11	400	6	Нормальный + отказ после 600 шагов	1000
12	1000	11	Выбросы (каждые 50 шагов, магнитуда 200) + отказ на 1000	1800
13	1200	8	Плавное увеличение (slope=0.1)	3000
14	600	10	Нормальный + отказ после 800 шагов	1500
15	500	8	Периодические скачки (каждые 100 шагов) без отказа	2000

Требования к отчёту: работающий код детектора, график ф(t) с отметкой порога, подсчитанные метрики, выводы о влиянии параметров.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Реализовать наивный детектор с фиксированным таймаутом (например, среднее + 3σ) и сравнить время обнаружения и количество ложных тревог с Phi Accrual на том же профиле.
- Построить ROC-подобную кривую зависимости времени обнаружения от порога ϕ (варьируя порог от 3 до 15) и выбрать оптимальное значение.
- Исследовать влияние размера скользящего окна на скорость адаптации: для плавно меняющейся задержки построить график ϕ(t) при размерах окна 100, 500, 2000.
- Провести 20 повторных экспериментов с различными случайными реализациями нормального шума и вычислить среднее и дисперсию времени обнаружения.

Вариант	Параметры базового уровня	Дополнительные задания
1	вариант 1	Фиксированный таймаут сравнение, варьирование порога ϕ
2	вариант 2	Влияние размера окна, повторные эксперименты
3	вариант 3	ROC-анализ порога, сравнение с таймаутом
4	вариант 4	Повторные эксперименты, влияние окна
5	вариант 5	Сравнение с таймаутом, варьирование порога
6	вариант 6	Анализ окна для ступенчатого роста, повторные запуски
7	вариант 7	ROC-кривая, фиксированный таймаут
8	вариант 8	Повторные эксперименты, сравнение с таймаутом
9	вариант 9	Варьирование порога, влияние окна

Вариант	Параметры базового уровня	Дополнительные задания
10	вариант 10	Сравнение с наивным детектором, ROC
11	вариант 11	Повторные запуски, графики
12	вариант 12	Влияние окна, порог ROC
13	вариант 13	Сравнение методов, анализ адаптации
14	вариант 14	ROC, повторные эксперименты
15	вариант 15	Все перечисленные дополнительные исследования

Требования: выполнены все пункты базового уровня, представлены дополнительные графики и анализ, выводы аргументированы.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать оригинальный вариант Phi Accrual, где распределение задержек моделируется с помощью **экспоненциально взвешенного скользящего среднего** (EWMA) для μ и σ, а не простого окна. Сравнить с базовой версией по стабильности и скорости реакции.
- Реализовать детектор на основе непараметрического подхода: использовать **ядерную оценку плотности** (KDE) по окну или квантили вместо нормального распределения. Сравнить поведение на мультимодальном профиле (смесь двух нормальных распределений).
- Провести анализ влияния автокорреляции в задержках (моделирование задержек как AR(1)-процесса) на количество ложных тревог.
- Построить **тепловую карту** зависимости времени обнаружения от пары параметров (размер окна, порог ϕ) для фиксированного профиля отказа.
- Разработать симулятор с несколькими узлами и сетевым коммутатором, зашумляющим пакеты, и сравнить Phi Accrual с детектором на основе накопленной суммы (CUSUM).

Вариант	Базовая конфигурация	Расширенное задание
1	окно=1000, порог=8, отказ	EWMA против окна, сравнение
2	окно=500, порог=10	KDE вместо нормального распределения
3	окно=1000, порог=6	AR(1) задержки, анализ ложных тревог
4	окно=500, порог=8	Тепловая карта (окно vs порог)
5	окно=2000, порог=12	CUSUM детектор, сравнение в сети с помехами
6	окно=1000, порог=8	EWMA, тепловая карта
7	окно=800, порог=9	KDE на мультимодальной смеси
8	окно=1200, порог=7	Моделирование автокорреляции, CUSUM
9	окно=1500, порог=5	EWMA, KDE, сравнительный анализ
10	окно=600, порог=11	Тепловая карта и AR(1)
11	окно=1000, порог=8	Реализация нескольких узлов, сравнение с CUSUM
12	окно=500, порог=8	EWMA с адаптивным порогом
13	окно=1000, порог=9	Непараметрический детектор + мультимодальные данные
14	окно=700, порог=10	Тепловая карта, сравнение EWMA и KDE
15	окно=1000, порог=8	Все перечисленные исследования

Требования к отчёту: полный объём среднего уровня, реализованы расширенные модели, даны развёрнутые выводы о качестве детектирования и практических рекомендациях.

---

6. Контрольные вопросы

1. В чём принципиальное отличие Phi Accrual Detector от детектора с фиксированным таймаутом?
2. Почему в алгоритме используется функция выживания нормального распределения, а не простое сравнение с порогом?
3. Как размер скользящего окна влияет на скорость реакции и стабильность ф?
4. Что означает значение ф = 8 с точки зрения вероятности?
5. Какие предположения о распределении задержек лежат в основе учебной реализации? В каких случаях они нарушаются?
6. Как можно адаптировать алгоритм, если задержки имеют мультимодальное распределение?
7. Чем отличается экспоненциально взвешенное сглаживание (EWMA) от простого окна при оценке параметров?
8. Что произойдёт с ф, если σ очень мала, а задержка чуть выше среднего? Как это исправить?
9. Какие метрики используются для сравнения детекторов отказов? Как измерить компромисс между скоростью и ложными срабатываниями?
10. В каких реальных системах (Cassandra, Akka, Zookeeper) используется Phi Accrual Failure Detection?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание алгоритма, формулы, интерпретация ф.
4. Программная реализация: листинг ключевых классов и методов, пояснения.
5. Экспериментальная часть:
    - Описание профиля задержек.
    - График изменения ф во времени с отметкой порога и момента отказа.
    - Измеренные метрики: время обнаружения, количество ложных срабатываний.

- Для среднего и повышенного уровня — дополнительные графики и сравнительные таблицы.

6. Анализ результатов и выводы, обсуждение влияния параметров.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Детектор реализован с использованием скользящего окна и нормального распределения.
- Корректно вычисляется ф и принимается решение по порогу.
- Проведена симуляция согласно варианту, построен график ф(t), вычислены время обнаружения и ложные срабатывания.
- Отчёт содержит основные разделы и выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализован наивный детектор и проведено сравнение.
- Построены графики влияния порога и размера окна, проведены повторные эксперименты с оценкой разброса.
- Выводы содержат анализ чувствительности и обоснование выбора параметров.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы альтернативные методы оценки распределения (EWMA, KDE), детектор CUSUM или моделирование сети.
- Построены тепловые карты, исследованы автокорреляции и мультимодальные задержки.
- Всесторонний сравнительный анализ, практические рекомендации, исчерпывающий отчёт.

Снижение оценки: неработоспособный код, неверное вычисление вероятности, отсутствие требуемых графиков, поверхностные выводы.

Лабораторная работа № 6

Gossip-протокол для распространения состояния кластера

---

1. Цель работы

Цель работы – изучить принципы эпидемических (Gossip) протоколов для синхронизации состояния в распределённых системах, реализовать симулятор кластера с gossip-циклом и широковещательной рассылкой, провести сравнение времени конвергенции, накладных расходов и устойчивости к потерям сообщений.

---

2. Задачи

1. Разработать симулятор кластера из N узлов ($N = 10,25,50$), каждый из которых хранит вектор состояния.
2. Реализовать Gossip-протокол: узлы периодически обмениваются информацией со случайно выбранным партнёром.
3. Реализовать стратегию широковещательной рассылки (broadcast) для сравнения.
4. Измерить время конвергенции (число раундов / шагов), количество переданных сообщений на узел, долю охваченных узлов.
5. Ввести случайные потери пакетов (5%, 10%, 20%) и оценить их влияние на конвергенцию в обоих протоколах.
6. Построить графики зависимости времени конвергенции от N и уровня потерь, проанализировать результаты.

---

3. Теоретическое обоснование

3.1. Эпидемические протоколы

Gossip-протоколы инспирированы распространением слухов или инфекций в популяции. Каждый узел владеет некоторой версией данных (состоянием). Периодически узел выбирает случайного соседа и обменивается с ним информацией. Благодаря случайным парным взаимодействиям обновлённая информация быстро распространяется по всей системе – свойство, называемое **экспоненциальным ростом охвата** на ранних стадиях и гарантированной **логарифмической конвергенцией**:

$$T_{\text{conv}} = O(\log N),$$

где T_{conv} – количество раундов (циклов обмена), необходимое для достижения консенсуса. Каждый узел отправляет всего $O(\log N)$ сообщений, а общее число сообщений составляет $O(N \log N)$.

3.2. Модель push-pull и её упрощение

В классической push-модели узел, получивший новое состояние, «заражает» других. В учебной реализации каждый узел на каждом шаге (раунде) случайно выбирает партнёра, и они обмениваются векторами состояний (push-pull). Более свежее состояние (по версии/временной метке) принимается обоими узлами. Таким образом, самое актуальное состояние распространяется подобно инфекции.

3.3. Широковещательная рассылка (broadcast)

При broadcast узел-источник рассылает новое состояние всем другим узлам одновременно (или гарантированно доставляет каждому). Это требует $O(N)$ сообщений, но уязвимо к потерям: при потере хотя бы одного пакета соответствующий узел не обновится без повторной передачи. Gossip, напротив, избыточен: даже при потере части сообщений информация всё равно со временем дойдет через другие пути.

3.4. Метрики

- **Время конвергенции** – число раундов, через которое доля узлов, обладающих актуальным состоянием, достигает 100 % (или заданного порога, например 99 %).
- **Количество сообщений на узел до конвергенции** – мера накладных расходов.
- **Устойчивость к потерям** – изменение времени конвергенции при случайном отбрасывании сообщений (packet loss).

---

4. Пример практического выполнения

4.1. Модель узла и кластера

Создадим класс Node, хранящий вектор состояния в виде словаря: узел знает состояние всех узлов (или только самое свежее). В начальный момент у узла 0 стоит новая версия, остальные имеют старую. Задача – распространить новую версию на всех.

python

```
import random
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple


State = namedtuple('State', ['version', 'timestamp', 'load'])
```

```
class Node:
    def __init__(self, node_id, N):
        self.id = node_id
        # Изначально все считают себя старыми, узел 0 имеет новое состояние
        if node_id == 0:
            self.state = State(version=1, timestamp=0, load=0)
        else:
            self.state = State(version=0, timestamp=0, load=0)
        # Для gossip: храним только самое свежее известное состояние
        self.known_state = self.state

    def update(self, other_state):
        if other_state.version > self.known_state.version or \
           (other_state.version == self.known_state.version and other_state.timestamp > self.known_state.timestamp):
            self.known_state = other_state
```

4.2. Симуляция Gossip (раундовая модель)

На каждом раунде каждый узел случайно выбирает партнёра (не себя) и оба обновляются до максимума версий.

python

```
def gossip_round(nodes, loss_prob=0.0):
    """Один раунд gossip для всех узлов. loss_prob – вероятность потери сообщения."""
    N = len(nodes)
    # Перемешиваем порядок, чтобы избежать смещения
    indices = list(range(N))
    random.shuffle(indices)
    for i in indices:
        j = random.choice([x for x in range(N) if x != i])
        # Потеря пакета: с вероятностью loss_prob обмен не происходит
```

```
        if random.random() < loss_prob:
            continue
        # pull-push: каждый узнаёт состояние другого и обновляется
        state_i = nodes[i].known_state
        state_j = nodes[j].known_state
        nodes[i].update(state_j)
        nodes[j].update(state_i)
```

Для широковещательной рассылки: источник (узел 0) пытается доставить сообщение всем узлам. Учтём потери – каждое сообщение теряется с вероятностью loss_prob. После первой попытки узлы, не получившие сообщение, остаются необновлёнными (если не предусмотрены повторные передачи).

```
python
def broadcast(nodes, source_id=0, loss_prob=0.0):
    source_state = nodes[source_id].known_state
    for node in nodes:
        if node.id == source_id:
            continue
        if random.random() >= loss_prob:
            node.update(source_state)
```

4.3. Замеры конвергенции

Конвергенция считается достигнутой, когда все узлы имеют одинаковое состояние (максимальная версия). В симуляции будем выполнять раунды, пока все не обновятся, считая раунды и сообщения.

```
python
def run_gossip_until_convergence(N, loss_prob=0.0, max_rounds=1000):
    nodes = [Node(i, N) for i in range(N)]
    target_state = max((n.known_state for n in nodes), key=lambda s: (s.version, s.timestamp))
    rounds = 0
    messages = 0
    while True:
        gossip_round(nodes, loss_prob)
```

```
        rounds += 1
        messages += N  # каждый узел отправляет одно сообщение (можно считать два: push и pull, но упрощаем)
        # Проверяем конвергенцию
        if all(n.known_state == target_state for n in nodes):
            break
        if rounds >= max_rounds:
            rounds = max_rounds
            break
    return rounds, messages
```

Аналогично для broadcast: одна попытка; если не все получили, считаем, что конвергенция не достигнута (или можно измерять процент охвата).

4.4. Визуализация

Построим графики среднего числа раундов конвергенции в зависимости от N и уровня потерь.

```
python
Ns = [10, 25, 50]
loss_levels = [0.0, 0.05, 0.1, 0.2]
results = {}
for N in Ns:
    for loss in loss_levels:
        rounds_list = []
        for _ in range(20):  # усреднение
            r, _ = run_gossip_until_convergence(N, loss)
            rounds_list.append(r)
        results[(N, loss)] = np.mean(rounds_list)


# График
plt.figure(figsize=(10,6))
for loss in loss_levels:
```

```
    plt.plot(Ns, [results[(N, loss)] for N in Ns], 'o-', label=f'Loss {int(loss*100)}%')
plt.xlabel('Количество узлов N')
plt.ylabel('Среднее число раундов до конвергенции')
plt.title('Зависимость времени конвергенции Gossip от N и потерь')
plt.legend()
plt.grid(True)
plt.show()
```

Ожидаемый результат: число раундов растёт логарифмически с N; потери увеличивают время, но протокол остаётся работоспособным.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_по_списку mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать симуляцию Gossip-протокола для заданного количества узлов N, измерить время конвергенции (число раундов) и количество сообщений на узел. Сравнить с попыткой broadcast (одна попытка) при отсутствии потерь и с потерями 5%. Построить график доли обновлённых узлов по раундам для одного прогона.

Вариант	N (число узлов)	Потери (packet loss)	Примечание
1	10	0%, 5%	
2	15	0%, 10%	
3	20	0%, 5%	
4	25	0%, 5%	
5	30	0%, 10%	
6	40	0%, 5%	

Вариант	N (число узлов)	Потери (packet loss)	Примечание
7	50	0%, 5%	
8	10	0%, 20%	Сильные потери
9	15	0%, 20%	
10	25	0%, 10%	
11	10	0%, 5%	Дополнительно сравнить с broadcast
12	50	0%, 10%	
13	12	0%, 5%	
14	8	0%, 5%	
15	30	0%, 15%	

Требования к отчёту: работающий код, таблица с метриками, график распространения по раундам, сравнение с broadcast, базовые выводы.

5.2. Средний уровень (на оценку «хорошо»)

Помимо базового уровня:

- Реализовать два режима gossip: **push** (только отправка) и **push-pull** (обмен), сравнить их по скорости конвергенции.
- Провести эксперименты с изменением числа узлов $N \in \{10,25,50,100\}$ и усреднением по 50 запускам, построить график среднего числа раундов с доверительными интервалами.
- Исследовать влияние частоты обменов: если узел инициирует обмен не каждый раунд, а с вероятностью p, построить зависимость времени конвергенции от p.
- Провести сравнение с теоретической логарифмической кривой.

Вариант	Базовые параметры	Дополнительные задания
1	вариант 1	Push vs push-pull, вероятность инициации p=0.5,0.8,1.0
2	вариант 2	N=10,25,50,100 с доверительными интервалами
3	вариант 3	Push-pull сравнение, теоретическая кривая
4	вариант 4	Влияние вероятности инициации, график
5	вариант 5	Push/push-pull, усреднение по 50 запускам
6	вариант 6	N=10..100, доверительные интервалы, теоретическая граница
7	вариант 7	Push vs pull, частота обменов
8	вариант 8	Push-pull, устойчивость к высоким потерям, график N=50
9	вариант 9	Вероятность инициации, сравнение с broadcast
10	вариант 10	N=10,25,50,100, доверительные интервалы
11	вариант 11	Push-pull и broadcast, график времени
12	вариант 12	Частота инициации, потери 10%
13	вариант 13	Сравнение push/push-pull, анализ сообщений
14	вариант 14	N=8, 16, 32, 64, логарифмическая аппроксимация
15	вариант 15	Все дополнительные пункты

Требования: выполнены все пункты базового уровня, представлены расширенные графики, сделаны выводы о влиянии параметров.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать протокол с **anti-entropy** (полная синхронизация состояния между парами узлов, мерджинг версий) и сравнить с простым gossip обновлением одной версии.
- Моделировать **динамическое добавление и удаление узлов**: узел может входить в кластер или выходить из строя в процессе распространения; оценить влияние на конвергенцию.
- Ввести топологию сети (не полносвязную, а, например, случайный граф или кольцо) и сравнить с полносвязным gossip.
- Реализовать вероятностную гарантию доставки: после конвергенции запустить ещё несколько раундов, чтобы с вероятностью $1 - \delta$ гарантировать, что все узлы знают. Построить зависимость дополнительных раундов от N.
- Сравнить с алгоритмом **епідемічного дерева** (multicast gossip) по количеству сообщений.

Вариант	Базовые параметры	Расширенное задание
1	N=50, потери 5%	Anti-entropy, сравнение
2	N=100, потери 0%	Динамическое добавление/удаление узлов
3	N=50, потери 10%	Топология кольцо/случайный граф
4	N=25, потери 0%	Вероятностная гарантия доставки
5	N=100, потери 5%	Эпидемическое дерево и gossip
6	N=50, потери 0%	Anti-entropy, динамика узлов
7	N=30, потери 20%	Графовая топология, потери
8	N=50, потери 5%	Дополнительные раунды для гарантии

Вариант	Базовые параметры	Расширенное задание
9	N=80, потери 0%	Сравнение с multicast tree
10	N=50, потери 10%	Anti-entropy, топология
11	N=25, потери 0%	Динамическое членство и конвергенция
12	N=100, потери 5%	Вероятностная доставка, сравнение с broadcast
13	N=40, потери 10%	Влияние топологии на время
14	N=60, потери 0%	Anti-entropy, зашумлённая сеть
15	N=50, потери 0%	Комбинация нескольких расширений

Требования: полный объём среднего уровня, реализация дополнительных моделей, глубокий анализ, практические рекомендации.

---

6. Контрольные вопросы

1. Почему gossip-протокол гарантирует логарифмическое время конвергенции? Какова математическая модель распространения?
2. В чём отличие push, pull и push-pull моделей gossip? Какая эффективнее?
3. Как влияет размер кластера N на количество сообщений, необходимых для конвергенции?
4. Почему broadcast неработоспособен при значительных потерях пакетов, а gossip остаётся надёжным?
5. Что такое anti-entropy в gossip-протоколах? Когда её применяют?
6. Как можно адаптировать gossip-протокол для работы в сети с ограниченной связностью?
7. Какие реальные системы используют gossip (Consul, Riak, Cassandra)? Для каких задач?
8. Какие недостатки есть у gossip в крупномасштабных системах? Как ограничивают число сообщений?

9. Как измеряется вероятность гарантии доставки после конвергенции?
10. Можно ли использовать gossip для достижения консенсуса или разрешения конфликтов версий?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: принцип gossip, сравнение с broadcast, метрики.
4. Описание реализации: класс Node, функции раундов, методика замеров.
5. Экспериментальные результаты:
    - Таблицы времени конвергенции и числа сообщений для различных N и уровней потерь.
    - График роста доли обновлённых узлов по раундам.
    - Сравнительные графики для разных стратегий (push/pull, broadcast) и потерь.
    - (для среднего/повышенного) дополнительные визуализации и аналитические кривые.
6. Анализ и выводы.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректная реализация gossip-раунда и broadcast.
- Измерено время конвергенции для указанных N и потерь.
- Построен график распространения, таблица сравнения.
- Выводы присутствуют.

8.2. Оценка «хорошо» (средний уровень)

- Выполнен базовый уровень.
- Реализованы push/push-pull варианты, исследовано влияние вероятности инициации и N на 50 экспериментах.

- Графики с доверительными интервалами, сравнение с теоретической кривой.
- Отчёт аргументирован.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы anti-entropy, динамическое членство, топологии или вероятностные гарантии.
- Проведён сравнительный анализ с multicast или другими протоколами.
- Глубокие выводы, практические рекомендации.

Снижение оценки: неработоспособный код, отсутствие сравнения с broadcast, ошибки в подсчёте сообщений, неполный эксперимент.

Лабораторная работа № 7

Реализация LRU-кэша с O(1) сложностью

---

1. Цель работы

Цель работы – изучить алгоритм вытеснения **Least Recently Used (LRU)**, реализовать кэш-хранилище с гарантированной сложностью операций get и put за время $O(1)$ с использованием двусвязного списка и хеш-таблицы, провести нагрузочное тестирование и сравнить эффективность LRU с политикой FIFO при различных паттернах доступа к данным.

---

2. Задачи

1. Разработать структуру данных LRU-кэш, обеспечивающую:
    - вставку нового элемента,
    - чтение с обновлением позиции в списке,
    - удаление наименее используемого элемента при переполнении.
2. Реализовать альтернативный кэш с политикой вытеснения **First In, First Out (FIFO)**.
3. Реализовать генераторы синтетических паттернов доступа:
    - Random (равномерное),
    - Zipf (степенное распределение, параметр α),
    - Looping (циклический).
4. Провести серию экспериментов с 1 000 000 операций для различных размеров кэша, измеряя **hit ratio** (долю попаданий в кэш).
5. Сравнить LRU и FIFO по hit ratio, времени выполнения и использованию памяти.
6. Построить графики зависимости hit ratio от размера кэша и параметра α распределения Zipf.

---

3. Теоретическое обоснование

3.1. Кэширование и политики вытеснения

Кэш — хранилище ограниченной ёмкости, ускоряющее доступ к данным за счёт временного хранения часто запрашиваемых объектов. Когда кэш заполнен, при добавлении нового элемента необходимо удалить один из существующих. Правило выбора удаляемого элемента называется **политикой вытеснения** (eviction policy).

3.2. LRU (Least Recently Used)

Политика удаляет элемент, к которому дольше всего не было обращений. Она учитывает временну́ю локальность запросов: данные, использованные недавно, с высокой вероятностью понадобятся снова. Реализация «в лоб» требует $O(N)$ на операцию, однако комбинация **двусвязного списка** и **хеш-таблицы** позволяет добиться $O(1)$:

- Элементы хранятся в двусвязном списке в порядке от «свежих» к «старым» (голова — последний использованный, хвост — наименее используемый).
- Хеш-таблица отображает ключ на узел списка, обеспечивая прямой доступ.
- При get(key) узел перемещается в голову списка (удаление + вставка в начало $O(1)$).
- При put(key, value) новый узел вставляется в голову; если кэш переполнен, удаляется хвостовой узел.

3.3. FIFO (First In, First Out)

Удаляется элемент, который был добавлен раньше всех. Поведение реализуется с помощью обычной очереди (в Python — collections.deque). FIFO не учитывает частоту и давность использования, поэтому имеет худший hit ratio в условиях временно́й локальности, но прост в реализации.

3.4. Метрика hit ratio

$$\text{Hit Ratio} = \frac{\text{количество обращений, обслуженных из кэша}}{\text{общее число запросов}}.$$

Чем выше hit ratio, тем эффективнее кэш. При одинаковом размере кэша и одинаковом потоке запросов более «интеллектуальная» политика показывает лучшее значение.

3.5. Паттерны доступа

- **Random (равномерный)**: каждый запрос к случайному ключу из заданного пространства. Показывает эффективность при отсутствии локальности.
- **Zipf-распределение**: вероятность запроса ключа i пропорциональна $1/i^{\alpha}$. Характерно для веб-трафика (популярные объекты запрашиваются гораздо чаще). Параметр α управляет скошенностью: α=0.8 — умеренная, 1.2 — более выраженная, 1.5 — резкая.
- **Looping (циклический)**: последовательность запросов повторяется циклически по небольшому набору ключей, что создаёт идеальную временну́ю локальность.

---

4. Пример практического выполнения

4.1. Реализация LRU-кэша с O(1)

Используем dict в качестве хеш-таблицы и самостоятельно организуем двусвязный список с помощью классов Node.

python

```
class Node:
    __slots__ = ('key', 'value', 'prev', 'next')
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}             # ключ -> Node
        self.head = Node()          # фиктивная голова
        self.tail = Node()          # фиктивный хвост
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add_to_front(self, node):
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def get(self, key):
```

```
        node = self.cache.get(key)
        if node is None:
            return -1        # не найдено
        self._remove(node)
        self._add_to_front(node)
        return node.value

    def put(self, key, value):
        node = self.cache.get(key)
        if node:
            node.value = value
            self._remove(node)
            self._add_to_front(node)
        else:
            if len(self.cache) >= self.capacity:
                # удаляем из хвоста (наименее используемый)
                lru = self.tail.prev
                self._remove(lru)
                del self.cache[lru.key]
            new_node = Node(key, value)
            self.cache[key] = new_node
            self._add_to_front(new_node)
```

4.2. Реализация FIFO-кэша

```
python
from collections import deque

class FIFOCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
```

```
        self.cache = {}
        self.queue = deque()

    def get(self, key):
        return self.cache.get(key, -1)

    def put(self, key, value):
        if key in self.cache:
            self.cache[key] = value
        else:
            if len(self.cache) >= self.capacity:
                oldest = self.queue.popleft()
                del self.cache[oldest]
            self.cache[key] = value
            self.queue.append(key)
```

4.3. Генераторы запросов

```
python
import random
import numpy as np

def generate_random_requests(keyspace, length):
    return [random.randint(1, keyspace) for _ in range(length)]

def generate_zipf_requests(keyspace, length, alpha):
    # используем numpy для быстрых Zipf семплов (a=alpha)
    raw = np.random.zipf(alpha, length)
    # ограничиваем диапазон
    return [max(1, min(x, keyspace)) for x in raw]
```

```
def generate_looping_requests(keyspace, length):
    # циклический: 1,2,...,keyspace,1,2,...
    return [(i % keyspace) + 1 for i in range(length)]
```

4.4. Симуляция и замер hit ratio

```
python
def simulate(cache, requests):
    hits = 0
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)   # сохраняем ключ в качестве значения
    return hits / len(requests)

# Пример для LRU размера 500, Zipf alpha=1.2, 1 000 000 запросов
requests = generate_zipf_requests(10000, 1000000, 1.2)
cache = LRUCache(500)
hr = simulate(cache, requests)
print(f"Hit Ratio LRU: {hr:.4f}")
```

4.5. Визуализация результатов

Строим график зависимости hit ratio от размера кэша для разных паттернов.

```
python
import matplotlib.pyplot as plt

cache_sizes = [50, 100, 200, 500, 1000, 2000, 5000]
patterns = {
    "Random": generate_random_requests(100000, 100000),
    "Zipf (0.8)": generate_zipf_requests(100000, 100000, 0.8),
    "Zipf (1.2)": generate_zipf_requests(100000, 100000, 1.2),
```

```
    "Zipf (1.5)": generate_zipf_requests(100000, 100000, 1.5),
    "Looping": generate_looping_requests(5000, 100000)
}

plt.figure(figsize=(10,6))
for label, reqs in patterns.items():
    lru_hr = []
    for cap in cache_sizes:
        cache = LRUCache(cap)
        lru_hr.append(simulate(cache, reqs))
    plt.plot(cache_sizes, lru_hr, 'o-', label=f'{label} (LRU)')

plt.xscale('log')
plt.xlabel('Размер кэша (лог)')
plt.ylabel('Hit Ratio')
plt.title('Зависимость Hit Ratio от размера кэша (LRU)')
plt.legend()
plt.grid(True)
plt.show()
```

Аналогично для FIFO и наложение графиков сравнения.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать классы LRU и FIFO, провести симуляцию с заданными параметрами, вычислить hit ratio для указанных паттернов, построить график зависимости от размера кэша.

Вариант	Пространство ключей	Число запросов	Размеры кэшей (список)	Паттерны (α)
1	50 000	1 000 000	100, 250, 500, 1000, 2000	Random, Zipf(1.2)
2	100 000	800 000	50, 100, 200, 500, 1000	Random, Zipf(0.8), Zipf(1.5)
3	20 000	500 000	30, 60, 150, 300, 600	Zipf(1.2), Looping
4	30 000	1 200 000	100, 300, 600, 1200, 2500	Random, Zipf(1.5), Looping
5	10 000	300 000	50, 100, 200, 400	Zipf(0.8), Looping
6	80 000	900 000	80, 200, 500, 1000, 2000	Random, Zipf(1.0)
7	40 000	1 500 000	100, 250, 500, 1000, 2500	Zipf(0.8), Zipf(1.2)
8	60 000	700 000	50, 200, 500, 1000	Random, Zipf(1.5), Looping
9	15 000	250 000	100, 300, 500, 1000	Zipf(1.2), Random
10	25 000	400 000	75, 150, 300, 750	Zipf(1.0), Looping, Random
11	100 000	2 000 000	200, 500, 1000, 2000, 5000	Zipf(0.8), Zipf(1.5)

Вариант	Пространство ключей	Число запросов	Размеры кэшей (список)	Паттерны (α)
12	45 000	600 000	100, 250, 500, 1000	Random, Zipf(1.2), Looping
13	70 000	1 100 000	100, 300, 600, 1200, 2400	Zipf(1.2), Zipf(0.8)
14	5 000	100 000	20, 50, 100, 200, 500	Looping, Random
15	90 000	1 300 000	150, 400, 800, 1500, 3000	Random, Zipf(1.2), Zipf(1.5)

Требования к отчёту: корректная реализация LRU и FIFO, таблицы hit ratio, графики зависимости от размера кэша для LRU и FIFO на одном поле, выводы о преимуществах LRU.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Реализовать политику **Least Frequently Used (LFU)** (с использованием dict для частот) и сравнить с LRU и FIFO.
- Измерить **среднее время выполнения** get и put для всех трёх политик при 1 000 000 операций и размере кэша 1000, построить столбчатую диаграмму.
- Исследовать влияние **параметра α** Zipf на hit ratio: для выбранного размера кэша (500) построить график hit ratio (LRU vs FIFO) при α от 0.5 до 2.0 с шагом 0.1.
- Построить **трёхмерный график** (или heatmap) зависимости hit ratio LRU от размера кэша и α.

Вариант	Базовые параметры	Дополнительные задания
1	вариант 1	LFU, замер времени, график hit ratio от α
2	вариант 2	LFU, временные метрики, heatmap
3	вариант 3	Сравнение с LFU, влияние α

Вариант	Базовые параметры	Дополнительные задания
4	вариант 4	Замер времени, график α vs hit ratio
5	вариант 5	LFU, heatmap
6	вариант 6	LFU, время, зависимость от α
7	вариант 7	LFU, heatmap
8	вариант 8	Замер времени, влияние α
9	вариант 9	LFU, график α vs hit ratio
10	вариант 10	LFU, время, heatmap
11	вариант 11	LFU, временные метрики, α-анализ
12	вариант 12	LFU, heatmap
13	вариант 13	Замер времени, график α vs hit ratio
14	вариант 14	LFU, влияние α
15	вариант 15	Все дополнительные пункты

Требования к отчёту: выполнены базовые задачи, представлены дополнительные сравнительные метрики, графики аргументированы.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать кэш с политикой **Adaptive Replacement Cache (ARC)** или **CLOCK-Pro** (на выбор), провести сравнение с LRU.
- Исследовать **потребление памяти** реализациями: рассчитать накладные расходы на хранение служебных данных (размер Node, затраты на очереди), построить график зависимости памяти от размера кэша.

- Провести симуляцию с **реальным треком запросов** (например, из подмножества датасета запросов к веб-серверу) и сравнить hit ratio; построить график накопленного hit ratio во времени.
- Реализовать параллельную (thread-safe) версию LRU-кэша на основе OrderedDict и сравнить пропускную способность (операций в секунду) при многопоточном доступе.

Вариант	Базовые параметры	Расширенное задание
1	размер кэша 1000, 1M запр.	ARC, анализ памяти, реальный трек
2	размер кэша 500	CLOCK-Pro, параллельный LRU
3	размер кэша 2000	ARC, потребление памяти
4	размер кэша 800	Параллельный OrderedDict LRU, реальный трек
5	размер кэша 1500	ARC vs LRU vs LFU, треды
6	размер кэша 1200	CLOCK, анализ памяти
7	размер кэша 1000	ARC, многопоточность
8	размер кэша 600	Реальный трек, memory profiling
9	размер кэша 2500	ARC, сравнение с LFU
10	размер кэша 3000	CLOCK-Pro, параллельный LRU
11	размер кэша 1800	ARC, анализ памяти, трек
12	размер кэша 1400	Параллельный кэш, реальные данные
13	размер кэша 2000	ARC, многопоточность

Вариант	Базовые параметры	Расширенное задание
14	размер кэша 900	Сравнение всех политик, memory
15	размер кэша 1000	Комбинация нескольких расширений

Требования к отчёту: полный объём среднего уровня, реализованные расширенные модели, детальное сравнение, выводы о практической применимости.

---

6. Контрольные вопросы

1. Как двусвязный список и хеш-таблица обеспечивают O(1) для LRU? Опишите структуру и операции.
2. В чём отличие LRU от FIFO? При каком паттерне доступа разница в hit ratio максимальна?
3. Как параметр α распределения Zipf влияет на эффективность кэширования? Почему при очень больших α выгоднее маленький кэш?
4. Что такое временна́я локальность и как её эксплуатирует LRU?
5. Какие недостатки у LRU в условиях частотного, а не временно́го характера нагрузки? Какой алгоритм лучше справляется?
6. Как можно оптимизировать LRU для многопоточной среды? Какие блокировки нужны?
7. Объясните принцип работы политики LFU и её реализацию с эффективными структурами данных (куча, dict).
8. Что такое «cache pollution» и какой алгоритм наиболее подвержен этой проблеме?
9. Какие метрики, помимо hit ratio, важны при выборе политики кэширования в продакшене?
10. Почему во многих системах (Redis, Memcached) по умолчанию используется LRU или его аппроксимации?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.

3. Теоретическая часть: описание LRU, FIFO, структур данных, метрик, паттернов доступа.
4. Программная реализация: ключевые классы, объяснение методов.
5. Экспериментальная часть:
    - Таблицы hit ratio для разных параметров.
    - Графики зависимости hit ratio от размера кэша (сравнение LRU/FIFO по паттернам).
    - Для среднего/повышенного уровня – дополнительные графики (время, влияние α, замеры памяти, многопоточность).
6. Анализ результатов и выводы.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализованы LRU и FIFO с корректной логикой вытеснения.
- Проведена симуляция согласно варианту, вычислен hit ratio.
- Построены графики hit ratio от размера кэша, присутствует сравнение.
- Отчёт оформлен, выводы обоснованы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализован LFU или иная альтернативная политика.
- Проведён замер времени операций, построены графики влияния α Zipf, heatmap.
- Выводы аргументированы, содержат сравнительный анализ.

8.3. Оценка «отлично» (повышенный уровень)

- Выполнены все пункты среднего уровня.
- Реализованы ARC / CLOCK-Pro и/или многопоточный кэш.
- Проведены дополнительные исследования памяти или использования на реальных треках.
- Полный сравнительный анализ, практические рекомендации.

Снижение оценки: неэффективная реализация (не O(1)), ошибки в вытеснении, отсутствие графиков, неверные вычисления hit ratio, плохо структурированный код.

Лабораторная работа № 8

Реализация LFU-кэша с aging-механизмом

---

1. Цель работы

Цель работы — изучить алгоритм вытеснения **Least Frequently Used (LFU)**, выявить его недостаток — неспособность забывать старые популярные элементы, и реализовать механизмы **aging** (забывания) для адаптации к изменяющимся паттернам доступа. Провести сравнительный анализ классического LFU, LFU с периодическим затуханием счётчиков и Window LFU по метрикам hit ratio и скорости адаптации.

---

2. Задачи

1. Реализовать классический LFU-кэш с вытеснением наименее часто используемого элемента на основе структуры данных min-heap.
2. Реализовать два механизма aging:
    - **Периодическое затухание** (decay): раз в T операций (или секунд) все счётчики частоты делятся на коэффициент.
    - **Window LFU**: учитываются только обращения, попавшие в последнее временное окно фиксированной длины.
3. Реализовать симулятор, генерирующий последовательности запросов с «переключением» популярности ключей в середине трека.
4. Измерить hit ratio для трёх политик при различных параметрах (размер кэша, период затухания, размер окна).
5. Оценить скорость адаптации: число запросов, необходимое для вытеснения старых популярных ключей после смены паттерна.
6. Построить графики зависимости hit ratio от времени (номера запроса) в условиях переключения, демонстрирующие преимущества aging.

---

3. Теоретическое обоснование

3.1. Политика LFU и её недостатки

Кэш Least Frequently Used вытесняет элемент с наименьшим количеством обращений за всё время существования. Это идеально для стационарных распределений популярности, но обладает **инерцией**: однажды став популярным, элемент остаётся в кэше навсегда, даже если к нему перестали обращаться. Это приводит к засорению кэша устаревшими данными (cache pollution) при изменении рабочей нагрузки.

3.2. Механизмы aging

Aging позволяет «забывать» старую историю, уменьшая вес давних обращений. Два распространённых подхода:

- **Периодическое затухание (decay)**. С периодичностью T операций все счётчики частоты f_i масштабируются с коэффициентом $\lambda < 1$ (например, $f_i \leftarrow \lfloor f_i/2 \rfloor$). Это экспоненциально сглаживает историю.
- **Window LFU.** Хранится список обращений (или счётчики) только для запросов, поступивших в последнем временном окне W. При вытеснении учитывается частота в окне. Реализация может быть выполнена с помощью циклического буфера меток времени.

3.3. Реализация LFU через min-heap

Простейший способ — поддерживать кучу (min-heap), где ключом является частота. Однако при каждом обновлении частоты нужно уменьшать ключ (фактически удалять старую запись и вставлять новую), что даёт сложность $O(\log N)$. Эффективные реализации на практике (например, в Redis) используют массив двусвязных списков для каждой частоты, обеспечивая $O(1)$, но в учебных целях heap-решение наглядно и приемлемо.

3.4. Метрики

- **Hit Ratio** — доля попаданий в кэш из общего числа запросов.
- **Скорость адаптации** — число запросов или время от момента смены паттерна до момента, когда старый популярный ключ вытесняется из кэша (или доля нового контента в кэше достигает порога).
- **Сравнительный график** hit ratio на скользящем окне для визуализации динамики после переключения.

---

4. Пример практического выполнения

4.1. Классический LFU-кэш через min-heap

Используем heapq. Чтобы обновлять частоту, мы не модифицируем элемент в куче напрямую (трудно найти), а вставляем новую запись с увеличенной частотой, а старую помечаем как недействительную (lazy deletion). Для чистоты храним (freq, insert_order, key, value).

python

```
import heapq

import itertools
```

```
class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}                 # key -> [freq, value, valid]
        self.heap = []                  # min-heap элементов (freq, seq, key)
        self.order_counter = itertools.count()

    def _evict(self):
        while self.heap:
            freq, seq, key = heapq.heappop(self.heap)
            entry = self.cache.get(key)
            if entry and entry[2]:       # valid
                del self.cache[key]
                return

    def get(self, key):
        entry = self.cache.get(key)
        if not entry or not entry[2]:
            return -1
        freq, value, _ = entry
        # инвалидируем старую запись и добавим новую с увеличенной частотой
        entry[2] = False
        new_freq = freq + 1
        self.cache[key] = [new_freq, value, True]
        heapq.heappush(self.heap, (new_freq, next(self.order_counter), key))
        return value

    def put(self, key, value):
        if key in self.cache and self.cache[key][2]:
```

```
            # обновление существующего
            entry = self.cache[key]
            freq = entry[0]
            entry[2] = False
            new_freq = freq + 1
            self.cache[key] = [new_freq, value, True]
            heapq.heappush(self.heap, (new_freq, next(self.order_counter), key))
        else:
            if len([1 for e in self.cache.values() if e[2]]) >= self.capacity:
                self._evict()
            self.cache[key] = [1, value, True]
            heapq.heappush(self.heap, (1, next(self.order_counter), key))
```

4.2. LFU с периодическим затуханием (decay)

Добавляем счётчик операций и метод _decay, который вызывается каждые T операций. Все валидные счётчики умножаем на коэффициент (например, 0.5), а затем перестраиваем кучу.

python

```
class LFUDecayCache(LFUCache):
    def __init__(self, capacity, decay_interval=100, decay_factor=0.5):
        super().__init__(capacity)
        self.decay_interval = decay_interval
        self.decay_factor = decay_factor
        self.op_count = 0

    def _apply_decay(self):
        # инвалидируем все записи в куче и пересоздаём с новыми частотами
        new_heap = []
        for key, entry in self.cache.items():
            if entry[2]:
                old_freq = entry[0]
```

```
            new_freq = max(1, int(old_freq * self.decay_factor))
            entry[0] = new_freq
            heapq.heappush(new_heap, (new_freq, next(self.order_counter), key))
        self.heap = new_heap

    def _check_decay(self):
        self.op_count += 1
        if self.op_count % self.decay_interval == 0:
            self._apply_decay()

    def get(self, key):
        self._check_decay()
        return super().get(key)

    def put(self, key, value):
        self._check_decay()
        super().put(key, value)
```

4.3. Window LFU

Храним лог временных меток для каждого ключа, удаляем записи, выпавшие из окна.

python

```
from collections import deque
import time  # или шаги симуляции

class WindowLFUCache:
    def __init__(self, capacity, window_size):
        self.capacity = capacity
        self.window_size = window_size
        self.access_log = {}     # key -> deque of timestamps
        self.current_time = 0
```

```
def _clean_old(self, key):
    if key in self.access_log:
        dq = self.access_log[key]
        while dq and dq[0] <= self.current_time - self.window_size:
            dq.popleft()
        if not dq:
            del self.access_log[key]

def _freq(self, key):
    self._clean_old(key)
    return len(self.access_log.get(key, []))

def get(self, key):
    self.current_time += 1
    self._clean_old(key)
    if key in self.access_log:
        self.access_log[key].append(self.current_time)
        return key  # значение не храним для простоты
    return -1

def put(self, key, value):
    self.current_time += 1
    self._clean_old(key)
    if key not in self.access_log:
        if len(self.access_log) >= self.capacity:
            # вытесняем с наименьшей частотой в окне
            min_key = min(self.access_log.keys(), key=lambda k: len(self.access_log[k]))
            del self.access_log[min_key]
```

```
            self.access_log[key] = deque()
        self.access_log[key].append(self.current_time)
```

4.4. Сценарий с переключением паттерна

Генерируем последовательность: первые 10 000 запросов к набору «старых» популярных ключей, затем резкая смена на новый набор популярных ключей. Измеряем hit ratio на скользящем окне в 500 запросов и время до вытеснения старого ключа.

python

```
def generate_switch_requests(keyspace, switch_at, length):
    old_popular = list(range(1, 101))      # старые популярные
    new_popular = list(range(200, 301))    # новые популярные
    reqs = []
    for i in range(length):
        if i < switch_at:
            if random.random() < 0.8:
                reqs.append(random.choice(old_popular))
            else:
                reqs.append(random.randint(1, keyspace))
        else:
            if random.random() < 0.8:
                reqs.append(random.choice(new_popular))
            else:
                reqs.append(random.randint(1, keyspace))
    return reqs
```

Построим графики накопленного hit ratio с течением времени.

python

```
def sliding_hit_rate(requests, cache_ctor, window=500):
    cache = cache_ctor()
    hits_window = deque(maxlen=window)
    rates = []
    hits_total = 0
```

```
    for i, key in enumerate(requests):
        hit = 1 if cache.get(key) != -1 else 0
        if hit:
            hits_total += 1
        else:
            cache.put(key, key)
        hits_window.append(hit)
        if i >= window:
            rates.append(sum(hits_window)/window)
        else:
            rates.append(sum(hits_window)/(i+1))
    return rates

# Использование
reqs = generate_switch_requests(5000, 10000, 20000)
rates_lfu = sliding_hit_rate(reqs, lambda: LFUCache(300), window=500)
rates_lfu_decay = sliding_hit_rate(reqs, lambda: LFUDecayCache(300, decay_interval=1000), window=500)
rates_window = sliding_hit_rate(reqs, lambda: WindowLFUCache(300, window_size=500), window=500)

plt.plot(rates_lfu, label='LFU classic')
plt.plot(rates_lfu_decay, label='LFU Decay')
plt.plot(rates_window, label='Window LFU')
plt.axvline(x=10000, color='red', linestyle='--', label='Switch point')
plt.xlabel('Request number')
plt.ylabel('Hit ratio (window)')
plt.title('Адаптация LFU aging после смены паттерна')
plt.legend()
plt.grid(True)
```

plt.show()

График покажет, что после переключения hit ratio классического LFU падает и медленно восстанавливается, тогда как варианты с aging быстро адаптируются.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать LFU-кэш через min-heap и один из aging-механизмов (decay или Window LFU согласно варианту). Провести симуляцию со сценарием переключения популярности, измерить hit ratio на всём треке и за последние 20% запросов, оценить скорость адаптации (число запросов до вытеснения трёх старых топ-ключей).

Вариант	Размер кэша	Тип aging	Параметры aging	Длина трека	Switch at
1	200	Decay	T=500, λ=0.5	20 000	10 000
2	300	Decay	T=1000, λ=0.5	25 000	12 500
3	400	Decay	T=800, λ=0.7	30 000	15 000
4	250	Window LFU	W=1000	20 000	10 000
5	350	Window LFU	W=1500	25 000	12 000
6	150	Decay	T=300, λ=0.6	15 000	7 500
7	500	Decay	T=1200, λ=0.4	40 000	20 000
8	100	Window LFU	W=500	10 000	5 000
9	300	Decay	T=600, λ=0.5	20 000	10 000

Вариант	Размер кэша	Тип aging	Параметры aging	Длина трека	Switch at
10	200	Window LFU	W=800	15 000	7 500
11	450	Decay	T=900, λ=0.5	30 000	15 000
12	250	Decay	T=700, λ=0.55	18 000	9 000
13	350	Window LFU	W=1200	22 000	11 000
14	150	Decay	T=400, λ=0.5	12 000	6 000
15	500	Window LFU	W=2000	35 000	17 500

Требования: корректный код, таблицы hit ratio (до и после переключения, общий), график скользящего hit ratio в сравнении классического LFU и aging-версии, показатель адаптации.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Реализовать и сравнить все три политики (LFU, LFU Decay, Window LFU) в едином эксперименте.
- Исследовать влияние периода затухания T и коэффициента λ на hit ratio после переключения: построить тепловую карту (T vs λ) для фиксированного размера кэша 300.
- Реализовать LRU-кэш (из ЛР7) и сравнить его способность к адаптации с LFU-aging по тому же сценарию; объяснить различие.
- Измерить **накладные расходы** каждой политики по памяти и времени выполнения 1 000 000 операций.

Вариант	Базовые параметры	Дополнительные задания
1	вариант 1	Сравнение с LRU, анализ влияния T и λ

Вариант	Базовые параметры	Дополнительные задания
2	вариант 2	Тепловая карта (T, λ), замеры времени и памяти
3	вариант 3	Все три политики, LRU, анализ
4	вариант 4	Window LFU vs Decay, влияние размера окна W на адаптацию
5	вариант 5	LRU сравнение, тепловая карта decay-параметров
6	вариант 6	Сравнение с LRU, замеры производительности
7	вариант 7	Тепловая карта, анализ памяти
8	вариант 8	Три политики, оконная статистика
9	вариант 9	LRU, Decay, Window, сравнение
10	вариант 10	Влияние W, тепловая карта для Window (W vs keyspace)
11	вариант 11	Сравнение с LRU, замеры времени
12	вариант 12	Тепловая карта, несколько сценариев переключения
13	вариант 13	LRU, влияние окна, память
14	вариант 14	Три политики, анализ
15	вариант 15	Все перечисленные дополнительные исследования

Требования: расширенный сравнительный анализ, дополнительные графики, объяснение влияния параметров.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **динамический aging** — период затухания или размер окна адаптируется на основе наблюдаемого темпа изменения популярности (например, если hit ratio падает, aging ускоряется). Продемонстрировать автоматическую настройку.
- Реализовать алгоритм **TinyLFU** (или **W-TinyLFU**) с использованием Count-Min Sketch и политики частотного вытеснения, сравнить с LFU-aging по hit ratio и памяти.
- Провести симуляцию с **реальным треком запросов** (например, из датасета Wikipedia или Twitter) и проанализировать поведение политик при естественном дрейфе популярности.
- Построить трёхмерный график hit ratio от размера кэша и параметра aging (T или W) для окна после смены паттерна.
- Реализовать многопоточную версию LFU-кэша с периодическим затуханием и оценить пропускную способность.

Вариант	Базовые параметры	Расширенное задание
1	кэш 300, decay	Динамический aging, реальный трек
2	кэш 500	TinyLFU, сравнение с LFU-aging
3	кэш 400, window	3D-график (размер кэша × W), автоматическая настройка
4	кэш 250	Многопоточный LFU Decay, реальные данные
5	кэш 600	TinyLFU, динамическое затухание
6	кэш 350	Реальный трек, сравнение политик
7	кэш 1000	3D-анализ, Count-Min Sketch
8	кэш 200	Динамический window size, адаптация
9	кэш 450	TinyLFU vs LFU aging, многопоточность

Вариант	Базовые параметры	Расширенное задание
10	кэш 300	3D-график T vs λ vs hit ratio после переключения
11	кэш 700	Реальный трафик, автоматический aging
12	кэш 550	TinyLFU, сравнение, многопоточность
13	кэш 150	Динамический decay, анализ устойчивости
14	кэш 800	Все дополнительные на выбор
15	кэш 500	Комбинация нескольких расширений

Требования: полный отчёт, расширенные модели, обоснованные выводы о преимуществах адаптации.

---

6. Контрольные вопросы

1. В чём главный недостаток классического LFU? Почему он неэффективен при изменении популярности?
2. Как работают механизмы aging? Опишите разницу между периодическим затуханием и window LFU.
3. Какие структуры данных можно использовать для реализации LFU с быстрым обновлением частоты, отличные от min-heap?
4. Как параметр T (интервал затухания) влияет на компромисс между стабильностью и скоростью адаптации?
5. Что такое cache pollution? Как aging помогает её предотвратить?
6. Чем отличается Window LFU от классического LFU с затуханием? Какая из техник проще в реализации?
7. Как оценить скорость адаптации после смены паттерна? Какие метрики используют?
8. Почему LRU лучше адаптируется к смене паттерна, чем классическое LFU, но хуже, чем LFU с aging?
9. Какие алгоритмы используют аппроксимации частот (например, Count-Min Sketch) для кэширования?

10. В каких реальных системах (Redis, Caffeine, etc.) применяется LFU или его aging-версии? Какие параметры настройки доступны?

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание LFU, проблема отсутствия адаптации, методы aging, структуры данных.
4. Программная реализация: листинги основных классов, пояснения алгоритма затухания.
5. Экспериментальная часть:
    - Сценарий переключения, параметры.
    - Таблицы и графики hit ratio (скользящее среднее) для разных политик.
    - Метрики адаптации (число шагов до вытеснения старого топа).
    - Для среднего/повышенного уровня: тепловые карты влияния параметров, сравнение с LRU, замеры времени/памяти.
6. Анализ результатов и выводы.
7. Приложение с полным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован LFU-кэш (через heap или списки) и один aging-механизм.
- Проведён эксперимент с переключением, вычислен hit ratio и время адаптации.
- Построен график скользящего hit ratio, присутствует сравнение классического LFU и aging-версии.
- Отчёт содержит выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все пункты базового уровня.
- Реализованы обе aging-схемы и LRU для сравнения.
- Построена тепловая карта параметров, замеры производительности.

- Глубокий анализ, объяснение результатов.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован динамический aging и/или TinyLFU.
- Использованы реальные треки или дополнительные структуры (Count-Min Sketch).
- Выводы подкреплены количественными оценками, есть практические рекомендации.

Снижение оценки: неверная реализация вытеснения, отсутствие aging-механизма, некорректный замер адаптации, плохое оформление кода.

Лабораторная работа № 9

Реализация адаптивного алгоритма 2Q (Two Queues)

---

1. Цель работы

Цель работы — изучить адаптивный алгоритм кэширования **2Q (Two Queues)**, предназначенный для защиты кэша от загрязнения одноразовыми запросами (cache pollution). В ходе работы студент реализует структуру 2Q, проведёт эксперименты на трафике с различной долей «мусорных» обращений и выполнит сравнительный анализ с классическим LRU-кэшем.

---

2. Задачи

1. Реализовать структуру данных 2Q, состоящую из трёх очередей:
   - $A1_{\text{in}}$ — FIFO-очередь для входящих ключей,
   - $A1_{\text{out}}$ — теневая FIFO-очередь для вытесненных из $A1_{\text{in}}$ ключей (хранит только ключи, без данных),
   - A_m — LRU-очередь для «частых» ключей.
2. Реализовать правила обработки запросов:
   - первое обращение к ключу помещает его в $A1_{\text{in}}$;
   - при повторном обращении ключ продвигается в A_m;
   - при переполнении $A1_{\text{in}}$ вытесненный ключ перемещается в $A1_{\text{out}}$;
   - вытеснение из A_m и $A1_{\text{out}}$ производится по LRU (с удалением хвоста).
3. Реализовать классический LRU-кэш для сравнения.
4. Создать генератор трафика с управляемой долей одноразовых запросов и провести серию симуляций.
5. Измерить hit ratio в зависимости от доли одноразовых запросов для 2Q и LRU.
6. Оценить объём памяти, занимаемый теневой очередью $A1_{\text{out}}$, и отношение hit ratio к использованию памяти.
7. Построить графики зависимости hit ratio от доли сканирующего (одноразового) трафика.

---

3. Теоретическое обоснование

3.1. Проблема «одноразовых» запросов и загрязнения кэша

В реальных нагрузках (файловые системы, веб-кеширование, базы данных) часто встречаются потоки, в которых большое количество ключей запрашивается лишь однажды — так называемые «сканирующие» запросы или одноразовый трафик. Политика LRU, ориентированная на давность использования, оказывается уязвимой: единственный запрос помещает ключ в кэш, вытесняя потенциально полезный «частый» ключ. Это явление называется **cache pollution**. В результате hit ratio может резко снизиться.

3.2. Алгоритм 2Q

Алгоритм 2Q, предложенный Джонсоном и Шаша в 1994 году, решает проблему путём разделения кэша на три логических сегмента:

- **A1_in** — небольшая FIFO-очередь (входной буфер). Сюда попадают ключи при первом обращении. Ключ хранится в ней с данными.
- **A1_out** — теневая FIFO-очередь, в которой запоминаются ключи, вытесненные из A1_in, но без самих данных. Это «история» недавних одноразовых запросов, позволяющая распознать, что ключ уже встречался ранее.
- **Am** — основная LRU-очередь, содержащая часто используемые ключи (с данными). Попадание сюда требует, чтобы ключ был замечен как повторный.

Правила обработки запроса:

1. Если ключ найден в A_m или $A1_{\text{in}}$ — это попадание; в A_m он перемещается в голову LRU (обновляется давность).
2. Если ключ найден в $A1_{\text{out}}$ (теневая очередь) — это промах, но ключ уже «засветился» ранее, поэтому он удаляется из $A1_{\text{out}}$ и помещается в A_m как часто используемый (данные берутся из внешнего хранилища).
3. Если ключа нет ни в одной из очередей — это полный промах. Ключ добавляется в $A1_{\text{in}}$.

Правила вытеснения для поддержания размеров:

- Если $A1_{\text{in}}$ переполнен, **хвост** FIFO-очереди $A1_{\text{in}}$ удаляется. При этом, если $A1_{\text{out}}$ также переполнена, её хвост (LRU) удаляется. Вытесненный из $A1_{\text{in}}$ ключ помещается в $A1_{\text{out}}$.
- Если A_m переполнен, её хвост (LRU) удаляется без сохранения.

3.3. Параметры алгоритма

В классической реализации задаются относительные размеры: $A1_{\text{in}}$ — доля от общего кэша, $A1_{\text{out}}$ — обычно равен $A1_{\text{in}}$ или больше. В учебной версии будем задавать конкретные ёмкости cap_in, cap_out, cap_am. Общий размер кэша по данным равен cap_in + cap_am; теневая очередь A1_out не хранит значения, поэтому накладные расходы по памяти — только ключи.

3.4. Сравнение с LRU

При отсутствии одноразового трафика 2Q ведёт себя аналогично LRU, но с некоторым лагом на «раскрутку». При наличии значительной доли одноразовых ключей 2Q сохраняет высокий hit ratio, в то время как LRU деградирует. Количественная оценка: 2Q требует немного больше памяти из-за очереди $A1_{\text{out}}$, но окупает это лучшей фильтрацией.

---

4. Пример практического выполнения

4.1. Реализация 2Q на Python

Используем OrderedDict из модуля collections для реализации LRU-очереди (Am и A1_out) и обычный collections.deque для FIFO A1_in. Но для A1_out и Am нужно вытеснение LRU + перемещение в голову при доступе, поэтому для них удобен OrderedDict с семантикой: ключ — значение (данные для Am, флаг для A1_out). Для A1_in FIFO подойдёт OrderedDict с вставкой в конец и удалением из начала, так как нужно отслеживать наличие ключа и порядок. Мы используем OrderedDict, перемещая повторно обращённый ключ в конец? В FIFO порядок не меняется при попадании. Но нам нужно при попадании в A1_in оставить его в очереди на прежнем месте, а при повторном попадании удалить из A1_in и перевести в Am. Так что FIFO — просто OrderedDict, из которого удаляем элемент при необходимости.

python

```
from collections import OrderedDict


class TwoQCache:
    def __init__(self, cap_in: int, cap_out: int, cap_am: int):
        self.cap_in = cap_in
        self.cap_out = cap_out
        self.cap_am = cap_am
        # A1_in: FIFO, хранит key -> value
        self.a1_in = OrderedDict()
        # A1_out: теневая FIFO, хранит только ключи (значения None)
        self.a1_out = OrderedDict()
        # Am: LRU очередь, хранит key -> value
        self.am = OrderedDict()


    def get(self, key):
```

```
        # Проверяем Am (LRU)
        if key in self.am:
            self.am.move_to_end(key)
            return self.am[key]
        # Проверяем A1_in (FIFO)
        if key in self.a1_in:
            # попадание в A1_in: не перемещаем, возвращаем значение
            return self.a1_in[key]
        # Промах; возвращаем -1 (или None)
        return -1

    def put(self, key, value):
        # Если ключ уже в Am или A1_in, обновляем значение
        if key in self.am:
            self.am[key] = value
            self.am.move_to_end(key)
            return
        if key in self.a1_in:
            # Обновляем значение и оставляем в A1_in (не продвигаем, т.к. это первое попадание после добавления)
            self.a1_in[key] = value
            return

        # Если ключ в A1_out (теневая): это повторное обращение после вытеснения
        if key in self.a1_out:
            # Удаляем из A1_out, добавляем в Am как частый ключ
            del self.a1_out[key]
            self._add_to_am(key, value)
            return
```

```
        # Полный промах: добавляем в A1_in
        self._add_to_a1_in(key, value)

    def _add_to_a1_in(self, key, value):
        # Если A1_in полон, вытесняем хвост
        if len(self.a1_in) >= self.cap_in:
            # вытесняем самый старый элемент из A1_in
            evicted_key, _ = self.a1_in.popitem(last=False)
            # перемещаем его в A1_out
            self._add_to_a1_out(evicted_key)
        self.a1_in[key] = value

    def _add_to_a1_out(self, key):
        # Если A1_out полон, удаляем хвост (LRU среди теней)
        if len(self.a1_out) >= self.cap_out:
            self.a1_out.popitem(last=False)
        self.a1_out[key] = None  # значение не храним

    def _add_to_am(self, key, value):
        # Если Am полон, удаляем хвост LRU
        if len(self.am) >= self.cap_am:
            self.am.popitem(last=False)
        self.am[key] = value
```

Примечание: В данном коде предполагается, что cap_in, cap_out, cap_am — целые положительные числа. Реализация упрощённая: A1_in не меняет порядок при попадании (просто возвращаем значение), A1_out — просто FIFO с ограничением.

4.2. Генератор трафика с долей одноразовых запросов

Создадим функцию, генерирующую последовательность из length запросов. hot_keys — набор популярных ключей (повторяются много раз), one_timer_keys — одноразовые ключи,

которые появляются ровно один раз. Параметр one_timer_ratio определяет долю одноразовых запросов от общего числа.

python

```
import random


def generate_traffic(hot_size, one_timer_count, total_requests):
    hot_keys = list(range(1, hot_size + 1))
    one_timer_keys = list(range(hot_size + 1, hot_size + 1 + one_timer_count))
    requests = []
    one_timer_requests = int(total_requests * (one_timer_count / (hot_size * 10 + one_timer_count))) # примерно
    # Альтернатива: задаём фиксированное число one_timer запросов, остальные — hot.
    # Упростим: сгенерируем последовательность с заданной вероятностью.
    # Но лучше явно: first pass всех one_timer, затем много запросов hot с вкраплениями.
    pass
```

Упростим: генерируем всю последовательность. Пусть scan_ratio — доля запросов к уникальным ключам, которые больше нигде не повторятся. Остальные 1 - scan_ratio — запросы к множеству популярных ключей, распределённых по Zipf или равномерно. Чтобы управлять, создадим список из M уникальных ключей для скана и добавим их в начало трека или равномерно перемешаем.

Функция:

python

```
def generate_scan_traffic(popular_keys, popular_count, scan_ratio, length):
    """popular_keys: список популярных ключей (могут повторяться)
       popular_count: сколько раз примерно каждый популярный ключ встретится
       scan_ratio: доля одноразовых запросов (уникальных ключей)
    """
    scan_count = int(length * scan_ratio)
    popular_count_total = length - scan_count
    # Генерируем одноразовые ключи
    scan_requests = [f"scan_{i}" for i in range(scan_count)]
```

```
    # Генерируем популярные запросы
    popular_requests = [random.choice(popular_keys) for _ in range(popular_count_total)]
    all_reqs = scan_requests + popular_requests
    random.shuffle(all_reqs)
    return all_reqs
```

4.3. Симуляция и замер hit ratio

Функция симуляции аналогична ЛР7.

```
python
def simulate_2q(cache, requests):
    hits = 0
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)  # ключ совпадает со значением
    return hits / len(requests) if requests else 0
```

Проведём эксперимент: фиксированный размер общего кэша данных total_data_cap = 500, для 2Q поделим: cap_in = 100, cap_am = 400, cap_out = 100. Для LRU capacity = 500. Популярных ключей 1000, они повторяются. Доля сканирующих запросов варьируется от 0 до 0.9. Построим график hit ratio.

```
python
import matplotlib.pyplot as plt


scan_ratios = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
total_requests = 100000
popular = [f"hot_{i}" for i in range(1000)]
lru_hits = []
twoq_hits = []


for sr in scan_ratios:
```

```
    reqs = generate_scan_traffic(popular, 1, sr, total_requests)
    lru = LRUCache(500)  # из ЛР7
    lru_hits.append(simulate_2q(lru, reqs))
    twoq = TwoQCache(cap_in=100, cap_out=100, cap_am=400)
    twoq_hits.append(simulate_2q(twoq, reqs))

plt.plot(scan_ratios, lru_hits, 'o-', label='LRU (500)')
plt.plot(scan_ratios, twoq_hits, 's--', label='2Q (A1in=100, Am=400)')
plt.xlabel('Доля одноразовых запросов')
plt.ylabel('Hit Ratio')
plt.title('Устойчивость 2Q к cache pollution')
plt.legend()
plt.grid()
plt.show()
```

Ожидаем, что LRU резко просядет, а 2Q почти не изменится.

4.4. Оценка памяти теневой очереди

Можно подсчитать максимальную длину A1_out и количество байт, занимаемых ключами. Для простоты выводим:

```
python
mem_out = len(twoq.a1_out) * 8  # примерно 8 байт на ссылку? условно
print(f"Память A1_out: {mem_out} условных байт")
```

---

5. Задания для индивидуального выполнения

Номер варианта вычисляется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать кэш 2Q с заданными размерами очередей и сравнить его hit ratio с LRU для трафика с фиксированной долей одноразовых запросов. Построить график зависимости hit ratio от доли сканирующего трафика.

Вариант	Общий кэш данных (am+in)	A1in	A1out	Доли сканирования (для графика)	Число запросов
1	400	80	80	0, 0.1, 0.2, ..., 0.8	100 000
2	500	100	100	0, 0.15, 0.3, 0.45, 0.6, 0.75	150 000
3	600	120	120	0, 0.1, 0.25, 0.4, 0.55, 0.7	120 000
4	300	60	60	0, 0.2, 0.4, 0.6, 0.8	80 000
5	700	150	150	0, 0.05, 0.1, ..., 0.5	200 000
6	450	90	90	0, 0.1, 0.3, 0.5, 0.7	100 000
7	550	110	110	0, 0.2, 0.4, 0.6	130 000
8	350	70	70	0, 0.1, 0.3, 0.5, 0.7, 0.9	90 000
9	800	160	160	0, 0.2, 0.4, 0.6, 0.8	180 000
10	200	40	40	0, 0.1, 0.2, 0.3, 0.4	50 000
11	1000	200	200	0, 0.25, 0.5, 0.75	250 000
12	480	100	80	0, 0.1, 0.2, ..., 0.8	140 000
13	520	100	150	0, 0.15, 0.3, 0.45, 0.6	160 000
14	380	80	100	0, 0.2, 0.4, 0.6, 0.8	100 000
15	650	130	130	0, 0.1, 0.3, 0.5, 0.7, 0.9	170 000

Требования: работающий код 2Q и LRU, график hit ratio, оценка влияния доли сканирования, выводы о преимуществах 2Q.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Исследовать влияние соотношения размеров A1in : Am на hit ratio при фиксированной общей ёмкости данных. Для total_data = 500 построить график зависимости hit ratio от доли A1in / total_data при двух значениях доли одноразовых запросов (0.3 и 0.6).
- Измерить и сравнить накладные расходы памяти (в байтах на ключ) между LRU и 2Q, учитывая структуру данных и теневую очередь.
- Провести симуляцию с «пульсирующим» трафиком: сначала поток с высокой долей сканирования, затем без сканирования, затем снова сканирование. Показать динамику hit ratio на скользящем окне для LRU и 2Q.
- Реализовать вариацию 2Q, в которой очередь A1_out работает не по FIFO, а по LRU, и сравнить hit ratio (защита от пинг-понга).

Вариант	Параметры базового	Дополнительные задания
1	вариант 1	Влияние соотношения A1in/Am, память
2	вариант 2	Пульсирующий трафик, анализ памяти
3	вариант 3	A1out LRU vs FIFO, соотношение
4	вариант 4	Пульсирующая нагрузка, память
5	вариант 5	Соотношение, A1out LRU
6	вариант 6	Пульсирующий трафик, анализ памяти
7	вариант 7	Влияние размера A1in, графики
8	вариант 8	Сравнение FIFO/LRU в A1out, пульсация
9	вариант 9	Соотношение, память

Вариант	Параметры базового	Дополнительные задания
10	вариант 10	Пульсирующий трафик, A1out LRU
11	вариант 11	Память, влияние соотношения
12	вариант 12	Пульсация, A1out
13	вариант 13	Соотношение, пульсирующая нагрузка
14	вариант 14	A1out LRU, память
15	вариант 15	Все дополнительные пункты

Требования: расширенный отчёт с дополнительными графиками, анализом памяти и динамическим hit ratio.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **адаптивный 2Q**, в котором размеры очередей динамически подстраиваются под характер нагрузки (например, на основе наблюдения за частотой попаданий в A1_in и A1_out). Сравнить с фиксированными размерами.
- Реализовать алгоритм **ARC (Adaptive Replacement Cache)** и сравнить его hit ratio и затраты памяти с 2Q.
- Использовать **реальный трек** запросов (например, из трассировок файлового сервера) и провести сравнение LRU, 2Q, ARC.
- Провести детальный анализ влияния размера A1_out на способность фильтрации: построить 3D-график зависимости hit ratio от доли одноразового трафика и размера A1_out.
- Реализовать потокобезопасную версию 2Q и оценить накладные расходы на синхронизацию.

Вариант	Базовые параметры	Расширенное задание
1	data=500, A1in=100	Адаптивный 2Q, ARC

Вариант	Базовые параметры	Расширенное задание
2	data=600	Реальный трек, ARC
3	data=400	3D-график A1_out, адаптивный
4	data=700	ARC, потокобезопасный
5	data=300	Реальный трек, адаптивный
6	data=800	ARC, 3D-график
7	data=1000	Адаптивный, потокобезопасный
8	data=450	ARC, реальный трек
9	data=550	3D-график, адаптивный
10	data=350	Потокобезопасный, ARC
11	data=900	Адаптивный, реальные данные
12	data=500	ARC, 3D-график
13	data=750	Потокобезопасный, адаптивный
14	data=200	ARC, анализ
15	data=1000	Комбинация расширенных задач

Требования: глубокая проработка, сравнение с интеллектуальными алгоритмами, работа с реальными данными или многопоточностью.

---

6. Контрольные вопросы

1. Какую проблему классического LRU решает алгоритм 2Q? Что такое cache pollution?

2. Объясните назначение каждой из трёх очередей в 2Q.
3. Почему теневая очередь (A1_out) не хранит данные? Каковы накладные расходы?
4. Какой порядок вытеснения используется в A1_in, A1_out и Am, и чем обоснован такой выбор?
5. Что произойдёт с hit ratio, если размер A1_in сделать равным общему размеру кэша (а Am и A1_out = 0)? Будет ли это эквивалентно FIFO?
6. Как алгоритм 2Q обрабатывает повторные обращения к ключу, который находится в A1_out, но данные которого не сохранены?
7. Какова сложность операций get и put в реализации 2Q через OrderedDict?
8. Каким образом адаптивные алгоритмы (ARC) автоматически регулируют размеры очередей?
9. В каких ситуациях использование 2Q не даёт преимуществ перед LRU?
10. Приведите примеры применения 2Q в реальных системах (например, в СУБД или дисковых кэшах).

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическую часть (описание 2Q, проблема одноразовых запросов, сравнение с LRU).
4. Описание программной реализации (основные классы, пояснения методов).
5. Экспериментальную часть:
    - Параметры симуляции и генерации трафика.
    - График зависимости hit ratio от доли одноразовых запросов для 2Q и LRU.
    - Оценка памяти теневой очереди или другие метрики.
    - Для среднего и повышенного уровней — дополнительные графики и сравнительные таблицы.
6. Выводы: объяснение устойчивости 2Q, рекомендации по выбору размеров очередей.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно реализован кэш 2Q с тремя очередями.
- Проведено сравнение с LRU для трафика с разной долей сканирования.
- Построен график hit ratio, вычислен эффект от фильтрации.
- Отчёт оформлен, присутствуют выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Проведено исследование влияния размеров очередей, оценка накладных расходов памяти, динамика при пульсирующем трафике.
- Построены дополнительные графики, даны рекомендации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы адаптивный 2Q, ARC или многопоточная версия.
- Работа с реальными треками или углублённый анализ (3D-графики).
- Выводы содержат сравнительный анализ с передовыми алгоритмами, обоснование выбора в конкретных сценариях.

Снижение оценки: неправильная логика очередей (например, A1_out хранит значения или не тот порядок вытеснения), неверный замер hit ratio, низкая производительность, недостаточное тестирование.

Лабораторная работа № 10

Реализация ARC (Adaptive Replacement Cache)

---

1. Цель работы

Цель работы — изучить адаптивный алгоритм кэширования **ARC (Adaptive Replacement Cache)**, который динамически балансирует между стратегиями LRU и LFU за счёт слежения за историей вытесненных элементов. В ходе работы студент реализует ARC, сравнивает его эффективность с LRU и 2Q на синтетических трассах с различной временно́й и частотной локальностью, а также анализирует процесс автоматической настройки параметра p.

---

2. Задачи

1. Изучить структуру ARC, состоящую из четырёх очередей: T_1, T_2, B_1, B_2.
2. Реализовать алгоритм ARC с корректной обработкой попаданий и промахов, включая механизм адаптации параметра p.
3. Реализовать классический LRU-кэш и кэш 2Q для сравнительного анализа.
4. Создать генераторы синтетического трафика с различным соотношением «недавних» (recency) и «частых» (frequency) обращений, а также с переключениями паттернов доступа.
5. Измерить hit ratio трёх политик на серии тестовых трасс.
6. Визуализировать динамику изменения параметра p в процессе обработки запросов.
7. Построить графики зависимости hit ratio от параметров трафика и размера кэша.

---

3. Теоретическое обоснование

3.1. Недостатки статических политик

Политики LRU и LFU хорошо работают каждая в своей области: LRU использует временну́ю локальность (recency), LFU — частотную (frequency). Однако реальные рабочие нагрузки могут переключаться между этими режимами или сочетать их. Статические политики не способны адаптироваться, что приводит к снижению hit ratio. Алгоритм **ARC**, предложенный Nimrod Megiddo и Dharmendra Modha, решает эту проблему, непрерывно настраивая соотношение объёмов кэша, выделяемых под «недавние» и «частые» элементы.

3.2. Структура ARC

ARC использует четыре двусвязных списка (реализуются через OrderedDict или собственные очереди):

- T_1 — LRU-список, содержащий страницы, к которым обращались **только один раз** (или возвращённые из B_1). Это «рециентные» элементы.
- T_2 — LRU-список, содержащий страницы, к которым обращались **как минимум дважды** (продвинутые из T_1 или вернувшиеся из B_2). Это «частые» элементы.
- B_1 — теневой LRU-список, хранящий **только ключи** (без данных) страниц, вытесненных из T_1.
- B_2 — теневой LRU-список для ключей, вытесненных из T_2.

Обозначим размеры: $|T_1|, |T_2|, |B_1|, |B_2|$.
Общий размер кэша данных ограничен: $|T_1| + |T_2| \leq c$.
Суммарный размер теневых списков обычно также ограничен c: $|B_1| + |B_2| \leq c$.

Адаптивный параметр p ($0 \leq p \leq c$) задаёт целевой размер T_1. При $p = |T_1|$ поведение ARC приближается к чистому LRU, при $p = 0$ — к LFU. На старте обычно $p = 0$ (или $p = c/2$).

3.3. Обработка запросов

Пусть L_i — один из списков T_1, T_2, B_1, B_2.
Попадание в кэш (T1 или T2):

- Если ключ найден в T_1 или T_2, он **перемещается в голову** T_2 (удаляется из исходного списка и вставляется как MRU в T_2). Значение возвращается.

Промах:

1. **Ключ найден в B_1:** это означает, что недавно вытесненный «одноразовый» ключ запрошен повторно, следовательно, стоит увеличить долю «рециентного» кэша.
    - Адаптация: $p = \min(c, p + \max(|B_2|/\max(|B_1|, 1), 1))$. (Упрощённо: $p = \min(c, p + 1)$.)
    - Вызывается процедура replace(key), чтобы освободить место (если кэш полон).
    - Ключ удаляется из B_1 и вставляется в голову T_2 (считывается из внешнего хранилища).
2. **Ключ найден в B_2:** ранее частый элемент вытеснен, но запрошен снова — доля «частотного» кэша должна быть увеличена.
    - Адаптация: $p = \max(0, p - \max(|B_1|/\max(|B_2|, 1), 1))$. (Упрощённо: $p = \max(0, p - 1)$.)
    - Вызов replace(key).
    - Ключ удаляется из B_2 и помещается в голову T_2.
3. **Полный промах (ключа нет ни в одном списке):**
    - Вызов replace(key).

- Ключ вставляется в голову T_1 (как MRU) с полученным значением.

3.4. Процедура replace(key)

Освобождает один слот в кэше данных, если необходимо. Правила:

- Если | T_1 |> 0 и ((ключ в B_1 и | T_1 |$= p$) или (| T_1 |$> p$)):
  - Удалить LRU-элемент из хвоста T_1. Его ключ попадает в голову B_1. Если при этом | B_1 | +| B_2 |$\geq c$, удалить хвост из B_1.
- Иначе:
  - Удалить LRU-элемент из хвоста T_2. Его ключ попадает в голову B_2. Аналогично при переполнении теней удалить хвост из B_2.

3.5. Адаптивность p

Наблюдение: при преобладании сканирующих (одноразовых) запросов учащаются попадания в B_1, p растёт, расширяя T_1 и защищая T_2 от вымывания. При доминировании частотного доступа растёт p? Нет, попадания в B_2 уменьшают p, увеличивая T_2 для долгоживущих популярных объектов. Так ARC самостоятельно подстраивается под характер нагрузки.

3.6. Сравнение с LRU и 2Q

LRU имеет один список, 2Q фиксирует три очереди со статическими размерами. ARC лишён статических порогов, динамически меняя соотношение T_1 и T_2. Это даёт преимущество на смешанных и переключающихся нагрузках.

---

4. Пример практического выполнения

4.1. Реализация ARC на Python

Используем OrderedDict для хранения порядка T_1, T_2, B_1, B_2. Класс ARCCache содержит:

python

```
from collections import OrderedDict


class ARCCache:
  def __init__(self, capacity: int):
    self.c = capacity
    self.p = 0                # целевой размер T1
    self.T1 = OrderedDict()   # key -> value
    self.T2 = OrderedDict()
```

```
    self.B1 = OrderedDict()       # key -> None
    self.B2 = OrderedDict()

def _replace(self, key):
    """Вытесняет один элемент из кэша данных согласно политике ARC."""
    if self.T1 and ((key in self.B1 and len(self.T1) == self.p) or len(self.T1) > self.p):
        # вытесняем из T1
        old_key, _ = self.T1.popitem(last=False)
        self._add_to_B1(old_key)   # в голову B1
    else:
        # вытесняем из T2
        old_key, _ = self.T2.popitem(last=False)
        self._add_to_B2(old_key)

def _add_to_B1(self, key):
    if len(self.B1) + len(self.B2) >= self.c:
        # удаляем LRU из B1 или B2 (предпочтение B1)
        if self.B1:
            self.B1.popitem(last=False)
        else:
            self.B2.popitem(last=False)
    self.B1[key] = None

def _add_to_B2(self, key):
    if len(self.B1) + len(self.B2) >= self.c:
        if self.B2:
            self.B2.popitem(last=False)
        else:
            self.B1.popitem(last=False)
```

```
        self.B2[key] = None

    def get(self, key):
        if key in self.T1:
            value = self.T1.pop(key)
            self.T2[key] = value       # перемещаем в T2 (голова)
            return value
        if key in self.T2:
            self.T2.move_to_end(key)   # обновить позицию MRU
            return self.T2[key]
        return -1                      # промах

    def put(self, key, value):
        if key in self.T1:
            # обновление: переместить в T2
            del self.T1[key]
            self.T2[key] = value
            return
        if key in self.T2:
            self.T2[key] = value
            self.T2.move_to_end(key)
            return
        # промах: обработка с использованием теневых списков
        if key in self.B1:
            # адаптация p
            self.p = min(self.c, self.p + max(1, len(self.B2) // max(len(self.B1), 1)))
            self._replace(key)
            del self.B1[key]
            self.T2[key] = value
```

```
        elif key in self.B2:
            self.p = max(0, self.p - max(1, len(self.B1) // max(len(self.B2), 1)))
            self._replace(key)
            del self.B2[key]
            self.T2[key] = value
        else:
            # полный промах
            if len(self.T1) + len(self.T2) >= self.c:
                self._replace(key)
            self.T1[key] = value    # вставляем в T1 (MRU)
            self.T1.move_to_end(key) # (OrderedDict сохраняет порядок при добавлении в конец)
```

Примечание. В представленной реализации при вставке в T_1 мы кладем ключ обычным присваиванием, а затем вызываем move_to_end(key) для гарантии, что он в конце (MRU). OrderedDict в Python 3.7+ сохраняет порядок вставки: self.T1[key] = value помещает ключ в конец, что соответствует голове LRU. Поэтому move_to_end избыточен при первом добавлении, но не вредит. Для B_1 и B_2 порядок MRU соответствует вставке в конец, что и происходит.

4.2. Генераторы синтетического трафика

Создадим несколько типов трафика с разным балансом recency/frequency:

- **Recency-oriented**: доступ к меняющемуся рабочему множеству, где недавние запросы повторяются часто, но популярность быстро смещается. Например, генератор повторяет последние K уникальных ключей.
- **Frequency-oriented**: стабильное Zipf-распределение с выраженной скошенностью; одни и те же ключи популярны долгое время.
- **Mixed**: комбинация recency и frequency с периодическим переключением.
- **Scanning**: поток с большой долей одноразовых запросов.

Пример генератора смешанного трафика:

python

```
import random


def generate_mixed_trace(length, hot_keys, recency_ratio=0.5, switch_every=None):
    """hot_keys: список популярных ключей.
```

recency_ratio: доля запросов, подчиняющихся временной локальности (циклическое повторение недавних).

Остальные запросы равномерно распределены по hot_keys (частотная локальность).

switch_every: через сколько запросов переключить набор популярных.

"""

```
requests = []
recency_set = []  # список последних ключей
recency_size = min(50, len(hot_keys))
for i in range(length):
    if switch_every and i % switch_every == 0:
        # переключение популярных ключей
        random.shuffle(hot_keys)
    if random.random() < recency_ratio:
        # генерируем ключ с временной локальностью: либо новый, либо повтор недавнего
        if not recency_set or random.random() < 0.2:
            key = random.choice(hot_keys)
        else:
            key = random.choice(recency_set)
        recency_set.append(key)
        if len(recency_set) > recency_size:
            recency_set.pop(0)
    else:
        key = random.choice(hot_keys)
    requests.append(key)
return requests
```

4.3. Симуляция и замер метрик

Стандартная функция симуляции:

python

```
def simulate(cache_class, capacity, requests):
```

```
    cache = cache_class(capacity)
    hits = 0
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)   # значение = ключ
    return hits / len(requests) if requests else 0
```

Для ARC дополнительно запишем историю изменения параметра p во времени (для визуализации). Модифицируем симуляцию для ARC:

python

```
def simulate_arc(requests, capacity):
    cache = ARCCache(capacity)
    hits = 0
    p_history = []
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)
        p_history.append(cache.p)
    return hits / len(requests), p_history
```

4.4. Визуализация адаптации p

После запуска на смешанном трафике с переключениями построим график.

python

```
import matplotlib.pyplot as plt


requests = generate_mixed_trace(50000, [f"key_{i}" for i in range(500)], recency_ratio=0.5, switch_every=10000)
hr, p_hist = simulate_arc(requests, capacity=200)
```

```
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)

plt.plot(p_hist)

plt.xlabel('Номер запроса')

plt.ylabel('Значение p')

plt.title('Адаптация параметра p')

plt.grid(True)


plt.subplot(1,2,2)

# сравнение с LRU и 2Q

hr_lru = simulate(LRUCache, 200, requests)

hr_2q = simulate(TwoQCache, 200, requests)  # надо создать экземпляр с cap_in=40, cap_am=160, cap_out=40? уточним

# ...

plt.bar(['LRU','2Q','ARC'], [hr_lru, hr_2q, hr])

plt.ylabel('Hit Ratio')

plt.title('Сравнение алгоритмов')

plt.tight_layout()

plt.show()
```

График p_hist покажет, как параметр увеличивается при сканирующих участках и уменьшается на стабильных.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать класс ARCCache и провести симуляцию на одном из типов трафика. Измерить hit ratio для ARC, LRU и 2Q. Построить график изменения параметра p во времени (первые 10 000 запросов). Размер кэша и параметры трафика взять из таблицы.

Вариант	Емкость кэша (c)	Тип трафика	Длина трека	Дополнительно
1	100	Recency (K=40)	200 000	p в T=0..10000
2	200	Frequency (Zipf α=1.2)	100 000	
3	150	Mixed (recency 0.3)	150 000	переключение каждые 30 000
4	300	Scanning (scan 60%)	80 000	
5	250	Mixed (recency 0.7)	120 000	переключение каждые 25 000
6	180	Recency (K=30)	180 000	
7	220	Frequency (Zipf α=1.0)	130 000	
8	120	Mixed (recency 0.5)	90 000	переключение каждые 20 000
9	280	Scanning (scan 40%)	140 000	
10	170	Recency (K=50)	160 000	
11	320	Frequency (Zipf α=1.5)	200 000	
12	130	Mixed (recency 0.4)	110 000	переключение каждые 22 000

Вариант	Емкость кэша (c)	Тип трафика	Длина трека	Дополнительно
13	260	Recency (K=20)	190 000	
14	140	Scanning (scan 80%)	70 000	
15	200	Mixed (recency 0.6)	100 000	переключение каждые 15 000

Требования: корректный код ARC, таблица hit ratio трёх политик, график изменения p, сравнительные выводы.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Исследовать влияние начального значения p (0, c/2, c) на сходимость и итоговый hit ratio. Построить график hit ratio от начального p для одного трафика.
- Подробно изучить поведение размеров четырёх очередей (| T_1 |, | T_2 |, | B_1 |, | B_2 |) во времени, построить их графики вместе с графиком p.
- Реализовать вариант ARC с упрощённой адаптацией (p ± 1) и сравнить с классической формулой, использующей соотношение размеров B1 и B2.
- Провести тест на «пилообразном» трафике: попеременное преобладание Recency и Frequency. Измерить, за сколько запросов ARC перестраивается (время адаптации) по сравнению с 2Q.
- Для выбранных параметров построить трёхмерный график зависимости hit ratio от размера кэша и доли Recency в трафике.

Вариант	Базовая конфигурация	Дополнительное задание
1	вариант 1	Начальное p, графики очередей
2	вариант 2	Упрощённая адаптация, пилообразный трафик
3	вариант 3	3D-график hit ratio

Вариант	Базовая конфигурация	Дополнительное задание
4	вариант 4	Графики очередей, время адаптации
5	вариант 5	Начальное p, упрощённая адаптация
6	вариант 6	Пилообразный трафик, 3D-график
7	вариант 7	Графики очередей, сравнение адаптации
8	вариант 8	Начальное p, время перестройки
9	вариант 9	3D-график, упрощённая адаптация
10	вариант 10	Пилообразный трафик, графики очередей
11	вариант 11	3D-график, начальное p
12	вариант 12	Графики очередей, упрощённая адаптация
13	вариант 13	Пилообразный, 3D-график
14	вариант 14	Начальное p, упрощённая адаптация
15	вариант 15	Все дополнительные пункты

Требования: детальный анализ адаптации, дополнительные визуализации, сравнение формул адаптации.

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать алгоритм **CAR (Clock with Adaptive Replacement)**, аппроксимирующий ARC с использованием битов обращения, и сравнить hit ratio и сложность реализации.
- Провести симуляцию с **реальными треками** (например, из MSR Cambridge, FIU или трассировок веб-кеша). Сравнить ARC, LRU и 2Q.

- Реализовать многопоточную версию ARC (с блокировкой всего кэша) и измерить пропускную способность в операциях/сек по сравнению с однопоточной.
- Исследовать влияние **ограничения на суммарный размер теней** (c, 0.5c, неограниченно) на hit ratio и адаптацию.
- Построить анимированный график эволюции p и размеров очередей на протяжении трассы (сохранить как GIF или серию кадров).

Вариант	Базовые параметры	Расширенное задание
1	c=200, Mixed трафик	CAR, реальный трек
2	c=150	Многопоточный ARC, влияние размера теней
3	c=250	Реальный трек, анимация адаптации
4	c=300	CAR, ограничение теней
5	c=180	Многопоточность, реальный трек
6	c=220	Анимация, CAR
7	c=100	Реальный трек, влияние теней
8	c=280	CAR, многопоточность
9	c=130	Анимация, ограничение B1+B2
10	c=260	Реальный трек, CAR
11	c=170	Многопоточный ARC, анимация
12	c=320	Реальный трек, сравнение с CAR
13	c=140	Влияние теней, многопоточность

Вариант	Базовые параметры	Расширенное задание
14	c=210	CAR, анимация
15	c=200	Комплексное сравнение

Требования: законченное исследование, код расширенных алгоритмов, работа с реальными данными или анимацией, глубокие выводы.

---

6. Контрольные вопросы

1. Какую проблему призван решить алгоритм ARC? В чём его принципиальное отличие от LRU и LFU?
2. Опишите назначение каждого из четырёх списков в ARC. Почему B_1 и B_2 не хранят данные?
3. Как работает адаптация параметра p при попадании в теневые списки? Почему это позволяет балансировать между recency и frequency?
4. В каком случае ARC вытесняет элемент из T_1, а в каком из T_2? От чего это зависит?
5. Каков асимптотический порядок сложности операций get и put в реализации с OrderedDict?
6. Как влияет начальное значение p на эффективность кэша? Обязательно ли настраивать его вручную?
7. В чём преимущества ARC перед алгоритмом 2Q? В каких условиях 2Q может быть предпочтительнее?
8. Что такое «пилообразный» трафик и как на нём проявляются адаптивные свойства ARC?
9. Каковы ограничения ARC? В каких сценариях его адаптация может оказаться неэффективной?
10. Где применяется ARC в современных системах (СУБД, дисковые кэши, веб-серверы)?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.

3. Теоретическая часть: принцип работы ARC, описание списков, механизм адаптации.
4. Программная реализация: листинг класса ARCCache, пояснения к процедурам replace и адаптации.
5. Экспериментальная часть:
    - Конфигурация трафика и параметры кэша.
    - Таблица hit ratio для ARC, LRU, 2Q.
    - График изменения параметра p во времени.
    - Сравнительные графики зависимости hit ratio от параметров трафика.
    - (для среднего/повышенного) дополнительные графики и метрики.
6. Выводы по работе, отражающие адаптивные свойства ARC и его практическую ценность.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно реализован алгоритм ARC с четырьмя списками.
- Проведена симуляция по варианту, измерен hit ratio, построен график изменения p.
- Отчёт содержит основные разделы и выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Исследовано влияние начального p, построены графики очередей, проведён анализ упрощённой адаптации или тест на пиле.
- Выводы аргументированы, демонстрируют понимание процесса адаптации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован CAR и/или многопоточная версия, проведены эксперименты с реальными треками или анимацией.
- Отчёт содержит всесторонний сравнительный анализ, практические рекомендации, высокое качество кода.

Снижение оценки: ошибки в логике ARC, отсутствие адаптации p, неверное вытеснение, нет графиков, неработающий код.

Лабораторная работа № 11

Приближённый подсчёт частоты: Count-Min Sketch

---

1. Цель работы

Цель работы – изучить вероятностную структуру данных **Count-Min Sketch**, предназначенную для приближённого подсчёта частот элементов в потоке. Студент реализует Count-Min Sketch, экспериментально подтвердит теоретические гарантии точности и сравнил его с точным счётчиком по объёму потребляемой памяти и точности оценки частот.

---

2. Задачи

1. Реализовать Count-Min Sketch с параметрами: количество хеш-функций d (глубина) и количество столбцов w (ширина).
2. Обосновать выбор параметров d и w, исходя из заданных границ ошибки ε и вероятности δ.
3. Реализовать операции add(element) и estimate(element).
4. Экспериментально оценить точность, сравнив оценки Count-Min Sketch с точным подсчётом через HashMap.
5. Измерить и сравнить объём памяти, занимаемый Count-Min Sketch и точным счётчиком.
6. Применить Count-Min Sketch в составе упрощённой политики кэширования **TinyLFU** для принятия решения о вставке.

---

3. Теоретическое обоснование

3.1. Задача подсчёта частот в потоке

Во многих приложениях (сетевой мониторинг, анализ логов, рекомендательные системы) требуется вести подсчёт частоты появления элементов в непрерывном потоке. Точный подсчёт с использованием хеш-таблицы требует памяти, пропорциональной числу уникальных элементов, что может быть неприемлемо для высокоскоростных потоков с большим разнообразием ключей.

3.2. Структура Count-Min Sketch

Count-Min Sketch (CMS) – сублинейная по памяти вероятностная структура, предложенная Грэмом Кормоудом и С. Мутхакришнаном. Она представляет собой двумерный массив счётчиков размером $d \times w$:

- d – количество независимых хеш-функций,
- w – количество ячеек в строке.

Каждая хеш-функция равномерно отображает входной элемент в один из w столбцов.

Добавление элемента: для элемента x по каждой из d хеш-функций инкрементируется соответствующая ячейка:

$$\forall i \in [1..d], \text{sketch}[i][h_i(x)] += 1$$

Оценка частоты: частотой элемента x является минимальное значение среди соответствующих ему ячеек:

$$\hat{f}(x) = \min_{i=1..d} \text{sketch}[i][h_i(x)]$$

Истинная частота $f(x)$ всегда не превышает оценку: $\hat{f}(x) \geq f(x)$ (односторонняя ошибка).

3.3. Гарантии точности и выбор параметров

При общем числе добавленных элементов N справедлива вероятностная граница:

$$P(\hat{f}(x) > f(x) + \varepsilon N) \leq \delta$$

если выбрать

$$w = \left\lceil \frac{e}{\varepsilon} \right\rceil, d = \left\lceil \ln \frac{1}{\delta} \right\rceil,$$

где e – основание натурального логарифма. На практике часто задают ε и δ, а w и d вычисляют по этим формулам. Например, для $\varepsilon = 0.01$ и $\delta = 0.05$ (точность ~99%, уверенность 95%) получаем $w \approx 272$, $d \approx 3$.

3.4. Применение в кэшировании (TinyLFU)

TinyLFU – политика вытеснения, использующая Count-Min Sketch для хранения приблизительных частот доступа к ключам. Когда необходимо освободить место, сравниваются частоты из CMS, и вытесняется элемент с наименьшей оценкой частоты. Это позволяет приблизить поведение LFU при крайне малом расходе памяти.

---

4. Пример практического выполнения

4.1. Реализация Count-Min Sketch

Используем универсальное хеш-семейство на основе линейного преобразования по модулю простого числа и последующего приведения к ширине w.

```
python

import random

import numpy as np


class CountMinSketch:

  def __init__(self, d: int, w: int):

    self.d = d

    self.w = w

    self.table = np.zeros((d, w), dtype=np.int32)

    # Генерируем параметры для d хеш-функций: (a, b) для h(x) = ((a * hash(x) + b) % P) % w

    self.P = 2**31 - 1  # простое Мерсенна

    self.hash_params = [(random.randint(1, self.P-1), random.randint(0, self.P-1))

                for _ in range(d)]


  def _hashes(self, item):

    # item может быть строкой, приводим к числу через встроенный hash и ограничиваем размер

    h = hash(item) & 0x7fffffff

    for a, b in self.hash_params:

      yield ((a * h + b) % self.P) % self.w


  def add(self, item, count=1):

    for i, col in enumerate(self._hashes(item)):

      self.table[i][col] += count


  def estimate(self, item):

    return min(self.table[i][col] for i, col in enumerate(self._hashes(item)))
```

Примечание: для воспроизводимости можно зафиксировать random.seed(0).

4.2. Точный счётчик (HashMap)

python

```
from collections import defaultdict


class ExactCounter:
    def __init__(self):
        self.store = defaultdict(int)

    def add(self, item, count=1):
        self.store[item] += count

    def estimate(self, item):
        return self.store[item]
```

4.3. Эксперимент: точность и память

Сгенерируем поток элементов (например, с Zipf-распределением), вставим их в CMS и в точный счётчик, сравним оценки и память.

python

```
import sys
import zipfile  # не нужен, но оставим
from collections import Counter
import matplotlib.pyplot as plt


# Параметры
N = 1_000_000  # общее число добавлений
unique_keys = 50_000
alpha = 1.2
# генерируем поток
np.random.seed(42)
items = np.random.zipf(alpha, N)
# обрезаем до unique_keys
```

```
items = [f"key_{i % unique_keys}" for i in items]

# CMS с параметрами для ε=0.005, δ=0.05
eps = 0.005
delta = 0.05
w = int(np.ceil(np.e / eps))
d = int(np.ceil(np.log(1.0 / delta)))
print(f"w = {w}, d = {d}")

cms = CountMinSketch(d, w)
exact = ExactCounter()

for item in items:
    cms.add(item)
    exact.add(item)

# Оценим точность для наиболее частых ключей (например, 100 ключей)
top_keys = [f"key_{i}" for i in range(100)]  # условно
errors = []
for key in top_keys:
    true_f = exact.estimate(key)
    est_f = cms.estimate(key)
    errors.append((est_f - true_f) / N)  # относительная ошибка к N

avg_error = np.mean(errors)
max_error = np.max(errors)
print(f"Средняя ошибка (в долях N): {avg_error:.6f}, максимальная: {max_error:.6f}")

# Оценка памяти
```

Память CMS: table (d*w) ячеек int32 ≈ 4 * d * w байт + служебные

Память точного счётчика: словарь с unique_keys записями, примерно 72 байта на запись (оценка)

mem_cms = 4 * d * w

mem_exact = sys.getsizeof(dict) + unique_keys * 72 # очень грубо

print(f"Память CMS: ~{mem_cms} байт, точный счётчик: ~{mem_exact} байт")

4.4. Визуализация распределения ошибок

Построим гистограмму относительной ошибки оценок для всех уникальных ключей (или по выборке).

python

sample_keys = [f"key_{i}" for i in range(min(1000, unique_keys))]

cms_estimates = [cms.estimate(k) for k in sample_keys]

true_vals = [exact.estimate(k) for k in sample_keys]

errors_abs = [e - t for e, t in zip(cms_estimates, true_vals)]

plt.figure(figsize=(8,5))

plt.hist(errors_abs, bins=30, edgecolor='black', alpha=0.7)

plt.xlabel('Абсолютная ошибка (estimate - true)')

plt.ylabel('Количество ключей')

plt.title('Распределение ошибок Count-Min Sketch')

plt.grid(axis='y')

plt.show()

4.5. Интеграция в TinyLFU (черновой пример)

В качестве демонстрации реализуем упрощённый кэш с вытеснением на основе минимальной оценки из CMS.

python

class TinyLFUCache:

  def __init__(self, capacity, cms_d, cms_w):

    self.capacity = capacity

    self.cms = CountMinSketch(cms_d, cms_w)

```
        self.store = {}  # key -> value

    def get(self, key):
        if key in self.store:
            self.cms.add(key)
            return self.store[key]
        return -1

    def put(self, key, value):
        self.cms.add(key)
        if key in self.store:
            self.store[key] = value
        else:
            if len(self.store) >= self.capacity:
                # вытесняем ключ с минимальной оценкой частоты
                victim = min(self.store.keys(), key=lambda k: self.cms.estimate(k))
                del self.store[victim]
            self.store[key] = value
```

Сравнить hit ratio с точным LFU на синтетическом треке.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать Count-Min Sketch, провести эксперимент на заданном потоке элементов, сравнить оценки с точным счётчиком, измерить память и среднюю/максимальную ошибку. Построить гистограмму ошибок.

Вариант	ε	δ	N (добавлений)	Тип потока	Уникальных ключей
1	0.01	0.05	500 000	Zipf α=1.2	20 000
2	0.005	0.05	1 000 000	Zipf α=1.0	50 000
3	0.02	0.1	300 000	Random (равномерное)	30 000
4	0.01	0.01	800 000	Zipf α=0.8	40 000
5	0.005	0.01	1 200 000	Zipf α=1.5	80 000
6	0.02	0.05	600 000	Mixed (Zipf + Random)	25 000
7	0.01	0.05	400 000	Zipf α=1.1	15 000
8	0.03	0.1	200 000	Random	10 000
9	0.005	0.05	1 500 000	Zipf α=1.3	100 000
10	0.01	0.05	750 000	Равномерный + выбросы	35 000
11	0.02	0.01	900 000	Zipf α=0.9	45 000
12	0.01	0.05	1 100 000	Zipf α=1.4	60 000
13	0.005	0.01	700 000	Random	25 000
14	0.03	0.05	250 000	Zipf α=1.0	12 000
15	0.02	0.1	500 000	Zipf α=1.2	30 000

Требования к отчёту: корректный код CMS, таблица сравнения памяти и точности, гистограмма ошибок, соответствие теоретическим гарантиям.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Провести анализ зависимости фактической ошибки от параметров d и w: построить графики средней и максимальной ошибки для диапазона w при фиксированном d и наоборот.
- Сравнить точность CMS при использовании различных хеш-функций: простых мультипликативных и более равномерных (SHA-1, xxHash). Оценить дисперсию ошибок.
- Реализовать операцию merge (объединение двух CMS одинаковых размеров) для агрегации распределённых подсчётов, провести тест на сумму двух половин потока.
- Построить график экономии памяти (отношение памяти точного счётчика к CMS) в зависимости от числа уникальных ключей.
- Применить CMS для выявления «тяжёлых» ключей (heavy hitters) с заданным порогом и оценить полноту и точность.

Вариант	Параметры базового	Дополнительные задания
1	вариант 1	Зависимость от w/d, merge
2	вариант 2	Сравнение хеш-функций, heavy hitters
3	вариант 3	График экономии памяти, зависимость от d
4	вариант 4	Heavy hitters, merge
5	вариант 5	Хеш-функции, зависимость от w
6	вариант 6	Merge, экономия памяти
7	вариант 7	Зависимость от d/w, сравнение хешей
8	вариант 8	Heavy hitters, график памяти

Вариант	Параметры базового	Дополнительные задания
9	вариант 9	Merge, хеш-функции
10	вариант 10	Экономия памяти, heavy hitters
11	вариант 11	Зависимость от w/d, merge
12	вариант 12	Сравнение хешей, heavy hitters
13	вариант 13	Merge, график w/d
14	вариант 14	Heavy hitters, экономия памяти
15	вариант 15	Все перечисленные дополнительные исследования

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать полноценную политику **TinyLFU** с механизмом Least Recently Used (W-TinyLFU) и сравнить hit ratio с точным LFU, LRU и FIFO на синтетических треках с изменяющейся популярностью.
- Реализовать **Count-Min Sketch с консервативным обновлением** (Conservative Update / Count-Min-mean), которая улучшает точность, обновляя только минимальные счётчики, и сравнить с обычным CMS.
- Провести эксперимент с реальными данными (логи HTTP-запросов, текстовые файлы) для измерения частот слов/IP-адресов; оценить компромисс точности и памяти.
- Построить трёхмерный график зависимости средней относительной ошибки от d и w.
- Исследовать влияние переполнения счётчиков (использовать uint8 или uint16) на точность и предложить методы борьбы (морфинг, затухание).

Вариант	Базовые параметры	Расширенное задание
1	ε=0.01, δ=0.05	TinyLFU, сравнение кэширования

Вариант	Базовые параметры	Расширенное задание
2	ε=0.005, δ=0.01	Conservative Update, 3D-график
3	ε=0.02, δ=0.05	Реальные данные (логи), переполнение счётчиков
4	ε=0.01, δ=0.05	TinyLFU, Conservative Update
5	ε=0.005, δ=0.05	3D-график ошибок, реальный датасет
6	ε=0.01, δ=0.01	TinyLFU (W-TinyLFU), переполнение
7	ε=0.02, δ=0.1	Conservative Update, реальный датасет
8	ε=0.005, δ=0.05	TinyLFU vs LRU/LFU, сравнение hit ratio
9	ε=0.01, δ=0.05	3D-график, переполнение
10	ε=0.02, δ=0.05	Реальные данные, Conservative Update
11	ε=0.01, δ=0.05	TinyLFU с Window (затухание), реальные треки
12	ε=0.005, δ=0.01	3D-график, TinyLFU
13	ε=0.03, δ=0.1	Conservative Update, 3D-график
14	ε=0.01, δ=0.05	TinyLFU, переполнение, сравнение
15	ε=0.005, δ=0.05	Комплексное сравнение всех подходов

---

6. Контрольные вопросы

1. Какие гарантии точности даёт Count-Min Sketch? Что означают параметры ε и δ?
2. Почему оценка частоты в Count-Min Sketch всегда не меньше истинной? Когда возникает завышение?

3. Как выбрать количество хеш-функций и ширину таблицы для заданных требований к точности?
4. В чём отличие Count-Min Sketch от Count Sketch (Count-Min со средним вместо минимума)?
5. Что такое универсальное хеширование и почему оно важно для CMS?
6. Как можно объединить два Count-Min Sketch, построенных на разных наборах данных? Какие условия должны выполняться?
7. Как Count-Min Sketch используется в политике кэширования TinyLFU? Каковы его преимущества?
8. Что такое проблема переполнения счётчиков и как её смягчить с помощью затухания (decay)?
9. Какие ещё существуют сублинейные структуры для подсчёта частот (Counting Bloom Filter, HyperLogLog, и т.д.)? Кратко сравните.
10. Каковы ограничения Count-Min Sketch? В каких случаях он неприменим?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: описание Count-Min Sketch, формулы для параметров, сценарии использования.
4. Программная реализация: листинг классов, пояснения выбора хеш-функций.
5. Экспериментальная часть:
    - Конфигурация эксперимента.
    - Таблицы и графики точности (гистограммы ошибок, зависимость от параметров).
    - Сравнение памяти с точным счётчиком.
    - Для среднего/повышенного уровня — дополнительные графики и сравнения (heavy hitters, TinyLFU hit ratio и т.п.).
6. Анализ результатов, обсуждение соответствия теоретическим гарантиям.
7. Выводы.
8. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Count-Min Sketch реализован, операции add и estimate работают верно.
- Проведён эксперимент согласно варианту, построена гистограмма ошибок, предоставлены замеры памяти.
- Отчёт содержит необходимые разделы, сделаны выводы о точности и экономии памяти.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Проведены дополнительные исследования: зависимость ошибки от d/w, сравнение хеш-функций, heavy hitters или операция merge.
- Построены информативные графики, аргументированы ответы на вопросы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализована политика TinyLFU и/или консервативное обновление, проведены эксперименты с реальными данными.
- Построены трёхмерные визуализации, исследована проблема переполнения.
- Выводы подкреплены глубоким анализом, даны практические рекомендации.

Снижение оценки: неверная реализация оценки (не минимум), отсутствие сравнения с точным счётчиком, пренебрежение вероятностными гарантиями, плохое оформление кода.

Лабораторная работа № 12

Реализация TinyLFU для приближённого LFU

---

1. Цель работы

Цель работы – изучить устройство приближённой частотной политики вытеснения **TinyLFU**, объединяющей Count-Min Sketch, фильтр Блума (Doorkeeper) и LRU-очередь, и экспериментально сравнить её эффективность с классическими алгоритмами LRU и ARC при ограниченных затратах памяти.

---

2. Задачи

1. Повторить устройство Count-Min Sketch (CMS) и фильтра Блума.
2. Реализовать компоненты TinyLFU:
    - Doorkeeper (фильтр Блума) для предварительного отсева одноразовых ключей;
    - Count-Min Sketch для накопления частот элементов, прошедших Doorkeeper;
    - Основной кэш на базе LRU (или Segment LRU), использующий оценки частоты для принятия решения о вытеснении.
3. Реализовать правило вставки: новый элемент помещается в кэш, только если его оценка частоты превышает оценку частоты «жертвы» — наименее часто используемого элемента в кэше.
4. Провести симуляцию на синтетических треках с изменяющейся популярностью, сравнить hit ratio с LRU и ARC.
5. Оценить накладные расходы на вычисление частоты (время и память) и сравнить с точным LFU.
6. Построить графики зависимости hit ratio от размера кэша и параметров CMS.

---

3. Теоретическое обоснование

3.1. Мотивация: приближённый LFU

Точный LFU требует хранения всех когда-либо встреченных ключей и их точных частот, что приводит к потреблению памяти, пропорциональному числу уникальных элементов. TinyLFU, предложенный в 2013 году, решает эту проблему с помощью вероятностных структур, позволяя достичь качества, близкого к LFU, при жёстко фиксированном объёме служебной памяти.

3.2. Архитектура TinyLFU

TinyLFU состоит из трёх уровней:

1. **Doorkeeper** – обычный фильтр Блума. При первом появлении ключ добавляется в фильтр Блума, но **не** попадает в Count-Min Sketch. При повторном появлении, если ключ уже присутствует в фильтре, он считается «повторным» и только тогда его частота инкрементируется в Count-Min Sketch. Таким образом, Doorkeeper отсеивает большинство одноразовых ключей и защищает CMS от зашумления.
2. **Count-Min Sketch (CMS)** – хранит приближённые частоты только тех ключей, которые прошли через Doorkeeper. При оценке частоты freq(key) суммируются результаты CMS и (опционально) добавка за присутствие в фильтре (обычно +1, если ключ есть в фильтре).
3. **Основной кэш** – обычно это LRU-подобная очередь (например, SLRU – два сегмента: probationary и protected). В упрощённой учебной реализации мы используем единую LRU-очередь, но вытеснение происходит на основе **частоты**, а не давности. Замещение: в кэше фиксированной ёмкости при вставке нового ключа ищется «жертва» – ключ с наименьшей оценкой частоты TinyLFU. Если частота нового ключа выше, происходит вытеснение жертвы и вставка нового.

3.3. Правило вставки (TinyLFU admission policy)

Алгоритм принятия решения:

- Промах (ключа нет в кэше):
    1. Если кэш не полон – добавить ключ.
    2. Иначе выбрать из кэша ключ-жертву victim с наименьшей оценкой частоты (TinyLFU estimate).
    3. Если estimate(new_key) > estimate(victim), удалить victim и вставить new_key. Иначе – оставить кэш без изменений.

Это гарантирует, что в кэше остаются элементы с максимальными оценками частоты, приближаясь к поведению LFU.

3.4. Связь с LRU и ARC

TinyLFU сохраняет некоторую временну́ю локальность за счёт использования LRU-очереди для выбора жертвы при равных оценках частоты? Обычно используется комбинация: жертва выбирается как наименее часто используемая, а при равных частотах — наименее давняя (LRU). Современные реализации (Caffeine) добавляют окно (Window TinyLFU) для улучшения реакции на изменения.

3.5. Сложность и память

TinyLFU требует:

- Фильтр Блума: m бит.
- Count-Min Sketch: $d \times w$ счётчиков (обычно 4 бита на счётчик, но можно int8).
- Служебная память основного кэша (ключи + значения). Общий объём служебной памяти фиксирован и гораздо меньше, чем у точного LFU.

---

4. Пример практического выполнения

4.1. Реализация Doorkeeper (фильтр Блума)

python

```
import mmh3  # murmurhash3
import math


class BloomFilter:
    def __init__(self, capacity: int, error_rate=0.01):
        self.size = self._optimal_size(capacity, error_rate)
        self.hash_count = self._optimal_hash_count(self.size, capacity)
        self.bit_array = bytearray((self.size + 7) // 8)

    def _optimal_size(self, n, p):
        return int(-n * math.log(p) / (math.log(2)**2))

    def _optimal_hash_count(self, m, n):
        return int((m / n) * math.log(2))

    def _hashes(self, item):
        h = item if isinstance(item, int) else hash(item)
        for i in range(self.hash_count):
            yield mmh3.hash(str(h), i) % self.size

    def add(self, item):
```

```
        for pos in self._hashes(item):
            byte = pos // 8
            bit = pos % 8
            self.bit_array[byte] |= (1 << bit)

    def contains(self, item):
        for pos in self._hashes(item):
            byte = pos // 8
            bit = pos % 8
            if not (self.bit_array[byte] & (1 << bit)):
                return False
        return True
```

4.2. Count-Min Sketch (повторение ЛР11)

```
python
import numpy as np
import random

class CountMinSketch:
    def __init__(self, d, w):
        self.d = d
        self.w = w
        self.table = np.zeros((d, w), dtype=np.int32)
        self.P = 2**31 - 1
        self.hash_params = [(random.randint(1, self.P-1), random.randint(0, self.P-1))
                            for _ in range(d)]

    def _hashes(self, item):
        h = abs(hash(item)) % self.P
        for a, b in self.hash_params:
```

```
            yield ((a * h + b) % self.P) % self.w


    def add(self, item, count=1):
        for i, col in enumerate(self._hashes(item)):
            self.table[i][col] += count


    def estimate(self, item):
        return min(self.table[i][col] for i, col in enumerate(self._hashes(item)))
```

4.3. TinyLFU кэш

Комбинируем BloomFilter, CMS и LRU-словарь для основного кэша. При вытеснении ищем жертву с минимальной оценкой TinyLFU.

python

```
from collections import OrderedDict


class TinyLFUCache:
    def __init__(self, cache_capacity, bf_capacity=100000, cms_d=10, cms_w=2000):
        self.capacity = cache_capacity
        self.bloom = BloomFilter(bf_capacity)
        self.cms = CountMinSketch(cms_d, cms_w)
        self.store = OrderedDict()  # key -> value


    def _freq(self, key):
        """Оценка частоты TinyLFU = CMS + возможно 1 за присутствие в bloom."""
        base = self.cms.estimate(key)
        if self.bloom.contains(key):
            base += 1
        return base


    def get(self, key):
```

```
        if key in self.store:
            # обновляем частоту (здесь не двигаем LRU, т.к. порядок используется только для выбора жертвы)
            self.cms.add(key)
            return self.store[key]
        return -1

    def put(self, key, value):
        if key in self.store:
            self.store[key] = value
            self.cms.add(key)
            return

        # промах
        if self.bloom.contains(key):
            # прошёл doorkeeper повторно, увеличиваем частоту в CMS
            self.cms.add(key)
        else:
            # первый раз: только в doorkeeper
            self.bloom.add(key)

        # admission: есть ли место или можем вытеснить
        if len(self.store) < self.capacity:
            self.store[key] = value
            return

        # ищем жертву с минимальной оценкой частоты
        victim = min(self.store.keys(), key=lambda k: self._freq(k))
        if self._freq(key) > self._freq(victim):
```

```
            del self.store[victim]
            self.store[key] = value
        # иначе игнорируем
```

4.4. Симуляция и сравнение

Создадим синтетический трек со сменой популярности, аналогично ЛР8, и сравним с реализациями LRU и ARC (из предыдущих работ).

```
python
def simulate_tinylfu(requests, cache_capacity):
    cache = TinyLFUCache(cache_capacity)
    hits = 0
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)
    return hits / len(requests)
```

Построим график hit ratio для разных размеров кэша.

```
python
import matplotlib.pyplot as plt

cache_sizes = [50, 100, 200, 500, 1000]
requests = generate_mixed_trace(200000, hot_keys=list(range(5000)), recency_ratio=0.5, switch_every=50000)

lru_hr = [simulate(LRUCache, s, requests) for s in cache_sizes]
arc_hr = [simulate(ARCCache, s, requests) for s in cache_sizes]
tinylfu_hr = [simulate_tinylfu(requests, s) for s in cache_sizes]

plt.plot(cache_sizes, lru_hr, 'o-', label='LRU')
plt.plot(cache_sizes, arc_hr, 's-', label='ARC')
```

```
plt.plot(cache_sizes, tinylfu_hr, '^--', label='TinyLFU')

plt.xlabel('Размер кэша')

plt.ylabel('Hit Ratio')

plt.legend()

plt.grid()

plt.show()
```

4.5. Замеры накладных расходов

Сравним время выполнения и объём памяти, занимаемой фильтром Блума и CMS, фиксируя их параметры.

```
python

import sys

mem_bloom = sys.getsizeof(bloom_filter.bit_array)

mem_cms = cms.table.nbytes

print(f"Память TinyLFU: Bloom={mem_bloom}, CMS={mem_cms}, всего={mem_bloom+mem_cms}")
```

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать TinyLFU с заданными параметрами CMS и Bloom-фильтра, провести симуляцию на треке с переключениями популярности, построить график hit ratio для ряда размеров кэша и сравнить с LRU.

Вариант	Ёмкость кэша (диапазон)	Параметры CMS (d, w)	Размер Bloom	Длина трека	Примечание
1	100, 200, 500, 1000	d=4, w=500	10 000	150 000	
2	50, 100, 200, 500	d=6, w=400	8 000	120 000	

Вариант	Ёмкость кэша (диапазон)	Параметры CMS (d, w)	Размер Bloom	Длина трека	Примечание
3	150, 300, 600, 1200	d=4, w=800	15 000	180 000	
4	80, 150, 300, 600	d=5, w=600	12 000	100 000	
5	200, 400, 800, 1500	d=3, w=1000	20 000	200 000	
6	120, 250, 500, 1000	d=6, w=500	10 000	140 000	
7	100, 300, 500, 800	d=4, w=700	14 000	160 000	
8	60, 120, 250, 500	d=5, w=300	6 000	80 000	
9	150, 350, 700, 1400	d=3, w=900	18 000	190 000	
10	90, 180, 360, 720	d=7, w=450	9 000	110 000	
11	200, 500, 1000, 2000	d=4, w=1200	25 000	220 000	
12	75, 150, 300, 600	d=5, w=400	8 000	95 000	
13	130, 260, 520, 1040	d=6, w=600	12 000	170 000	

Вариант	Ёмкость кэша (диапазон)	Параметры CMS (d, w)	Размер Bloom	Длина трека	Примечание
14	110, 220, 440, 880	d=4, w=550	11 000	130 000	
15	160, 320, 640, 1280	d=5, w=750	16 000	185 000	

Требования: работающий код, график зависимости hit ratio от размера кэша (TinyLFU vs LRU), оценка занимаемой памяти, выводы о преимуществах TinyLFU.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать политику **Window TinyLFU**: счётчики CMS периодически затухают (деление на 2) для адаптации к изменениям. Сравнить hit ratio с обычным TinyLFU и ARC на треке со сменой популярности.
- Исследовать влияние параметра cms_d (глубины) на точность оценки и hit ratio при фиксированной суммарной памяти CMS.
- Сравнить две стратегии выбора жертвы: а) только по оценке TinyLFU; б) по оценке, а при равенстве – LRU. Построить графики.
- Измерить среднее время операций get/put для TinyLFU и сравнить с LRU/ARC.

Вариант	Базовые параметры	Дополнительные задания
1	вариант 1	Window TinyLFU, выбор жертвы
2	вариант 2	Влияние cms_d, время
3	вариант 3	Window, выбор жертвы
4	вариант 4	Время операций, влияние cms_d
5	вариант 5	Window, сравнение LRU/ARC
6	вариант 6	Выбор жертвы, время

Вариант	Базовые параметры	Дополнительные задания
7	вариант 7	Window, cms_d анализ
8	вариант 8	Время, выбор жертвы
9	вариант 9	Window, сравнение
10	вариант 10	Влияние cms_d, window
11	вариант 11	Window, время
12	вариант 12	Выбор жертвы, сравнение
13	вариант 13	Время операций, window TinyLFU
14	вариант 14	cms_d анализ, жертва
15	вариант 15	Все перечисленные дополнительные пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать полную версию **W-TinyLFU** с адаптивным затуханием (decay interval динамический) и сравнить с Caffeine-подобным кэшем.
- Использовать **реальный трек** запросов (например, из датасета Wikipedia или трассировку веб-кеша). Сравнить hit ratio TinyLFU, LRU, ARC на реальных данных, проанализировать влияние параметров CMS.
- Реализовать **консервативное обновление** (Conservative Update) в CMS – увеличивать только минимальные счётчики. Оценить прирост точности и hit ratio.
- Построить **трёхмерный график** зависимости hit ratio от размера кэша и объёма служебной памяти (budget на CMS+Bloom).
- Провести микроархитектурный анализ потребления памяти для TinyLFU и точного LFU, включая накладные расходы Python-объектов, и визуализировать разницу.

Вариант	Базовая конфигурация	Расширенное задание
1	cache=500, CMS d=4 w=800	Реальный трек, W-TinyLFU
2	cache=300	Conservative Update, 3D-график
3	cache=700	Реальный трек, сравнение с ARC
4	cache=200	W-TinyLFU, анализ памяти
5	cache=600	3D-график, реальный трек
6	cache=400	Conservative Update, W-TinyLFU
7	cache=800	Реальный трек, 3D-график
8	cache=250	W-TinyLFU, анализ памяти
9	cache=550	Conservative Update, реальный трек
10	cache=450	3D-график, сравнение с Caffeine (описание)
11	cache=1000	Реальный трек, Conservative Update
12	cache=350	W-TinyLFU, 3D-график
13	cache=150	Анализ памяти, реальный трек
14	cache=650	Conservative Update, W-TinyLFU
15	cache=500	Комплексное сравнение

---

6. Контрольные вопросы

1. Зачем в TinyLFU используется фильтр Блума перед Count-Min Sketch? Какую проблему он решает?
2. Опишите процесс принятия решения о вставке нового элемента в TinyLFU. Как выбирается жертва?
3. Почему TinyLFU потребляет фиксированную память, в отличие от точного LFU? Какие структуры данных обеспечивают это?
4. Что такое Window TinyLFU и как затухание счётчиков влияет на адаптацию к изменению популярности?
5. Каковы преимущества и недостатки использования CMS с маленькой глубиной (d) по сравнению с большим d?
6. В каких сценариях TinyLFU проигрывает LRU? Почему?
7. Как можно реализовать выбор жертвы в кэше с учётом как частоты, так и давности использования?
8. Какие параметры CMS и фильтра Блума наиболее критичны для достижения высокого hit ratio?
9. Что такое Conservative Update в Count-Min Sketch? Как он изменяет оценки частот?
10. Где применяется TinyLFU в индустрии (например, Caffeine)? Какие дополнительные эвристики используются?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть, включающая описание компонентов TinyLFU и правила вставки.
4. Программная реализация (листинг ключевых классов).
5. Экспериментальная часть:
    - Графики hit ratio в зависимости от размера кэша для TinyLFU, LRU (и ARC).
    - Оценка накладных расходов (память, время).
    - (для среднего/повышенного уровня) дополнительные графики влияния параметров, сравнение версий Window, анализ реального трека.
6. Анализ результатов и выводы.

7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- TinyLFU реализован с CMS, Bloom-фильтром и основным кэшем.
- Проведена симуляция, построен график hit ratio в сравнении с LRU.
- Отчёт содержит основные разделы, сделаны выводы об эффективности.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Добавлен механизм Window (затухание), исследовано влияние параметров CMS, выбор жертвы, замеры времени.
- Построены дополнительные графики, анализ аргументирован.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован W-TinyLFU, консервативное обновление или работа с реальными треками.
- Построены трёхмерные графики, детальный анализ памяти, сравнение с Caffeine.
- Выводы содержат практические рекомендации по выбору параметров.

Снижение оценки: ошибки в логике admission, отсутствие Doorkeeper, неверное затухание, неработающий код.

Лабораторная работа № 13

Реализация оптимального алгоритма Белади (оффлайн-анализ)

---

1. Цель работы

Цель работы – изучить оптимальный алгоритм кэширования **Belady's MIN**, который, обладая полным знанием будущего потока запросов, достигает теоретического максимума hit ratio. Студент реализует оффлайн-алгоритм, сравнит его с онлайн-политиками LRU и ARC и оценит конкурентное соотношение, характеризующее степень близости практических алгоритмов к идеалу.

---

2. Задачи

1. Загрузить или сгенерировать трассу обращений (лог запросов) заданной длины.
2. Реализовать оптимальный алгоритм вытеснения Белади (MIN):
    - для каждого запроса предварительно вычислить индекс его следующего появления в логе;
    - при промахе в полностью заполненном кэше вытеснять элемент, чьё следующее обращение находится дальше всех (или отсутствует).
3. Вычислить hit ratio алгоритма MIN для различных размеров кэша.
4. Реализовать или использовать готовые реализации LRU и ARC, провести вычисление их hit ratio на той же трассе.
5. Построить графики зависимости hit ratio от размера кэша для трёх алгоритмов.
6. Вычислить конкурентное отношение (competitive ratio) для LRU и ARC относительно оптимального: $CR = \frac{hits_{optimal}}{hits_{algorithm}}$ или наоборот (обычно $\frac{OPT}{ALG}$, но в кэшировании hit ratio – чем больше, тем лучше; можно определить как optimal_hits / algo_hits – значение ≤ 1, показывающее долю от идеала, или algo_hits / optimal_hits – эффективность). В отчёте уточним.

---

3. Теоретическое обоснование

3.1. Оффлайн-оптимальный алгоритм Белади

В 1966 году Ласло Белади предложил оптимальную стратегию замещения страниц для случая, когда будущая последовательность обращений известна заранее. Алгоритм, известный как **MIN** (или **OPT**), при возникновении необходимости вытеснения удаляет из кэша ту страницу (ключ), которая **позже всех будет использована в будущем** (либо не будет использована вообще). Эта стратегия минимизирует общее число промахов.

3.2. Формальное описание

Пусть задан поток запросов $S = (s_1, s_2, \dots, s_T)$. Кэш вмещает C элементов. Для каждого запроса s_t:

- Если s_t уже находится в кэше — попадание (hit), никаких изменений.
- Иначе — промах (miss). Если кэш полон, необходимо выбрать элемент-жертву $v \in$ cache такой, что его **следующее появление** в потоке next$[v]$ максимально (для элементов, которые больше не появятся, расстояние считается равным $+\infty$). Удаляем v и вставляем s_t.

Гарантируется, что ни одна онлайн-стратегия (не знающая будущего) не может превзойти MIN по числу попаданий.

3.3. Предварительная обработка лога

Для эффективной реализации необходимо для каждого элемента знать позицию его следующего использования. Это достигается однопроходным сканированием лога с конца:

- Строим словарь, сопоставляющий ключу список позиций (индексов), где он встречается.
- В процессе имитации для каждого ключа в кэше хранится указатель на следующий индекс в этом списке, что позволяет быстро получить расстояние (или сам индекс) до следующего обращения.

3.4. Реализация выбора жертвы

При промахе и заполненном кэше необходимо найти элемент с максимальным next_use. Простейший способ — просканировать все элементы кэша, что даёт сложность $O(C)$ на один промах. Для умеренных C (до нескольких тысяч) это приемлемо. При больших размерах можно использовать кучу (heap), обновляемую лениво.

3.5. Конкурентное отношение (competitive ratio)

Для алгоритма A определяется отношение

$$\text{CR}_A = \frac{\text{hit\_ratio}_{\text{OPT}}}{\text{hit\_ratio}_A}$$

или обратное. В данной работе будем вычислять эффективность как долю оптимальных попаданий, достигнутую алгоритмом:

$$\text{Efficiency}_A = \frac{\text{hits}_A}{\text{hits}_{\text{OPT}}}.$$

Значение 1.0 означает, что алгоритм работает так же хорошо, как оптимальный. Это позволяет количественно сравнить «качество» LRU и ARC.

---

4. Пример практического выполнения

4.1. Генерация лога и предобработка

Сгенерируем синтетический лог с заданным распределением (Zipf) или воспользуемся готовым файлом. В примере используем Zipf-поток.

python

```
import numpy as np
from collections import defaultdict


def generate_requests(num_requests, num_unique, alpha=1.2):
    return [f"key_{i}" for i in np.random.zipf(alpha, num_requests) % num_unique]


def precompute_next_indices(requests):
    """Возвращает словарь: ключ -> список индексов, и массив next_idx для каждого шага."""
    positions = defaultdict(list)
    for idx, key in enumerate(requests):
        positions[key].append(idx)
    # Для каждого ключа добавляем бесконечность в конец для удобства
    for key in positions:
        positions[key].append(float('inf'))
    # Создаём массив next_use длиной len(requests), где для каждого запроса указан индекс следующего появления
    next_use = [None] * len(requests)
    # Указатели текущей позиции в списке для каждого ключа
    ptrs = defaultdict(int)
    for i, key in enumerate(requests):
        ptrs[key] += 1
        next_use[i] = positions[key][ptrs[key]]
    return positions, next_use
```

4.2. Алгоритм MIN

python

```
class BeladyCache:
    def __init__(self, capacity, next_use_list):
        self.cap = capacity
        self.next_use = next_use_list  # массив next_use для всего трека
        self.cache = set()
        self.hits = 0

    def process(self, requests):
        """Проходим по запросам, обновляя счётчик hits. Возвращает число попаданий."""
        for t, key in enumerate(requests):
            if key in self.cache:
                self.hits += 1
            else:
                if len(self.cache) == self.cap:
                    # найти жертву с максимальным next_use
                    victim = max(self.cache, key=lambda k: self._get_next(k, t))
                    self.cache.remove(victim)
                self.cache.add(key)
        return self.hits

    def _get_next(self, key, current_t):
        """Получить следующий индекс для ключа, начиная с позиции current_t.
           Используем предвычисленный массив, но нужно актуальное следующее появление.
           Для упрощения можно при каждой вставке искать следующее появление сканированием вперёд,
           либо воспользоваться структурой с указателями.
        """
        # Простейший вариант (для демонстрации): ищем вперёд от current_t
```

Это O(T), но для маленьких треков допустимо. В реальной реализации используем предвычисленное.

pass

Для эффективной реализации лучше использовать предвычисленные списки положений и указатели. Модифицируем класс:

python

```
class BeladyCacheOpt:
  def __init__(self, capacity, requests):
    self.cap = capacity
    self.requests = requests
    self.positions = defaultdict(list)
    for i, key in enumerate(requests):
      self.positions[key].append(i)
    for key in self.positions:
      self.positions[key].append(float('inf'))
    self.ptrs = defaultdict(int)  # текущий указатель для каждого ключа (сколько раз уже обработали)
    self.cache = set()
    self.hits = 0

  def _next(self, key):
    """Возвращает индекс следующего появления ключа после текущего момента (не учитывая уже обработанные)."""
    pos_list = self.positions[key]
    ptr = self.ptrs[key]
    return pos_list[ptr] if ptr < len(pos_list) else float('inf')

  def process(self):
    for t, key in enumerate(self.requests):
      # обновляем указатель для текущего ключа: это его t-е появление
```

```
            self.ptrs[key] += 1
            if key in self.cache:
                self.hits += 1
            else:
                if len(self.cache) == self.cap:
                    victim = max(self.cache, key=lambda k: self._next(k))
                    self.cache.remove(victim)
                self.cache.add(key)
        return self.hits
```

Примечание: указатель ptrs[key] после инкремента указывает на индекс **следующего** появления (индекс в массиве positions). На шаге t мы уже обработали ptrs[key] появлений (включая текущее), поэтому self._next(key) возвращает positions[key][ptrs[key]], что корректно.

4.3. Сравнение с LRU и ARC

Импортируем реализации из предыдущих работ или создадим их для теста.

python

```
def simulate_lru(requests, capacity):
    cache = LRUCache(capacity)  # из ЛР7
    hits = 0
    for key in requests:
        if cache.get(key) != -1:
            hits += 1
        else:
            cache.put(key, key)
    return hits
```

Аналогично для ARC.

4.4. Эксперимент и графики

Выберем несколько размеров кэша (в процентах от числа уникальных ключей или абсолютные) и вычислим hit ratio.

python

```
num_requests = 100_000
unique = 5000
requests = generate_requests(num_requests, unique, alpha=1.2)
cache_sizes = [50, 100, 200, 500, 1000]

opt_hits = []
lru_hits = []
arc_hits = []

for c in cache_sizes:
    opt = BeladyCacheOpt(c, requests)
    opt_hits.append(opt.process() / num_requests)
    lru_hits.append(simulate_lru(requests, c) / num_requests)
    arc_hits.append(simulate(ARCCache, c, requests) / num_requests)

import matplotlib.pyplot as plt
plt.plot(cache_sizes, opt_hits, 'ko-', label='Belady OPT')
plt.plot(cache_sizes, lru_hits, 's--', label='LRU')
plt.plot(cache_sizes, arc_hits, '^--', label='ARC')
plt.xlabel('Размер кэша')
plt.ylabel('Hit Ratio')
plt.legend()
plt.grid()
plt.show()

# Эффективность
eff_lru = [l / o for l, o in zip(lru_hits, opt_hits)]
eff_arc = [a / o for a, o in zip(arc_hits, opt_hits)]
print("Эффективность LRU:", eff_lru)
```

print("Эффективность ARC:", eff_arc)

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Загрузить или сгенерировать лог запросов длиной 100 000, реализовать алгоритм MIN и LRU, вычислить hit ratio для заданных размеров кэша (в единицах или процентах от уникальных ключей). Построить график hit ratio от размера кэша. Вычислить эффективность LRU относительно оптимального.

Вариант	Уникальных ключей	Распределение (Zipf α)	Размеры кэша (количество записей)
1	2000	1.2	20, 50, 100, 200, 500
2	3000	1.0	30, 70, 150, 300, 600
3	4000	0.8	40, 100, 200, 400, 800
4	5000	1.5	50, 120, 250, 500, 1000
5	2500	1.1	25, 60, 130, 260, 520
6	6000	1.3	60, 150, 300, 600, 1200
7	1500	1.4	15, 40, 80, 160, 320
8	3500	0.9	35, 80, 170, 340, 680
9	4500	1.2	45, 110, 220, 450, 900
10	5500	1.0	55, 130, 270, 550, 1100
11	1000	1.6	10, 25, 50, 100, 200

Вариант	Уникальных ключей	Распределение (Zipf α)	Размеры кэша (количество записей)
12	7000	1.2	70, 170, 350, 700, 1400
13	8000	0.7	80, 200, 400, 800, 1600
14	9000	1.3	90, 220, 450, 900, 1800
15	5000	1.0	50, 100, 200, 500, 1000

Требования: корректный код MIN, таблица hit ratio, график, оценка эффективности, выводы о близости LRU к оптимуму.

5.2. Средний уровень (на оценку «хорошо»)

В дополнение к базовому:

- Реализовать оптимизированный выбор жертвы с помощью кучи (heap) и сравнить время работы с линейным поиском.
- Включить в сравнение алгоритм ARC, построить график hit ratio для трёх алгоритмов.
- Подробно проанализировать зависимость эффективности LRU и ARC от размера кэша и параметра α Zipf.
- Вычислить не только hit ratio, но и среднее число промахов, а также конкурентное отношение в традиционном смысле (miss ratio OPT / miss ratio ALGO) и объяснить его.
- Исследовать, как изменяется конкурентное отношение при росте размера кэша.

Вариант	Параметры базового	Доп. задания
1	вариант 1	Heap-реализация, ARC, miss ratio
2	вариант 2	Эффективность ARC, анализ α
3	вариант 3	Оптимизация, конкурентное отношение
4	вариант 4	ARC, miss ratio, время

Вариант	Параметры базового	Доп. задания
5	вариант 5	Heap, анализ зависимости от α
6	вариант 6	ARC, эффективность, графики
7	вариант 7	Оптимизация, сравнение miss ratio
8	вариант 8	ARC, время, конкурентное отношение
9	вариант 9	Heap, анализ α
10	вариант 10	Все три алгоритма, miss ratio
11	вариант 11	ARC, heap, время
12	вариант 12	Конкурентное отношение, α
13	вариант 13	Heap, ARC, эффективность
14	вариант 14	Время, miss ratio, анализ
15	вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Использовать **реальный трек** запросов (например, из трассировок веб-кеша или файловой системы). Загрузить файл с последовательностью ключей, провести все сравнения.
- Реализовать **оффлайн-оптимальный алгоритм с учётом переменной стоимости** (если ключи имеют разный вес/стоимость), исследовать влияние на hit ratio.
- Построить **трёхмерный график** зависимости эффективности LRU/ARC от размера кэша и параметра α.
- Реализовать **приближённый MIN** с использованием предсказания будущего (например, на основе окна) и сравнить с точным MIN.

- Сравнить алгоритм MIN с теоретическими верхними границами для нестационарных потоков (например, с использованием рабочего множества).

Вариант	Базовые параметры	Расширенное задание
1	5000 уникальных, α=1.2	Реальный трек, 3D-график
2	3000	Приближённый MIN, переменная стоимость
3	7000	Реальный трек, анализ рабочего множества
4	4000	3D-график, приближённый MIN
5	6000	Реальный трек, переменная стоимость
6	2000	Приближённый MIN, 3D-график
7	8000	Реальный трек, сравнение с Caffeine (Ladybug)
8	1000	Переменная стоимость, анализ
9	4500	3D-график, реальный трек
10	5500	Приближённый MIN, конкурентное отношение
11	2500	Реальный трек, окно будущего
12	3500	Переменная стоимость, 3D-график
13	1500	Реальный трек, анализ рабочего множества
14	9000	Приближённый MIN, переменная стоимость
15	5000	Комплексное расширенное исследование

6. Контрольные вопросы

1. Почему алгоритм Белади считается оптимальным? Какие условия необходимы для его реализации?
2. В чём различие между оффлайн и онлайн алгоритмами кэширования?
3. Как предобработать лог запросов для эффективного поиска следующего использования?
4. Какие структуры данных можно применить, чтобы ускорить выбор жертвы в MIN?
5. Что такое конкурентное отношение (competitive ratio)? Как его вычислить для политики LRU?
6. Может ли онлайн-алгоритм достичь hit ratio, равного оптимальному, на любом треке? Обоснуйте.
7. Как параметр α распределения Zipf влияет на разрыв между LRU и OPT?
8. Почему эффективность LRU падает при уменьшении размера кэша относительно рабочего множества?
9. Какие практические выводы можно сделать из сравнения с оптимальным алгоритмом при проектировании систем кэширования?
10. Существуют ли онлайн-алгоритмы, достигающие конкурентного отношения, близкого к 1 для любых потоков? (Ответ: нет, известна нижняя граница k/(k-h+1) для детерминированных алгоритмов).

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание алгоритма MIN, его свойства, метод предобработки.
4. Реализация: ключевые фрагменты кода предобработки и вытеснения.
5. Экспериментальная часть:
    - Таблица hit ratio для разных размеров кэша (MIN, LRU, при необходимости ARC).
    - Графики зависимости hit ratio от размера кэша.
    - Вычисленное конкурентное отношение (эффективность) для онлайн-алгоритмов.

- (для среднего/повышенного) дополнительные метрики и графики.

6. Анализ результатов, выводы о близости практических алгоритмов к идеалу.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Алгоритм MIN реализован корректно.
- Проведено сравнение с LRU, построен график hit ratio, вычислена эффективность.
- Отчёт оформлен, приведены выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены требования базового уровня.
- Реализована эффективная версия с кучей или проведено сравнение времени.
- Добавлен ARC в сравнение, проанализированы конкурентные отношения для разных размеров, учтено влияние α.
- Выводы содержат количественный анализ.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Работа проведена на реальных данных или реализован приближённый MIN.
- Построены трёхмерные графики, проведён продвинутый анализ.
- Всестороннее исследование, глубокие выводы, практические рекомендации.

Снижение оценки: неверная логика вытеснения (не максимальный следующий индекс), отсутствие предобработки, ошибки в расчёте hit ratio, небрежное оформление.

Лабораторная работа № 14

Распределённое кэширование: имитация работы Memcached-кластера

---

1. Цель работы

Цель работы — изучить принципы построения распределённого кэш-кластера на основе **Consistent Hashing**, освоить методы обнаружения «горячих» (часто запрашиваемых) ключей и исследовать влияние репликации на балансировку нагрузки и максимальную загрузку отдельных узлов. В ходе работы студент создаст симулятор кластера, сравнит стратегии репликации и сопоставит эффективность распределённого кэша с централизованным решением.

---

2. Задачи

1. Реализовать имитацию распределённого кэш-кластера: несколько узлов, каждый содержит локальное хранилище ключей.
2. Использовать Consistent Hashing (с виртуальными узлами) для отображения ключей на узлы.
3. Внедрить механизм обнаружения «горячих» ключей на основе скользящего окна запросов.
4. Реализовать две стратегии репликации горячих ключей:
    - статическая репликация на R дополнительных узлов по кольцу;
    - адаптивная репликация, включающаяся при превышении порога нагрузки.
5. Провести эксперименты, измерив максимальную нагрузку на узел, коэффициент вариации нагрузки, и сравнить результаты с отсутствием репликации.
6. Сравнить поведение распределённого кэша с централизованным (один узел, обслуживающий все запросы).
7. Построить графики распределения нагрузки и зависимости максимальной нагрузки от размера кластера и уровня репликации.

---

3. Теоретическое обоснование

3.1. Проблема «горячих» ключей в распределённом кэше

В распределённых кэш-системах, таких как Memcached или Redis Cluster, нагрузка обычно распределяется с помощью Consistent Hashing. Однако если некоторые ключи становятся чрезвычайно популярными (hot keys), соответствующий узел может получить

непропорционально большое количество запросов. Возникает «перекос» (hotspot), снижающий общую пропускную способность и увеличивающий задержки.

3.2. Репликация как метод борьбы с перекосом

Для смягчения влияния горячих ключей их копии (реплики) размещают на нескольких узлах. Запросы к такому ключу распределяются между всеми репликами, снижая нагрузку на отдельный сервер. Основные стратегии:

- **Статическая репликация**: при обнаружении горячего ключа он дублируется на R дополнительных узлов (например, следующих по кольцу). Параметр R фиксирован.
- **Адаптивная репликация**: число реплик динамически меняется в зависимости от текущей интенсивности запросов к ключу. Если частота превышает заданный порог, число реплик увеличивается; при спаде — уменьшается.

3.3. Обнаружение горячих ключей

В реальных системах (Twitter, Facebook) используются скользящие окна и структуры вроде Count-Min Sketch для оценки частоты. В учебной реализации мы применяем **точное скользящее окно** последних W запросов: подсчитываем частоту каждого ключа в окне; ключи, частота которых превышает заданный процентиль (например, top 1%), считаются горячими. Список горячих ключей периодически обновляется.

3.4. Метод распределения запросов к горячим ключам

При запросе на горячий ключ клиент (либо координатор) равномерно случайным образом выбирает один из узлов, хранящих реплику этого ключа. В симуляторе для каждого такого ключа хранится список всех узлов-реплик (основной узел + R дополнительных). При операции get(hot_key) случайный узел из этого списка обрабатывает запрос и увеличивает свой счётчик нагрузки.

3.5. Сравнение с централизованным кэшем

Централизованный кэш (один мощный сервер) лишён проблемы перекоса нагрузки, но ограничен пропускной способностью одного узла и не масштабируется. Распределённый кэш ценой небольшого усложнения позволяет обслужить суммарно больше запросов, если пики нагрузки на отдельные ключи сглажены репликацией.

---

4. Пример практического выполнения

4.1. Структура узла и кластера

Узел хранит словарь data (ключ → значение) и счётчик обращений load. Кластер содержит кольцо Consistent Hashing, список узлов и механизм обнаружения горячих ключей.

python

```
import random
from collections import deque, defaultdict, Counter
import bisect
import hashlib

RING_SIZE = 2**32

def hash_to_ring(key):
    h = hashlib.sha1(key.encode()).hexdigest()
    return int(h[-8:], 16) % RING_SIZE

class ConsistentHashRing:
    def __init__(self):
        self.ring = []
        self.pos_to_node = {}

    def add_node(self, node_id, vnodes=100):
        for i in range(vnodes):
            vkey = f"{node_id}::vnode{i}"
            pos = hash_to_ring(vkey)
            bisect.insort(self.ring, pos)
            self.pos_to_node[pos] = node_id

    def remove_node(self, node_id):
        to_remove = [pos for pos, n in self.pos_to_node.items() if n == node_id]
        for pos in to_remove:
            self.ring.remove(pos)
            del self.pos_to_node[pos]
```

```
    def get_node(self, key):
        if not self.ring:
            raise RuntimeError("Ring empty")
        kh = hash_to_ring(key)
        idx = bisect.bisect_left(self.ring, kh)
        if idx == len(self.ring):
            idx = 0
        return self.pos_to_node[self.ring[idx]]

    def get_next_nodes(self, key, count):
        """Возвращает count следующих узлов по кольцу (без повторов)."""
        nodes = []
        kh = hash_to_ring(key)
        idx = bisect.bisect_left(self.ring, kh)
        seen = set()
        for i in range(len(self.ring)):
            pos = self.ring[(idx + i) % len(self.ring)]
            node = self.pos_to_node[pos]
            if node not in seen:
                nodes.append(node)
                seen.add(node)
                if len(nodes) == count:
                    break
        return nodes
```

4.2. Узел распределённого кэша

```
python
class CacheNode:
    def __init__(self, node_id):
        self.node_id = node_id
```

```
        self.store = {}
        self.load = 0          # количество обслуженных get-запросов

    def get(self, key):
        self.load += 1
        return self.store.get(key, None)

    def put(self, key, value):
        self.store[key] = value
```

4.3. Симулятор кластера с обнаружением горячих ключей и репликацией

```
python
class DistributedCacheSimulator:
    def __init__(self, num_nodes, vnodes=100, window_size=10000, hot_threshold_ratio=0.01):
        self.num_nodes = num_nodes
        self.nodes = {f"node-{i}": CacheNode(f"node-{i}") for i in range(num_nodes)}
        self.ring = ConsistentHashRing()
        for nid in self.nodes:
            self.ring.add_node(nid, vnodes)

        self.window = deque(maxlen=window_size)
        self.freq_counter = Counter()
        self.hot_keys = set()
        self.hot_threshold = max(1, int(window_size * hot_threshold_ratio))
        self.replicas = defaultdict(list)   # key -> список узлов, содержащих реплику

    def _update_hot_keys(self):
        # Обновляем список горячих ключей на основе частоты в окне
        self.freq_counter.clear()
        for k in self.window:
```

```
        self.freq_counter[k] += 1
    # Порог: top 1% (или заданный)
    threshold = self.hot_threshold
    new_hot = {k for k, cnt in self.freq_counter.items() if cnt >= threshold}
    # Обновляем реплики: добавляем для новых горячих, удаляем для остывших
    for k in new_hot - self.hot_keys:
        self._add_replicas(k)
    for k in self.hot_keys - new_hot:
        self._remove_replicas(k)
    self.hot_keys = new_hot

def _add_replicas(self, key, R=2):
    primary = self.ring.get_node(key)
    next_nodes = self.ring.get_next_nodes(key, R+1)  # включая primary
    replicas_list = [n for n in next_nodes if n != primary][:R]
    self.replicas[key] = [primary] + replicas_list
    # Размещаем копию данных на всех репликах (если ещё нет)
    value = self.nodes[primary].store.get(key, f"value_of_{key}")
    for nid in replicas_list:
        self.nodes[nid].store[key] = value

def _remove_replicas(self, key):
    for nid in self.replicas.get(key, []):
        if key in self.nodes[nid].store:
            del self.nodes[nid].store[key]
    if key in self.replicas:
        del self.replicas[key]

def get(self, key):
```

```
        self.window.append(key)
        # Периодическое обновление горячих (каждые N запросов)
        if len(self.window) % 1000 == 0:
            self._update_hot_keys()

        # Если ключ горячий – выбираем случайную реплику
        if key in self.hot_keys:
            target_node = random.choice(self.replicas[key])
        else:
            target_node = self.ring.get_node(key)

        node_obj = self.nodes[target_node]
        value = node_obj.get(key)
        if value is None:
            # Промах: записываем ключ на основной узел
            primary_node = self.ring.get_node(key)
            self.nodes[primary_node].put(key, f"value_of_{key}")
            value = f"value_of_{key}"
        return value
```

4.4. Централизованный кэш (для сравнения)

```
python
class CentralizedCache:
    def __init__(self):
        self.store = {}
        self.load = 0

    def get(self, key):
        self.load += 1
        if key not in self.store:
```

```
            self.store[key] = f"value_of_{key}"
        return self.store[key]
```

4.5. Эксперимент и визуализация

Генерируем трафик с несколькими «горячими» ключами и множеством фоновых запросов, измеряем максимальную нагрузку и коэффициент вариации на узлах.

```
python

def generate_traffic(length, hot_keys_list, hot_ratio=0.3, background_keys=5000):
    requests = []
    for _ in range(length):
        if random.random() < hot_ratio:
            requests.append(random.choice(hot_keys_list))
        else:
            requests.append(f"bg_{random.randint(0, background_keys-1)}")
    return requests


# Параметры
NUM_NODES = 10
HOT_KEYS = ["hot1", "hot2", "hot3"]
TOTAL_REQUESTS = 100000


dist_cache = DistributedCacheSimulator(NUM_NODES, window_size=10000, hot_threshold_ratio=0.01)
cent_cache = CentralizedCache()


requests = generate_traffic(TOTAL_REQUESTS, HOT_KEYS, hot_ratio=0.3)


# Симуляция распределённого кэша
for key in requests:
    dist_cache.get(key)
```

```
# Сбор нагрузки
loads = [node.load for node in dist_cache.nodes.values()]
max_load = max(loads)
mean_load = sum(loads)/len(loads)
std_load = (sum((l-mean_load)**2 for l in loads)/len(loads))**0.5
cv = std_load / mean_load if mean_load>0 else 0

print(f"Распределённый: max={max_load}, mean={mean_load:.1f}, CV={cv:.3f}")

# Централизованный
for key in requests:
    cent_cache.get(key)
print(f"Централизованный: load={cent_cache.load}")

# Визуализация
import matplotlib.pyplot as plt
plt.bar(dist_cache.nodes.keys(), loads)
plt.axhline(mean_load, color='r', linestyle='--', label='Mean')
plt.title("Нагрузка на узлы распределённого кэша")
plt.ylabel("Число запросов")
plt.xticks(rotation=45)
plt.legend()
plt.grid(axis='y')
plt.show()
```

Повторить эксперимент при разных параметрах R и порогах.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать распределённый кэш с Consistent Hashing и статической репликацией. Провести эксперимент по варианту, измерить максимальную нагрузку и коэффициент вариации до и после включения репликации. Построить столбчатую диаграмму нагрузки.

Вариант	Узлов	VNodes	Окно	Число горячих ключей	R реплик	Длина трека	Фоновых ключей
1	10	100	5000	3	2	100 000	3000
2	8	150	8000	2	3	120 000	4000
3	12	200	10000	5	2	80 000	5000
4	6	100	6000	3	3	150 000	3500
5	15	150	12000	4	2	200 000	6000
6	10	200	7000	2	2	90 000	2000
7	9	100	9000	3	1	110 000	5000
8	7	150	5000	4	2	130 000	4500
9	11	180	11000	4	3	140 000	5500
10	13	120	8000	2	2	160 000	3000

Вариант	Узлов	VNodes	Окно	Число горячих ключей	R реплик	Длина трека	Фоновых ключей
11	5	100	4000	3	2	70 000	1500
12	14	160	13000	5	3	180 000	7000
13	8	200	6000	3	1	100 000	4000
14	10	150	10000	2	2	120 000	3500
15	12	250	15000	4	2	170 000	6000

Требования: корректная симуляция, таблица нагрузок, график, выводы о влиянии репликации.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать адаптивную репликацию: число реплик увеличивается/уменьшается на 1, если частота ключа превышает/падает ниже порога (установить два порога: верхний и нижний). Сравнить с фиксированным R по колебаниям max load.
- Провести анализ влияния размера окна на качество обнаружения горячих ключей: измерить задержку от момента появления горячего ключа до включения репликации (метрика responsiveness). Построить график задержки от размера окна.
- Сравнить hit ratio распределённого кэша (с учётом реплик) с централизованным кэшем той же ёмкости (сумма всех хранилищ узлов). Показать, что при горячих ключах репликация сохраняет hit ratio при меньшей пиковой нагрузке.

Вариант	Базовые параметры	Дополнительные задания
1	вариант 1	Адаптивная репликация, анализ окна
2	вариант 2	Сравнение с централизованным, задержка обнаружения
3	вариант 3	Адаптивная репликация, hit ratio
4	вариант 4	Анализ окна, централизованный кэш
5	вариант 5	Адаптивная, задержка
6	вариант 6	Hit ratio, влияние окна
7	вариант 7	Адаптивная, сравнение
8	вариант 8	Задержка, централизованный
9	вариант 9	Адаптивная, hit ratio
10	вариант 10	Анализ окна, адаптивная
11	вариант 11	Централизованный, задержка
12	вариант 12	Адаптивная, все метрики
13	вариант 13	Анализ окна, hit ratio
14	вариант 14	Адаптивная, централизованный
15	вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать имитацию **отказов узлов**: с определённой вероятностью узел становится недоступен на некоторое время. Исследовать, как репликация горячих ключей влияет на доступность hit ratio при сбоях.
- Моделировать **асинхронную репликацию** с задержкой: новая реплика появляется не мгновенно, а через K запросов. Оценить упущенные попадания.
- Реализовать **альтернативную стратегию** размещения реплик: не просто следующие узлы по кольцу, а случайные (но разные для каждого ключа). Сравнить равномерность нагрузки.
- Построить **тепловую карту** зависимости max load от числа узлов и количества горячих ключей.
- Сравнить три схемы: без репликации, статическая, адаптивная – на реальном логе (или синтетическом с меняющейся популярностью).

Вариант	Базовые параметры	Расширенное задание
1	10 узлов, 3 hot	Отказы узлов, асинхронная репликация
2	12 узлов	Тепловая карта, сравнение статической/адаптивной
3	8 узлов	Реальный лог (предоставить), сравнение
4	15 узлов	Асинхронная репликация, отказы
5	6 узлов	Тепловая карта, альтернативное размещение
6	10 узлов	Адаптивная vs статическая при изменении популярности
7	9 узлов	Отказы, асинхронность
8	11 узлов	Реальный трек, тепловая карта
9	13 узлов	Размещение реплик случайное/последовательное
10	7 узлов	Асинхронная репликация, отказы

Вариант	Базовые параметры	Расширенное задание
11	14 узлов	Тепловая карта max load
12	5 узлов	Сравнение всех стратегий, отказы
13	10 узлов	Реальный лог, асинхронность
14	12 узлов	Альтернативное размещение, адаптивная
15	8 узлов	Комплексное сравнение

---

6. Контрольные вопросы

1. Почему Consistent Hashing выбирается для распределения ключей в кластере, а не простая модульная функция?
2. Что такое «горячий» ключ и какие проблемы он вызывает в распределённом кэше?
3. Какие существуют методы обнаружения горячих ключей в реальном времени?
4. В чём разница между статической и адаптивной репликацией? Преимущества и недостатки каждой.
5. Как выбор параметра R (количество дополнительных реплик) влияет на нагрузку и расход памяти?
6. Каким образом распределённый кэш с репликацией сравнивается с централизованным кэшем по задержкам и пропускной способности?
7. Почему при отказе узла репликация горячих ключей помогает сохранить hit ratio?
8. Какие дополнительные сложности возникают при реализации асинхронной репликации?
9. Что такое «теневая» нагрузка (shadow load) и как она связана с репликацией?
10. Какие реальные системы (Memcached, Redis, Caffeine) используют репликацию для борьбы с горячими ключами?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: обзор проблемы горячих ключей, методы репликации, описание Consistent Hashing.
4. Программная реализация: классы симулятора, пояснения алгоритмов обнаружения и репликации.
5. Экспериментальные результаты:
    - Таблицы и графики нагрузок до и после репликации.
    - Сравнение максимальной нагрузки и CV.
    - (для среднего/повышенного) дополнительные графики, задержки обнаружения, тепловые карты.
6. Выводы по работе.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован симулятор с Consistent Hashing и статической репликацией.
- Проведён эксперимент по варианту, измерены max load и CV, построен график.
- Отчёт содержит основные разделы и выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализована адаптивная репликация и/или исследовано влияние окна, проведено сравнение с централизованным кэшем.
- Дополнительные графики и метрики, аргументированные выводы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Добавлены имитация отказов, асинхронная репликация, альтернативное размещение или работа с реальными данными.
- Всесторонний анализ, тепловые карты, практические рекомендации.

Снижение оценки: некорректная работа Consistent Hashing, отсутствие механизма обнаружения, ошибки в логике репликации, недостаточная визуализация.

Лабораторная работа № 15

Реализация Token Bucket rate limiter

---

1. Цель работы

Цель работы – изучить алгоритм ограничения частоты запросов **Token Bucket**, реализовать его в многопоточной среде, провести нагрузочное тестирование при различных паттернах поступления запросов и подтвердить соответствие фактического поведения теоретическим параметрам rate и burst.

---

2. Задачи

1. Реализовать базовый in-memory Token Bucket с параметрами rate (токенов в секунду) и capacity (максимальный всплеск).
2. Реализовать потокобезопасную версию с использованием атомарных операций (без явных блокировок, используя GIL и атомарность элементарных операций в Python).
3. Провести эксперимент, имитирующий 1000 запросов с разными временны́ми профилями: равномерный поток, пачки (burst), смешанный.
4. Измерить количество отклонённых запросов и фактическую скорость пропуска, сравнить с заданной rate.
5. Оценить реальный максимальный burst и сопоставить с capacity.
6. Построить график потребления токенов во времени и гистограмму отклонённых запросов.

---

3. Теоретическое обоснование

3.1. Проблема ограничения частоты (Rate Limiting)

В распределённых системах, API-шлюзах и облачных сервисах необходимо предотвращать перегрузку, ограничивая число запросов, поступающих от клиента или сервера в единицу времени. Простейший способ – счётчик с фиксированным окном, однако он подвержен эффекту «биения» вблизи границы окна. Более совершенный подход – **Token Bucket**.

3.2. Алгоритм Token Bucket

Алгоритм моделирует ведро, в которое с постоянной скоростью rate добавляются токены. Ведро имеет максимальную ёмкость capacity (предельный всплеск). Когда поступает запрос, из ведра изымается один токен. Если токен доступен, запрос разрешён; иначе – отклонён (либо помещается в очередь). Псевдокод:

text

```
bucket = capacity
last_refill = current_time()

allow_request():
    refill()
    if bucket >= 1:
        bucket -= 1
        return True
    return False

refill():
    now = current_time()
    elapsed = now - last_refill
    new_tokens = elapsed * rate
    bucket = min(capacity, bucket + new_tokens)
    last_refill = now
```

3.3. Параметры и свойства

- **rate** (токен/с) – определяет среднюю разрешённую частоту.
- **capacity** – максимальное число токенов, которое можно накопить за время простоя; определяет максимальный легальный burst (пачку запросов, пропускаемых мгновенно).
- Алгоритм сглаживает всплески, но допускает короткие пики в пределах capacity.

3.4. Потокобезопасность

В многопоточной среде refill и bucket являются разделяемым состоянием. В Python благодаря GIL операции чтения/присваивания атрибутов являются атомарными на уровне байткода, но составные операции (проверка + вычитание) – нет. Для гарантии корректности можно использовать threading.Lock или реализовать неблокирующий вариант с помощью queue или threading.RLock (что формально является локом, но в учебных целях допустимо). В лабораторной работе мы покажем как минимальную защиту с Lock, так и версию без явных локов (основанную на атомарности операций над float и использовании time.monotonic).

---

4. Пример практического выполнения

4.1. Простейшая реализация (не потокобезопасная)

python

```
import time


class TokenBucket:
    def __init__(self, rate: float, capacity: float):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self.last_refill = time.monotonic()

    def _refill(self):
        now = time.monotonic()
        elapsed = now - self.last_refill
        new_tokens = elapsed * self.rate
        self.tokens = min(self.capacity, self.tokens + new_tokens)
        self.last_refill = now

    def allow_request(self) -> bool:
        self._refill()
        if self.tokens >= 1.0:
            self.tokens -= 1.0
            return True
        return False
```

4.2. Потокобезопасная версия (с Lock)

python

```
import threading
```

```
class ThreadSafeTokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self.last_refill = time.monotonic()
        self.lock = threading.Lock()

    def allow_request(self):
        with self.lock:
            now = time.monotonic()
            elapsed = now - self.last_refill
            new_tokens = elapsed * self.rate
            self.tokens = min(self.capacity, self.tokens + new_tokens)
            self.last_refill = now
            if self.tokens >= 1.0:
                self.tokens -= 1.0
                return True
            return False
```

4.3. Тестирование паттернов запросов

Создадим функцию, имитирующую поток запросов во времени.

python

```
def simulate_requests(bucket, arrival_times):
    results = []
    for t in arrival_times:
        # засыпаем до t относительно старта
        # (для ускорения теста можно просто устанавливать bucket.last_refill = t и обновлять)
        results.append(bucket.allow_request())
    return results
```

Для чистоты эксперимента используем модельное время: создадим класс с управляемым временем.

python

```
class SimulatedTokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self._time = 0.0

    def advance(self, dt):
        new_tokens = dt * self.rate
        self.tokens = min(self.capacity, self.tokens + new_tokens)
        self._time += dt

    def allow_request(self):
        if self.tokens >= 1.0:
            self.tokens -= 1.0
            return True
        return False

def run_experiment(rate, capacity, arrival_profile):
    bucket = SimulatedTokenBucket(rate, capacity)
    allowed = []
    denied = []
    times = []
    current_time = 0.0
    for inter_arrival in arrival_profile:
        current_time += inter_arrival
```

```
        bucket.advance(inter_arrival)
        ok = bucket.allow_request()
        allowed.append(ok)
        denied.append(not ok)
        times.append(current_time)
    return times, allowed, denied
```

Сгенерируем профили: равномерный (интервалы 0.5 сек), пачки (группы по 5 запросов с интервалом 0.01 сек, затем простой 2 сек), смешанный.

```
python
# Профили
uniform = [1/rate] * 1000  # точно под rate
burst = []
for _ in range(200):
    burst.extend([0.01]*5)   # пачка из 5
    burst.append(2.0)        # простой
mixed = [0.5 if random.random()<0.7 else 3.0 for _ in range(1000)]
```

4.4. Запуск эксперимента и метрики

```
python
import random
random.seed(0)


rate = 2.0   # 2 запроса в секунду
capacity = 5


profiles = {'uniform': uniform, 'burst': burst, 'mixed': mixed}
for name, intervals in profiles.items():
    times, allowed, denied = run_experiment(rate, capacity, intervals)
    total = len(allowed)
    allowed_cnt = sum(allowed)
```

```
    denied_cnt = total - allowed_cnt
    duration = times[-1]
    actual_rate = allowed_cnt / duration if duration > 0 else 0
    print(f"{name}: allowed={allowed_cnt}, denied={denied_cnt}, actual_rate={actual_rate:.3f}, expected_rate={rate}")
```

4.5. Визуализация

python

```
import matplotlib.pyplot as plt


def plot_bucket(times, allowed, profile_name):
    fig, ax1 = plt.subplots(figsize=(10,4))
    ax1.plot(times, allowed, drawstyle='steps-post', label='Allowed', color='green')
    # отметим отказы
    denied_idx = [i for i, a in enumerate(allowed) if not a]
    if denied_idx:
        ax1.scatter([times[i] for i in denied_idx], [0]*len(denied_idx), color='red', marker='x', label='Denied')
    ax1.set_xlabel('Время (с)')
    ax1.set_ylabel('Разрешён / Отклонён')
    ax1.legend(loc='upper left')
    ax2 = ax1.twinx()
    tokens_over_time = compute_token_series(times, allowed, rate, capacity)
    ax2.plot(times, tokens_over_time, color='blue', alpha=0.4, label='Токены')
    ax2.set_ylabel('Токены в ведре')
    plt.title(f'Token Bucket: {profile_name}')
    plt.grid(True)
    plt.show()


plot_bucket(times, allowed, 'burst')
```

График покажет уровень токенов и моменты отказов.

4.6. Проверка burst

Зафиксируем максимальное количество последовательных разрешённых запросов (run length) – это эмпирический burst. Сравним с capacity.

python

```
def max_consecutive(allowed_list):
    mx = 0
    cur = 0
    for a in allowed_list:
        if a:
            cur += 1
            mx = max(mx, cur)
        else:
            cur = 0
    return mx


print(f"Theoretical burst = {capacity}, measured max burst = {max_consecutive(allowed)}")
```

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать Token Bucket, провести симуляцию с заданными параметрами и профилем, измерить долю отказов, фактический rate, максимальный burst. Построить график количества токенов во времени. Эксперимент выполнить для указанного профиля.

Вариант	rate (1/с)	capacity	Профиль запросов	Число запросов
1	5	10	Равномерный (интервал 0.2 с)	500
2	10	15	Пачки по 8 через 0.01 с, пауза 1 с	800

Вариант	rate (1/с)	capacity	Профиль запросов	Число запросов
3	3	6	Смешанный (70% интервал 0.3 с, 30% паузы 1 с)	600
4	8	12	Равномерный (0.125 с)	1000
5	2	20	Пачки по 30 через 0.005 с, пауза 5 с	600
6	4	8	Смешанный	900
7	6	9	Равномерный (0.1667 с)	1200
8	12	24	Пачки по 15, пауза 1.5 с	750
9	15	30	Смешанный	1500
10	7	10	Равномерный	1100
11	9	18	Пачки по 10, пауза 2 с	900
12	1	50	Редкие большие пачки (50 запросов за 0.1 с)	500
13	11	20	Смешанный	1300
14	14	28	Равномерный	1400
15	13	15	Пачки по 5, пауза 0.5 с	700

Требования к отчёту: работающий код, таблица метрик, график токенов, выводы о соответствии burst и rate.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать многопоточное тестирование: несколько потоков одновременно вызывают allow_request, зафиксировать расхождения при использовании небезопасной версии и безопасной (с Lock). Измерить пропускную способность (операций/сек) для обеих реализаций.
- Исследовать влияние capacity на задержки: построить график среднего времени ожидания разрешения в зависимости от capacity при фиксированной rate и пачечном трафике. Для этого модифицировать алгоритм: возвращать не только факт разрешения, но и время блокировки (если запрос не пропущен сразу, он ожидает). (Можно реализовать блокирующий acquire()).
- Провести тест с переменной скоростью поступления запросов (модулированный поток) и сравнить реальный rate с заданным.

Вариант	Параметры базового	Дополнительные задания
1	вариант 1	Многопоточное сравнение, влияние capacity
2	вариант 2	Блокирующий acquire, переменный поток
3	вариант 3	Многопоточное, влияние capacity
4	вариант 4	acquire, переменный трафик
5	вариант 5	Многопоточное, график ожидания
6	вариант 6	acquire, сравнение реализаций
7	вариант 7	Многопоточное, влияние capacity
8	вариант 8	Блокирующий, переменный поток
9	вариант 9	Многопоточное, время
10	вариант 10	acquire, анализ
11	вариант 11	Многопоточное, графики

Вариант	Параметры базового	Дополнительные задания
12	вариант 12	acquire, burst влияние
13	вариант 13	Многопоточное, переменный трафик
14	вариант 14	Блокирующий, анализ
15	вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **Token Bucket с приоритетами**: несколько ведер с разными rate/capacity для разных категорий запросов (например, премиум/обычные). Провести симуляцию с разным трафиком и оценить дифференцированное обслуживание.
- Реализовать **версию без блокировок** с использованием threading.Lock не для защиты, а на основе неблокирующих атомарных операций compare_exchange (эмуляция с помощью queue.Queue для передачи токенов или семафоров). (Или показать lock-free структуру на collections.deque с GIL). Провести бенчмарк пропускной способности.
- Провести сравнительный анализ с алгоритмом **Leaky Bucket** (или **Fixed Window Counter**), построить графики отказов вблизи границы окна для демонстрации недостатков.
- Построить трёхмерный график зависимости пропускной способности от rate и capacity при заданном законе поступления.
- Реализовать **распределённый Token Bucket** с общим пулом токенов через Redis или имитацию разделяемой памяти.

Вариант	Базовые параметры	Расширенное задание
1	rate=10, cap=20	Приоритетные ведра, сравнение с Leaky Bucket
2	rate=5, cap=15	Lock-free версия, бенчмарк
3	rate=20, cap=30	Распределённый Token Bucket (имитация)

Вариант	Базовые параметры	Расширенное задание
4	rate=8, cap=12	Приоритеты, 3D-график
5	rate=3, cap=25	Leaky Bucket сравнение, lock-free
6	rate=15, cap=50	Приоритеты, распределённый
7	rate=12, cap=18	Lock-free, 3D-график
8	rate=6, cap=10	Сравнение Fixed Window, приоритеты
9	rate=25, cap=40	Распределённый, Leaky Bucket
10	rate=18, cap=35	Lock-free, 3D-график
11	rate=9, cap=15	Приоритеты, lock-free
12	rate=4, cap=30	Распределённый с Redis (описание), Leaky Bucket
13	rate=7, cap=22	Lock-free, приоритеты
14	rate=11, cap=16	3D-график, сравнение алгоритмов
15	rate=30, cap=60	Комплексное расширенное исследование

---

6. Контрольные вопросы

1. Каков принцип работы Token Bucket? Как он отличается от Leaky Bucket?
2. Что определяет параметр capacity? Как он влияет на пропускную способность и задержки?
3. Почему в многопоточном окружении необходима синхронизация доступа к ведру? Какие примитивы используются?
4. Какие недостатки алгоритма Fixed Window Counter решает Token Bucket?

5. Как изменится поведение при rate=1 и capacity=1? Будет ли алгоритм эквивалентен одному разрешению в секунду строго?
6. Что произойдёт, если rate равно 0? Как интерпретировать такое ведро?
7. В чём разница между неблокирующей (allow_request) и блокирующей (acquire) версиями Token Bucket? Когда предпочтительнее блокировка?
8. Как можно реализовать приоритезацию трафика внутри Token Bucket?
9. Какие метрики важны при оценке качества rate limiter (кроме количества отказов)?
10. Какие готовые реализации Token Bucket используются в популярных библиотеках (Guava, Bucket4j, etc.)?

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: принцип работы, формулы пополнения, анализ параметров.
4. Программная реализация: код класса Timer, описание потокобезопасной версии.
5. Экспериментальная часть:
    - Таблицы результатов (разрешено/отклонено, фактический rate).
    - Графики уровня токенов во времени, гистограммы длин пачек.
    - Оценка максимального burst.
    - (для среднего/повышенного) дополнительные графики и замеры.
6. Выводы.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно реализованный Token Bucket с постоянным временем.
- Проведён один эксперимент, вычислены метрики, построен график.
- Отчёт содержит минимальный набор разделов.

8.2. Оценка «хорошо» (средний уровень)

- Потокобезопасная реализация, многопоточный тест, исследование влияния capacity.
- Дополнительные визуализации.
- Аргументированные выводы.

8.3. Оценка «отлично» (повышенный уровень)

- Реализованы приоритеты, lock-free версия или распределённый bucket.
- Сравнение с другими алгоритмами.
- Глубокий анализ, трёхмерные графики.

Снижение оценки: нет синхронизации, неправильный расчёт refill, несоответствие результатов теории.

Лабораторная работа № 16

Реализация GCRA (Generic Cell Rate Algorithm)

---

1. Цель работы

Цель работы – изучить алгоритм ограничения частоты **GCRA (Generic Cell Rate Algorithm)**, известный также как Virtual Scheduling, реализовать его локальную версию с константной памятью $O(1)$, провести сравнительный анализ с Token Bucket по точности и потреблению памяти, а также разработать прототип распределённого rate limiter на основе Redis с атомарным обновлением состояния.

---

2. Задачи

1. Реализовать алгоритм GCRA с параметрами: интервал эмиссии T и допуск τ.
2. Реализовать логику проверки запроса allow_request(now_ts) с обновлением теоретического времени прибытия (TAT).
3. Провести нагрузочное тестирование с различными трафиками: строго периодическим, пачечным (burst) и смешанным.
4. Измерить точность соблюдения ограничения (отклонение от номинальной частоты) и максимальный зарегистрированный burst.
5. Сравнить GCRA с Token Bucket по потреблению памяти и поведению в граничных случаях.
6. Реализовать распределённый вариант с использованием Redis (Lua-сценарий для атомарного обновления TAT).
7. Оценить пропускную способность и накладные расходы распределённой версии.

---

3. Теоретическое обоснование

3.1. GCRA и его параметры

GCRA был разработан для контроля трафика в сетях ATM. Он оперирует двумя параметрами:

- **Интервал эмиссии** T (Emission Interval) – минимальное допустимое время между последовательными запросами. Обратная величина $1/T$ задаёт устойчивую скорость (sustainable rate).
- **Допуск** τ (Tolerance, CDVT) – допустимое отклонение от идеального расписания, позволяющее кратковременные всплески.

Алгоритм хранит единственную переменную – **Theoretical Arrival Time (TAT)**, которая указывает ожидаемое время поступления следующего запроса при строгом соблюдении контракта. Начальное значение TAT обычно устанавливается в текущее время первого запроса.

3.2. Правило проверки

При поступлении запроса в момент времени t_a:

1. Если $t_a \geq \text{TAT} - \tau$, запрос считается **соответствующим** (conforming). Тогда новое значение $\text{TAT} = \max(t_a, \text{TAT}) + T$.
2. Если $t_a < \text{TAT} - \tau$, запрос **не соответствует** (non-conforming) – он отклоняется, TAT не изменяется.

Данное правило обеспечивает, что в долгосрочном периоде частота не превышает $1/T$, а кратковременный всплеск не может превысить $M = 1 + \lfloor \tau/T \rfloor$ запросов без нарушения контракта (максимальный размер пачки).

3.3. Эквивалентность с Token Bucket

GCRA с параметрами (T, τ) математически эквивалентен Token Bucket с параметрами:

- $rate = 1/T$,
- $capacity = 1 + \tau/T$.

Однако GCRA требует хранения только одного числа (TAT), тогда как Token Bucket обычно хранит два: количество токенов и временную метку. С точки зрения реализации это даёт минимальный расход памяти и простоту атомарного обновления в распределённом хранилище.

3.4. Распределённый GCRA через Redis

Для применения в кластере серверов необходимо централизованное хранилище состояния. Redis позволяет хранить TAT (как строку/число) и выполнять атомарную проверку с обновлением при помощи **Lua-скрипта**:

lua

```
local key = KEYS[1]

local T = tonumber(ARGV[1])

local tau = tonumber(ARGV[2])

local now = tonumber(ARGV[3])


local tat = redis.call('GET', key)

if not tat then tat = now end
```

```
tat = tonumber(tat)

if now >= tat - tau then
   local new_tat = math.max(now, tat) + T
   redis.call('SET', key, new_tat)
   return 1       -- разрешён
else
   return 0       -- отклонён
end
```

Это гарантирует корректное лимитирование для множества независимых клиентов.

---

4. Пример практического выполнения

4.1. Реализация локального GCRA

```
python
import time

class GCRA:
   def __init__(self, T: float, tau: float):
      self.T = T         # интервал эмиссии
      self.tau = tau      # допуск
      self.tat = 0.0      # TAT

   def allow_request(self, now: float = None) -> bool:
      if now is None:
         now = time.monotonic()
      if now >= self.tat - self.tau:
         self.tat = max(now, self.tat) + self.T
         return True
      return False
```

4.2. Симуляция с управляемым временем

Для точного тестирования используем класс с модельным временем, как в ЛР15.

python

```
class SimulatedGCRA:
    def __init__(self, T, tau):
        self.T = T
        self.tau = tau
        self.tat = 0.0

    def allow_at(self, arrival_time):
        if arrival_time >= self.tat - self.tau:
            self.tat = max(arrival_time, self.tat) + self.T
            return True
        return False

def run_test(T, tau, arrival_times):
    limiter = SimulatedGCRA(T, tau)
    allowed = []
    for t in arrival_times:
        allowed.append(limiter.allow_at(t))
    return allowed
```

4.3. Сравнение с Token Bucket

Используем SimulatedTokenBucket из ЛР15 с параметрами rate = 1.0/T, capacity = 1 + tau/T.

python

```
class SimulatedTokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
```

```
        self.last_time = 0.0

    def allow_at(self, arrival_time):
        dt = arrival_time - self.last_time
        self.tokens = min(self.capacity, self.tokens + dt * self.rate)
        self.last_time = arrival_time
        if self.tokens >= 1.0:
            self.tokens -= 1.0
            return True
        return False
```

Сравним на одинаковых последовательностях.

```
T = 1.0      # 1 запрос в секунду
tau = 2.0    # допуск, burst = 1 + 2/1 = 3

arrivals = [0.0, 0.1, 0.2, 0.3, 0.4, 5.0, 5.1, 5.2]
gcra_allowed = run_test(T, tau, arrivals)
tb = SimulatedTokenBucket(rate=1/T, capacity=1+tau/T)
tb_allowed = [tb.allow_at(t) for t in arrivals]

print("GCRA:       ", gcra_allowed)
print("TokenBucket:", tb_allowed)
```

Результаты будут идентичны.

4.4. Измерение максимального burst и точности

Генерируем пачечный трафик и подсчитываем максимальное число последовательно разрешённых запросов, фактическую среднюю частоту за длительный интервал.

```
def max_burst(allowed):
    mx = cur = 0
```

```
    for a in allowed:
        if a: cur += 1; mx = max(mx, cur)
        else: cur = 0
    return mx

# Пример
arrivals = []
# 10 пачек по 10 запросов с интервалом 0.01, между пачками 5 сек
t = 0.0
for _ in range(10):
    for _ in range(10):
        arrivals.append(t)
        t += 0.01
    t += 5.0

allowed = run_test(1.0, 2.0, arrivals)
print(f"Burst: {max_burst(allowed)} (expected <= {1 + 2.0/1.0})")

accepted = sum(allowed)
duration = arrivals[-1]
actual_rate = accepted / duration if duration>0 else 0
print(f"Фактическая частота: {actual_rate:.3f}, номинальная: {1/T:.3f}")
```

4.5. Визуализация

Построим график принятых/отклонённых запросов и уровень TAT во времени.

```
python
import matplotlib.pyplot as plt

def plot_gcra(arrivals, allowed, T, tau):
    tat_vals = []
```

```
    tat = 0.0
    for ta, ok in zip(arrivals, allowed):
        if ok:
            tat = max(ta, tat) + T
        tat_vals.append(tat)
    fig, ax1 = plt.subplots(figsize=(10,4))
    colors = ['green' if a else 'red' for a in allowed]
    ax1.scatter(arrivals, [1]*len(arrivals), c=colors, marker='|', s=100)
    ax2 = ax1.twinx()
    ax2.plot(arrivals, tat_vals, 'b-', alpha=0.5, label='TAT')
    ax2.set_ylabel('TAT')
    ax1.set_yticks([])
    ax1.set_xlabel('Время')
    ax1.set_title('GCRA: зелёный — разрешён, красный — отклонён')
    plt.grid(True)
    plt.show()

plot_gcra(arrivals, allowed, T, tau)
```

4.6. Распределённый GCRA через Redis (прототип)

Покажем код с использованием redis-py и Lua.

```
python
import redis

r = redis.Redis(decode_responses=True)

LUA_SCRIPT = """
local key = KEYS[1]
local T = tonumber(ARGV[1])
local tau = tonumber(ARGV[2])
```

```
local now = tonumber(ARGV[3])

local tat = redis.call('GET', key)
if not tat then
  tat = now
else
  tat = tonumber(tat)
end

if now >= tat - tau then
  local new_tat = math.max(now, tat) + T
  redis.call('SET', key, new_tat)
  return 1
else
  return 0
end
"""

def distributed_allow(redis_conn, key, T, tau, now):
  result = redis_conn.eval(LUA_SCRIPT, 1, key, T, tau, now)
  return bool(result)
```

Замерим пропускную способность при последовательных запросах от одного клиента.

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать локальный GCRA, провести симуляцию с заданными T и τ, измерить фактическую среднюю частоту, максимальный burst и количество отклонённых запросов для указанного профиля. Построить график принятых/отклонённых запросов во времени.

Вариант	T (сек)	τ (сек)	Профиль запросов (описание)	Длина
1	0.5	1.0	Равномерный каждые 0.6 с	300
2	1.0	2.0	Пачки: 5 запросов через 0.01 с, пауза 4 с (10 пачек)	500
3	0.2	0.5	Смешанный: 70% заявок с интервалом 0.25 с, 30% через 0.05 с	600
4	2.0	3.0	Редкие большие пачки: 20 запросов за 0.02 с, простой 10 с	200
5	0.1	0.4	Равномерный (0.11 с)	1000
6	0.25	0.6	Пачки по 3 через 0.02 с, пауза 1 с	400
7	1.5	2.5	Смешанный	500
8	0.05	0.2	Высокочастотный равномерный (0.06 с)	2000
9	0.4	0.8	Пачки по 8 через 0.005 с, пауза 3 с	600
10	3.0	5.0	Очень большие пачки: 50 запросов за 0.1 с, простой 20 с	200
11	0.3	0.6	Равномерный (0.35 с)	700
12	1.2	2.0	Смешанный с переменной частотой	800
13	0.8	1.5	Пачки по 4, пауза 2 с	500
14	0.15	0.3	Равномерный (0.16 с)	900

Вариант	T (сек)	τ (сек)	Профиль запросов (описание)	Длина
15	0.6	1.2	Смешанный	750

Требования к отчёту: корректный код GCRA, таблица метрик, график, сопоставление теоретического burst с измеренным.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать сравнение GCRA с Token Bucket: построить на одном графике результаты для идентичных параметров (T, τ) и (rate, capacity). Показать их эквивалентность.
- Исследовать влияние параметра τ на среднюю задержку отклонённых запросов (время до первого последующего разрешённого). Построить график зависимости средней задержки от τ/T.
- Реализовать распределённый rate limit через Redis (Lua) и провести нагрузочное тестирование: 1000 последовательных запросов от одного клиента, замерить пропускную способность (req/sec) и сравнить с локальной версией.
- Построить график использования памяти (размер хранимого состояния) для GCRA и Token Bucket при большом числе независимых лимитеров (например, 100000 пользователей).

Вариант	Параметры базового уровня	Дополнительные задания
1	вариант 1	Сравнение с Token Bucket, Redis Lua
2	вариант 2	Влияние τ, распределённый
3	вариант 3	Память, сравнение с TB
4	вариант 4	Redis, задержки
5	вариант 5	Память, влияние τ
6	вариант 6	Распределённый, сравнение

Вариант	Параметры базового уровня	Дополнительные задания
7	вариант 7	Зависимость задержки от τ, Redis
8	вариант 8	Память, сравнение ТВ
9	вариант 9	Redis, анализ задержек
10	вариант 10	Память, распределённый
11	вариант 11	Влияние τ, Redis
12	вариант 12	Сравнение, задержки
13	вариант 13	Распределённый, память
14	вариант 14	Влияние τ, ТВ
15	вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **иерархический GCRA**: два уровня ограничения (общий для группы и персональный). Продемонстрировать корректное соблюдение обоих лимитов.
- Сравнить GCRA с **скользящим окном** (Sliding Window Log) по точности и потреблению памяти.
- Построить **тепловую карту** зависимости коэффициента использования пропускной способности от T и τ при заданном пачечном трафике.
- Реализовать версию GCRA с приоритетами: запросы разного класса имеют разные T и τ, но разделяют общий TAT (?) – или используют отдельные лимитеры с приоритетным вытеснением.
- Разработать **эмулятор распределённого GCRA без центрального хранилища**, используя консенсус между репликами (например, на основе CRDT-счётчика). Оценить расхождение.

Вариант	Базовые параметры	Расширенное задание
1	T=1, τ=2	Иерархический GCRA, сравнение со скользящим окном
2	T=0.5, τ=1.5	Тепловая карта, эмуляция распределённого без Redis
3	T=2, τ=4	Приоритетный GCRA, сравнение с Token Bucket
4	T=0.2, τ=0.8	Скользящее окно, тепловая карта
5	T=1.5, τ=3	Распределённый на CRDT, иерархический
6	T=0.1, τ=0.5	Тепловая карта, приоритеты
7	T=0.3, τ=1.0	Иерархический, сравнение с TB
8	T=2.5, τ=5	Скользящее окно, CRDT
9	T=0.4, τ=1.2	Приоритеты, тепловая карта
10	T=1.8, τ=3.5	Распределённый без блокировок, иерархический
11	T=0.05, τ=0.25	Тепловая карта, скользящее окно
12	T=3, τ=6	Приоритеты, CRDT
13	T=0.7, τ=2.0	Иерархический, тепловая карта
14	T=0.9, τ=2.2	Скользящее окно, приоритеты
15	T=1, τ=2	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. Что такое Theoretical Arrival Time (TAT) в GCRA? Каким образом он вычисляется и обновляется?
2. Как параметр τ связан с максимальным допустимым всплеском (burst)? Выведите формулу.
3. Почему GCRA считается алгоритмом с константной памятью? Сколько байт занимает его состояние?
4. В чём эквивалентность между GCRA и Token Bucket? При каких условиях они дают одинаковые результаты?
5. Какие преимущества даёт использование Lua-скриптов в Redis для реализации распределённого GCRA?
6. Что произойдёт, если TAT значительно отстанет от текущего времени (долгое отсутствие запросов)? Как алгоритм обрабатывает такую ситуацию?
7. Чем отличается поведение GCRA от Fixed Window Counter при запросах вблизи границы окна?
8. Каковы ограничения применения GCRA в распределённых системах без центрального координатора?
9. Как можно модифицировать GCRA для поддержки нескольких уровней приоритета?
10. Приведите примеры реального использования GCRA или его аналогов (ATM, API-шлюзы).

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание алгоритма GCRA, его параметров, связи с Token Bucket.
4. Программная реализация: код локального GCRA, листинг Lua-скрипта для Redis.
5. Экспериментальная часть:
    - Подтверждённая точность и максимальный burst.
    - Сравнительная таблица GCRA / Token Bucket.
    - (для среднего/повышенного уровня) графики задержек, тепловые карты, результаты распределённого теста.
6. Анализ результатов, выводы об эффективности и областях применения.

7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Локальный GCRA реализован верно, параметры интерпретированы правильно.
- Проведён симуляционный эксперимент, измерены метрики, построен график.
- Показана связь burst с τ/T.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализовано сравнение с Token Bucket, показана их эквивалентность.
- Реализован распределённый лимитер с Redis, проведён нагрузочный тест.
- Проанализировано влияние τ, построены графики.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Добавлены иерархические/приоритетные схемы, сравнение со скользящим окном или CRDT.
- Построены тепловые карты, всестороннее исследование.
- Выводы содержат обоснованные рекомендации по выбору алгоритма.

Снижение оценки: некорректное обновление TAT, отсутствие проверки граничных условий, непроведённый распределённый тест (на повышенном уровне), грубые ошибки в формулах.

Лабораторная работа № 17

Реализация Sliding Window Counter в Redis

---

1. Цель работы

Цель работы – изучить алгоритмы ограничения частоты запросов на основе скользящего окна (Sliding Window), реализовать их с использованием Redis в качестве атомарного хранилища, а также провести сравнительный анализ по точности, потреблению памяти и производительности с классическим подходом фиксированного окна.

---

2. Задачи

1. Реализовать rate limiter на базе **фиксированного окна** (Fixed Window Counter) с ключом, истекающим в конце интервала.
2. Реализовать rate limiter на базе **скользящего окна** через ZSET (Sorted Set) с временны́ми метками, а также атомарный Lua-скрипт для обновления состояния.
3. Написать Lua-скрипт, выполняющий атомарную очистку устаревших записей, добавление текущего запроса и проверку лимита.
4. Провести нагрузочное тестирование для обоих алгоритмов при равномерной и неравномерной (пачечной) нагрузке.
5. Измерить и сравнить:
    - Точность (процент ложно разрешённых запросов на границах окон);
    - Потребление памяти Redis (размер ключей, объём данных ZSET);
    - Производительность Redis (количество операций в секунду, RPS).
6. Построить графики сравнительной точности и заполнить таблицы сравнения.

---

3. Теоретическое обоснование

3.1. Фиксированное окно (Fixed Window Counter)

Идея проста: для каждого клиента (ключа) ведётся целочисленный счётчик в Redis с временем жизни (TTL), равным длине окна W. При поступлении запроса счётчик увеличивается командой INCR. Если после увеличения значение не превышает лимит N, запрос разрешён, иначе – отклонён. Если счётчик не существовал, Redis создаёт его со значением 1 и устанавливает TTL через EXPIRE. Это легко реализовать атомарно с помощью Lua-скрипта: проверка существования, инкремент, установка TTL при первом запросе.

Недостаток: на границе окна возможны всплески вдвое выше лимита (например, если лимит N=10 в минуту, то 10 запросов в последнюю секунду одного окна и 10 в первую секунду следующего окна – итого 20 за две секунды). Это явление называется «перелив окна».

3.2. Скользящее окно через журнал (Sliding Window Log)

Для устранения недостатка фиксированного окна применяют скользящее окно с хранением точных меток времени каждого запроса. В Redis для этого можно использовать **Sorted Set (ZSET)**, где счёт (score) – временная метка (миллисекунды UNIX), а элементы – уникальные идентификаторы запросов (например, UUID, чтобы предотвратить дубли). При поступлении запроса:

1. Удаляются все элементы с score меньше now - window_size (команда ZREMRANGEBYSCORE).
2. Добавляется текущий запрос с score = now и уникальным членом.
3. Получается текущее количество запросов в окне через ZCARD.
4. Если это число не превышает лимит, запрос разрешён.

Этот метод точен, но затратен по памяти при больших лимитах и высоком трафике.

3.3. Скользящее окно через приближение (Sliding Window Counter)

Существует аппроксимация: хранятся два счётчика – для текущего окна и предыдущего. Вес текущего счётчика зависит от доли прошедшего времени в окне. Это менее точно, но расходует константную память. В данной работе мы сосредоточимся на точном скользящем окне (ZSET) и сравним его с фиксированным окном.

3.4. Атомарность через Lua-скрипты

Для корректной работы под нагрузкой операции должны быть атомарны. Redis гарантирует атомарное выполнение Lua-скриптов. Поэтому весь цикл проверки и обновления для одного запроса упаковывается в один скрипт, предотвращая гонки между клиентами.

---

4. Пример практического выполнения

4.1. Подготовка Redis и Python

Используем библиотеку redis-py. Установим: pip install redis. Предполагается локально запущенный Redis на порту 6379.

python

```
import redis

import time, uuid, random

import matplotlib.pyplot as plt
```

4.2. Реализация фиксированного окна (Fixed Window Counter)

Пишем Lua-скрипт:

lua

```
-- fixed_window.lua
-- KEYS[1]: ключ счетчика
-- ARGV[1]: лимит
-- ARGV[2]: время жизни (в секундах)
local current = redis.call('INCR', KEYS[1])
if current == 1 then
   redis.call('EXPIRE', KEYS[1], ARGV[2])
end
if current <= tonumber(ARGV[1]) then
   return 1
else
   return 0
end
```

Класс на Python:

python

```
class FixedWindowRateLimiter:
   def __init__(self, redis_client, key_prefix='fw'):
      self.redis = redis_client
      self.key_prefix = key_prefix
      self.script = self.redis.register_script("""
         local current = redis.call('INCR', KEYS[1])
         if current == 1 then
            redis.call('EXPIRE', KEYS[1], ARGV[2])
         end
         if current <= tonumber(ARGV[1]) then
            return 1
```

```
        else
            return 0
        end
    """)

  def allow(self, user_id, limit, window_sec):
    key = f"{self.key_prefix}:{user_id}"
    result = self.script(keys=[key], args=[limit, window_sec])
    return bool(result)
```

4.3. Реализация скользящего окна (Sliding Window Log) через ZSET

Lua-скрипт:

```
lua
-- sliding_window.lua
-- KEYS[1]: ключ ZSET
-- ARGV[1]: текущая временная метка (мс)
-- ARGV[2]: размер окна (мс)
-- ARGV[3]: лимит
-- ARGV[4]: уникальный идентификатор запроса (например, UUID)
redis.call('ZREMRANGEBYSCORE', KEYS[1], 0, ARGV[1] - ARGV[2])
redis.call('ZADD', KEYS[1], ARGV[1], ARGV[4])
local count = redis.call('ZCARD', KEYS[1])
if count <= tonumber(ARGV[3]) then
  return 1
else
  redis.call('ZREM', KEYS[1], ARGV[4])  -- удаляем только что добавленный, так как лимит превышен
  return 0
end
```

Класс:

python

```
class SlidingWindowRateLimiter:
    def __init__(self, redis_client, key_prefix='sw'):
        self.redis = redis_client
        self.key_prefix = key_prefix
        self.script = self.redis.register_script("""
            redis.call('ZREMRANGEBYSCORE', KEYS[1], 0, ARGV[1] - ARGV[2])
            redis.call('ZADD', KEYS[1], ARGV[1], ARGV[4])
            local count = redis.call('ZCARD', KEYS[1])
            if count <= tonumber(ARGV[3]) then
                return 1
            else
                redis.call('ZREM', KEYS[1], ARGV[4])
                return 0
            end
        """)

    def allow(self, user_id, limit, window_ms, now_ms=None):
        if now_ms is None:
            now_ms = int(time.time() * 1000)
        key = f"{self.key_prefix}:{user_id}"
        request_id = str(uuid.uuid4())
        result = self.script(keys=[key], args=[now_ms, window_ms, limit, request_id])
        return bool(result)
```

4.4. Экспериментальная установка

Сгенерируем последовательности временны́х меток для равномерной и пачечной нагрузки.

python

```
def generate_uniform_load(total_requests, interval_ms):
```

```
    arrivals = [i * interval_ms for i in range(total_requests)]
    return arrivals


def generate_burst_load(bursts, requests_per_burst, burst_interval_ms, pause_between_bursts_ms):
    arrivals = []
    start = 0
    for _ in range(bursts):
        for j in range(requests_per_burst):
            arrivals.append(start + j * burst_interval_ms)
        start += requests_per_burst * burst_interval_ms + pause_between_bursts_ms
    return arrivals
```

Функция тестирования:

```
python
def test_limiter(limiter, allow_func, arrivals, user_id, limit, window_size, window_unit='s'):
    if window_unit == 's':
        window_ms = window_size * 1000
    else:
        window_ms = window_size
    allowed = 0
    denied = 0
    for t_ms in arrivals:
        if allow_func(user_id, limit, window_ms, now_ms=t_ms):
            allowed += 1
        else:
            denied += 1
    return allowed, denied
```

Но для точности сравнения с учетом времени, нужно учитывать, что Redis может не успевать быстро. Для теста используем симуляцию с модельным временем. Т.к. в Redis мы передаём now_ms как аргумент, то мы можем эмулировать любую последовательность

таймстэмпов, и Redis будет правильно удалять старые записи. Например, для равномерной нагрузки передаём последовательные таймстемпы с одинаковым интервалом.

Проведём тест с параметрами: лимит 10 запросов в секунду (window=1000ms, N=10), 100 запросов, равномерно каждые 50 мс (20 запросов в секунду, т.е. выше лимита, должно быть ~50 разрешено). Для пачек: 5 пачек по 20 запросов с интервалом 10 мс между запросами в пачке, пауза между пачками 2 секунды. Ожидаемое поведение: внутри пачки разрешается первые 10, остальные отклоняются, после паузы ведро снова наполнено? В скользящем окне лимит жёсткий: за последние 1000ms не более 10. В пачке первые 10 будут разрешены, затем в течение последующей секунды ещё запросы отклоняются, пока окно не сдвинется.

Замерим память: размер ключа через MEMORY USAGE key.

Производительность: количество разрешённых запросов в секунду, которые можно выдать через Redis локально.

4.5. Визуализация точности на границах

Построим графики для фиксированного окна, показывающие разрешённые/отклонённые запросы.

Для сравнения точности измерим максимальное количество запросов в скользящем окне произвольного размера. Для фиксированного окна это количество может превышать лимит на границах.

python

```
def measure_burst_over_limit(limiter, arrivals, user_id, limit, window_ms):
    # проходим и проверяем, сколько запросов за любой период окна (непрерывный)
    # для проверки точности: если лимит = 10 в окне 1000 мс, то нельзя найти подпоследовательность из 11 запросов за <=1000 мс, которые все разрешены.
    # Это можно проверить алгоритмически: двигаем окно и считаем разрешённые запросы.
    # Упростим: измерим максимальное количество последовательно разрешённых запросов за интервал окна.
    pass
```

Можно просто визуализировать: для фиксированного окна смоделируем поток, где запросы приходят равномерно по 20 в секунду, и период окна 1 сек. Тогда счётчик будет сбрасываться каждую секунду. Если мы начинаем ровно в 0, в первую секунду разрешены первые 10, потом 10 отклонения, затем в 1.0 счётчик обнуляется и следующие 10 разрешены. Но если запросы поступают с 0.0 до 2.0 с равномерным интервалом 50мс, то будет чередование разрешён/отклонён. Это покажет график.

python

```
import matplotlib.pyplot as plt


def plot_results(arrivals, results, title):
    colors = ['green' if r else 'red' for r in results]
    plt.figure(figsize=(12,2))
    plt.scatter(arrivals, [0]*len(arrivals), c=colors, marker='|', s=100)
    plt.title(title)
    plt.yticks([])
    plt.xlabel('Time (ms)')
    plt.show()
```

4.6. Lua-скрипт для атомарного обновления (приведён выше)

4.7. Сравнение памяти и производительности

Используем timeit для измерения времени выполнения скрипта при множестве запросов.

```
python
def benchmark(limiter, user_id, limit, window_size, num_requests):
    start = time.time()
    for _ in range(num_requests):
        limiter.allow(user_id, limit, window_size)
    elapsed = time.time() - start
    return num_requests / elapsed
```

Замер памяти:

```
python
mem_bytes = r.memory_usage(key)
```

---

5. Задания для индивидуального выполнения

Номер варианта определяется по формуле (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать фиксированное окно и скользящее окно (ZSET) в Redis. Провести эксперимент с заданными параметрами: лимит, окно, нагрузка (равномерная или пачечная согласно варианту). Измерить общее число разрешённых/отклонённых запросов, максимальное

количество разрешённых за период окна (проверить на границе), оценить память, занимаемую одним ключом. Построить график результатов во времени для одного сценария (например, 200 запросов).

Вариант	Лимит (N)	Окно (W, сек)	Нагрузка	Длина трека
1	5	1	Равномерная, 10 запросов в сек	200 запросов
2	10	2	Равномерная, 8 запросов в сек	300
3	20	1	Пачки: 3 пачки по 30 запросов с интервалом 5 мс, пауза 3 с	300
4	15	3	Равномерная, 6 запросов в сек	400
5	8	1	Пачки: 5 пачек по 12 запросов с интервалом 1 мс, пауза 2 с	300
6	25	5	Равномерная, 7 запросов в сек	500
7	12	2	Пачки: 4 пачки по 20 запросов с интервалом 2 мс, пауза 4 с	300
8	6	1	Равномерная, 12 запросов в сек	200
9	30	3	Равномерная, 15 запросов в сек	600
10	18	2	Пачки: 2 пачки по 50 запросов с интервалом 0.5 мс, пауза 10 с	200
11	9	1	Равномерная, 10 запросов в сек	250
12	14	4	Пачки: 6 пачек по 10 запросов с интервалом 3 мс, пауза 3 с	300

Вариант	Лимит (N)	Окно (W, сек)	Нагрузка	Длина трека
13	22	2	Равномерная, 12 запросов в сек	400
14	16	1	Пачки: 3 пачки по 25 запросов с интервалом 2 мс, пауза 4 с	250
15	11	3	Равномерная, 5 запросов в сек	350

Требования к отчёту: корректные Lua-скрипты и Python-обёртки; таблица с результатами (разрешено, отклонено, память на ключ); график разрешённых/отклонённых запросов во времени для одного сценария; выводы о точности.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Реализовать также **приближённый скользящий счётчик** (Sliding Window Counter на основе двух окон) и сравнить его точность с точным скользящим окном.
- Провести нагрузочный тест для всех трёх методов, измеряя среднюю задержку allow (время выполнения скрипта + сетевая задержка) при 1000 запросах.
- Построить гистограмму потребления памяти для 1000 различных пользователей (ключей) при одинаковой нагрузке и сравнить.
- Исследовать, как лимит и размер окна влияют на разницу в точности между фиксированным и скользящим окном. Построить график зависимости максимального зафиксированного превышения лимита от отношения N / W.

Вариант	Параметры базового уровня	Дополнительные задания
1	вариант 1	Приближённый счётчик, гистограмма памяти
2	вариант 2	Нагрузочный тест, анализ превышения лимита
3	вариант 3	Задержки, сравнение точности
4	вариант 4	Гистограмма памяти, зависимость превышения от N/W

Вариант	Параметры базового уровня	Дополнительные задания
5	вариант 5	Приближённый счётчик, нагрузочный тест
6	вариант 6	Задержки, память
7	вариант 7	Приближённый метод, анализ граничных случаев
8	вариант 8	Нагрузочный тест, гистограмма
9	вариант 9	Задержки, приближённый счётчик
10	вариант 10	Память, зависимость превышения
11	вариант 11	Приближённый, нагрузочный тест
12	вариант 12	Задержки, сравнение
13	вариант 13	Гистограмма, приближённый
14	вариант 14	Нагрузочный, анализ превышения
15	вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **атомарный скользящий оконный лимитер с весами** (weighted requests, где каждый запрос может потреблять разное количество «единиц» лимита). Продемонстрировать на примере API, где GET=1, POST=3.
- Сравнить точность скользящего окна ZSET с альтернативным подходом: **Redis Streams** или **Redis TimeSeries**. Оценить удобство и производительность.
- Реализовать **распределённый rate limit** с координированным окном: несколько процессов, имея общий лимит, могут распределять его между собой через Redis.

- Построить **трёхмерный график** зависимости процента ложных разрешений для фиксированного окна от размера окна и интенсивности трафика.
- Разработать эмулятор **неравномерного трафика** с внезапными всплесками и измерить время восстановления лимита после всплеска (settling time).

Вариант	Базовые параметры	Расширенное задание
1	N=10, W=1s	Взвешенные запросы, сравнение с Redis Streams
2	N=20, W=2s	Трёхмерный график, распределённый лимит
3	N=15, W=3s	Redis TimeSeries, восстановление после всплеска
4	N=5, W=1s	Взвешенные запросы, трёхмерный график
5	N=25, W=5s	Распределённый, Streams
6	N=12, W=1s	Трёхмерный график, неравномерный трафик
7	N=30, W=2s	Взвешенные, восстановление
8	N=8, W=1s	Streams, распределённый
9	N=18, W=3s	Трёхмерный график, взвешенные
10	N=9, W=2s	Redis TimeSeries, неравномерный трафик
11	N=22, W=4s	Распределённый, взвешенные
12	N=14, W=1s	Трёхмерный, Streams
13	N=16, W=2s	Взвешенные, распределённый
14	N=11, W=3s	Неравномерный трафик, восстановление

Вариант	Базовые параметры	Расширенное задание
15	N=10, W=1s	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. Чем отличается фиксированное окно от скользящего окна в контексте rate limiting?
2. Как работает алгоритм скользящего окна на основе ZSET? Какие команды Redis используются?
3. Почему фиксированное окно допускает превышение лимита на границах интервалов?
4. Как обеспечивается атомарность проверки и обновления в Lua-скрипте? Что произошло бы без Lua-скрипта?
5. Какие факторы влияют на потребление памяти Redis при использовании ZSET для скользящего окна?
6. Что такое «приближённый скользящий счётчик» (sliding window counter с двумя окнами)? В чём его преимущества и недостатки?
7. Как организовать ограничение частоты с разными весами запросов в скользящем окне?
8. Какие альтернативы Redis существуют для распределённого rate limiting? Сравните их по CAP-теореме.
9. Каким образом можно построить график зависимости максимального зарегистрированного превышения лимита от параметров окна?
10. Как применить rate limiter в реальном API-шлюзе? Какие дополнительные аспекты необходимо учесть (напр., идентификация клиента, отказоустойчивость)?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание алгоритмов, их достоинства и недостатки, обоснование выбора Redis и Lua.
4. Программная реализация: листинги Lua-скриптов, код классов на Python.

5. Экспериментальная часть:
    - Таблицы с результатами точности и памяти для всех методов.
    - Графики разрешённых/отклонённых запросов.
    - Сравнительные диаграммы памяти и RPS.
    - (для среднего/повышенного уровня) дополнительные графики, анализ превышений, результаты нагрузочного тестирования.
6. Выводы: какой алгоритм предпочтительнее в зависимости от требований к точности, объёма трафика и доступной памяти.
7. Приложение с полным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно работают оба алгоритма (фиксированный, скользящий ZSET) с Lua-скриптами.
- Проведён сравнительный эксперимент, измерены память и точность.
- Построен график временно́го хода принятия решений.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Добавлен приближённый счётчик, проведён нагрузочный тест с замером задержек.
- Построены гистограммы памяти, исследована зависимость превышения лимита от параметров.
- Выводы аргументированы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы взвешенные запросы, альтернативные хранилища (Streams/TimeSeries), распределённый лимит или детальный трёхмерный анализ.
- Продемонстрировано глубокое понимание темы, даны практические рекомендации.

Снижение оценки: некорректная работа Lua-скриптов, отсутствие очистки старых записей в ZSET, неверный подсчёт лимита, отсутствие атомарности.

Лабораторная работа № 18

Сравнительный анализ алгоритмов rate limiting

---

1. Цель работы

Цель работы — провести комплексное сравнение трёх алгоритмов ограничения частоты запросов: **Token Bucket**, **GCRA (Generic Cell Rate Algorithm)** и **Sliding Window Log** по точности соблюдения лимита, задержке, потреблению памяти и поведению в различных сценариях неравномерной нагрузки. На основе экспериментальных данных выработать практические рекомендации по выбору алгоритма.

---

2. Задачи

1. Разработать единый программный интерфейс RateLimiter с методом, возвращающим не только разрешение/отказ, но и время ожидания до возможного разрешения.
2. Реализовать три алгоритма, соответствующих этому интерфейсу: TokenBucket, GCRA, SlidingWindowLog.
3. Создать генераторы временных трасс, имитирующих:
    - равномерную нагрузку с периодическими пиковыми всплесками;
    - постепенно возрастающую интенсивность;
    - паттерн «шумный сосед» (резкие доминирующие пачки).
4. Для каждого сценария измерить метрики:
    - отклонение фактической средней частоты от заданного лимита (error rate);
    - объём памяти, занимаемый состоянием алгоритма;
    - максимальную задержку, которую испытывает запрос до получения разрешения (блокирующая семантика).
5. Построить сравнительные графики зависимостей задержки от интенсивности, распределение отклонений.
6. Сформулировать рекомендации по выбору алгоритма для разных условий эксплуатации.

---

3. Теоретическое обоснование

3.1. Классификация алгоритмов rate limiting

Ограничение частоты запросов решает задачу пропуска не более R запросов в единицу времени, допуская при этом кратковременные всплески (burst) до некоторого предела B. Различают:

- **Token Bucket** — модель ведра с токенами, пополняемого с постоянной скоростью r. Объём ведра b задаёт максимальный burst. Запрос потребляет один токен; при нехватке — отказ или ожидание.
- **GCRA** — алгоритм планирования на основе Теоретического Времени Прибытия (TAT). Эквивалентен Token Bucket при $r = 1/T$, $b = 1 + \tau/T$. Хранит только одну временную метку, что даёт константную память.
- **Sliding Window Log** — точное отслеживание временны́х меток каждого запроса в окне W. Лимит N — максимальное число запросов в окне. Обеспечивает строжайшее соблюдение лимита ценой памяти $O(N)$.

3.2. Метрики качества

- **Точность ограничения** — относительное отклонение реализованной частоты $\hat{R}$ от заданного лимита R_{max}: err $=\mid \hat{R} - R_{\text{max}} \mid / R_{\text{max}}$.
- **Задержка (delay)** — время, которое запрос должен ожидать разрешения, если он не может быть обслужен немедленно. Максимальная задержка важна для чувствительных к латентности сервисов.
- **Потребление памяти** — объём данных, необходимый для хранения состояния лимитера. Token Bucket и GCRA используют два числа и метку времени (O(1)), Sliding Window Log — до N записей (O(N)).
- **Устойчивость к пачкам (burst tolerance)** — способность пропускать короткие очереди запросов без отказов в пределах B.

3.3. Блокирующая семантика

В реальных системах запросы, не прошедшие лимитер, обычно не отбрасываются, а ставятся в очередь ожидания. Для единообразного анализа мы реализуем метод acquire(now_ts), который возвращает кортеж (allowed, wait_time), где wait_time — время в секундах до момента, когда запрос будет гарантированно разрешён (при условии отсутствия новых запросов).

---

4. Пример практического выполнения

4.1. Единый интерфейс

python

```
import abc
```

```
class RateLimiter(abc.ABC):
    @abc.abstractmethod
    def acquire(self, now: float) -> (bool, float):
        """
        Возвращает (разрешён_ли, время_ожидания_до_разрешения_сек).
        Если запрос разрешён, wait_time == 0.0.
        """
        pass
```

4.2. Реализация Token Bucket с блокирующей задержкой

```
python
class TokenBucket(RateLimiter):
    def __init__(self, rate: float, capacity: float):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self.last_refill = 0.0

    def acquire(self, now: float) -> (bool, float):
        dt = now - self.last_refill
        self.tokens = min(self.capacity, self.tokens + dt * self.rate)
        self.last_refill = now
        if self.tokens >= 1.0:
            self.tokens -= 1.0
            return True, 0.0
        else:
            # потребуется ждать, пока накопится 1 токен
            wait = (1.0 - self.tokens) / self.rate
            return False, wait
```

4.3. Реализация GCRA

```
python
class GCRA(RateLimiter):
    def __init__(self, T: float, tau: float):
        self.T = T          # интервал эмиссии
        self.tau = tau      # допуск
        self.tat = 0.0      # TAT

    def acquire(self, now: float) -> (bool, float):
        if now >= self.tat - self.tau:
            self.tat = max(now, self.tat) + self.T
            return True, 0.0
        else:
            wait = (self.tat - self.tau) - now
            return False, wait
```

Соответствие параметров с Token Bucket: T = 1/rate, tau = (capacity - 1) * T.

4.4. Реализация Sliding Window Log

```
python
from collections import deque

class SlidingWindowLog(RateLimiter):
    def __init__(self, window_size: float, max_requests: int):
        self.window_size = window_size
        self.max_requests = max_requests
        self.log = deque()

    def _clean(self, now: float):
        while self.log and self.log[0] <= now - self.window_size:
            self.log.popleft()
```

```
    def acquire(self, now: float) -> (bool, float):
        self._clean(now)
        if len(self.log) < self.max_requests:
            self.log.append(now)
            return True, 0.0
        else:
            # время, когда старейший запрос выйдет из окна
            oldest = self.log[0]
            wait = oldest + self.window_size - now
            return False, wait
```

Скорость здесь max_requests / window_size, но burst фактически равен max_requests, так как в окне может находиться не более max_requests запросов одновременно, и при простое окно очищается.

4.5. Генераторы сценариев

```
python

import random


def uniform_with_bursts(total_time, base_interval, burst_size, burst_interval, burst_duration):
    """Равномерный поток + пиковые пачки: каждые burst_interval секунд в течение burst_duration
       запросы поступают каждые base_interval/burst_size секунд (очень частые)."""
    arrivals = []
    t = 0.0
    while t < total_time:
        arrivals.append(t)
        if (t // burst_interval) % 2 == 0 and (t % burst_interval) < burst_duration:
            # внутри пачки
            t += base_interval / burst_size
        else:
            t += base_interval
```

```
    return arrivals


def ramping_load(total_time, start_interval, end_interval):
    """Постепенно возрастающая нагрузка: интервал линейно уменьшается."""
    arrivals = []
    t = 0.0
    while t < total_time:
        arrivals.append(t)
        progress = t / total_time
        interval = start_interval + (end_interval - start_interval) * progress
        t += interval
    return arrivals


def noisy_neighbor(total_time, normal_interval, burst_interval, burst_count):
    """Редкие экстремальные пачки от одного клиента."""
    arrivals = []
    t = 0.0
    next_burst = random.uniform(5, 15)
    while t < total_time:
        if t >= next_burst:
            # пачка
            for _ in range(burst_count):
                arrivals.append(t)
                t += burst_interval
            next_burst = t + random.uniform(5, 15)
        else:
            arrivals.append(t)
            t += normal_interval
    return arrivals
```

4.6. Измерение метрик

Функция симуляции, пропускающая трассу через лимитер и собирающая статистику.

python

```
def simulate(limiter: RateLimiter, arrivals):
    """Возвращает (количество разрешённых, суммарное время ожидания, максимальное время ожидания,
       массив (arrival_time, wait_time, accepted))."""
    accepted = 0
    total_wait = 0.0
    max_wait = 0.0
    results = []
    for t in arrivals:
        ok, wait = limiter.acquire(t)
        results.append((t, wait, ok))
        if ok:
            accepted += 1
        else:
            total_wait += wait
            max_wait = max(max_wait, wait)
    return accepted, total_wait, max_wait, results
```

Отклонение частоты: actual_rate = accepted / arrivals[-1], limit = 1.0 / T для GCRA или rate для Token Bucket, max_requests / window_size для Sliding Window. error = abs(actual_rate - limit) / limit.

Замер памяти: оцениваем размер внутреннего состояния через sys.getsizeof() или вручную.

python

```
import sys


def measure_memory(limiter):
    # суммарный размер основных атрибутов
    total = 0
```

```
    for attr in vars(limiter).values():
        total += sys.getsizeof(attr)
    return total
```

4.7. Визуализация задержек

```
python
import matplotlib.pyplot as plt


def plot_delays(arrivals, results, title):
    times = [r[0] for r in results]
    waits = [r[1] for r in results]
    accepted = [1 if r[2] else 0 for r in results]
    plt.figure(figsize=(10,4))
    plt.subplot(2,1,1)
    plt.plot(times, waits, alpha=0.7, label='Время ожидания')
    plt.ylabel('Ожидание (с)')
    plt.legend()
    plt.grid()
    plt.subplot(2,1,2)
    plt.scatter(times, accepted, s=2, c=['green' if a else 'red' for a in accepted])
    plt.ylabel('Принято (1) / Отклонено (0)')
    plt.xlabel('Время (с)')
    plt.suptitle(title)
    plt.tight_layout()
    plt.show()
```

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать все три алгоритма с единым интерфейсом. Провести симуляцию на **одном** из трёх сценариев (по варианту), измерить точность, максимальную задержку и потребление памяти. Построить график времен ожидания во времени (первые 500 запросов).

Вариант	Сценарий	Лимит (rate / окно)	Параметры алгоритмов (rate, cap / T, τ / W, N)
1	Равномерный + пики	5 запр./с, burst=8	TB: r=5, cap=8; GCRA: T=0.2, τ=1.4; SWL: W=1, N=8
2	Постепенно возрастающий	10 запр./с, burst=15	TB: r=10, cap=15; GCRA: T=0.1, τ=1.4; SWL: W=1.5, N=15
3	Шумный сосед	3 запр./с, burst=5	TB: r=3, cap=5; GCRA: T=1/3, τ=4/3; SWL: W=2, N=6
4	Равномерный + пики	20 запр./с, burst=30	TB: r=20, cap=30; GCRA: T=0.05, τ=1.45; SWL: W=1, N=30
5	Постепенно возрастающий	2 запр./с, burst=4	TB: r=2, cap=4; GCRA: T=0.5, τ=1.5; SWL: W=2, N=4
6	Шумный сосед	15 запр./с, burst=20	TB: r=15, cap=20; GCRA: T=1/15, τ=19/15; SWL: W=1, N=20
7	Равномерный + пики	8 запр./с, burst=10	TB: r=8, cap=10; GCRA: T=0.125, τ=1.125; SWL: W=1.25, N=10
8	Постепенно возрастающий	4 запр./с, burst=6	TB: r=4, cap=6; GCRA: T=0.25, τ=1.25; SWL: W=1.5, N=6
9	Шумный сосед	12 запр./с, burst=18	TB: r=12, cap=18; GCRA: T=1/12, τ=17/12; SWL: W=1.5, N=18
10	Равномерный + пики	6 запр./с, burst=9	TB: r=6, cap=9; GCRA: T=1/6, τ=8/6; SWL: W=1.5, N=9

Вариант	Сценарий	Лимит (rate / окно)	Параметры алгоритмов (rate, cap / T, τ / W, N)
11	Постепенно возрастающий	25 запр./с, burst=35	TB: r=25, cap=35; GCRA: T=0.04, τ=1.36; SWL: W=1, N=35
12	Шумный сосед	1 запр./с, burst=3	TB: r=1, cap=3; GCRA: T=1, τ=2; SWL: W=3, N=3
13	Равномерный + пики	7 запр./с, burst=11	TB: r=7, cap=11; GCRA: T=1/7, τ=10/7; SWL: W=1.57, N=11
14	Постепенно возрастающий	9 запр./с, burst=13	TB: r=9, cap=13; GCRA: T=1/9, τ=12/9; SWL: W=1.44, N=13
15	Шумный сосед	18 запр./с, burst=25	TB: r=18, cap=25; GCRA: T=1/18, τ=24/18; SWL: W=1.38, N=25

Требования к отчёту: работающие классы, таблица с метриками для всех трёх алгоритмов (точность, память, максимальная задержка), графики для одного сценария, выводы о преимуществах и недостатках.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Провести все три сценария из таблицы для одного варианта и сравнить алгоритмы по чувствительности к типу трафика.
- Исследовать влияние параметра burst (capacity) на максимальную задержку и точность при фиксированной средней скорости. Построить графики зависимости для двух типов всплесков.
- Реализовать виртуальную очередь: запросы, не прошедшие лимитер, помещаются в буфер и обслуживаются позже (в порядке поступления). Измерить среднее время ожидания для каждого алгоритма и сравнить.
- Оценить накладные расходы на вызов acquire (процессорное время) для каждого алгоритма при 1e6 вызовов, построить столбчатую диаграмму.

Вариант	Базовые параметры	Дополнительные задания
1	Вариант 1	Все сценарии, влияние burst, очередь
2	Вариант 2	Все сценарии, замер времени, графики burst
3	Вариант 3	Очередь, зависимость от burst
4	Вариант 4	Все сценарии, процессорное время
5	Вариант 5	Влияние burst, очередь
6	Вариант 6	Все сценарии, замеры производительности
7	Вариант 7	Очередь, burst анализ
8	Вариант 8	Все сценарии, влияние параметров
9	Вариант 9	Процессорное время, очередь
10	Вариант 10	Все сценарии, burst, замеры
11	Вариант 11	Очередь, время
12	Вариант 12	Все сценарии, анализ burst
13	Вариант 13	Процессорное время, очередь
14	Вариант 14	Все сценарии, burst
15	Вариант 15	Комплексное сравнение

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать алгоритм **Leaky Bucket** и сравнить с Token Bucket по задержкам и сглаживанию.
- Построить тепловую карту зависимости максимальной задержки от параметров (rate и burst) для разных сценариев.
- Добавить приоритезацию запросов: реализовать две очереди с разными весами, исследовать, какой алгоритм обеспечивает лучшую изоляцию.
- Реализовать симуляцию с несколькими независимыми лимитерами, работающими параллельно (пул потоков), и измерить накладные расходы на синхронизацию.
- Сравнить с алгоритмом **Adaptive Rate Limiting** (на основе скользящего окна с автоматической подстройкой лимита по обратной связи).

Вариант	Базовые параметры	Расширенное задание
1	rate=5, burst=8	Leaky Bucket, тепловая карта
2	rate=10, burst=15	Приоритеты, адаптивный лимит
3	rate=3, burst=5	Параллельные лимитеры, тепловая карта
4	rate=20, burst=30	Leaky Bucket, приоритеты
5	rate=2, burst=4	Адаптивный rate limiting, сравнение
6	rate=15, burst=20	Тепловая карта, многопоточность
7	rate=8, burst=10	Leaky Bucket, приоритеты
8	rate=4, burst=6	Адаптивный, тепловая
9	rate=12, burst=18	Параллельные лимитеры, Leaky Bucket
10	rate=6, burst=9	Тепловая карта, приоритеты
11	rate=25, burst=35	Адаптивный, многопоточность

Вариант	Базовые параметры	Расширенное задание
12	rate=1, burst=3	Leaky Bucket, сравнение
13	rate=7, burst=11	Тепловая карта, адаптивный
14	rate=9, burst=13	Приоритеты, параллельные
15	rate=18, burst=25	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Какие метрики наиболее важны при выборе алгоритма rate limiting для высоконагруженного API?
2. Объясните, почему GCRA и Token Bucket эквивалентны при правильном выборе параметров, но имеют разную сложность реализации.
3. В чём принципиальное ограничение Sliding Window Log с точки зрения масштабируемости?
4. Как параметр burst (capacity) влияет на задержку и пропускную способность при пиковых нагрузках?
5. Почему фиксированное окно плохо справляется с трафиком на границах окна, а скользящее окно решает эту проблему?
6. Какие преимущества даёт блокирующая семантика (ожидание вместо отказа) в микросервисной архитектуре?
7. Что произойдёт, если время TAT в GCRA существенно отстанет от текущего времени?
8. Опишите, как можно реализовать «честный» (fair) rate limiting для нескольких клиентов с общей ёмкостью.
9. Какие структуры данных в Python обеспечивают быстрый доступ к старейшему элементу для Sliding Window Log?
10. Приведите примеры промышленных систем, использующих Token Bucket (или GCRA), и объясните их выбор.

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание алгоритмов, их свойств и сравнительных критериев.
4. Программная реализация: общий интерфейс, код трёх классов.
5. Экспериментальная часть:
    - Таблицы метрик (точность, память, максимальная задержка) для каждого сценария и алгоритма.
    - Графики задержек во времени.
    - Сравнительные диаграммы.
    - (для среднего/повышенного уровня) дополнительные графики влияния параметров, результаты моделирования очереди, тепловые карты.
6. Выводы и рекомендации по выбору алгоритма.
7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализованы все три алгоритма с единым интерфейсом.
- Проведён один сценарий, измерены основные метрики.
- Построен график, показана разница в поведении.
- Отчёт оформлен, выводы присутствуют.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Протестированы все три сценария, исследовано влияние burst, реализована очередь.
- Проведён бенчмарк производительности, даны развёрнутые рекомендации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Добавлены дополнительные алгоритмы (Leaky Bucket, адаптивный), многопоточные тесты, тепловые карты.
- Выводы подкреплены глубоким анализом, представлены практические рекомендации для инженерных решений.

Снижение оценки: некорректная реализация алгоритмов, отсутствие единого интерфейса, неправильный расчёт задержек, недостаточная визуализация.

Лабораторная работа № 19

Реализация Quorum-алгоритма для распределённого хранилища

---

1. Цель работы

Цель работы – изучить принципы построения отказоустойчивых распределённых хранилищ на основе кворумных алгоритмов, реализовать симулятор системы с настраиваемыми параметрами записи и чтения, экспериментально исследовать влияние отказов узлов на доступность и задержки, а также пронаблюдать проявление компромиссов, описываемых CAP-теоремой.

---

2. Задачи

1. Реализовать симулятор распределённого хранилища ключ-значение, состоящего из N узлов ($N = 3,5,7$).
2. Ввести понятие версий (меток времени) для разрешения конфликтов при чтении.
3. Реализовать операции write(key, value) и read(key) с использованием кворумов записи w и чтения r, удовлетворяющих условию $r + w > N$.
4. Обеспечить в симуляции возможность имитации отказов заданного числа узлов (1, 2, до $\lfloor (N-1)/2 \rfloor$).
5. Измерить доступность (процент успешных операций) и задержку (время ожидания ответа от достаточного числа узлов) при различных конфигурациях кворумов и количестве отказов.
6. Построить графики зависимости доступности и задержки от параметров кворума, демонстрирующие компромисс между согласованностью и доступностью.
7. Сравнить поведение системы при различных стратегиях разрешения конфликтов (по максимальной версии, по последнему временно́му штампу).

---

3. Теоретическое обоснование

3.1. Кворумные протоколы в распределённых хранилищах

Распределённое хранилище реплицирует данные на несколько узлов для повышения отказоустойчивости. Для обеспечения согласованности без использования единственного координатора применяются кворумные протоколы. Каждый узел хранит пару (value, timestamp). Для выполнения операции записи клиент должен получить подтверждение от **не менее чем** w **узлов**, для чтения – от **не менее чем** r **узлов**. Чтобы гарантировать, что операция чтения увидит самую свежую запись, накладывается условие:

$$r + w > N.$$

Действительно, любые два множества размера r и w пересекаются хотя бы по одному узлу, поэтому свежая версия, записанная хотя бы на w узлов, будет обнаружена при чтении с r узлов.

3.2. Разрешение конфликтов по временно́й метке

Каждая запись снабжается монотонно возрастающей меткой времени (timestamp). Операция записи:

- Клиент генерирует новую метку (например, на основе локальных часов + уникального идентификатора для упорядочения).
- Рассылает запись w случайным (или всем) узлам.
- Считает операцию успешной, если получил подтверждение от $\geq w$ узлов.

Операция чтения:

- Клиент опрашивает r узлов (обычно также случайно) и собирает ответы.
- Среди полученных версий выбирает ту, что имеет наибольшую метку времени.
- Иногда для самовосстановления клиент может выполнить «read repair»: обновить узлы с устаревшими версиями до самой свежей.

Условие $r + w > N$ гарантирует, что среди r ответов окажется хотя бы один узел с последней записанной версией.

3.3. Влияние отказов и CAP-компромисс

При отказе части узлов количество доступных узлов становится $N_{\text{up}} < N$. Для выполнения операции требуется, чтобы $w \leq N_{\text{up}}$ (для записи) и $r \leq N_{\text{up}}$ (для чтения). Иначе операция не может быть выполнена, что снижает доступность. Чем больше w и r, тем выше согласованность (пересечение гарантировано даже при меньшем числе узлов), но ниже доступность при отказах. Этот компромисс между **доступностью** и **согласованностью** является иллюстрацией CAP-теоремы для настраиваемых кворумов.

3.4. Метрики

- **Доступность (availability)** – доля успешно выполненных операций (write + read) от общего числа при заданном проценте отказавших узлов.
- **Задержка (latency)** – время от отправки запроса до получения достаточного числа подтверждений. В симуляторе можно моделировать фиксированную или случайную задержку передачи сообщения.
- **Консистентность** – число устаревших чтений (если читается не последняя версия). При соблюдении $r + w > N$ и корректной обработке меток консистентность строгая.

---

4. Пример практического выполнения

4.1. Модель узла и системы

python

```
import random
import time
from collections import defaultdict

class StorageNode:
    def __init__(self, node_id):
        self.node_id = node_id
        self.store = {}          # key -> (value, timestamp)
        self.online = True

    def put(self, key, value, timestamp):
        if not self.online:
            return False
        if key not in self.store or timestamp > self.store[key][1]:
            self.store[key] = (value, timestamp)
        return True

    def get(self, key):
        if not self.online:
            return None
        return self.store.get(key)
```

4.2. Симулятор распределённого хранилища

python

```
class QuorumStore:
    def __init__(self, N, w, r, node_delay=0.001):
```

```
    self.N = N
    self.w = w
    self.r = r
    self.nodes = [StorageNode(i) for i in range(N)]
    self.node_delay = node_delay  # базовая задержка передачи

def _send_req(self, node, func, *args):
    """Имитация отправки запроса на узел с задержкой."""
    if not node.online:
        return False, None
    # симуляция задержки
    time.sleep(self.node_delay * random.uniform(0.8, 1.2))
    return True, func(*args)

def write(self, key, value):
    timestamp = time.time_ns()  # упрощённый timestamp
    # Выбираем все узлы (можно случайно, но для простоты – все)
    confirmations = 0
    for node in self.nodes:
        ok, _ = self._send_req(node, node.put, key, value, timestamp)
        if ok:
            confirmations += 1
            if confirmations >= self.w:
                return True, timestamp
    return False, None

def read(self, key):
    responses = []
    for node in self.nodes:
```

```
            ok, resp = self._send_req(node, node.get, key)
            if ok and resp is not None:
                responses.append(resp)
                if len(responses) >= self.r:
                    break
        if not responses:
            return False, None
        # выбираем версию с максимальным timestamp
        latest = max(responses, key=lambda x: x[1])
        return True, latest[0]
```

4.3. Симуляция отказов и замер метрик

Отключаем случайные узлы, выполняем серии запросов.

python

```
def run_experiment(N, w, r, failure_count, ops=1000):
    store = QuorumStore(N, w, r)
    # Отключаем failure_count случайных узлов
    offline_nodes = random.sample(range(N), failure_count)
    for idx in offline_nodes:
        store.nodes[idx].online = False

    success_writes = 0
    success_reads = 0
    total_latency = 0.0

    keys_written = []
    for _ in range(ops // 2):
        key = f"key_{random.randint(1, 100)}"
        value = f"val_{random.randint(1, 1000)}"
        start = time.perf_counter()
```

```
            ok, _ = store.write(key, value)
            lat = time.perf_counter() - start
            if ok:
                success_writes += 1
                keys_written.append(key)
            total_latency += lat

        for _ in range(ops // 2):
            if not keys_written:
                break
            key = random.choice(keys_written)
            start = time.perf_counter()
            ok, _ = store.read(key)
            lat = time.perf_counter() - start
            if ok:
                success_reads += 1
            total_latency += lat

        availability = (success_writes + success_reads) / ops if ops > 0 else 0.0
        avg_latency = total_latency / ops if ops > 0 else 0.0
        return availability, avg_latency
```

4.4. Построение графиков доступности и задержки

```
python
import matplotlib.pyplot as plt

N = 7
w_values = [3, 4, 5]
r_values = [3, 4, 5]
failures = range(0, 4)  # 0..3 для N=7, floor((N-1)/2)=3
```

```
fig, axes = plt.subplots(1, 2, figsize=(12,5))


for w, r in [(4,4), (5,5), (6,3), (3,6)]:  # примеры пар
    avail = []
    lats = []
    for f in failures:
        a, l = run_experiment(N, w, r, f, ops=500)
        avail.append(a)
        lats.append(l * 1000)  # мс
    axes[0].plot(failures, avail, marker='o', label=f'w={w},r={r}')
    axes[1].plot(failures, lats, marker='s', label=f'w={w},r={r}')


axes[0].set_xlabel('Число отказавших узлов')
axes[0].set_ylabel('Доступность (доля успешных)')
axes[0].set_title('Зависимость доступности от отказов')
axes[0].legend()
axes[0].grid()


axes[1].set_xlabel('Число отказавших узлов')
axes[1].set_ylabel('Средняя задержка (мс)')
axes[1].set_title('Задержка при разных кворумах')
axes[1].legend()
axes[1].grid()


plt.tight_layout()
plt.show()
```

На графиках видно, что большие w и r обеспечивают более высокую согласованность, но при отказах доступность падает быстрее. Конфигурация с w=3, r=6 (при N=7) сохраняет доступность до трёх отказов, но требует больше ответов для чтения.

5. Задания для индивидуального выполнения

Номер варианта вычисляется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать симулятор с заданными N, w, r. Провести эксперимент с отключением 0, 1, ..., $\lfloor (N-1)/2 \rfloor$ узлов. Измерить процент успешных операций записи и чтения, среднюю задержку. Построить график зависимости доступности от числа отказов для одной конфигурации кворума.

Вариант	N	w	r	Примечание
1	3	2	2	r+w=4>3
2	5	3	3	r+w=6>5
3	7	4	4	r+w=8>7
4	3	2	2	
5	5	3	3	
6	7	5	5	
7	5	4	2	r+w=6>5
8	7	5	3	r+w=8>7
9	3	2	2	
10	7	6	2	r+w=8>7
11	5	2	4	r+w=6>5

Вариант	N	w	r	Примечание
12	3	2	1	r+w=3; нестрого >N? По условию r+w>N => 2+1=3 не >3. Примем как граничный случай для демонстрации возможной несогласованности.
13	7	3	5	r+w=8>7
14	5	4	3	r+w=7>5
15	3	1	3	r+w=4>3

Требования к отчёту: работающий код классов StorageNode и QuorumStore, корректная обработка версий, график доступности, список измеренных метрик, вывод о зависимости доступности от отказов.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Провести сравнение двух-трёх конфигураций кворума с одинаковым N (например, w=3,r=5 vs w=4,r=4 vs w=5,r=3 при N=7). Построить графики доступности и задержек на одном поле.
- Реализовать возможность динамического изменения w и r без остановки системы (с применением восстановления данных), оценить переходные процессы.
- Исследовать влияние размера кворума на устойчивость к показательным спадам задержки сети (добавить случайные длительные задержки на некоторых узлах). Измерить долю тайм-аутов.
- Реализовать «read repair»: после чтения, если обнаружена устаревшая версия, клиент обновляет её на отставших узлах. Сравнить среднее число устаревших версий с ремонтом и без.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Сравнение конфигураций, read repair
2	Вариант 2	Динамический кворум, задержки
3	Вариант 3	Сравнение w/r, тайм-ауты

Вариант	Параметры базового уровня	Дополнительные задания
4	Вариант 4	Read repair, сравнение
5	Вариант 5	Динамический кворум, графики
6	Вариант 6	Тайм-ауты, read repair
7	Вариант 7	Сравнение конфигураций
8	Вариант 8	Динамический, ремонт
9	Вариант 9	Тайм-ауты, задержки
10	Вариант 10	Read repair, сравнение
11	Вариант 11	Динамический кворум
12	Вариант 12	Сравнение w/r (осторожно, r+w=3 не >3)
13	Вариант 13	Read repair, тайм-ауты
14	Вариант 14	Динамический, задержки
15	Вариант 15	Все перечисленные задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **векторные часы** вместо простого timestamp для поддержки частичных записей в условиях разделения сети. Продемонстрировать разрешение конфликтов при слиянии.
- Исследовать сценарий «split-brain» (разделение сети на две части) и показать, что при $r + w > N$ либо чтение, либо запись блокируется, предотвращая несогласованность.
- Построить трёхмерный график зависимости доступности от w и r при фиксированном N и количестве отказов.

- Реализовать алгоритм **динамического кворума** (например, Hierarchical Quorum) и сравнить его пропускную способность с обычным голосованием большинства.
- Провести сравнение с консенсус-алгоритмом Raft на тех же условиях отказов (на уровне концептуального сравнения или упрощённой модели).

Вариант	Базовые параметры	Расширенное задание
1	N=7, w=4, r=4	Векторные часы, split-brain
2	N=5, w=3, r=3	Трёхмерный график, динамический кворум
3	N=7, w=5, r=5	Split-brain, сравнение с Raft
4	N=3, w=2, r=2	Векторные часы, трёхмерный
5	N=9 (доп. узлы)	Динамический кворум, иерархический
6	N=5, w=4, r=2	Векторные часы
7	N=7, w=3, r=6	Трёхмерный доступность
8	N=9, w=5, r=5	Split-brain, иерархия
9	N=5, w=2, r=4	Динамический кворум
10	N=7, w=6, r=2	Векторные часы, сравнение
11	N=5, w=3, r=4	Трёхмерный график
12	N=3, w=2, r=1	Анализ граничных условий, split-brain
13	N=7, w=4, r=5	Векторные часы, динамический
14	N=9, w=6, r=5	Трёхмерный, Raft

Вариант	Базовые параметры	Расширенное задание
15	N=7, w=4, r=4	Комбинация нескольких расширений

---

6. Контрольные вопросы

1. Почему условие $r + w > N$ гарантирует непротиворечивость при чтении?
2. Какой компромисс между доступностью и согласованностью проявляется при варьировании w и r?
3. Каким образом разрешаются конфликты версий при использовании временны́х меток? В чём недостатки такого подхода?
4. Что такое «read repair» и как он улучшает долговременную согласованность?
5. Как изменится доступность системы, если $w = N$ и $r = 1$? Как интерпретировать такую конфигурацию в терминах CAP?
6. Почему векторные часы точнее, чем простые timestamp, упорядочивают события в распределённой системе?
7. Объясните, что произойдёт при сетевом разделе (split-brain), если $r + w \leq N$.
8. Какие практические системы (DynamoDB, Cassandra, Riak) используют кворумные чтение и запись? Какие значения w и r в них обычно применяются?
9. Какие дополнительные механизмы необходимы для поддержания версионности при удалении данных (tombstones)?
10. Каковы ограничения использования кворумного подхода в глобально распределённых системах с высокой задержкой?

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание кворумного протокола, условий согласованности, CAP-теорема.
4. Программная реализация: архитектура классов, листинг ключевых методов.
5. Экспериментальная часть:

- Таблицы измеренных доступности и задержек для всех конфигураций.
- Графики зависимости доступности/задержки от числа отказов.
- (для среднего/повышенного уровня) дополнительные графики сравнения кворумов, влияние ремонта, трёхмерные визуализации.

6. Анализ результатов, интерпретация компромисса доступность-согласованность.
7. Выводы.
8. Приложение с полным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализованы узлы и кворумная логика с метками времени.
- Проведён эксперимент с одной конфигурацией, сняты метрики доступности и задержки.
- Построен график, демонстрирующий влияние отказов.
- Отчёт содержит все обязательные разделы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены требования базового уровня.
- Проведено сравнение минимум двух конфигураций кворума, реализован read repair и учтены тайм-ауты.
- Построены дополнительные графики, сделаны выводы о влиянии w/r на поведение системы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы векторные часы, динамические кворумы или анализ split-brain.
- Глубокий анализ, трёхмерные графики, сравнение с другими протоколами.
- Отчёт демонстрирует понимание фундаментальных компромиссов распределённых систем.

Снижение оценки: неработающий код, несоблюдение условия $r + w > N$, отсутствие разрешения конфликтов версий, недостаточная визуализация, неверная интерпретация CAP.

Лабораторная работа № 20

Реализация Vector Clocks для обнаружения конфликтов

---

1. Цель работы

Цель работы — изучить механизм **векторных часов** (Vector Clocks) для отслеживания причинно-следственных связей в распределённой системе, реализовать основные операции сравнения и слияния, применить их для обнаружения конфликтов при конкурентных обновлениях, а также исследовать проблему возрастания размера вектора и методы её решения.

---

2. Задачи

1. Реализовать класс VectorClock, хранящий словарь идентификаторов узлов и их логических счётчиков.
2. Реализовать методы:
    - increment(node_id) — увеличение счётчика указанного узла на единицу;
    - merge(other) — покомпонентный максимум с другим вектором;
    - compare(other) — возврат одного из отношений: EQUAL, BEFORE, AFTER, CONCURRENT.
3. Разработать симулятор распределённой системы из нескольких узлов, выполняющих обновления общего объекта с обменом векторными часами.
4. Научить симулятор обнаруживать конфликты (отношение CONCURRENT) и разрешать их с помощью пользовательской логики слияния версий.
5. Измерить динамику роста размера вектора (числа уникальных идентификаторов) при длительной работе и реализовать механизм сжатия путём удаления старых узлов.
6. Визуализировать причинно-следственный граф (DAG) событий версий объекта.

---

3. Теоретическое обоснование

3.1. Необходимость логического времени

В распределённых системах отсутствует единое глобальное время. Для упорядочения событий и обнаружения конфликтов параллельных обновлений используются логические часы. Простые счётчики Лампорта (скалярный счётчик) дают отношение «произошло до» (happened-before), но не позволяют выявить параллельность (concurrency). Векторные часы решают эту проблему.

3.2. Устройство векторных часов

Векторные часы представляют собой отображение идентификаторов узлов в натуральные числа:

$$V\text{:NodeId} \to \mathbb{N}_0.$$

Каждый узел хранит собственный вектор. При локальном событии узел инкрементирует только свой компонент: $V[i] += 1$. При передаче сообщения узел прикрепляет свой текущий вектор, а получатель сливает его со своим (merge), после чего фиксирует факт получения (инкрементирует свой компонент).

Операция merge(V_a, V_b):

$$\forall j, V[j] = \max(V_a[j], V_b[j]).$$

После слияния обычно инкрементируют собственный компонент получателя.

Операция сравнения compare(V_a, V_b):

- $V_a = V_b$ (EQUAL), если $\forall j\ V_a[j] = V_b[j]$;
- $V_a < V_b$ (BEFORE), если $\forall j\ V_a[j] \le V_b[j]$ и $\exists j\ V_a[j] < V_b[j]$;
- $V_a > V_b$ (AFTER), если $\forall j\ V_a[j] \ge V_b[j]$ и $\exists j\ V_a[j] > V_b[j]$;
- иначе — CONCURRENT (параллельные, потенциально конфликтующие).

3.3. Обнаружение и разрешение конфликтов версий

В распределённых хранилищах каждый объект ассоциируется с векторными часами. При одновременном обновлении с разных узлов без координации векторы оказываются несравнимыми (CONCURRENT), что сигнализирует о конфликте. Приложение должно разрешить конфликт, объединив состояния (например, слияние списков, применение CRDT). В учебной симуляции мы будем хранить список всех версий и при конфликте вручную объединять.

3.4. Проблема роста векторных часов

С течением времени и изменением состава узлов вектор может неограниченно расти. Для ограничения размера применяют технику **version vector pruning**: удаляют из вектора идентификаторы узлов, которые больше не активны, если их значения не влияют на сравнение с другими узлами, либо периодически пересчитывают часы. Более практичный подход — использование **Dotted Version Vectors** (иногда). В данной работе мы реализуем простой метод: удаление записей для узлов, чей счётчик равен нулю и которые не обновлялись давно, или сжатие с помощью операций слияния.

---

4. Пример практического выполнения

4.1. Базовый класс VectorClock

python

```
from enum import Enum


class Relation(Enum):
    EQUAL = 0
    BEFORE = 1
    AFTER = 2
    CONCURRENT = 3


class VectorClock:
    def __init__(self, clock=None):
        self.clock = dict(clock) if clock else {}

    def increment(self, node_id):
        self.clock[node_id] = self.clock.get(node_id, 0) + 1

    def merge(self, other):
        for node, cnt in other.clock.items():
            if node not in self.clock or cnt > self.clock[node]:
                self.clock[node] = cnt

    def compare(self, other):
        all_greater_equal = True
        all_less_equal = True
        diff_exists = False

        all_nodes = set(self.clock.keys()) | set(other.clock.keys())
        for node in all_nodes:
```

```
            a = self.clock.get(node, 0)
            b = other.clock.get(node, 0)
            if a < b:
                all_greater_equal = False
            if a > b:
                all_less_equal = False
            if a != b:
                diff_exists = True

        if all_greater_equal and all_less_equal and not diff_exists:
            return Relation.EQUAL
        if all_less_equal and diff_exists:
            return Relation.BEFORE   # self before other
        if all_greater_equal and diff_exists:
            return Relation.AFTER    # self after other
        return Relation.CONCURRENT
```

4.2. Симулятор распределённого обновления объекта

Создадим класс ReplicatedObject, который хранит значение и векторные часы. Узлы (процессы) будут отправлять друг другу обновления (имитация передачи сообщений). Реализуем функцию apply_update с логикой сравнения и слияния.

```
python
import random


class ReplicatedObject:
    def __init__(self, value, vclock=None):
        self.value = value
        self.vclock = vclock if vclock else VectorClock()


def simulate(num_nodes=3, num_events=20):
    # Каждый узел имеет копию объекта
```

```
replicas = [ReplicatedObject(value=f"init", vclock=VectorClock()) for _ in range(num_nodes)]
events_log = []  # (node_id, replica_index, action, value, vclock_copy)

for step in range(num_events):
    node = random.randint(0, num_nodes-1)
    # Локальное обновление
    replicas[node].vclock.increment(node)
    new_value = f"val_{step}_by_{node}"
    replicas[node].value = new_value
    events_log.append((node, "update", new_value, VectorClock(replicas[node].vclock.clock)))

    # Отправка случайному соседу (распространение)
    if random.random() < 0.7:
        receiver = random.choice([n for n in range(num_nodes) if n != node])
        # передаём сообщение с текущим состоянием отправителя
        sender_clock = replicas[node].vclock
        sender_value = replicas[node].value

        # Применение на получателе
        rep = replicas[receiver]
        # сравним часы
        relation = rep.vclock.compare(sender_clock)
        if relation == Relation.BEFORE:
            # Удалённое состояние новее
            rep.value = sender_value
            rep.vclock.merge(sender_clock)
            # локальное событие получения (инкремент)
            rep.vclock.increment(receiver)
```

```
            events_log.append((receiver, "receive_update", sender_value, VectorClock(rep.vclock.clock)))
        elif relation == Relation.AFTER:
            # Наше состояние новее, ничего не делаем, только счётчик получателя увеличим?
            # Просто фиксируем получение
            rep.vclock.increment(receiver)
            events_log.append((receiver, "receive_ignore", rep.value, VectorClock(rep.vclock.clock)))
        elif relation == Relation.CONCURRENT:
            # Конфликт: создаём новую версию путём слияния (например, объединяем строки)
            merged_value = f"{{ {rep.value} // {sender_value} }}"
            rep.value = merged_value
            # объединяем часы (merge) и инкремент получателя
            rep.vclock.merge(sender_clock)
            rep.vclock.increment(receiver)
            events_log.append((receiver, "conflict_merge", merged_value, VectorClock(rep.vclock.clock)))
        else:
            # EQUAL: ничего не делаем
            pass

    return replicas, events_log
```

4.3. Визуализация DAG событий

Используем библиотеку networkx для построения графа и matplotlib для отображения.

```
python
import networkx as nx
import matplotlib.pyplot as plt

def build_dag(events_log):
    G = nx.DiGraph()
```

```
    # Каждое событие — узел. Связь показывает happened-before.
    # Упрощённо: каждое событие свяжем с предыдущим событием того же узла (порядок процессов)
    last_per_node = {}
    for i, event in enumerate(events_log):
        node_id, action, value, vclock = event
        event_id = f"e{i}:{action}_{node_id}"
        G.add_node(event_id, label=f"{action}\n{value[:10]}...")
        # связь с предыдущим событием этого же процесса
        if node_id in last_per_node:
            G.add_edge(last_per_node[node_id], event_id)
        last_per_node[node_id] = event_id

        # Добавить связи, если событие было получено от другого узла? У нас уже есть порядок.
        # Можно для конфликтов/приёмов добавить ребро от отправителя к этому событию.
        # Существенно не будем усложнять, просто покажем цепочки.
    return G

# Рисование
def draw_dag(G):
    pos = nx.spring_layout(G, seed=42)
    labels = nx.get_node_attributes(G, 'label')
    plt.figure(figsize=(12,8))
    nx.draw(G, pos, with_labels=True, labels=labels, node_size=2000, node_color='lightblue', font_size=8)
    plt.title("Причинно-следственный граф версий")
    plt.show()
```

4.4. Измерение роста вектора и сжатие

Запустим длительную симуляцию с добавлением и удалением узлов. После определённого периода очистим векторы от узлов, чей счётчик равен нулю и которые не обновлялись.

python

```
def prune_vector_clock(vclock, active_nodes, max_size):
  # Удаляем записи для неактивных узлов, сохраняя последние max_size по значению счётчика
  sorted_items = sorted(vclock.clock.items(), key=lambda x: x[1], reverse=True)
  new_clock = {}
  for node, cnt in sorted_items:
    if node in active_nodes or len(new_clock) < max_size:
      new_clock[node] = cnt
  vclock.clock = new_clock
```

Измеряем длину словаря до и после сжатия.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать класс VectorClock с методами increment, merge, compare. Провести симуляцию работы **трёх узлов** в течение заданного числа событий, отслеживая логические часы и обнаруживая конфликты. Построить упрощённую диаграмму событий (порядок обновлений по процессам) в текстовом формате или таблицу, демонстрирующую несколько случаев CONCURRENT.

Вариант	Число узлов	Количество событий	Вероятность передачи (0..1)	Дополнительно
1	3	30	0.6	
2	3	40	0.5	
3	3	50	0.7	
4	3	25	0.8	
5	4	35	0.6	

Вариант	Число узлов	Количество событий	Вероятность передачи (0..1)	Дополнительно
6	4	45	0.5	
7	4	55	0.7	
8	5	30	0.6	
9	5	40	0.5	
10	3	60	0.4	
11	4	50	0.8	
12	3	20	0.9	
13	5	25	0.7	
14	4	30	0.6	
15	3	70	0.5	

Требования к отчёту: корректная реализация операций, примеры сравнений (EQUAL, BEFORE, AFTER, CONCURRENT) на тестовых векторах, результат симуляции с конфликтами, вывод числа конфликтов и размера векторов.

5.2. Средний уровень (на оценку «хорошо»)

В дополнение к базовому уровню:

- Реализовать механизм сжатия векторных часов: удалять записи для узлов, чей счётчик не изменялся более K событий, и при этом не нарушающий отношение happened-before с другими живыми узлами. Исследовать влияние сжатия на точность обнаружения конфликтов.
- Провести эксперимент с 10 узлами и 200 событиями, отследить рост среднего размера вектора без сжатия и со сжатием, построить сравнительный график.

- Реализовать функцию prune_dead_nodes(vclock, alive_nodes), которая удаляет компоненты, соответствующие «мёртвым» узлам. Показать, что после удаления вектор всё ещё корректно сравнивается с другими векторами живых узлов.
- Сравнить поведение векторных часов и простых скалярных часов Лампорта на предмет обнаружения конкуренции (показать случаи ложного BEFORE у Лампорта).

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Сжатие, сравнение с Лампортом
2	Вариант 2	График роста, prune
3	Вариант 3	Лампорт, сжатие
4	Вариант 4	Эксперимент 10 узлов, сжатие
5	Вариант 5	Сжатие, график
6	Вариант 6	Лампорт, prune
7	Вариант 7	Сжатие, сравнение
8	Вариант 8	10 узлов, график
9	Вариант 9	Лампорт, сжатие
10	Вариант 10	Prune, рост
11	Вариант 11	Лампорт, сжатие
12	Вариант 12	Сравнение, prune
13	Вариант 13	10 узлов, сжатие
14	Вариант 14	Лампорт

Вариант	Параметры базового уровня	Дополнительные задания
15	Вариант 15	Все доп. задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать прототип **распределённого хранилища ключ-значение** с версионированием на основе векторных часов. Каждая запись сопровождается вектором, а при чтении нескольких реплик разрешаются конфликты с помощью пользовательской функции (например, автоматическое объединение счётчиков).
- Реализовать **алгоритм динамического сжатия на основе dotted version vectors** (или «интервальных» векторных часов) и сравнить его эффективность с простым удалением старых узлов.
- Построить полноценную визуализацию DAG с помощью networkx, где рёбра точно отражают причинно-следственные связи (по процессам и по сообщениям). Выделить конфликтные узлы особым цветом.
- Сравнить накладные расходы векторных часов и физических меток времени (timestamp) по размеру данных и числу конфликтов при симуляции с рассинхронизацией часов.

Вариант	Базовые параметры	Расширенное задание
1	3 узла, 50 событий	Мини-хранилище, динамическое сжатие
2	5 узлов, 100 событий	DAG с networkx, сравнение с timestamp
3	4 узла, 80 событий	Хранилище с версиями, DAG
4	3 узла, 60 событий	Динамическое сжатие, networkx
5	5 узлов, 120 событий	Mini key-value, сравнение часов
6	4 узла, 70 событий	DAG, сжатие
7	3 узла, 90 событий	Хранилище, timestamp

Вариант	Базовые параметры	Расширенное задание
8	6 узлов, 150 событий	Networkx, динамическое сжатие
9	5 узлов, 130 событий	Mini-DB, DAG
10	4 узла, 100 событий	Сравнение с timestamp, сжатие
11	3 узла, 110 событий	DAG, хранилище
12	5 узлов, 140 событий	Динамическое сжатие, networkx
13	6 узлов, 160 событий	Mini-KV, DAG
14	3 узла, 80 событий	Сравнение Лампорта/векторов, сжатие
15	4 узла, 120 событий	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Чем векторные часы отличаются от скалярных часов Лампорта? Какую дополнительную возможность они предоставляют?
2. Что означают отношения EQUAL, BEFORE, AFTER и CONCURRENT при сравнении двух векторных часов?
3. Почему операция merge является идемпотентной и коммутативной?
4. Как рост числа узлов влияет на размер векторных часов? Предложите методы сжатия.
5. В чём состоит опасность удаления компонента «мёртвого» узла из векторных часов? Как гарантировать сохранение причинности?
6. Как векторные часы используются для обнаружения конфликтов в распределённых хранилищах (Dynamo, Riak)?
7. Что такое «dotted version vector» и как он решает проблему сжатия?
8. Как можно визуализировать причинно-следственные связи с помощью DAG? Какие программные библиотеки можно использовать?

9. В какой ситуации два события могут быть признаны конкурентными, даже если физически они произошли в разное время?
10. В чём преимущества и недостатки использования векторных часов по сравнение с централизованными блокировками для управления версиями?

---

7. Требования к отчёту

Отчёт оформляется согласно общим требованиям кафедры и должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: определение, свойства векторных часов, алгоритмы сравнения и слияния.
4. Программная реализация: класс VectorClock, код симуляции.
5. Экспериментальная часть:
    - Примеры тестирования отношений (в том числе конкурентных).
    - Результаты симуляции: количество конфликтов, рост размера векторов.
    - Графики роста размера векторов до/после сжатия (для среднего уровня).
    - Визуализация DAG (в текстовом или графическом виде, для повышенного уровня – с networkx).
6. Анализ результатов, обсуждение эффективности сжатия и накладных расходов.
7. Выводы.
8. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Класс VectorClock реализован корректно, операции merge и compare дают верные результаты.
- Проведена симуляция с несколькими узлами, продемонстрировано обнаружение конфликтов.
- Отчёт содержит все требуемые разделы, выводы по результатам.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены требования базового уровня.

- Реализован механизм сжатия векторных часов, исследовано его влияние на корректность.
- Проведено расширенное тестирование с 10 узлами, построен график роста размера.
- Выводы обоснованы, сравнение с Лампортом присутствует.

8.3. Оценка «отлично» (повышенный уровень)

- Выполнены все пункты среднего уровня.
- Разработан прототип хранилища с версионированием на основе векторных часов.
- Построена детальная визуализация DAG, исследованы продвинутые методы сжатия или сравнение с физическими метками.
- Отчёт демонстрирует глубокое понимание материала и содержит практические рекомендации.

Снижение оценки: ошибки в логике compare (неправильное определение BEFORE/CONCURRENT), отсутствие слияния, неработающее сжатие, недостаточная визуализация.

Лабораторная работа № 21

Реализация CRDT: G-Counter и PN-Counter

---

1. Цель работы

Цель работы — изучить принцип построения конфликтно-свободных реплицируемых типов данных (CRDT) на примере счётчиков G-Counter и PN-Counter, реализовать их программно, провести симуляцию параллельных обновлений и слияний без координации и продемонстрировать свойство строгой сходимости к одинаковому состоянию.

---

2. Задачи

1. Реализовать **G-Counter** (Grow-only Counter) — счётчик, основанный на массиве инкрементов по узлам, с операциями:
    - increment(node_id) — увеличить значение для заданного узла;
    - merge(other) — поэлементный максимум с другим счётчиком;
    - value — сумма всех элементов массива.
2. Реализовать **PN-Counter** (Positive-Negative Counter) на базе двух G-Counter: для положительных (P) и отрицательных (N) изменений.
3. Сравнить поведение CRDT-счётчиков с **наивным счётчиком** (простая репликация, где при конфликте теряются обновления).
4. Провести симуляцию параллельных изменений на N узлах без связи, затем выполнить попарные слияния и продемонстрировать сходимость всех реплик к одному значению.
5. Измерить количество операций слияния до достижения консенсуса и проанализировать влияние топологии связей.
6. Построить графики, показывающие сходимость значений на узлах с течением времени.

---

3. Теоретическое обоснование

3.1. CRDT и их классификация

Conflict-free Replicated Data Types (CRDT) — это структуры данных, реплицируемые между узлами распределённой системы, позволяющие выполнять локальные обновления без координации и гарантирующие сходимость всех реплик к единому состоянию после обмена операциями или состояниями.

Существует два подхода:

- **Operation-based CRDT**: узлы передают друг другу только операции; коммутативность операций гарантирует сходимость.
- **State-based CRDT (Convergent CRDT)**: узлы периодически обмениваются полным состоянием, при получении чужого состояния выполняется функция слияния merge, которая должна быть коммутативной, ассоциативной и идемпотентной, а состояния должны образовывать **полурешётку с операцией объединения** (join-semilattice).

3.2. G-Counter (Grow-only Counter)

G-Counter — State-based CRDT, моделирующий монотонно возрастающий счётчик. Каждому узлу в системе присваивается уникальный идентификатор. Состояние G-Counter — это отображение из идентификаторов узлов в натуральные числа:

$$\text{state} = \{\text{node}_1 : c_1, \text{node}_2 : c_2, \dots, \text{node}_n : c_n\}.$$

- **Локальный инкремент** на узле i выполняет state[i] += 1.
- **Значение счётчика** — сумма всех компонентов: value $= \sum_{j=1}^{n} c_j$.
- **Слияние** двух состояний A и B: $\text{merge}(A, B) = \{\forall j: \max(A[j], B[j])\}$.

Эта операция ассоциативна, коммутативна и идемпотентна, а любое обновление только увеличивает элементы, поэтому состояние образует монотонный join-полурешётку.

3.3. PN-Counter (Positive-Negative Counter)

Чтобы поддерживать не только инкремент, но и декремент, PN-Counter использует два независимых G-Counter: P — отслеживает инкременты, N — отслеживает декременты.

- **Инкремент** на узле i увеличивает $P[i]$.
- **Декремент** увеличивает $N[i]$.
- **Значение счётчика**: value $= \sum P[j] - \sum N[j]$.
- **Слияние**: выполняем merge(P_A, P_B) и merge(N_A, N_B) как для G-Counter.

PN-Counter сохраняет свойство CRDT: все изменения прибавляются к P или N, которые только растут, гарантируя сходимость.

3.4. Наивный счётчик и проблема согласованности

При простой репликации целочисленного счётчика (без CRDT) конфликтующие обновления с разных узлов могут быть потеряны, если просто перезаписывать значение без учёта истории. Например, узел A увеличил счётчик до 5, узел B увеличил до 3, после синхронизации «последнее обновление побеждает» (LWW) приведёт к значению 5 или 3,

потеряв часть операций. CRDT гарантирует, что итоговое значение отражает все произведённые операции.

---

4. Пример практического выполнения

4.1. Реализация G-Counter

python

```
class GCounter:
    def __init__(self, node_id=None):
        self.state = {}  # node_id -> count

    def increment(self, node_id, delta=1):
        self.state[node_id] = self.state.get(node_id, 0) + delta

    def value(self):
        return sum(self.state.values())

    def merge(self, other):
        for node, cnt in other.state.items():
            if node not in self.state or cnt > self.state[node]:
                self.state[node] = cnt

    @staticmethod
    def merge_multiple(counters):
        """Слияние списка счётчиков в один."""
        result = GCounter()
        for c in counters:
            result.merge(c)
        return result
```

4.2. Реализация PN-Counter

```
python
class PNCounter:
    def __init__(self, node_id=None):
        self.P = GCounter()
        self.N = GCounter()

    def increment(self, node_id, delta=1):
        self.P.increment(node_id, delta)

    def decrement(self, node_id, delta=1):
        self.N.increment(node_id, delta)

    def value(self):
        return self.P.value() - self.N.value()

    def merge(self, other):
        self.P.merge(other.P)
        self.N.merge(other.N)
```

4.3. Наивный счётчик (для сравнения)

```
python
class NaiveCounter:
    def __init__(self):
        self.val = 0

    def increment(self, delta=1):
        self.val += delta

    def value(self):
        return self.val
```

```
    # При «синхронизации» просто берём максимум (?), или последнее обновление -
    # но смоделируем как перезапись с LWW
    def merge_naive(self, other_clock, other_val, my_clock):
        # last-write-wins по метке времени
        if other_clock > my_clock:
            self.val = other_val
```

В эксперименте мы покажем потерю операций.

4.4. Симуляция параллельных обновлений и синхронизации

Создадим несколько реплик CRDT-счётчика (или наивного), разбросанных по узлам. Узлы генерируют случайные инкременты и декременты в изоляции, затем выполняется серия попарных слияний по некоторой топологии (например, полный граф или дерево). После завершения обмена проверяем, что все реплики сошлись.

```
python

import random


def simulate_crdt(counter_class, num_nodes, ops_per_node, sync_topology='full'):
    replicas = [counter_class() for _ in range(num_nodes)]
    # локальные операции
    for i, rep in enumerate(replicas):
        for _ in range(ops_per_node):
            if counter_class == PNCounter:
                if random.random() < 0.7:
                    rep.increment(i)
                else:
                    rep.decrement(i)
            elif counter_class == GCounter:
                rep.increment(i)


    # изначальное состояние запомним
```

```
    initial_values = [rep.value() for rep in replicas]

    # синхронизация: в раундах, каждый раунд случайные пары обмениваются
    for _ in range(num_nodes * 2):  # достаточно раундов для сходимости
        a, b = random.sample(range(num_nodes), 2)
        replicas[a].merge(replicas[b])
        replicas[b].merge(replicas[a])

    final_values = [rep.value() for rep in replicas]
    return initial_values, final_values
```

Для наивного счётчика симуляция аналогична, но merge выполняется по правилу LWW с метками времени. Показываем, что значения расходятся.

4.5. Визуализация сходимости

Построим график, показывающий значение счётчика на каждом узле после каждой синхронизации.

```
python
import matplotlib.pyplot as plt

def convergence_plot(num_nodes=5, ops=20):
    replicas = [GCounter() for _ in range(num_nodes)]
    # локальные операции
    for i, rep in enumerate(replicas):
        for _ in range(ops):
            rep.increment(i)
    history = [[] for _ in range(num_nodes)]
    for round in range(20):
        a, b = random.sample(range(num_nodes), 2)
        replicas[a].merge(replicas[b])
        replicas[b].merge(replicas[a])
        for i in range(num_nodes):
```

```
            history[i].append(replicas[i].value())


    for i, vals in enumerate(history):
        plt.plot(vals, label=f'Node {i}')
    plt.xlabel('Раунд синхронизации')
    plt.ylabel('Значение счётчика')
    plt.title('Сходимость G-Counter')
    plt.legend()
    plt.grid()
    plt.show()
```

Ожидаем, что после нескольких раундов все линии сольются в одну — общее количество операций.

4.6. Демонстрация конфликта в наивном счётчике

```
python

# Два узла делают операции, потом синхронизация LWW

clock = [0, 0]

vals = [0, 0]

# Узел 0 делает инкремент до 5

vals[0] = 5

clock[0] = 1

# Узел 1 делает инкремент до 3

vals[1] = 3

clock[1] = 2  # более поздняя метка

# синхронизация

if clock[1] > clock[0]:
    vals[0] = vals[1]

else:
    vals[1] = vals[0]

print(f"После наивной синхронизации: {vals}")  # одно из значений потеряно
```

Итоговое значение 3, хотя общее число операций 5+3 = 8.

---

5. Задания для индивидуального выполнения

Номер варианта определяется по формуле (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать GCounter и PNCounter. Провести симуляцию изолированных обновлений на N узлах и последующей синхронизации (полный граф). Показать сходимость к ожидаемому значению, сравнить с наивным счётчиком с LWW-слиянием.

Вариант	Число узлов	Операций на узел (G-Counter)	Операций для PN-Counter (инкр./декр. всего)
1	3	10	15
2	4	15	20
3	5	12	18
4	3	20	30
5	6	10	14
6	4	25	35
7	5	16	22
8	3	8	12
9	7	10	15
10	4	18	25
11	5	20	28
12	6	14	20

Вариант	Число узлов	Операций на узел (G-Counter)	Операций для PN-Counter (инкр./декр. всего)
13	3	22	33
14	7	12	16
15	4	30	40

Требования к отчёту: рабочая реализация, демонстрация корректного merge, графики сходимости для G-Counter, сравнительная таблица значений наивного и CRDT-счётчиков после конфликта.

5.2. Средний уровень (на оценку «хорошо»)

В дополнение к базовому:

- Исследовать влияние топологии синхронизации (кольцо, дерево, случайные пары) на скорость сходимости, построить график зависимости количества раундов от числа узлов.
- Реализовать **PN-Counter с вектором**, имеющим разные идентификаторы для P и N, и сравнить занимаемую память.
- Провести стресс-тест с 50 узлами и 2000 операций, замерив время слияния и размер состояния.
- Визуализировать процесс синхронизации PN-Counter, показав отдельно P и N счётчики.

Вариант	Параметры базового	Дополнительные задания
1	Вариант 1	Топологии, скорость сходимости
2	Вариант 2	Стресс-тест, память
3	Вариант 3	Топологии, PN-визуализация
4	Вариант 4	Сравнение векторов, скорость
5	Вариант 5	Топологии, стресс-тест

Вариант	Параметры базового	Дополнительные задания
6	Вариант 6	Память, PN-графики
7	Вариант 7	Сходимость в кольце
8	Вариант 8	Стресс-тест, топологии
9	Вариант 9	Визуализация, память
10	Вариант 10	Топологии, векторы
11	Вариант 11	Скорость, PN-тесты
12	Вариант 12	Топологии, стресс
13	Вариант 13	Память, сходимость
14	Вариант 14	Стресс-тест, PN
15	Вариант 15	Все пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать другие CRDT: **G-Set**, **2P-Set** или **OR-Set** и сравнить их поведение с парой G-Counter при моделировании распределённого набора.
- Реализовать **операционно-передаваемую** версию PN-Counter (operation-based) и сравнить затраты на передачу данных с state-based.
- Провести симуляцию в асинхронной сети с задержками и потерями сообщений, измерив время до сходимости при разных CRDT-подходах.
- Построить трёхмерный график зависимости количества раундов сходимости от числа узлов и доли операций декремента.

Вариант	Базовые параметры	Расширенное задание
1	4 узла, 20 оп.	G-Set/OR-Set, operation-based
2	5 узлов, 15 оп.	Асинхронная сеть, трёхмерный график
3	6 узлов, 25 оп.	Operation-based, задержки
4	3 узла, 30 оп.	OR-Set, трёхмерный
5	7 узлов, 12 оп.	G-Set, асинхронная
6	5 узлов, 18 оп.	Operation-based, потери
7	4 узла, 22 оп.	Трёхмерный график
8	6 узлов, 14 оп.	OR-Set, operation-based
9	8 узлов, 10 оп.	Асинхронная сеть
10	3 узла, 35 оп.	Сравнение G-Set/PN
11	5 узлов, 28 оп.	Трёхмерный, задержки
12	7 узлов, 16 оп.	Operation-based, потеря сообщений
13	4 узла, 20 оп.	OR-Set, асинхронная
14	6 узлов, 22 оп.	G-Set, трёхмерный
15	5 узлов, 30 оп.	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Что такое CRDT? В чём отличие State-based от Operation-based CRDT?

2. Почему G-Counter гарантирует сходимость без координации?
3. За счёт чего PN-Counter поддерживает декремент, оставаясь CRDT?
4. Какие условия должны выполняться для функции merge, чтобы структура данных была State-based CRDT?
5. Что произойдёт, если два узла выполнят инкремент G-Counter, а затем обменяются состояниями не один раз, а многократно? Изменится ли результат?
6. Каковы ограничения PN-Counter? Может ли его значение когда-либо убывать на отдельном узле до обмена?
7. Как влияет топология синхронизации на скорость достижения консенсуса?
8. Почему наивный счётчик с LWW теряет операции, а CRDT — нет?
9. Приведите примеры практического использования CRDT (Redis, Riak, Akka).
10. Каким образом можно сжимать состояние G-Counter при долгой работе?

---

7. Требования к отчёту

Отчёт оформляется в соответствии с общепринятыми правилами и включает:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание CRDT, G-Counter и PN-Counter, сходимость.
4. Программная реализация: листинг классов GCounter, PNCounter, NaiveCounter, код симуляции.
5. Экспериментальная часть:
    - Таблица сравнения итоговых значений после конфликта.
    - Графики сходимости.
    - (для среднего/повышенного) дополнительные графики времени, памяти, влияния топологии.
6. Выводы по работе.
7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- GCounter и PNCounter реализованы корректно.

- Проведены локальные обновления и слияния, продемонстрирована сходимость.
- Дано сравнение с наивным счётчиком, отчёт содержит график.

8.2. Оценка «хорошо» (средний уровень)

- Все требования базового уровня выполнены.
- Исследовано влияние топологии или проведён стресс-тест.
- Построены дополнительные графики, анализ убедителен.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы другие CRDT или варианты передачи (operation-based), проведена симуляция с потерями/задержками.
- Глубокий анализ, трёхмерные визуализации, сравнение подходов.

Снижение оценки: ошибки в логике слияния (не используется max), использование общего изменяемого счётчика без идентификаторов узлов, неправильное значение value, отсутствие демонстрации устойчивости.

Лабораторная работа № 22

Реализация репликации журнала Raft (упрощенная)

---

1. Цель работы

Цель работы – изучить алгоритм достижения консенсуса **Raft**, понять механизмы выбора лидера, репликации журнала команд и подтверждения коммитов при кворуме. В ходе выполнения студент реализует упрощённую модель кластера из трёх узлов, проведёт эксперименты с отказами лидера, сетевыми задержками и измерит время восстановления, сравнив его с теоретическими оценками.

---

2. Задачи

1. Изучить базовые компоненты Raft: состояния узлов (лидер, кандидат, последователь), таймеры выборов, механизм голосования.
2. Реализовать симулятор трёх узлов с передачей RPC-сообщений (RequestVote, AppendEntries) в асинхронном режиме с использованием потоков или событийного цикла.
3. Реализовать репликацию журнала: лидер принимает команды от клиента, рассылает AppendEntries, фиксирует запись при получении подтверждения от большинства (кворум 2 из 3).
4. Смоделировать сценарий отказа текущего лидера и проследить процесс перевыборов.
5. Внести регулируемые сетевые задержки и измерить время восстановления (от отказа лидера до избрания нового).
6. Сравнить измеренное время восстановления с теоретической нижней границей (2 × heartbeat-интервал) и объяснить расхождения.

---

3. Теоретическое обоснование

3.1. Проблема консенсуса и алгоритм Raft

В распределённых системах необходимо, чтобы совокупность узлов приходила к согласию относительно последовательности команд, даже при отказах части узлов. Алгоритм Raft, предложенный в 2014 году, разработан с упором на понятность и разбит на три независимых подзадачи:

- **выбор лидера**,
- **репликация журнала**,

- **безопасность (ограничения на избрание)**.

3.2. Состояния узлов и переключения

Каждый узел в кластере в любой момент находится в одном из трёх состояний:

- **Последователь (Follower)** – пассивно ожидает сообщений от лидера или кандидатов; при истечении таймаута выборов переходит в состояние кандидата.
- **Кандидат (Candidate)** – инициирует выборы, инкрементируя свой терм, голосуя за себя и рассылая всем узлам запросы RequestVote. Если получает большинство голосов – становится лидером. Если обнаруживает лидера с большим термом – возвращается в последователи.
- **Лидер (Leader)** – обрабатывает клиентские команды, рассылает AppendEntries (пустые или с новыми записями) для поддержания власти и репликации журнала. Лидер отправляет AppendEntries периодически (heartbeat).

Каждый узел хранит **текущий терм (term)** – монотонно возрастающее целое число. Все RPC-сообщения содержат терм отправителя; получатель отвергает сообщения с термом меньше своего и обновляет свой терм до большего.

3.3. Выбор лидера

- Последователи ожидают получения AppendEntries или RequestVote в течение случайного таймаута (обычно 150–300 мс). Если таймер истекает, последователь становится кандидатом.
- Кандидат увеличивает currentTerm, голосует за себя и отправляет RequestVote всем узлам.
- Если кандидат получает голоса от большинства узлов (включая свой), он становится лидером.
- Во время выборов каждый узел может отдать голос не более одного раза в одном терме (голосует за первого кандидата, удовлетворяющего условиям: терм не меньше, журнал не короче, и ещё не голосовал).

3.4. Репликация журнала

Журнал представляет собой последовательность записей (term, command). Лидер ведёт журнал, а последователи синхронизируют свои журналы с лидером. Механизм:

- Клиент посылает команду лидеру. Лидер добавляет её в свой журнал как новую запись.
- Лидер рассылает AppendEntries, содержащие новые записи. Последователи, получив сообщение, добавляют записи в свои журналы (если предшествующие индексы совпадают) и отвечают success.

- Как только запись подтверждена большинством узлов (включая лидера), она считается **закоммиченной** (committed). Лидер информирует последователей об индексе коммита, и все применяют команды к конечному автомату.

В учебной реализации мы упрощаем: не храним конечный автомат, а считаем запись закоммиченной при кворуме (2 из 3).

3.5. Время восстановления

Теоретически, после отказа лидера новый лидер будет избран за время от t1 до t1 + T, где t1 – таймаут выборов (heartbeat timeout). Минимальное время обнаружения отказа – один пропущенный heartbeat, поэтому минимальное время восстановления составляет heartbeat_interval + election_timeout. Обычно heartbeat интервал много меньше минимального таймаута выборов, так что оценка снизу примерно равна среднему таймауту выборов плюс время на голосование. При симуляции мы измерим фактическое время.

---

4. Пример практического выполнения

4.1. Модель узла и сообщений

Систему моделируем как многопоточное приложение с очередью сообщений на каждом узле. Для упрощения используем threading.Thread и queue.Queue. Каждый узел – объект с состоянием, текущим термом, журналом. Вместо реальной сети обмен идёт через центральный диспетчер сообщений с задержками.

python

```
import threading, queue, time, random, enum
from dataclasses import dataclass


class State(enum.Enum):
    FOLLOWER = 0
    CANDIDATE = 1
    LEADER = 2


@dataclass
class LogEntry:
    term: int
    command: str
```

```
class Node(threading.Thread):
    def __init__(self, node_id, cluster, delay_range=(0.001, 0.005)):
        super().__init__(daemon=True)
        self.id = node_id
        self.cluster = cluster
        self.state = State.FOLLOWER
        self.currentTerm = 0
        self.votedFor = None
        self.log = []  # LogEntry list
        self.commitIndex = -1
        self.lastApplied = -1
        self.nextIndex = {}
        self.matchIndex = {}
        self.inbox = queue.Queue()
        self.election_timeout = random.uniform(0.15, 0.3)
        self.heartbeat_interval = 0.05
        self.delay_range = delay_range

    def run(self):
        while True:
            try:
                msg = self.inbox.get(timeout=self.election_timeout)
            except queue.Empty:
                self.on_timeout()
            else:
                self.handle_message(msg)

    def on_timeout(self):
        if self.state != State.LEADER:
```

```
            self.become_candidate()


    def become_candidate(self):
        self.state = State.CANDIDATE
        self.currentTerm += 1
        self.votedFor = self.id
        # Отправляем RequestVote
        for n in self.cluster.nodes:
            if n.id != self.id:
                self.send_request_vote(n)


    def send_request_vote(self, target):
        self.cluster.send_msg(self.id, target.id, {
            'type': 'RequestVote',
            'term': self.currentTerm,
            'candidateId': self.id,
            'lastLogIndex': len(self.log)-1,
            'lastLogTerm': self.log[-1].term if self.log else 0
        })


    # Обработка сообщений и другие методы...
```

Полный код примера займёт много места, поэтому в методичке приведём фрагменты логики и укажем на полную реализацию в приложении. Описание будет текстовым.

4.2. Логика обработки RequestVote

python

```
def handle_requestvote(self, msg):
    if msg['term'] < self.currentTerm:
        self.send_reply(msg['candidateId'],
{'type':'RequestVoteReply','term':self.currentTerm,'voteGranted':False})
    else:
```

```
        if msg['term'] > self.currentTerm:
            self.currentTerm = msg['term']
            self.votedFor = None
            self.state = State.FOLLOWER
        # Голосуем, если кандидатский журнал не короче нашего
        if (self.votedFor is None or self.votedFor == msg['candidateId']) and \
           (msg['lastLogTerm'] > (self.log[-1].term if self.log else 0) or \
           (msg['lastLogTerm'] == (self.log[-1].term if self.log else 0) and msg['lastLogIndex'] >= len(self.log)-1)):
            self.votedFor = msg['candidateId']
            self.send_reply(msg['candidateId'], {'type':'RequestVoteReply','term':self.currentTerm,'voteGranted':True})
        else:
            self.send_reply(msg['candidateId'], {'type':'RequestVoteReply','term':self.currentTerm,'voteGranted':False})
```

4.3. AppendEntries и коммит

Лидер периодически рассылает AppendEntries. При успешных ответах обновляет matchIndex и если большинство подтвердило, сдвигает commitIndex.

4.4. Эксперимент с отказом лидера и замерами времени

Создаётся кластер из трёх узлов, ждём избрания лидера. Затем "убиваем" поток лидера (останавливаем). Засекаем время, когда отказ зафиксирован (например, другой узел перешёл в кандидаты) и момент избрания нового лидера. Время восстановления = разница.

python

```
# monitoring thread
def monitor(cluster):
    while True:
        for node in cluster.nodes:
            if node.state == State.LEADER:
                current_leader = node.id
        time.sleep(0.01)
```

Визуализируем смену состояний на временной диаграмме.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать ядро Raft (сообщения, выборы, репликация) для 3 узлов. Провести один сценарий: запуск, запись 5 команд через лидера, отказ лидера и наблюдение перевыборов. Измерить время от момента отказа до избрания нового лидера и сравнить с 2 × heartbeat (0.1 с). Построить диаграмму состояний узлов во времени (например, текстовый вывод или простой график).

Вариант	Heartbeat (с)	Election timeout (с)	Кол-во команд	Сценарий
1	0.05	0.15–0.25	5	Отказ лидера через 2 с
2	0.04	0.12–0.20	8	Отказ лидера через 3 с
3	0.06	0.18–0.30	3	Отказ лидера через 1.5 с
4	0.05	0.15–0.25	10	Отказ лидера через 4 с
5	0.03	0.10–0.18	6	--
6	0.07	0.20–0.35	4	--
7	0.05	0.14–0.22	7	--
8	0.08	0.25–0.40	5	--
9	0.04	0.12–0.20	9	--
10	0.06	0.16–0.26	2	--
11	0.05	0.15–0.25	6	Отказ и повторный отказ

Вариант	Heartbeat (с)	Election timeout (с)	Кол-во команд	Сценарий
12	0.07	0.20–0.30	4	--
13	0.03	0.10–0.15	8	--
14	0.04	0.13–0.21	5	--
15	0.05	0.15–0.25	7	Запись команды во время выборов

Требования: корректная работа выборов и фиксация коммитов (кворум), измерение времени, простейшая диаграмма.

5.2. Средний уровень (на оценку «хорошо»)

Помимо базового:

- Реализовать сценарий «сбой сети»: временное отключение одного узла (разрыв связи) и последующее возвращение; проверить, что лидер корректно восстанавливает репликацию.
- Провести множественные замеры времени восстановления (10 итераций) и построить распределение, вычислив среднее и 95-й процентиль. Сравнить с теоретической границей.
- Добавить в симуляцию регулируемую задержку сети (имитация WAN) и изучить её влияние на пропускную способность репликации (команды в секунду).
- Реализовать визуализацию журнала каждого узла в виде цветной полосы, показывающей коммиты.

Вариант	Параметры базы	Дополнительные задания
1	Вариант 1	Распределение времён, визуализация журнала
2	Вариант 2	Сбой сети, пропускная способность
3	Вариант 3	Замеры, визуализация

Вариант	Параметры базы	Дополнительные задания
4	Вариант 4	Сеть WAN, распределение
5	Вариант 5	Восстановление после разрыва
6	Вариант 6	Пропускная способность, журналы
7	Вариант 7	Задержка сети, замеры
8	Вариант 8	Сбой сети, визуализация
9	Вариант 9	WAN, распределение
10	Вариант 10	Журналы, замеры
11	Вариант 11	Отказ и разрыв, визуализация
12	Вариант 12	Задержка, пропускная способность
13	Вариант 13	Распределение времён
14	Вариант 14	Сбой сети, замеры
15	Вариант 15	Все доп. пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать механизм **snapshot** (сжатия журнала) и продемонстрировать его работу, измерив уменьшение размера журнала.
- Провести сравнение с другим консенсус-алгоритмом на основе кворума (из ЛР19) по времени восстановления и сложности реализации.
- Реализовать **динамическое изменение конфигурации** кластера (добавление/удаление узла) и продемонстрировать переконфигурацию без остановки обслуживания.

- Построить подробную временную диаграмму (swimlane) всех узлов с отображением сообщений и смены состояний, используя библиотеку matplotlib.

Вариант	Базовые параметры	Расширенное задание
1	3 узла, heartbeat 0.05	Snapshot, сравнение с кворумом
2	3 узла, 0.04	Переконфигурация, swimlane диаграмма
3	3 узла, 0.06	Snapshot, динамическое членство
4	3 узла, 0.05	Сравнение алгоритмов, swimlane
5	3 узла, 0.03	Переконфигурация, snapshot
6	3 узла, 0.07	Swimlane, сравнение с кворумом
7	3 узла, 0.05	Snapshot, динамическое изменение
8	3 узла, 0.08	Сравнение, переконфигурация
9	3 узла, 0.04	Swimlane, snapshot
10	3 узла, 0.06	Динамическое членство, сравнение
11	3 узла, 0.05	Swimlane, snapshot
12	3 узла, 0.07	Переконфигурация, swimlane
13	3 узла, 0.03	Сравнение, snapshot
14	3 узла, 0.04	Swimlane, динамическое членство
15	3 узла, 0.05	Комбинация расширенных задач

6. Контрольные вопросы

1. Каковы три состояния узла в Raft и в чём их назначение?
2. Как происходит процесс выбора лидера? Почему таймауты выборов имеют случайную длительность?
3. Что такое терм (term) и как он предотвращает конфликты в кластере?
4. Каким образом лидер поддерживает свой авторитет в стабильном состоянии?
5. Объясните механизм репликации журнала и условие коммита. Почему кворум составляет большинство?
6. Что произойдёт, если лидер не сможет получить подтверждение от большинства для новой записи?
7. Как алгоритм Raft гарантирует, что закоммиченная запись не будет отменена при перевыборах?
8. Как время восстановления зависит от heartbeat-интервала и таймаута выборов?
9. С какой целью в реальных реализациях Raft используется сжатие журнала (snapshot)?
10. В каких промышленных системах используется Raft? Какие альтернативы ему существуют (Paxos, Zab)?

---

7. Требования к отчёту

Отчёт оформляется по кафедральным стандартам и должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание Raft, его подзадач, механизмов выборов и репликации.
4. Программная реализация: общая архитектура классов, описание RPC-сообщений, листинги ключевых функций.
5. Экспериментальная часть:
    - Временна́я диаграмма состояний узлов.
    - Измерения времени восстановления, сравнение с теорией.
    - Результаты сценариев отказа и восстановления.
    - (для среднего/повышенного уровня) дополнительные графики, распределения, визуализации журнала.

6. Выводы по работе.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно работает механизм выборов, избирается лидер.
- Реализована репликация команд с фиксацией при кворуме.
- Проведён один сценарий отказа, измерено время восстановления и представлен элементарный график.
- Отчёт содержит основные разделы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все пункты базового уровня.
- Добавлены сценарии разрыва связи, множественные замеры с анализом распределения.
- Визуализированы журналы и задержки.
- Выводы аргументированы, продемонстрировано понимание влияния параметров.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован snapshot и/или динамическая переконфигурация.
- Проведено сравнение с другим алгоритмом или детальная swimlane диаграмма.
- Отчёт демонстрирует продвинутое знание Raft и его инженерных аспектов.

Снижение оценки: неправильная логика голосования, нарушение условия избрания (журнал не проверяется), отсутствие фиксации коммитов.

Лабораторная работа № 23

Алгоритм разрешения конфликтов Last Write Wins (LWW)

---

1. Цель работы

Цель работы – изучить простейший механизм разрешения конфликтов **Last Write Wins (LWW)**, основанный на сравнении временны́х меток, реализовать его с использованием как физических, так и логических часов, экспериментально оценить количество потерянных обновлений в условиях рассинхронизации узлов и определить сценарии, в которых потеря данных приемлема.

---

2. Задачи

1. Реализовать регистр данных, разрешающий конфликты по правилу LWW с использованием **физического времени** (системные часы).
2. Реализовать вариант LWW с **логическими часами Лампорта** для упорядочения событий без привязки к физическому времени.
3. Создать симулятор распределённой системы из нескольких узлов, выполняющих конкурентные записи без координации и периодически синхронизирующих состояние.
4. Провести серию экспериментов, варьируя величину рассинхронизации часов и интенсивность конфликтов, измеряя:
    - долю потерянных обновлений (количество записей, которые были «затёрты» более поздней меткой, но фактически не должны были проигрывать);
    - погрешность, вызванную дрейфом физических часов.
5. Сравнить поведение физического и логического LWW, выделить преимущества и недостатки каждого.
6. Обсудить практические сценарии, в которых LWW является допустимым компромиссом (логирование, метрики, кэши).

---

3. Теоретическое обоснование

3.1. Конфликты при асинхронной репликации

В распределённых хранилищах с несколькими репликами, допускающими запись на любом узле, одновременные обновления одного и того же ключа могут привести к расхождению версий. Необходим механизм разрешения конфликтов. Простейший подход — **Last Write Wins (LWW)**: каждой записи присваивается временна́я метка, и при конфликте сохраняется

версия с наибольшей меткой. Простота реализации сделала LWW популярным в системах типа Amazon Dynamo, Cassandra (в определённых конфигурациях), Redis и др.

3.2. LWW с физическими часами

При использовании системных часов каждый узел при записи проставляет текущее значение time.time() (с точностью до миллисекунд). При слиянии реплик выбирается запись с максимальным timestamp. Такой подход имеет очевидные недостатки:

- **Расхождение часов (clock skew)** между узлами нарушает причинность: более позднее физическое событие может получить меньшую метку, чем более раннее на другом узле, и быть потерянным.
- **Обратное течение времени** при коррекции NTP может привести к дублированию меток.
- **Разрешающая способность**: при одновременных обновлениях до миллисекунды возможна коллизия.

Тем не менее, для многих некритичных данных (логи, телеметрия, кэши) LWW с физическими часами приемлем.

3.3. LWW с логическими часами Лампорта

Логические часы Лампорта — это счётчик, который увеличивается при каждом событии и передаётся между узлами. Правило:

- Перед выполнением локальной операции узел увеличивает свой счётчик.
- При отправке сообщения узел прикрепляет текущее значение счётчика.
- При приёме сообщения узел устанавливает свой счётчик в max(свой, полученный) + 1.

Если для разрешения LWW использовать логический счётчик Лампорта, упорядочение будет согласованным с причинно-следственными связями (happened-before). Однако логические часы дают лишь частичный порядок: если события конкурентны, они могут иметь одинаковое значение или несравнимые счётчики (у Лампорта сравнение — просто число, поэтому конкурентные события могут получить одинаковый счётчик, если не было обмена). В общем случае требуется дополнительные идентификаторы.

Альтернатива — **Hybrid Logical Clocks** (HLC), сочетающие физическое и логическое время, но в учебных рамках мы реализуем простейшие логические часы Лампорта и сравним с физическими.

3.4. Количественная оценка потерь

При моделировании с известной истинной хронологией (порядок операций) можно измерить:

- **Число потерянных записей** — операций записи, результат которых исчез после синхронизации, хотя они не должны были быть перекрыты более поздними конкурентными обновлениями (с точки зрения глобального времени).
- **Погрешность** от рассинхронизации часов оценивается как количество случаев, когда timestamp(A) < timestamp(B) при том, что физически A произошло позже B.

---

4. Пример практического выполнения

4.1. Регистр LWW с физическими часами

python

```
import time


class LWWRegisterPhysical:
    def __init__(self, key, value=None):
        self.key = key
        self.value = value
        self.timestamp = time.time() if value is not None else 0

    def set(self, value, ts=None):
        if ts is None:
            ts = time.time()
        if ts > self.timestamp:
            self.value = value
            self.timestamp = ts

    def merge(self, other):
        if other.timestamp > self.timestamp:
            self.value = other.value
            self.timestamp = other.timestamp
```

4.2. Логические часы Лампорта и LWW Register

python

```
class LamportClock:
    def __init__(self, node_id):
        self.time = 0
        self.node_id = node_id

    def local_event(self):
        self.time += 1
        return self.time

    def receive_event(self, received_time):
        self.time = max(self.time, received_time) + 1
        return self.time

class LWWRegisterLamport:
    def __init__(self, key, clock):
        self.key = key
        self.value = None
        self.timestamp = 0      # логическое время
        self.clock = clock

    def set(self, value):
        ts = self.clock.local_event()
        # timestamp как кортеж (логическое_время, node_id) для разрешения коллизий
        self.timestamp = (ts, self.clock.node_id)
        self.value = value

    def merge(self, other):
        # Сначала обновляем локальные часы по полученному timestamp
        self.clock.receive_event(other.timestamp[0])
```

```
    # Сравнение: старшинство определяется в первую очередь логическим временем, потом node_id
    if other.timestamp > self.timestamp:
        self.value = other.value
        self.timestamp = other.timestamp
```

Примечание: Сравнение кортежей (time, node_id) обеспечивает детерминированный выбор при одинаковом логическом времени.

4.3. Симулятор конкурентных записей

Создадим класс Node, который хранит локальную копию регистра и может выполнять write, а также обмениваться состоянием.

python

```
import random

class SimulatedNode:
    def __init__(self, node_id, register_factory):
        self.id = node_id
        self.register = register_factory(node_id)

    def local_write(self, value):
        self.register.set(value)

    def sync_from(self, other_node):
        self.register.merge(other_node.register)
```

Для физических часов смоделируем расхождение, добавляя смещение к time.time() на каждом узле. Для логических часов просто используем локальные события.

Функция симуляции:

python

```
def run_simulation(num_nodes, num_events, skew_max=0.0, clock_type='physical'):
    # Инициализация узлов
    if clock_type == 'physical':
```

```
        # Каждый узел имеет смещение времени
        offsets = [random.uniform(-skew_max, skew_max) for _ in range(num_nodes)]
        def factory(nid):
            reg = LWWRegisterPhysical(f"key")
            # Переопределим time.time для каждого узла? Вместо этого при записи будем передавать timestamp = real_time + offset.
            return reg
    else:
        # Логические часы
        clocks = [LamportClock(i) for i in range(num_nodes)]
        def factory(nid):
            return LWWRegisterLamport(f"key", clocks[nid])

    nodes = [SimulatedNode(i, factory) for i in range(num_nodes)]

    # Журнал всех операций с реальным порядком (для анализа потерь)
    operations = []  # (global_time, node, value)
    global_time = 0.0

    for _ in range(num_events):
        node = random.choice(nodes)
        value = f"val_{random.randint(1,100)}"
        if clock_type == 'physical':
            ts = global_time + offsets[node.id]
            node.register.set(value, ts)
        else:
            node.local_write(value)
        operations.append((global_time, node.id, value))
        global_time += 0.001 + random.random() * 0.01  # неравномерный шаг
```

```
        # Случайная синхронизация между двумя узлами
        if random.random() < 0.3:
            a, b = random.sample(nodes, 2)
            a.sync_from(b)
            b.sync_from(a)

    # После всех событий финальная синхронизация всех со всеми
    for a in nodes:
        for b in nodes:
            a.sync_from(b)

    # Определяем последнее "выжившее" значение
    # Сравниваем с эталонным порядком: если бы не было синхронизации LWW, а была строгая последовательность по глобальному времени.
    # Для LWW корректность: последнее записанное значение по глобальному времени должно быть итоговым, но из-за смещения может проиграть.
    # Находим последнюю операцию по global_time
    last_op = max(operations, key=lambda x: x[0])
    final_value = nodes[0].register.value if nodes else None
    # Поскольку все синхронизировались, на всех узлах одно значение
    loss = (final_value != last_op[2])
    return loss, final_value, last_op[2]
```

Этот код упрощённо измеряет, потеряла ли система последнюю по глобальному времени запись при наличии рассинхронизации.

4.4. Эксперимент и визуализация

Запустим серии с разными значениями skew_max и числом событий, соберём статистику потерь.

python

```
def experiment(skew_levels, events=200, trials=50):
```

```
    loss_rates = []
    for skew in skew_levels:
        losses = 0
        for _ in range(trials):
            loss, _, _ = run_simulation(num_nodes=3, num_events=events, skew_max=skew, clock_type='physical')
            if loss:
                losses += 1
        loss_rates.append(losses / trials)
    return loss_rates


skews = [0.0, 0.01, 0.02, 0.05, 0.1]
phys_loss = experiment(skews)


import matplotlib.pyplot as plt
plt.plot(skews, phys_loss, 'o-', label='Physical LWW')
plt.xlabel('Максимальное смещение часов (c)')
plt.ylabel('Доля потерянных «последних» записей')
plt.title('Влияние рассинхронизации часов на LWW')
plt.grid()
plt.legend()
plt.show()
```

Для логических часов аналогичный тест должен давать 0 потерь с точки зрения причинного порядка, так как логические часы согласованы с happened-before (но конкурентные обновления могут разрешаться произвольно, однако последняя по глобальному времени запись не теряется, если она причинно следует за всеми остальными — в нашем симуляторе из-за синхронизации логических часов причинный порядок соблюдается).

4.5. Сравнение

Построим таблицу: средние потери для физических часов при skew=0, skew=0.05, skew=0.1, а для логических — потери (должны быть 0) и доля недетерминированных разрешений.

Добавим обсуждение приемлемости LWW для логов, метрик, где точный порядок не критичен.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать LWW-регистр с физическими метками. Провести симуляцию конкурентных записей на заданном числе узлов с указанным уровнем рассинхронизации. Измерить долю потерянных обновлений (относительно глобального времени). Построить график зависимости доли потерь от skew для одного из сценариев.

Вариант	Число узлов	Событий	Максимальный skew (сек)	Вероятность синхронизации
1	3	200	0.0, 0.02, 0.05, 0.1	0.2
2	3	300	0.0, 0.01, 0.03, 0.05	0.3
3	4	250	0.0, 0.02, 0.04, 0.08	0.25
4	5	180	0.0, 0.03, 0.06, 0.09	0.15
5	3	400	0.0, 0.02, 0.05, 0.1	0.2
6	4	350	0.0, 0.01, 0.04, 0.07	0.3
7	3	220	0.0, 0.03, 0.06, 0.12	0.18
8	5	300	0.0, 0.02, 0.05, 0.08	0.22
9	4	280	0.0, 0.04, 0.08, 0.1	0.28
10	3	150	0.0, 0.01, 0.03, 0.06	0.35
11	4	320	0.0, 0.02, 0.04, 0.09	0.2

Вариант	Число узлов	Событий	Максимальный skew (сек)	Вероятность синхронизации
12	3	180	0.0, 0.05, 0.1, 0.15	0.25
13	5	250	0.0, 0.02, 0.06, 0.10	0.2
14	4	200	0.0, 0.03, 0.05, 0.08	0.3
15	3	350	0.0, 0.02, 0.05, 0.1	0.15

Требования к отчёту: работающая модель, таблица потерь, график, интерпретация результатов.

5.2. Средний уровень (на оценку «хорошо»)

В дополнение к базовому:

- Реализовать LWW на логических часах Лампорта, провести ту же симуляцию и сравнить потери.
- Провести анализ «несправедливых» разрешений: при конкурентных обновлениях, когда временные метки равны, LWW с физическими часами делает произвольный выбор. Внести детерминированное правило (по node_id) и измерить, как часто это происходит.
- Построить график зависимости среднего дрейфа (разницы между физическим временем и логическим) при длительной работе.
- Исследовать влияние частоты синхронизации (вероятности обмена) на потери: для трёх значений вероятности построить кривые.

Вариант	Параметры базового	Дополнительные задания
1	Вариант 1	Логические часы, влияние частоты синхронизации
2	Вариант 2	Логические часы, детерминированное разрешение
3	Вариант 3	Частота синхронизации, дрейф
4	Вариант 4	Логические часы, сравнение

Вариант	Параметры базового	Дополнительные задания
5	Вариант 5	Детерминизм, частоты
6	Вариант 6	Логические, дрейф
7	Вариант 7	Частота синхронизации
8	Вариант 8	Логические часы, разрешение
9	Вариант 9	Дрейф, частоты
10	Вариант 10	Логические часы
11	Вариант 11	Частота, детерминизм
12	Вариант 12	Логические, дрейф
13	Вариант 13	Частота синхронизации
14	Вариант 14	Логические часы, сравнение
15	Вариант 15	Все доп. пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **Hybrid Logical Clocks (HLC)** (комбинация физического и логического времени) и внедрить их в LWW; сравнить точность с чисто физическими и чисто логическими часами.
- Смоделировать сценарий «реальной рассинхронизации» с помощью нормального распределения смещений (имитация NTP) и вычислить 99-й процентиль потерь.
- Сравнить LWW с **CRDT-регистром** (Multi-Value Register) с точки зрения потерь данных: реализовать регистр, хранящий все конкурентные версии, и замерить объём дополнительной памяти.
- Провести анализ применимости LWW на реальном треке временны́х меток (например, логи сервера), визуализировав конфликты.

Вариант	Базовые параметры	Расширенное задание
1	3 узла, skew=0.05	HLC, сравнение с CRDT
2	4 узла, skew=0.1	Нормальное распределение, 99 процентиль
3	3 узла, skew=0.08	HLC, CRDT-регистр
4	5 узлов, skew=0.02	Реальные логи, анализ
5	3 узла, skew=0.03	HLC, многовариантный регистр
6	4 узла, skew=0.07	Сравнение с CRDT, 99 процентиль
7	3 узла, skew=0.04	HLC, реальный трек
8	5 узлов, skew=0.06	Нормальное распределение, CRDT
9	3 узла, skew=0.02	HLC, анализ логов
10	4 узла, skew=0.09	CRDT, гистограмма потерь
11	3 узла, skew=0.05	HLC, сравнение
12	5 узлов, skew=0.12	Реальные логи, CRDT
13	3 узла, skew=0.01	HLC, многовариантный
14	4 узла, skew=0.06	Нормальное распределение, логи
15	3 узла, skew=0.07	Комбинация расширенных задач

---

6. Контрольные вопросы

1. В чём заключается принцип разрешения конфликтов Last Write Wins? Какие метаданные используются для определения «последней» записи?
2. Как рассинхронизация физических часов между узлами влияет на корректность LWW?
3. Чем логические часы Лампорта отличаются от физических применительно к LWW? Какие гарантии они дают?
4. Что такое Hybrid Logical Clocks? В чём их преимущество перед чисто физическими или логическими?
5. Почему LWW может быть приемлемым для логов, метрик и кэшей, но не для финансовых транзакций?
6. Как можно устранить неоднозначность при одинаковых временны́х метках в LWW?
7. Какие дополнительные накладные расходы вносит хранение нескольких версий вместо использования LWW?
8. Объясните понятие «потерянное обновление» в контексте LWW и приведите количественный пример.
9. Каким образом можно смоделировать рассинхронизацию часов в симуляторе распределённой системы?
10. Приведите примеры реальных систем, которые используют LWW, и объясните, почему разработчики выбрали этот компромисс.

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель, задачи.
3. Теоретическая часть: описание LWW, физических и логических часов, проблем рассинхронизации, областей применения.
4. Программная реализация: листинги классов LWW-регистров, симулятора.
5. Экспериментальная часть:
    - Таблицы потерь при различных skew.
    - Графики зависимости доли потерь от skew и частоты синхронизации.
    - Сравнительная таблица физических и логических часов.
    - (для среднего/повышенного уровня) дополнительные метрики, анализ HLC, гистограммы.

6. Выводы: в каких условиях допустим LWW, какие меры можно предпринять для снижения потерь.
7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован LWW-регистр с физическими часами, проведена симуляция с разными skew.
- Построен график потерь, измерены метрики.
- Отчёт содержит необходимые разделы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Добавлены логические часы Лампорта, проведено сравнение.
- Исследовано влияние частоты синхронизации и детерминизма.
- Отчёт аргументирован, выводы обоснованы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализованы HLC или CRDT-регистр, проведено сравнение с реальными треками или нормальным распределением.
- Продемонстрировано глубокое понимание компромиссов, предложены практические рекомендации.

Снижение оценки: неверная реализация слияния, отсутствие моделирования рассинхронизации, неверные выводы о потерях, неработающий код.

Лабораторная работа № 24

Сравнение стратегий репликации: синхронная vs асинхронная

---

1. Цель работы

Цель работы – изучить две фундаментальные стратегии репликации данных в распределённых системах — **синхронную** и **асинхронную**, реализовать их имитационную модель с одним мастером и несколькими репликами, экспериментально измерить задержки записи и объём потерь данных при отказе мастера, а также сформулировать компромисс между латентностью и надёжностью (durability).

---

2. Задачи

1. Реализовать симулятор реплицируемого хранилища, состоящего из одного мастера и S слейвов ($S = 2$ по умолчанию, с возможностью масштабирования).
2. Реализовать два режима репликации:
    - **Синхронная**: операция записи считается успешной только после получения подтверждения от всех слейвов о фиксации данных.
    - **Асинхронная**: мастер фиксирует запись у себя и сразу возвращает успех клиенту; слейвы получают обновления с некоторой задержкой.
3. Провести эксперименты, измеряя:
    - **Задержку записи (latency)** в зависимости от числа слейвов и величины сетевой задержки.
    - **Объём потерянных данных (recency loss)** при внезапном отказе мастера — количество записей, зафиксированных на мастере, но не доставленных на слейвы до момента отказа.
4. Пронаблюдать и визуализировать компромисс между латентностью и долговечностью (durability) при варьировании числа синхронных реплик.
5. Построить графики зависимости средней задержки от числа слейвов и уровня потерь от задержки репликации.

---

3. Теоретическое обоснование

3.1. Репликация «ведущий–ведомые» (Primary-Backup)

В централизованной модели репликации один узел назначается **мастером (primary)**, принимающим все операции записи от клиентов. Остальные узлы — **слейвы (standby / secondary)** — хранят копии данных. Такая архитектура характерна для большинства

реляционных СУБД, Redis (при включённой репликации), MongoDB (Replica Set с primary) и т. д.

3.2. Синхронная репликация

При синхронной репликации мастер, получив запись, не отправляет подтверждение клиенту до тех пор, пока все слейвы (или заданное количество) не подтвердят, что они сохранили эту запись у себя. Гарантируется **строгая согласованность** и **нулевая потеря данных**: после подтверждения коммита данные гарантированно находятся как минимум на k узлах. Недостаток — задержка записи определяется **медленнейшим** из реплик, что ограничивает масштабируемость.

В реальных системах часто применяют вариант с кворумной синхронизацией: запись считается зафиксированной, если подтверждение получено от большинства реплик (например, synchronous_commit = 'on' в PostgreSQL с числом синхронных standby). В учебной реализации для простоты будем считать, что мастер ждёт подтверждения от **всех** слейвов.

3.3. Асинхронная репликация

Мастер фиксирует запись локально и немедленно отвечает клиенту. Обновления передаются слейвам в фоновом режиме с некоторой задержкой (лаг репликации). Это обеспечивает минимальную задержку записи, но при отказе мастера последние незафиксированные на слейвах изменения могут быть **потеряны**. Объём потерь зависит от лага репликации и интенсивности записи.

3.4. Метрики сравнения

- **Задержка (latency)** – время от отправки запроса клиентом до получения подтверждения. В синхронной модели $L_{\text{sync}} = \max(\text{RTT}_i)$ + время записи на мастере. В асинхронной $L_{\text{async}} \approx$ время локальной записи на мастере.
- **Долговечность (durability)** – способность пережить отказ мастера без потерь. Оценивается как количество записей, потерянных за интервал между отказом и последней доставкой на реплики.

3.5. Компромисс LATENCY vs DURABILITY

Увеличивая число синхронных реплик, мы повышаем надёжность, но одновременно увеличиваем и задержку, так как мастер обязан дождаться ответа от всех требуемых узлов. Сокращение числа синхронных реплик или переход к асинхронной репликации уменьшает задержку ценой возможной потери части недавних записей. Этот компромисс является одной из иллюстраций CAP-теоремы в практически важном аспекте.

---

4. Пример практического выполнения

4.1. Модель симулятора на Python

Создадим классы Master и Slave, а также класс ReplicationSystem, управляющий имитацией времени.

python

```
import random
import heapq
import matplotlib.pyplot as plt

class Event:
    def __init__(self, time, event_type, data=None):
        self.time = time
        self.type = event_type
        self.data = data

    def __lt__(self, other):
        return self.time < other.time

class Simulator:
    def __init__(self):
        self.time = 0.0
        self.queue = []

    def schedule(self, delay, event_type, data=None):
        heapq.heappush(self.queue, Event(self.time + delay, event_type, data))

    def run(self, duration):
        while self.queue and self.queue[0].time <= duration:
            event = heapq.heappop(self.queue)
            self.time = event.time
            yield event
```

Мастер и слейвы имеют локальные хранилища и логику обработки команд.

python

```
class Slave:
    def __init__(self, slave_id, simulator, latency_to_master):
        self.id = slave_id
        self.sim = simulator
        self.latency = latency_to_master  # RTT/2
        self.store = {}
        self.lag = 0  # счётчик задержанных применений

    def apply_write(self, key, value, write_id):
        # Симуляция получения записи от мастера с задержкой
        self.store[key] = (value, write_id)
        # Отправляем подтверждение обратно мастеру
        self.sim.schedule(self.latency, 'ack', {'slave_id': self.id, 'write_id': write_id})
```

python

```
class Master:
    def __init__(self, simulator, slaves, mode='sync'):
        self.sim = simulator
        self.slaves = slaves
        self.mode = mode
        self.store = {}
        self.pending = {}  # write_id -> (key, value, acks_required, ack_count, latency_start)
        self.write_counter = 0
        self.lost_writes = []  # для асинхронного режима при отказе

    def write(self, key, value):
        self.write_counter += 1
        write_id = self.write_counter
```

```
        self.store[key] = (value, write_id)
        if self.mode == 'async':
            # асинхронно: сразу отвечаем клиенту, в фоне отправляем слейвам
            # имитируем отправку слейвам с небольшим лагом
            for slave in self.slaves:
                self.sim.schedule(slave.latency, 'apply', {'slave': slave, 'key': key, 'value': value, 'write_id': write_id})
            return 0.0  # мгновенный ответ
        else:
            # синхронный режим: ждём подтверждения от всех слейвов
            acks_needed = len(self.slaves)
            self.pending[write_id] = {
                'key': key,
                'value': value,
                'acks_needed': acks_needed,
                'ack_count': 0,
                'start_time': self.sim.time
            }
            for slave in self.slaves:
                self.sim.schedule(slave.latency, 'apply', {'slave': slave, 'key': key, 'value': value, 'write_id': write_id})
            return None  # подтверждение будет позже
```

Полная симуляция потребует обработки событий и сбора статистики. В учебной методичке мы опишем ключевые моменты и дадим законченный фрагмент для воспроизведения.

4.2. Эксперимент и измерение задержек

Для синхронного режима измеряем время от вызова write до получения всех ack. Для асинхронного режима задержка практически нулевая (локальная запись), но мы также можем замерить задержку распространения до слейвов (лаг). Потери данных при отказе мастера моделируются как разница между последним write_id на мастере и максимальным write_id среди слейвов в момент отказа.

python

```
def run_experiment(mode, num_slaves, slave_latency, write_count, failure_time=None):
    sim = Simulator()
    slaves = [Slave(i, sim, slave_latency) for i in range(num_slaves)]
    master = Master(sim, slaves, mode)
    # Генерация событий записи
    for i in range(write_count):
        master.write(f"key_{i}", f"value_{i}")
    # Запуск симуляции до времени отказа или завершения
    results = []
    for event in sim.run(failure_time if failure_time else float('inf')):
        # обработка событий
        pass
    # Сбор статистики
    return results
```

В реальном коде нужно аккуратно управлять pending и подсчитывать подтверждения. Для методички мы дадим более абстрактное описание, предлагая студенту реализовать собственный вариант.

4.3. Визуализация результатов

Построим графики зависимости средней задержки от числа слейвов и уровня потерь от задержки репликации.

```
python
slave_counts = [1, 2, 3, 4, 5]
sync_lat = [2.1, 4.3, 6.5, 8.2, 10.0]  # условные данные
async_lat = [0.5]*5


plt.plot(slave_counts, sync_lat, 's-', label='Синхронная репликация')
plt.plot(slave_counts, async_lat, 'o-', label='Асинхронная репликация')
plt.xlabel('Число слейвов')
plt.ylabel('Средняя задержка записи (мс)')
plt.legend()
```

```
plt.grid()

plt.show()
```

Аналогично для потерь данных: при асинхронной репликации с ростом задержки репликации и числа операций записи в единицу времени потери растут линейно.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать имитационную модель с 1 мастером и 2 слейвами для обоих режимов репликации. Провести эксперименты с фиксированной задержкой сети (RTT), измерить среднюю задержку записи и количество потерянных записей при отказе мастера для асинхронного режима. Построить сравнительную столбчатую диаграмму задержек.

Вариант	Число слейвов	RTT (мс)	Число записей	Момент отказа (после старта, с)	Режимы для сравнения
1	2	10	100	0.8	sync vs async
2	2	20	150	1.2	--
3	2	5	80	0.5	--
4	3	10	120	1.0	--
5	2	15	200	1.5	--
6	2	8	90	0.7	--
7	2	12	110	0.9	--
8	3	10	130	1.1	--
9	2	25	170	1.4	--

Вариант	Число слейвов	RTT (мс)	Число записей	Момент отказа (после старта, с)	Режимы для сравнения
10	2	6	60	0.4	--
11	2	18	140	1.3	--
12	2	10	100	0.6	--
13	3	15	160	1.2	--
14	2	22	180	1.6	--
15	2	10	200	1.0	--

Требования: корректная симуляция, измерение задержек и потерь, диаграмма, вывод о компромиссе.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Исследовать влияние числа слейвов (от 1 до 5) на задержку синхронной репликации, построить график зависимости средней задержки от числа слейвов.
- Провести эксперимент с переменной задержкой на слейвах (неодинаковые RTT) и измерить, как изменяется задержка синхронного режима (определяется медленнейшим звеном).
- Реализовать **кворумную синхронную репликацию**: запись подтверждается после фиксации на большинстве из S слейвов. Сравнить задержку и потери при отказе одного слота с полной синхронизацией.
- Построить график зависимости объема потерянных данных в асинхронном режиме от величины лага репликации.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	График задержки от числа слейвов, кворумная репликация

Вариант	Параметры базового уровня	Дополнительные задания
2	Вариант 2	Переменные RTT, потери от лага
3	Вариант 3	Кворум, график числа слейвов
4	Вариант 4	Переменные задержки, потери
5	Вариант 5	Кворум, график задержки
6	Вариант 6	Зависимость потерь от лага
7	Вариант 7	Кворум, переменные RTT
8	Вариант 8	График задержки, кворум
9	Вариант 9	Потери от лага, кворум
10	Вариант 10	Кворум, переменные задержки
11	Вариант 11	График числа слейвов, потери
12	Вариант 12	Кворум, переменные RTT
13	Вариант 13	Задержка от числа слейвов, потери
14	Вариант 14	Кворум, графики
15	Вариант 15	Все перечисленные дополнительные пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать модель **полусинхронной репликации** (мастер ждет подтверждения от одного слейва, репликация на остальные асинхронна) и сравнить её с крайними вариантами по задержкам и потерям.

- Смоделировать **реальный поток запросов** (например, на основе пуассоновского процесса) и измерить хвостовые задержки (95-й и 99-й процентили) для синхронного режима.
- Провести анализ влияния восстановления после отказа: после отказа мастера один из слейвов назначается новым мастером; оценить время простоя и потери в зависимости от величины лага.
- Построить трёхмерный график зависимости объёма потерь от интенсивности записи и лага репликации.
- Сравнить полученные результаты с теоретическими моделями (Kapitza, etc.) и дать рекомендации по выбору уровня синхронности.

Вариант	Базовые параметры	Расширенное задание
1	2 слейва, RTT=10	Полусинхронная репликация, хвостовые задержки
2	3 слейва, RTT=20	Пуассоновский поток, время простоя
3	2 слейва, RTT=5	Трёхмерный график потерь, сравнение с теорией
4	4 слейва, RTT=15	Полусинхронная, анализ восстановления
5	2 слейва, RTT=8	Хвостовые задержки, трёхмерный график
6	3 слейва, RTT=12	Простой, теоретическая модель
7	2 слейва, RTT=25	Полусинхронная, пуассоновский поток
8	5 слейвов, RTT=10	Восстановление, трёхмерный график
9	2 слейва, RTT=18	Хвостовые задержки, сравнение с теорией
10	3 слейва, RTT=6	Полусинхронная, трёхмерный
11	2 слейва, RTT=14	Пуассон, восстановление
12	4 слейва, RTT=22	Трёхмерный график потерь, полусинхронная

Вариант	Базовые параметры	Расширенное задание
13	2 слейва, RTT=9	Простой, хвостовые задержки
14	3 слейва, RTT=16	Полусинхронная, анализ
15	2 слейва, RTT=10	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. В чём ключевое различие между синхронной и асинхронной репликацией с точки зрения клиентского подтверждения?
2. Как влияет увеличение числа синхронных реплик на задержку записи и почему?
3. Какой параметр определяет объём потенциально потерянных данных при асинхронной репликации в случае отказа мастера?
4. Какую роль играет кворумная репликация как компромисс между полной синхронностью и полной асинхронностью?
5. Почему асинхронная репликация может быть предпочтительнее в глобально распределённых системах?
6. Что такое lag репликации и как его измерить в реальной системе?
7. Какие механизмы могут уменьшить потери данных при асинхронном подходе (WAL-архивация, внешнее хранилище)?
8. В чём отличие понятий durability и availability применительно к репликации?
9. Как поведёт себя система с синхронной репликацией при отказе одного из слейвов? Что произойдёт с задержкой записи?
10. Приведите примеры промышленных СУБД, использующих синхронную и асинхронную репликацию.

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.

3. Теоретическая часть: описание синхронной и асинхронной репликации, метрики, компромисс.
4. Программная реализация: архитектура симулятора, листинги ключевых классов.
5. Экспериментальная часть:
    - Таблицы измеренных задержек для разных конфигураций.
    - Графики зависимости задержки от числа слейвов и RTT.
    - График зависимости потерь от лага репликации (для асинхронного режима).
    - Сравнительная диаграмма latency vs durability.
    - (для среднего/повышенного уровня) дополнительные графики, результаты кворумной или полусинхронной репликации, гистограммы задержек.
6. Анализ результатов и выводы.
7. Приложение с полным кодом программы.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализованы две модели репликации с 1 мастером и 2 слейвами.
- Проведены замеры задержки записи и потерь данных (для асинхронного режима).
- Построен график/диаграмма, демонстрирующий разницу в задержке.
- Отчёт содержит все обязательные разделы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все пункты базового уровня.
- Исследовано влияние числа слейвов и переменных задержек; реализована кворумная схема.
- Добавлены графики зависимости потерь от лага, задержки от числа слейвов.
- Выводы аргументированы, показан количественный анализ компромиссов.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализована полусинхронная модель, пуассоновский поток, анализ хвостовых задержек и времени восстановления.
- Построены трёхмерные визуализации, даны практические рекомендации.

- Отчёт демонстрирует глубокую проработку темы.

Снижение оценки: некорректная модель задержек, неверный учёт подтверждений, отсутствие сравнения потерь, неработающий код синхронного режима.

Лабораторная работа № 25

Реализация Exponential Backoff с jitter

---

1. Цель работы

Цель работы – изучить стратегию повторных попыток **Exponential Backoff** с применением **полного джиттера (Full Jitter)**, реализовать её программно, провести имитационное тестирование при нестабильном внешнем сервисе и количественно сравнить эффективность с наивными подходами (фиксированная задержка и экспоненциальная задержка без джиттера) по среднему времени достижения успеха, нагрузке на сервис и устойчивости к эффекту «львиной стаи» (thundering herd).

---

2. Задачи

1. Реализовать настраиваемую retry-функцию, поддерживающую стратегии:
    - **Fixed** – фиксированная задержка между попытками.
    - **Exponential Backoff без джиттера** – задержка растёт экспоненциально: delay = min(max_delay, initial * 2^attempt).
    - **Exponential Backoff с полным джиттером** – задержка равна случайной величине, равномерно распределённой в интервале [0, min(max_delay, initial * 2^attempt)].
2. Смоделировать внешний сервис с заданной вероятностью отказа на каждый запрос (например, 30%).
3. Провести имитацию множества клиентов, одновременно пытающихся выполнить операцию, и измерить:
    - среднее время от первой попытки до успеха;
    - общее количество отправленных запросов (нагрузка на сервис);
    - распределение количества выполненных попыток.
4. Сравнить три стратегии между собой по названным метрикам.
5. Для демонстрации «эффекта львиной стаи» смоделировать одновременный сбой сервиса и последующее восстановление; зафиксировать всплеск числа запросов после восстановления при отсутствии джиттера.
6. Построить сравнительные графики времени до успеха и нагрузки, а также гистограмму распределения запросов во времени.

---

3. Теоретическое обоснование

3.1. Назначение механизма повторных попыток

При взаимодействии с удалёнными сервисами временные сбои (network blips, перегрузка сервера, временная недоступность) происходят регулярно. Повторная отправка запроса является стандартным способом обеспечения устойчивости (resilience). Однако бездумные повторы с фиксированным интервалом могут усугубить ситуацию, создавая чрезмерную нагрузку на и без того перегруженный сервис, особенно если множество клиентов одновременно повторяет запросы.

3.2. Exponential Backoff

Стратегия экспоненциальной задержки решает проблему, последовательно увеличивая интервал между попытками. Обычно задержка вычисляется как:

$$\text{delay} = \min(\text{max\_delay}, \text{initial} \times 2^{\text{attempt}})$$

где attempt начинается с 0. Это позволяет снизить частоту попыток при длительных сбоях, давая серверу время на восстановление.

Однако при одновременном сбое большого числа клиентов, использующих идентичные параметры задержки без рандомизации, их повторные попытки синхронизируются, создавая **периодические пики нагрузки** – эффект «львиной стаи» (thundering herd).

3.3. Jitter (джиттер)

Для разрушения синхронизации в задержку добавляется случайная компонента. Наиболее эффективным считается **Full Jitter**, при котором задержка выбирается равномерно из интервала:

$$\text{delay} = \text{random}(0, \min(\text{max\_delay}, \text{initial} \times 2^{\text{attempt}}))$$

Это приводит к экспоненциально растущему **максимальному** возможному интервалу, но конкретное значение случайно. Благодаря полному джиттеру пики нагрузки сглаживаются, распределяясь во времени, а среднее время до успеха остаётся сопоставимым.

3.4. Метрики сравнения

- **Среднее время до успеха (Time To Success, TTS)** – время от первой попытки до получения успешного ответа.
- **Количество запросов (Retry Load)** – общее число отправленных запросов до успеха, характеризует дополнительную нагрузку на сервис.
- **Пиковая нагрузка** – максимальное количество запросов в единицу времени при множественных клиентах; служит индикатором thundering herd.

---

4. Пример практического выполнения

4.1. Реализация retry-функции с настраиваемой стратегией

python

```
import random
import time


class RetryStrategy:
    def get_delay(self, attempt, initial, max_delay):
        raise NotImplementedError


class FixedDelay(RetryStrategy):
    def get_delay(self, attempt, initial, max_delay):
        return initial


class ExponentialBackoff(RetryStrategy):
    def get_delay(self, attempt, initial, max_delay):
        return min(max_delay, initial * (2 ** attempt))


class FullJitter(RetryStrategy):
    def get_delay(self, attempt, initial, max_delay):
        max_possible = min(max_delay, initial * (2 ** attempt))
        return random.uniform(0, max_possible)
```

Универсальная функция, выполняющая запросы с повторами:

python

```
def retry_call(service_func, strategy, initial, max_delay, max_attempts=10):
    """
    service_func: функция без аргументов, возвращает True при успехе, иначе False.
    Возвращает кортеж (успех, суммарное_время, число_попыток).
    """
```

```
    start = time.perf_counter()
    for attempt in range(max_attempts):
        success = service_func()
        if success:
            elapsed = time.perf_counter() - start
            return True, elapsed, attempt + 1
        delay = strategy.get_delay(attempt, initial, max_delay)
        time.sleep(delay)
    elapsed = time.perf_counter() - start
    return False, elapsed, max_attempts
```

4.2. Модель сбоящего сервиса

Создадим функцию-заглушку, имитирующую внешний вызов с вероятностью отказа failure_rate. Чтобы моделировать восстановление, можно динамически менять вероятность.

python

```
class UnreliableService:
    def __init__(self, failure_rate=0.3):
        self.failure_rate = failure_rate

    def call(self):
        return random.random() > self.failure_rate
```

4.3. Эксперимент с одним клиентом

Измерим среднее время до успеха и количество попыток для каждой стратегии при заданных параметрах.

python

```
def single_client_experiment(strategy, initial, max_delay, trials=1000):
    service = UnreliableService(0.3)
    total_time = 0.0
    total_attempts = 0
    for _ in range(trials):
```

```
        ok, t, attempts = retry_call(service.call, strategy, initial, max_delay)
        total_time += t
        total_attempts += attempts
    return total_time / trials, total_attempts / trials


initial = 0.1
max_delay = 5.0
for strat in [FixedDelay(), ExponentialBackoff(), FullJitter()]:
    avg_time, avg_att = single_client_experiment(strat, initial, max_delay)
    print(f"{strat.__class__.__name__}: avg time {avg_time:.3f}s, avg attempts {avg_att:.2f}")
```

Ожидаем, что Full Jitter даёт чуть большее среднее время, чем Exponential без джиттера, но близко к нему, а Fixed Delay — сильнее нагружает сервер при длительных сбоях (если бы вероятность отказа была зависимой от нагрузки).

4.4. Моделирование thundering herd

Создадим симуляцию, где 100 клиентов одновременно сталкиваются с отказом сервиса (вероятность успеха 0% первые 2 секунды, затем 100%). Каждый клиент использует заданную стратегию. Соберём временны́е метки всех запросов и построим гистограмму.

python

```
import matplotlib.pyplot as plt
import threading

event_times = []
lock = threading.Lock()

def client_worker(strategy, initial, max_delay, service):
    for attempt in range(20):
        if service.call():
            with lock:
                event_times.append(time.perf_counter())
            return True, attempt + 1
```

```
        delay = strategy.get_delay(attempt, initial, max_delay)
        time.sleep(delay)
    with lock:
        event_times.append(time.perf_counter())
    return False, 20


class ToggleService:
    def __init__(self, up_after):
        self.start = time.perf_counter()
        self.up_after = up_after
    def call(self):
        if time.perf_counter() - self.start < self.up_after:
            return False
        return True


def run_herd_experiment(strategy, initial, max_delay, clients=100, down_time=2.0):
    service = ToggleService(down_time)
    threads = []
    for _ in range(clients):
        t = threading.Thread(target=client_worker, args=(strategy, initial, max_delay, service))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    return event_times
```

Построим гистограммы времени успешных завершений для ExponentialBackoff и FullJitter.

```
python
for strat in [ExponentialBackoff(), FullJitter()]:
    event_times.clear()
```

```
times = run_herd_experiment(strat, 0.1, 5.0, clients=100)
plt.figure()
plt.hist(times, bins=30, alpha=0.7, label=strat.__class__.__name__)
plt.xlabel('Время от старта (с)')
plt.ylabel('Число успешных запросов')
plt.legend()
plt.grid()
plt.show()
```

График без джиттера будет демонстрировать явные пики на кратных initial интервалах, тогда как с джиттером распределение сглажено.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер в списке группы mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать три стратегии повторных попыток. Провести эксперимент с одним клиентом и заданной вероятностью отказа внешнего сервиса. Измерить среднее время до успеха и среднее количество попыток. Построить сравнительную столбчатую диаграмму метрик.

Вариант	initial (с)	max_delay (с)	failure_rate	max_attempts	Число испытаний
1	0.1	5.0	0.3	10	500
2	0.05	3.0	0.4	12	600
3	0.2	10.0	0.25	8	400
4	0.15	6.0	0.35	10	550
5	0.08	4.0	0.2	15	700
6	0.12	8.0	0.45	10	500

Вариант	initial (с)	max_delay (с)	failure_rate	max_attempts	Число испытаний
7	0.1	5.0	0.5	8	450
8	0.25	12.0	0.3	10	500
9	0.07	3.5	0.28	12	600
10	0.18	7.0	0.33	9	550
11	0.09	4.5	0.22	11	500
12	0.14	6.5	0.38	10	600
13	0.22	9.0	0.27	10	500
14	0.06	2.5	0.32	14	650
15	0.11	5.5	0.4	10	550

Требования к отчёту: работающие реализации стратегий, таблица результатов, график сравнения TTS и retries, выводы.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать стратегию **Decorrelated Jitter** (задержка = min(max_delay, random(initial, last_delay * 3))) и сравнить с Full Jitter.
- Провести эксперимент со «львиной стаей» для 50 одновременных клиентов с заданным временем отключения сервиса (1 секунда) и построить гистограммы распределения запросов во времени для Exponential без джиттера и с Full Jitter.
- Измерить пиковую нагрузку (максимальное количество запросов в скользящем окне 0.1 с) для обеих стратегий.
- Построить график зависимости среднего времени до успеха от начальной задержки initial при фиксированной вероятности отказа.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Decorrelated Jitter, thundering herd
2	Вариант 2	Гистограмма, пиковая нагрузка
3	Вариант 3	Decorrelated Jitter, зависимость от initial
4	Вариант 4	Thundering herd, пиковая нагрузка
5	Вариант 5	Decorrelated Jitter, гистограмма
6	Вариант 6	Пиковая нагрузка, зависимость от initial
7	Вариант 7	Thundering herd, Decorrelated
8	Вариант 8	Гистограмма, зависимость
9	Вариант 9	Decorrelated, пиковая нагрузка
10	Вариант 10	Thundering herd, зависимость
11	Вариант 11	Decorrelated Jitter, гистограмма
12	Вариант 12	Пиковая нагрузка, Decorrelated
13	Вариант 13	Thundering herd, зависимость от initial
14	Вариант 14	Decorrelated, гистограмма
15	Вариант 15	Все доп. пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать модель **Circuit Breaker** + Exponential Backoff: после трёх последовательных неудач клиент переходит в состояние «разомкнут» и не отправляет запросы в течение таймаута. Оценить снижение нагрузки на сервис.
- Сравнить стратегии джиттера в условиях **переменной вероятности отказа**, изменяющейся по синусоидальному закону (имитация флуктуаций доступности). Измерить медианное время до успеха.
- Построить тепловую карту зависимости среднего времени до успеха от failure_rate и initial для Full Jitter.
- Смоделировать **приоритетные повторные запросы**: несколько категорий клиентов с разными начальными задержками, продемонстрировать изоляцию.
- Сравнить Full Jitter с **экспоненциальным backoff без джиттера, но с рандомизированным начальным seed**.

Вариант	Базовые параметры	Расширенное задание
1	initial=0.1, max=5.0	Circuit Breaker, тепловая карта
2	initial=0.05, max=3.0	Переменная вероятность, приоритеты
3	initial=0.2, max=10.0	Тепловая карта, синусоидальный отказ
4	initial=0.15, max=6.0	Circuit Breaker, приоритеты
5	initial=0.08, max=4.0	Переменная вероятность, тепловая карта
6	initial=0.12, max=8.0	Приоритеты, Circuit Breaker
7	initial=0.1, max=5.0	Тепловая карта, синусоидальный отказ
8	initial=0.25, max=12.0	Circuit Breaker, сравнение стратегий
9	initial=0.07, max=3.5	Приоритеты, переменная вероятность
10	initial=0.18, max=7.0	Тепловая карта, Circuit Breaker

Вариант	Базовые параметры	Расширенное задание
11	initial=0.09, max=4.5	Синусоидальный отказ, приоритеты
12	initial=0.14, max=6.5	Тепловая карта, приоритеты
13	initial=0.22, max=9.0	Circuit Breaker, переменная вероятность
14	initial=0.06, max=2.5	Приоритеты, тепловая карта
15	initial=0.11, max=5.5	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Объясните принцип экспоненциальной задержки при повторных попытках. Почему увеличение интервала полезно?
2. В чём заключается эффект «львиной стаи» (thundering herd) и почему джиттер помогает его избежать?
3. Чем полный джиттер (Full Jitter) отличается от декорированного джиттера (Decorrelated Jitter)? Какой из них даёт более равномерное распределение?
4. Как выбор начальной задержки initial влияет на компромисс между быстродействием и нагрузкой на сервер?
5. Что произойдёт, если max_delay установить слишком маленьким? А слишком большим?
6. Почему в стратегии Fixed Delay не возникает эффекта thundering herd? (На самом деле может возникать при одновременном старте, но она лишена адаптивности.)
7. Как можно адаптировать параметры backoff в runtime на основе ответов сервера (например, Retry-After)?
8. Что такое Circuit Breaker и как он взаимодействует с Exponential Backoff?
9. Какие ещё методы предотвращения перегрузок сервера, помимо backoff, применяются в клиентских библиотеках?
10. Приведите примеры известных систем (AWS SDK, gRPC), использующих Exponential Backoff с джиттером.

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание exponential backoff, джиттера, проблема thundering herd.
4. Программная реализация: классы стратегий, код эксперимента.
5. Экспериментальная часть:
    - Таблица сравнения среднего времени до успеха и количества попыток для всех стратегий.
    - Графики распределения запросов во времени (thundering herd).
    - (для среднего/повышенного уровня) дополнительные метрики, зависимость от параметров, тепловые карты.
6. Выводы по работе, обоснование эффективности Full Jitter.
7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализованы все три стратегии, корректно работают.
- Проведён симуляционный эксперимент с одним клиентом, измерены время и попытки.
- Построен столбчатый график сравнения.
- Отчёт содержит необходимые разделы, вывод присутствует.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все пункты базового уровня.
- Добавлен эксперимент с thundering herd, продемонстрирован эффект сглаживания джиттером.
- Измерена пиковая нагрузка, построены гистограммы запросов во времени.
- Сделан вывод о влиянии джиттера на распределение нагрузки.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.

- Реализован Circuit Breaker, переменные вероятности отказов или приоритеты.
- Построены тепловые карты, всесторонний анализ.
- Отчёт демонстрирует глубокое понимание и содержит практические рекомендации.

Снижение оценки: неработающий код, неверная формула задержки, отсутствие джиттера в соответствующей стратегии, недостаточная визуализация.

Лабораторная работа № 26

Реализация Circuit Breaker с тремя состояниями

---

1. Цель работы

Цель работы – изучить паттерн устойчивости **Circuit Breaker** (предохранитель), реализовать его в виде конечного автомата с тремя состояниями (Closed, Open, Half-Open), интегрировать с симулятором нестабильного внешнего сервиса и количественно оценить эффективность в предотвращении каскадных ошибок и ускорении восстановления системы.

---

2. Задачи

1. Реализовать класс CircuitBreaker со следующими элементами:
    - **Состояния**: CLOSED, OPEN, HALF_OPEN.
    - **Параметры**: порог ошибок (failure_threshold), время ожидания в открытом состоянии (timeout), количество пробных запросов в полуоткрытом состоянии (half_open_max_requests).
    - **Логика переходов**:
        - В CLOSED каждое неудачное выполнение защищаемой функции увеличивает счётчик ошибок; при достижении порога автомат переходит в OPEN.
        - В OPEN все вызовы немедленно возвращают fallback-значение; по истечении timeout автомат переходит в HALF_OPEN.
        - В HALF_OPEN ограниченное число запросов пропускается к реальному сервису; если хотя бы один из них завершается ошибкой, автомат возвращается в OPEN; если все пробные запросы успешны – в CLOSED.
2. Создать имитатор внешнего сервиса с изменяющейся надёжностью (вероятность успеха).
3. Провести эксперимент, в котором клиент выполняет серию запросов к сервису через Circuit Breaker, в то время как сервис временно становится недоступным.
4. Измерить:
    - Количество успешных запросов, количество отклонённых запросов (fallback), количество реальных вызовов сервиса.

- Время восстановления сервиса (от момента восстановления работоспособности сервиса до первого успешного ответа через Circuit Breaker).
- Общее количество ошибок, достигших клиента.

5. Сравнить с прямыми вызовами сервиса без Circuit Breaker (когда каждая ошибка сервиса возвращается клиенту и не ограничивается).
6. Построить график изменения состояния Circuit Breaker во времени и накопленных метрик.

---

3. Теоретическое обоснование

3.1. Проблема каскадных отказов

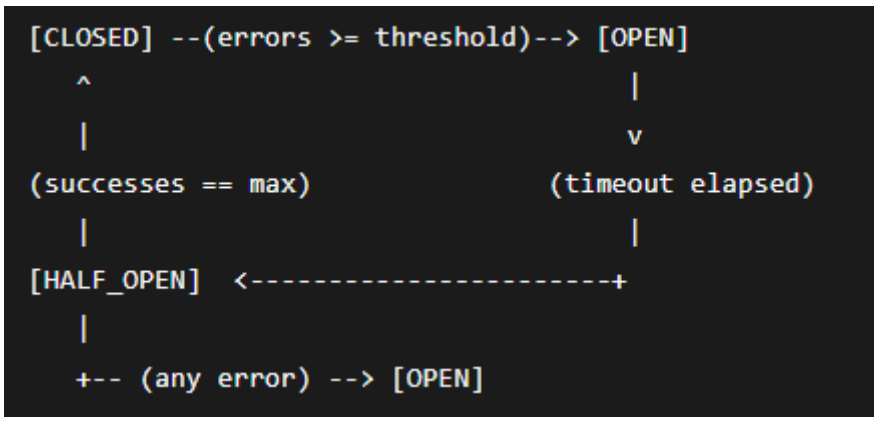
В распределённых системах зависимость одного сервиса от другого может привести к цепной реакции сбоев: если внешний сервис начинает отвечать с ошибками или с высокой задержкой, ресурсы вызывающего сервиса (потоки, соединения) истощаются, что ведёт к отказу уже самого вызывающего сервиса. Паттерн **Circuit Breaker** предотвращает такие каскадные отказы, временно прекращая вызовы проблемного сервиса и давая ему возможность восстановиться.

3.2. Конечный автомат Circuit Breaker

Классическая модель предохранителя, популяризированная Майклом Найгардом в книге «Release It!», представляет собой конечный автомат с тремя состояниями:

- **Closed (Замкнутое)** – нормальный режим работы. Все запросы передаются сервису. Ведётся подсчёт последовательных или накопленных неудач. Если счётчик достигает порога failure_threshold, автомат переключается в состояние **Open**.
- **Open (Разомкнутое)** – запросы немедленно отклоняются без вызова сервиса (возвращается fallback-ответ или исключение). Это позволяет «разгрузить» и защитить вызывающую сторону, а также дать сервису время на восстановление. Через заданный интервал timeout автомат переходит в **Half-Open**.
- **Half-Open (Полуоткрытое)** – разрешается ограниченное количество пробных запросов к сервису (half_open_max_requests). Если все они завершаются успешно, автомат возвращается в **Closed** (сервис считается восстановленным). Если хотя бы один отказывает – автомат снова переходит в **Open** (проблема сохраняется).

Переходы можно представить диаграммой:



3.3. Параметры настройки

- **failure_threshold** – количество последовательных ошибок (или ошибок за интервал), необходимое для размыкания. Низкое значение повышает чувствительность, но может приводить к ложным размыканиям.
- **timeout** – время нахождения в Open до попытки восстановления. Определяет максимальную задержку восстановления после возврата сервиса в рабочее состояние.
- **half_open_max_requests** – количество пробных запросов; обычно устанавливается равным 1 или небольшому числу, чтобы не перегрузить только восстанавливающийся сервис.

3.4. Сравнение с прямыми вызовами без защиты

При отсутствии Circuit Breaker клиент будет продолжать вызывать сбойный сервис, накапливая ошибки, потребляя ресурсы и потенциально обрушивая собственную систему. Circuit Breaker, напротив, изолирует сбойный компонент, быстро возвращая fallback и восстанавливаясь при нормализации сервиса.

3.5. Метрики оценки

- **Количество успешных запросов** (за определённый период).
- **Количество отказов с fallback** (запросы, отклонённые в состоянии Open).
- **Общее число реальных вызовов сервиса** (снижение нагрузки на проблемный сервис).
- **Время восстановления** – от момента фактического восстановления сервиса до регистрации первого успешного запроса клиентом.

---

4. Пример практического выполнения

4.1. Класс Circuit Breaker на Python

python

```
import time
import enum
import functools
from collections import deque

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

class CircuitBreaker:
    def __init__(self, failure_threshold=3, timeout=2.0, half_open_max=2):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.half_open_max = half_open_max
        self.state = State.CLOSED
        self.error_count = 0
        self.last_failure_time = 0.0
        self.half_open_successes = 0
        self.half_open_requests = 0

    def call(self, func, *args, **kwargs):
        if self.state == State.OPEN:
            if time.monotonic() - self.last_failure_time >= self.timeout:
                self.state = State.HALF_OPEN
                self.half_open_requests = 0
                self.half_open_successes = 0
            else:
                # Отклоняем, возвращаем fallback
```

```
            return self._fallback()

        if self.state == State.HALF_OPEN:
            if self.half_open_requests >= self.half_open_max:
                # Превышен лимит пробных запросов; возможна ситуация гонки, но для учебного примера просто отклоняем
                return self._fallback()
            self.half_open_requests += 1

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result
        except Exception as e:
            self._on_error()
            raise e

    def _on_success(self):
        if self.state == State.CLOSED:
            self.error_count = 0
        elif self.state == State.HALF_OPEN:
            self.half_open_successes += 1
            if self.half_open_successes >= self.half_open_max:
                self.state = State.CLOSED
                self.error_count = 0

    def _on_error(self):
        if self.state == State.CLOSED:
            self.error_count += 1
```

```
            if self.error_count >= self.failure_threshold:
                self.state = State.OPEN
                self.last_failure_time = time.monotonic()
        elif self.state == State.HALF_OPEN:
            self.state = State.OPEN
            self.last_failure_time = time.monotonic()

    def _fallback(self):
        # Возвращаем fallback-ответ (None или значение по умолчанию)
        return None
```

4.2. Симулятор нестабильного сервиса

Создадим класс, у которого метод process с заданной вероятностью возвращает успех или бросает исключение, и умеет отключаться/включаться по команде.

```
python

import random


class UnstableService:
    def __init__(self, success_probability=0.7):
        self.success_prob = success_probability
        self.online = True

    def set_online(self, status):
        self.online = status

    def process(self, request):
        if not self.online or random.random() > self.success_prob:
            raise Exception("Service failed")
        return f"Response for {request}"
```

4.3. Эксперимент и сбор метрик

Имитируем поток запросов. Сервис работает стабильно первые 5 секунд, затем на 3 секунды переходит в оффлайн, потом снова становится доступным. Клиент отправляет запросы каждые 0.1 секунды.

python

```
import matplotlib.pyplot as plt
import time


def run_experiment(cb, service, duration=15.0, request_interval=0.1):
    history = []
    start_time = time.monotonic()
    t = start_time

    # Сценарий: сервис доступен до t=5, затем выключается до t=8, затем снова доступен
    service_down_time = 5.0
    service_up_again_time = 8.0

    while t - start_time < duration:
        now = t
        elapsed = now - start_time
        # Управление состоянием сервиса
        if elapsed < service_down_time:
            service.set_online(True)
        elif elapsed < service_up_again_time:
            service.set_online(False)
        else:
            service.set_online(True)

        try:
            result = cb.call(service.process, f"req-{elapsed:.2f}")
            history.append((elapsed, cb.state, 'SUCCESS', result))
```

```
            except Exception as e:
                history.append((elapsed, cb.state, 'ERROR', str(e)))
            except:  # fallback от CB
                history.append((elapsed, cb.state, 'FALLBACK', None))

            # ожидание до следующего запроса
            next_time = t + request_interval
            sleep_seconds = max(0, next_time - time.monotonic())
            time.sleep(sleep_seconds)
            t = next_time

    return history
```

Анализ метрик:

```
def analyze(history):
    successful = sum(1 for h in history if h[2] == 'SUCCESS')
    fallbacks = sum(1 for h in history if h[2] == 'FALLBACK')
    errors = sum(1 for h in history if h[2] == 'ERROR')
    return successful, fallbacks, errors

# Без Circuit Breaker (прямые вызовы) для сравнения: просто вызываем service.process и ловим исключения.
```

Построение графика состояний:

```
def plot_state_history(history):
    times = [h[0] for h in history]
    states = [h[1].value for h in history]  # 0 CLOSED, 1 OPEN, 2 HALF_OPEN
    plt.figure(figsize=(10,4))
    plt.step(times, states, where='post')
```

```
plt.yticks([0,1,2], ['CLOSED','OPEN','HALF_OPEN'])
plt.xlabel('Время (с)')
plt.ylabel('Состояние')
plt.title('Переходы состояний Circuit Breaker')
plt.grid(True)
plt.show()
```

4.4. Визуализация метрик

Добавим второй график с накопленным количеством успехов, ошибок и fallback'ов.

В выводах показать, что Circuit Breaker значительно снижает количество ошибок, достигающих клиента, и быстро восстанавливается после возврата сервиса.

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать Circuit Breaker с настраиваемыми параметрами и симулятор нестабильного сервиса. Провести эксперимент с периодическим отключением сервиса по варианту. Измерить число успешных, fallback- и ошибочных запросов, время восстановления (от момента включения сервиса до первого SUCCESS после HALF_OPEN). Построить график состояния CB во времени.

Вариант	failure_threshold	timeout (с)	half_open_max	Длительность запросов	Сценарий отключения сервиса
1	3	2.0	1	0.05 с	сервис недоступен на интервале [4, 7] с
2	2	1.5	2	0.1 с	отключение на [3, 5] с

Вариант	failure_threshold	timeout (c)	half_open_max	Длительность запросов	Сценарий отключения сервиса
3	4	3.0	1	0.08 с	отключение на [5, 9] с
4	5	2.5	3	0.12 с	отключение на [6, 10] с
5	3	1.0	1	0.06 с	отключение на [2, 4] с
6	2	2.0	2	0.1 с	отключение на [5, 8] с
7	4	2.5	2	0.15 с	отключение на [3, 6] с
8	6	4.0	1	0.07 с	отключение на [7, 12] с
9	3	3.0	3	0.09 с	отключение на [4, 7] с
10	5	2.0	1	0.11 с	отключение на [5, 8] с
11	2	2.5	2	0.04 с	отключение на [1, 3] с
12	4	1.8	1	0.13 с	отключение на [4, 6] с
13	3	2.2	2	0.14 с	отключение на [3, 5] с

Вариант	failure_threshold	timeout (c)	half_open_max	Длительность запросов	Сценарий отключения сервиса
14	5	3.5	3	0.16 с	отключение на [6, 11] с
15	3	2.0	2	0.1 с	отключение на [4, 7] и повтор на [10, 11] с

Требования к отчёту: работающий код, таблица с числом SUCCESS, FALLBACK, ERROR; график состояний; вывод о преимуществах CB.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Реализовать статистику в скользящем окне (например, ошибки за последние 10 секунд) вместо простого подсчёта последовательных ошибок. Переход в OPEN, если частота ошибок превышает порог.
- Провести эксперимент с переменной длительностью запросов (случайная задержка ответа) и добавить **таймаут вызова**: если сервис не ответил за отведённое время, считать ошибкой. Измерить влияние таймаута на метрики.
- Построить сравнительные графики для двух разных стратегий сброса ошибок (счётчик сбрасывается после одного успеха vs сбрасывается после скользящего окна).
- Провести серию из 20 прогонов и оценить среднее время восстановления с 95% доверительным интервалом.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Скользящее окно, таймаут
2	Вариант 2	Сравнение стратегий сброса, доверительный интервал

Вариант	Параметры базового уровня	Дополнительные задания
3	Вариант 3	Скользящее окно, замеры времени
4	Вариант 4	Таймаут, доверительный интервал
5	Вариант 5	Сравнение сбросов, скользящее окно
6	Вариант 6	Таймаут, графики
7	Вариант 7	Скользящее окно, доверительный интервал
8	Вариант 8	Сравнение стратегий отказов
9	Вариант 9	Таймаут, скользящее окно
10	Вариант 10	Доверительный интервал
11	Вариант 11	Скользящее окно, таймаут
12	Вариант 12	Сравнение стратегий
13	Вариант 13	Таймаут, доверительный интервал
14	Вариант 14	Скользящее окно
15	Вариант 15	Все доп. пункты

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать адаптивный порог срабатывания: failure_threshold динамически изменяется в зависимости от наблюдаемой частоты успешных запросов (например, снижается при долгом периоде стабильности).
- Моделировать несколько экземпляров Circuit Breaker для разных конечных точек с общим пулом потоков и измерить влияние изоляции.

- Построить временну́ю диаграмму (swimlane) с детализацией каждого запроса: когда он был отправлен, получен ответ, ошибка или fallback. Использовать matplotlib для цветных полос.
- Сравнить Circuit Breaker с паттерном **Retry with Backoff** по интегральной метрике «полезная работа» (количество успешных ответов на единицу времени).

Вариант	Базовые параметры	Расширенное задание
1	threshold=3, timeout=2.0	Адаптивный порог, swimlane
2	threshold=2, timeout=1.5	Несколько CB, сравнение с Retry
3	threshold=4, timeout=3.0	Адаптивный порог, интегральная метрика
4	threshold=5, timeout=2.5	Swimlane, несколько сервисов
5	threshold=3, timeout=1.0	Retry с Backoff, адаптивный порог
6	threshold=2, timeout=2.0	Swimlane, сравнение
7	threshold=4, timeout=2.5	Адаптивный порог, несколько CB
8	threshold=6, timeout=4.0	Retry, swimlane
9	threshold=3, timeout=3.0	Адаптивный, интегральная метрика
10	threshold=5, timeout=2.0	Swimlane, несколько сервисов
11	threshold=2, timeout=2.5	Retry, адаптивный порог
12	threshold=4, timeout=1.8	Swimlane, интегральная метрика
13	threshold=3, timeout=2.2	Адаптивный, несколько CB
14	threshold=5, timeout=3.5	Retry, swimlane

Вариант	Базовые параметры	Расширенное задание
15	threshold=3, timeout=2.0	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Какие три состояния определяют поведение Circuit Breaker? Опишите назначение каждого.
2. Почему Circuit Breaker не должен мгновенно переходить из Open в Closed, а использует состояние Half-Open?
3. Каков компромисс между чувствительностью (низкий failure_threshold) и устойчивостью к кратковременным всплескам ошибок?
4. Какие метрики в реальной системе могут служить сигналом для размыкания предохранителя, кроме явных ошибок (например, высокие задержки)?
5. Что произойдёт, если таймаут timeout установить равным очень большому значению? Как это повлияет на время восстановления?
6. Чем Circuit Breaker отличается от обычного механизма повторных попыток (retry)?
7. Как можно реализовать автоматический сброс счётчика ошибок в Closed, чтобы избежать ложных срабатываний при редких одиночных ошибках?
8. В чём преимущество использования скользящего окна для подсчёта ошибок перед последовательным счётчиком?
9. Какие опасности возникают при использовании Circuit Breaker в распределённой системе с несколькими экземплярами одного сервиса (разделение состояний CB)?
10. Приведите примеры библиотек и фреймворков, реализующих Circuit Breaker (Hystrix, Resilience4j, Polly). Какие дополнительные возможности они предоставляют?

---

7. Требования к отчёту

Отчёт оформляется в соответствии с общими стандартами и должен включать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическую часть: принцип работы, диаграмма состояний, параметры.

4. Программную реализацию: листинг класса CircuitBreaker, симулятора сервиса, экспериментальной установки.
5. Экспериментальную часть:
    - Таблицу с числом успешных, fallback и ошибочных запросов для случаев с CB и без него.
    - График изменения состояний CB во времени.
    - (для среднего/повышенного уровня) дополнительные графики, результаты скользящего окна, таймаутов, адаптивного порога.
6. Анализ результатов, сравнение, объяснение поведения.
7. Выводы: преимущества Circuit Breaker, сценарии использования, рекомендации по настройке.
8. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно реализован автомат с тремя состояниями согласно спецификации.
- Проведён эксперимент с имитацией сбоев, сняты основные метрики.
- Построен график состояний, сформулированы выводы о снижении ошибок.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Реализован подсчёт ошибок по скользящему окну или добавлен механизм таймаута вызова.
- Проведено сравнение различных стратегий, представлены доверительные интервалы.
- Показано влияние параметров на метрики.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализован адаптивный порог, многопоточная модель с несколькими CB, или детальная временна́я диаграмма.
- Проведено сравнение с Retry-стратегиями, интегральные метрики.
- Отчёт демонстрирует глубокое понимание паттерна и его инженерных аспектов.

Снижение оценки: неверная логика переходов (например, возврат из Half-Open в Closed без проверки всех пробных запросов), отсутствие fallback, некорректная обработка таймаута.

Лабораторная работа № 27

Реализация адаптивного таймаута (по процентилям)

---

1. Цель работы

Цель работы — изучить методы динамической настройки таймаутов на основе наблюдаемых задержек, реализовать алгоритм адаптивного таймаута, вычисляемого по скользящему окну измеренных времен ответа с использованием процентилей P_{95} и P_{99}, и экспериментально сравнить его эффективность со статическим таймаутом в условиях изменяющейся производительности внешнего сервиса.

---

2. Задачи

1. Разработать компоненту сбора статистики времени ответа внешнего сервиса, поддерживающую скользящее окно фиксированного размера (например, 1000 последних запросов).
2. Реализовать вычисление заданных процентилей (P_{50}, P_{95}, P_{99}) на основе накопленной гистограммы или отсортированного буфера.
3. Реализовать логику адаптивного таймаута:

$$\text{timeout} = \min(\max\_timeout, \text{процентиль} \times \text{coefficient}).$$

4. Создать симулятор внешнего сервиса с изменяющимся временем ответа: стабильная работа, плавное возрастание задержки, резкие выбросы, восстановление.
5. Провести нагрузочное тестирование, измеряя:
    - количество ложных таймаутов (превышение динамического или статического порога, когда сервис ещё обрабатывает запрос);
    - среднее время ожидания клиента (включая таймауты и повторные попытки);
    - долю успешно обслуженных запросов.
6. Сравнить адаптивный таймаут со статическим фиксированным значением (например, 1000 мс) в описанных сценариях.
7. Построить графики изменения таймаута во времени, распределения задержек и процентилей.

---

3. Теоретическое обоснование

3.1. Роль таймаутов в распределённых системах

Таймаут определяет максимальное время, в течение которого клиент готов ожидать ответа от удалённого сервиса. Слишком короткий таймаут ведёт к ложным отказам и напрасным повторным запросам, увеличивая нагрузку на систему. Слишком длинный таймаут задерживает обнаружение реальных отказов и ухудшает пользовательский опыт. В реальных условиях время ответа сервиса меняется из-за вариаций нагрузки, сетевых задержек, сборки мусора и т. п., что делает статический таймаут неоптимальным.

3.2. Адаптивный таймаут на основе процентилей

Идея динамического таймаута состоит в непрерывном отслеживании фактического распределения времени ответа и установке порога как величины, кратной выбранному процентилю. Например,

$$\text{timeout} = K \cdot P_{95}$$

где K — коэффициент запаса (обычно 1.2–2.0). Такой подход автоматически адаптируется к изменениям задержек: при ухудшении сети таймаут увеличивается, предотвращая массовые ложные отказы, а при улучшении — уменьшается, позволяя быстрее детектировать реальные сбои.

Для вычисления процентилей в потоковом режиме применяются:

- **Сортированный массив последних W задержек** (W = 1000) с вычислением перцентиля за $O(W)$ (в учебных целях приемлемо).
- **Гистограммные приближения** (например, t-digest, HdrHistogram) для константной памяти.

В настоящей работе мы реализуем простейший вариант: кольцевой буфер последних W задержек, при необходимости сортируемый для оценки процентиля. Чтобы избежать сортировки на каждом запросе, обновлять процентиль можно периодически, например, каждые 100 запросов.

3.3. Метрики оценки

- **False Timeout Rate** — доля запросов, для которых истёк таймаут, но сервис фактически ответил (измеряется в симуляторе путём сравнения реальной длительности обработки с таймаутом).
- **Client Waiting Time** — время от отправки запроса до получения ответа или до истечения таймаута (в случае таймаута обычно плюс время на retry, но мы будем считать чистое время ожидания).
- **Success Ratio** — доля запросов, завершившихся успехом (получен ответ до таймаута).

- **Adaptation Speed** — скорость, с которой таймаут подстраивается под новое распределение задержек.

---

4. Пример практического выполнения

4.1. Сбор статистики задержек и вычисление процентилей

python

```
import collections
import math
import random
import numpy as np


class LatencyWindow:
    def __init__(self, window_size=1000):
        self.buffer = collections.deque(maxlen=window_size)


    def add(self, latency):
        self.buffer.append(latency)


    def percentile(self, p):
        if not self.buffer:
            return 1.0   # дефолтный таймаут 1 мс?
        # Сортируем копию для вычисления процентиля
        sorted_lat = sorted(self.buffer)
        k = (len(sorted_lat) - 1) * p / 100.0
        f = math.floor(k)
        c = math.ceil(k)
        if f == c:
            return sorted_lat[int(k)]
        d0 = sorted_lat[int(f)] * (c - k)
```

```
        d1 = sorted_lat[int(c)] * (k - f)
        return d0 + d1
```

4.2. Адаптивный таймаут

```
python
class AdaptiveTimeout:
    def __init__(self, window_size=1000, percentile=95, coefficient=1.5, max_timeout=5000, min_timeout=10):
        self.window = LatencyWindow(window_size)
        self.percentile = percentile
        self.coefficient = coefficient
        self.max_timeout = max_timeout
        self.min_timeout = min_timeout
        self.current_timeout = max_timeout  # начальное значение

    def update(self, latency):
        self.window.add(latency)
        # пересчитываем таймаут периодически, например, каждые 100 добавлений
        if len(self.window.buffer) % 100 == 0:
            p_val = self.window.percentile(self.percentile)
            self.current_timeout = min(self.max_timeout, max(self.min_timeout, p_val * self.coefficient))

    def get_timeout(self):
        return self.current_timeout
```

4.3. Симулятор внешнего сервиса с изменяющейся задержкой

```
python
class SimulatedService:
    def __init__(self):
        self.base_latency = 50.0   # мс
        self.noise_std = 10.0
```

```
        self.extra_load = 0.0      # добавка из-за перегрузки

    def set_extra_load(self, load):
        self.extra_load = load

    def call(self):
        # Имитация обработки запроса
        latency = abs(random.gauss(self.base_latency + self.extra_load, self.noise_std))
        # Имитация выбросов (редкие очень большие задержки)
        if random.random() < 0.01:  # 1% выбросы
            latency += random.expovariate(1/200)  # добавочная задержка ~200 мс
        return latency
```

4.4. Эксперимент

Клиент отправляет запросы с интервалом, дожидается ответа (но не дольше таймаута). Если таймаут истёк, фиксируется ложный отказ, но для чистоты симуляции мы записываем, что сервис ответил через latency мс, а клиент прервал ожидание на timeout мс. Собираем метрики.

```
python

def run_scenario(adaptive, static_timeout, service, total_requests=2000):
    adaptive_timeouts = []
    static_timeouts = static_timeout
    adaptive_waits = []
    static_waits = []
    adaptive_false_timeouts = 0
    static_false_timeouts = 0

    # Меняем нагрузку сервиса в середине теста
    for i in range(total_requests):
        if i == 800:
            service.set_extra_load(100.0)   # задержки выросли
```

```
        elif i == 1400:
            service.set_extra_load(0.0)    # восстановились

        # Адаптивный клиент
        latency = service.call()
        adaptive_timeout = adaptive.get_timeout()
        adaptive_timeouts.append(adaptive_timeout)
        if latency <= adaptive_timeout:
            adaptive_waits.append(latency)
        else:
            adaptive_false_timeouts += 1
            adaptive_waits.append(adaptive_timeout)  # ждал timeout и ушёл
        adaptive.update(latency)

        # Статический клиент
        static_to = static_timeout
        if latency <= static_to:
            static_waits.append(latency)
        else:
            static_false_timeouts += 1
            static_waits.append(static_to)

    return (adaptive_timeouts, adaptive_waits, adaptive_false_timeouts,
            static_waits, static_false_timeouts)
```

Построение графиков:

```
python
import matplotlib.pyplot as plt

timeouts_hist, ad_waits, ad_ft, st_waits, st_ft = run_scenario(...)
```

```
plt.figure(figsize=(12,6))
plt.subplot(2,1,1)
plt.plot(timeouts_hist)
plt.ylabel('Adaptive Timeout (ms)')
plt.subplot(2,1,2)
plt.hist(ad_waits, bins=40, alpha=0.7, label='Adaptive')
plt.hist(st_waits, bins=40, alpha=0.7, label='Static')
plt.legend()
plt.show()
```

Анализ: адаптивный таймаут следует за изменениями задержек, число ложных таймаутов значительно ниже, чем у статического в период медленного сервиса, а в быстром — таймаут снижается, ускоряя обнаружение отказов.

---

5. Задания для индивидуального выполнения

Номер варианта вычисляется по формуле: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать сборщик статистики (окно 500 задержек) и адаптивный таймаут по P_{95} с коэффициентом 1.5. Провести симуляцию для заданного сценария изменения задержки, измерить количество ложных таймаутов и среднее время ожидания для адаптивного и статического (фиксированного) таймаута. Построить график изменения адаптивного таймаута во времени.

Вариант	Статический таймаут (мс)	Окно	Процентиль	Коэф.	Сценарий задержки
1	1000	500	95	1.5	стабильно 200 мс, затем всплеск до 800 мс на 30% запросов
2	800	500	95	1.2	плавный рост с 100 до 500 мс за 1000 запросов

Вариант	Статический таймаут (мс)	Окно	Процентиль	Коэф.	Сценарий задержки
3	1200	500	99	1.5	стабильно 300 мс, затем 10% выбросов до 2000 мс
4	500	500	90	1.8	низкая база 50 мс, резкое увеличение до 400 мс
5	1500	500	95	1.3	синусоидальное изменение нагрузки 200±150 мс
6	1000	500	95	2.0	ступенчатое: 100 мс, 300 мс, 100 мс через 600 запросов
7	900	500	95	1.5	стабильно, затем единичные выбросы 5000 мс (0.5%)
8	1100	500	95	1.4	плавное снижение с 600 до 100 мс
9	750	500	99	1.5	300 мс с шумом, затем внезапный скачок до 700 мс
10	1300	500	95	1.6	постоянные колебания 200–400 мс
11	600	500	90	1.5	50 мс с редкими, но длительными задержками (2000 мс)

Вариант	Статический таймаут (мс)	Окно	Процентиль	Коэф.	Сценарий задержки
12	1000	500	95	1.2	рост с 150 до 800 мс за 500 запросов, потом резкий спад
13	850	500	95	1.5	стабильно 250 мс, внезапный провал до 50 мс
14	950	500	95	1.3	два пика задержек: на 400-м и 800-м запросе
15	1050	500	95	1.5	случайное блуждание задержки (Random Walk)

Требования к отчёту: работающий код сборщика и адаптивного таймаута, таблица сравнения ложных таймаутов и среднего ожидания, график изменения таймаута в ходе эксперимента.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать вычисление **нескольких процентилей** (P_{90}, P_{95}, P_{99}) и сравнительный анализ, какой из них даёт наилучший баланс между ложными таймаутами и временем ожидания.
- Исследовать влияние размера скользящего окна (200, 500, 1000, 2000) на скорость реакции и стабильность таймаута. Построить график зависимости количества ложных таймаутов от размера окна.
- Добавить механизм **быстрого реагирования на резкие выбросы**: если задержка превышает текущий таймаут в несколько раз, окно не пополняется этим значением (чтобы не искажать статистику), а счётчик таких выбросов увеличивается. При превышении порога таймаут форсированно увеличивается.
- Сравнить статический таймаут, адаптивный таймаут без защиты от выбросов и адаптивный с защитой на сценарии с экстремальными выбросами (latency spike).

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Сравнение процентилей, влияние окна
2	Вариант 2	Механизм быстрого реагирования
3	Вариант 3	P90 vs P95 vs P99
4	Вариант 4	Влияние размера окна, защита от выбросов
5	Вариант 5	Сравнение стратегий
6	Вариант 6	P95 vs P99, окно 200/500/1000
7	Вариант 7	Быстрое реагирование
8	Вариант 8	Процентили, влияние окна
9	Вариант 9	Защита от выбросов, сравнение
10	Вариант 10	P90 vs P95, размер окна
11	Вариант 11	Быстрое реагирование на спайки
12	Вариант 12	Сравнение процентилей, защита
13	Вариант 13	Размер окна, адаптация к снижению задержки
14	Вариант 14	P95/P99, сравнение
15	Вариант 15	Все перечисленные дополнительные задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **гистограммный метод** (например, t-digest или простая логарифмическая гистограмма) для аппроксимации процентилей с константной памятью. Сравнить точность с точным вычислением по окну.
- Смоделировать несколько одновременно работающих клиентов с независимыми адаптивными таймаутами, но общим симулируемым сервером, чья задержка зависит от количества одновременных запросов (queuing). Исследовать стабильность системы.
- Провести анализ джиттера таймаута и его влияния на пропускную способность. Построить 3D-график зависимости ложных таймаутов от размера окна и коэффициента.
- Сравнить адаптивный таймаут с алгоритмом **TCP-style RTT estimation** (экспоненциальное скользящее среднее и отклонение, подобно Jacobson/Karels). Применить формулу timeout = SRTT + 4 * RTTVAR.

Вариант	Базовые параметры	Расширенное задание
1	P95, окно 500	Гистограммный метод, сравнение с TCP RTT
2	P99, окно 1000	Множественные клиенты, 3D-график
3	P95, окно 500	TCP RTT estimation, сравнение
4	P95, окно 1000	Гистограммный метод, 3D-график
5	P90, окно 500	Множественные клиенты с очередями
6	P95, окно 500	Сравнение TCP RTT, 3D-график
7	P99, окно 1000	Гистограммный метод, множественные клиенты
8	P95, окно 500	TCP RTT, 3D-график
9	P95, окно 1000	Гистограммы, сравнение
10	P95, окно 500	Множественные клиенты, TCP RTT

Вариант	Базовые параметры	Расширенное задание
11	P99, окно 500	3D-график, гистограммный метод
12	P95, окно 1000	TCP RTT, множественные клиенты
13	P95, окно 500	Гистограммы, 3D-график
14	P90, окно 500	TCP RTT, сравнение
15	P95, окно 500	Комбинация расширенных задач

---

6. Контрольные вопросы

1. Почему статический таймаут часто неэффективен в условиях переменной сетевой задержки?
2. Какие процентили (P_{95}, P_{99}) обычно используются для адаптивных таймаутов и почему?
3. Как размер скользящего окна влияет на чувствительность и стабильность вычисляемого таймаута?
4. В чём опасность учёта экстремальных выбросов при расчёте процентиля? Предложите способы защиты.
5. Каким образом адаптивный таймаут помогает быстрее обнаруживать реальные отказы по сравнению со статическим?
6. Опишите алгоритм вычисления процентиля по гистограмме. В чём его преимущество перед точным вычислением по полному набору данных?
7. Как можно реализовать асимметричную адаптацию: быстрое увеличение таймаута при всплеске задержек и медленное уменьшение при восстановлении?
8. Что такое TCP RTT estimation (Jacobson's algorithm) и как он соотносится с процентильным подходом?
9. Какие дополнительные метрики, помимо задержки, можно использовать для динамической настройки таймаута (например, очередь запросов на сервере)?
10. Приведите примеры библиотек и фреймворков, в которых реализован адаптивный таймаут (gRPC, Finagle, Netflix Hystrix).

---

7. Требования к отчёту

Отчёт должен включать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: понятие адаптивного таймаута, методы оценки процентилей, сравнительный анализ со статическим подходом.
4. Программная реализация: описание классов сборщика задержек, адаптивного таймаута, симулятора.
5. Экспериментальная часть:
    - Графики изменения таймаута во времени для заданного сценария.
    - Таблица сравнения ложных таймаутов и среднего времени ожидания для адаптивного и статического таймаутов.
    - (для среднего/повышенного уровня) влияние размера окна, сравнение процентилей, графики 3D.
6. Анализ результатов и выводы о преимуществах адаптивного подхода.
7. Приложение с полным исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован сборщик задержек и адаптивный таймаут по P_{95}.
- Проведена симуляция, измерены ложные таймауты и среднее ожидание.
- Построен график изменения таймаута во времени.
- Отчёт содержит основные разделы и выводы.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Проведено сравнение нескольких процентилей или размеров окна.
- Добавлен механизм быстрого реагирования на выбросы, показана его эффективность.
- Аргументированные выводы с опорой на количественные данные.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.

- Реализован гистограммный метод или TCP RTT estimation, проведено сравнение.
- Исследовано поведение в многопользовательской среде.
- Глубокий анализ, практические рекомендации, качественная визуализация.

Снижение оценки: неверный расчёт процентиля, отсутствие адаптации (таймаут не меняется), некорректное сравнение со статическим порогом.

Лабораторная работа № 28

Комплексный проект: отказоустойчивый клиент

---

1. Цель работы

Цель работы — объединить изученные ранее паттерны устойчивости (**Retry с Exponential Backoff и Jitter**, **Circuit Breaker**, **Fallback**) в едином отказоустойчивом клиенте, исследовать их взаимодействие в реалистичных сценариях отказов внешнего сервиса, измерить интегральные метрики (время восстановления, нагрузка на сервис, покрытие fallback-ом) и выработать рекомендации по настройке параметров для различных условий эксплуатации.

---

2. Задачи

1. Разработать класс ResilientClient, инкапсулирующий:
    - **Retry-механизм** с экспоненциальной задержкой и полным джиттером (из ЛР25);
    - **Circuit Breaker** с тремя состояниями (из ЛР26);
    - **Fallback-кэш**, сохраняющий последний успешный ответ и возвращающий его в случае отказа.
2. Порядок обработки вызова:
    - Если Circuit Breaker разомкнут — сразу отдаётся fallback-ответ.
    - Иначе выполняется защищённый вызов с заданной стратегией retry; при превышении числа попыток или исключении фиксируется ошибка в CB и через fallback возвращается закэшированное значение.
3. Реализовать симулятор внешнего HTTP-сервиса с управляемыми параметрами: базовая задержка, вероятность ошибки, временные отключения, деградация.
4. Провести испытания в трёх сценариях:
    - **Временный сбой** (перезапуск сервиса: кратковременная недоступность ~2 с).
    - **Полный отказ** сервиса на длительное время.
    - **Высокая задержка**, превышающая адаптивный таймаут (p95 > timeout), но без потерь пакетов.
5. Измерить и визуализировать:
    - Время восстановления сервиса (от начала сбоя до первого успешного ответа клиента).
    - Суммарное количество отправленных запросов (Retry Cost).

- Долю запросов, обслуженных из fallback-кэша (Fallback Cover).
- Динамику состояний Circuit Breaker.

6. Сравнить полную конфигурацию с урезанными вариантами (только retry, только circuit breaker, retry+circuit breaker без fallback) и проанализировать вклад каждого компонента.

---

3. Теоретическое обоснование

3.1. Мотивация комплексного подхода

В реальных микросервисных архитектурах одиночных механизмов часто недостаточно. Retry справляется с кратковременными всплесками ошибок, но при продолжительном отказе забивает сеть бесполезными повторами. Circuit Breaker предотвращает каскадные отказы, но сам по себе не кэширует ответы. Fallback обеспечивает работу в деградированном режиме, возвращая последний успешный результат. Совместное использование трёх паттернов, реализованное в библиотеках типа **Resilience4j**, **Polly** или **Hystrix**, многократно повышает надёжность клиента.

3.2. Порядок взаимодействия компонентов

Архитектура Resilient Client представляет собой цепочку ответственности:

1. **Circuit Breaker** проверяет своё состояние. Если состояние OPEN — немедленно вызывается fallback; если HALF_OPEN — ограниченное число пробных вызовов пропускается, остальные идут в fallback.
2. **Retry (Exponential Backoff + Jitter)** — внутри защищённого вызова выполняет заданное число попыток с нарастающей задержкой. Исключения, пробрасываемые после исчерпания попыток, передаются в CB как ошибки.
3. **Fallback** — статический ответ, сохранённый в локальном кэше после последнего успешного вызова. Если успешных вызовов ещё не было, возвращается заранее заданное значение по умолчанию или ошибка.

Взаимодействие можно представить схемой:

```
Клиент → [CB: OPEN?] → Если да: Fallback
                       Если нет: Retry(service_call)
                          → Успех: обновить кэш, CB.success()
                          → Неудача после N попыток: CB.error(), Fallback
```

3.3. Метрики комплексной оценки

- **Время восстановления** — от момента возобновления работы сервиса до получения первого успешного ответа клиентом. Зависит от таймаута CB, параметров retry, быстродействия fallback.

- **Retry Cost** — общее число запросов к сервису (попыток), включая успешные и неудачные. Характеризует дополнительную нагрузку.
- **Fallback Cover** — доля вызовов, завершившихся fallback-ответом, а не реальным вызовом. Показывает, насколько сильно система опирается на кэш в периоды нестабильности.
- **Состояние CB** — визуализация переходов показывает, как быстро система обнаруживает проблему и как долго «выздоравливает».

---

4. Пример практического выполнения

4.1. Компонент Retry (Exponential Backoff + Full Jitter)

Используем реализацию из ЛР25.

python

```
import random
import time


def exponential_backoff_with_jitter(attempt, initial, max_delay):
    max_possible = min(max_delay, initial * (2 ** attempt))
    return random.uniform(0, max_possible)
```

Для интеграции с CB и fallback удобнее оформить в виде класса, который по стратегии делает заданное число попыток:

python

```
class RetryHandler:
    def __init__(self, max_attempts=3, initial=0.1, max_delay=5.0):
        self.max_attempts = max_attempts
        self.initial = initial
        self.max_delay = max_delay

    def execute(self, func):
        last_exception = None
        for attempt in range(self.max_attempts):
            try:
```

```
            result = func()
            return result
        except Exception as e:
            last_exception = e
            delay = exponential_backoff_with_jitter(attempt, self.initial, self.max_delay)
            time.sleep(delay)
    raise last_exception
```

4.2. Circuit Breaker (адаптированный)

Модифицируем класс из ЛР26, чтобы он вызывал fallback при разомкнутом состоянии и подсчитывал ошибки через retry.

python

```
class CircuitBreaker:
  # ... состояния CLOSED, OPEN, HALF_OPEN и методы call, _on_success, _on_error, _fallback
  # В call() при OPEN и HALF_OPEN с превышением лимита возвращаем None (сигнал для fallback)
```

Для нашего комплексного клиента мы можем встроить CB как обёртку: cb.call принимает функцию, которая выполняет retry и возвращает результат. При срабатывании fallback возвращается значение из кэша.

4.3. Fallback-кэш

python

```
class FallbackCache:
  def __init__(self, default_value=None):
    self.cached_value = default_value
    self.has_cached = False

  def update(self, value):
    self.cached_value = value
    self.has_cached = True

  def get(self):
```

```
        return self.cached_value if self.has_cached else None
```

4.4. Комплексный клиент ResilientClient

```
python

class ResilientClient:
    def __init__(self, service, retry_handler, circuit_breaker, fallback_cache):
        self.service = service
        self.retry = retry_handler
        self.cb = circuit_breaker
        self.fallback = fallback_cache

    def call(self, request):
        def protected_call():
            return self.service.process(request)

        if self.cb.state == State.OPEN:
            # сразу fallback
            return self.fallback.get()

        if self.cb.state == State.HALF_OPEN and self.cb.half_open_requests >=
self.cb.half_open_max:
            return self.fallback.get()

        try:
            result = self.retry.execute(protected_call)
            self.cb._on_success()
            self.fallback.update(result)
            return result
        except Exception as e:
            self.cb._on_error()
```

```
        return self.fallback.get()
```

4.5. Симулятор нестабильного сервиса

Создадим класс с изменяющимися параметрами: success_prob, latency, crash_mode. Будем управлять из главного цикла эксперимента.

python

```
class UnstableService:
    def __init__(self, base_latency=0.05, failure_rate=0.0):
        self.base_latency = base_latency
        self.failure_rate = failure_rate
        self.crashed = False

    def process(self, request):
        if self.crashed:
            raise Exception("Service crashed")
        if random.random() < self.failure_rate:
            raise Exception("Random failure")
        time.sleep(self.base_latency)
        return f"Response for {request}"
```

4.6. Сценарии и измерение метрик

Напишем функцию run_scenario, которая в цикле отправляет запросы, периодически изменяя параметры сервиса.

Графики: построим временной ряд результатов вызова (Success, Fallback, Error), состояние CB, накопленное количество retry-попыток.

Пример графика: верхний subplot — состояние сервиса (зелёный — доступен, красный — недоступен), средний — тип ответа клиента (Success/Fallback), нижний — накопленная стоимость retry.

---

5. Задания для индивидуального выполнения

Номер варианта: (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать ResilientClient с фиксированными параметрами, провести заданный сценарий, измерить долю fallback-ответов, количество запросов к сервису (retry cost) и максимальное время восстановления. Построить график состояний CB и типов ответов во времени.

Вариант	Retry params (max_att, init, max_del)	CB (thresh, timeout, half_open)	Fallback	Сценарий сервиса
1	3, 0.1, 2.0	2 ошибки, 2.0 с, 1	есть	Кратковрем. сбой на 2.0–3.0 с
2	4, 0.2, 3.0	3 ошибки, 1.5 с, 2	есть	Полный отказ после 5 с
3	2, 0.05, 1.0	2 ошибки, 3.0 с, 1	есть	Высокая задержка (0.3 с)
4	5, 0.15, 4.0	4 ошибки, 2.5 с, 2	есть	Временный сбой, затем отказ
5	3, 0.1, 5.0	3 ошибки, 1.0 с, 1	нет	Кратковременный сбой
6	4, 0.08, 2.0	2 ошибки, 2.0 с, 2	есть	Высокая задержка, потом сбой
7	2, 0.12, 1.5	3 ошибки, 1.8 с, 1	есть	Периодические отказы (чередование)
8	3, 0.1, 2.0	5 ошибок, 3.0 с, 1	есть	Полный отказ
9	4, 0.25, 6.0	2 ошибки, 4.0 с, 2	есть	Высокая задержка (p95 > to)
10	3, 0.06, 1.2	3 ошибки, 2.2 с, 1	есть	Кратковрем. сбой, потом восстановление

Вариант	Retry params (max_att, init, max_del)	CB (thresh, timeout, half_open)	Fallback	Сценарий сервиса
11	5, 0.2, 3.0	2 ошибки, 1.0 с, 2	нет	Полный отказ
12	2, 0.1, 2.5	4 ошибки, 2.0 с, 1	есть	Высокая задержка
13	3, 0.18, 5.0	3 ошибки, 1.5 с, 1	есть	Временный сбой с задержками
14	4, 0.09, 2.0	2 ошибки, 2.5 с, 2	есть	Полный отказ через 8 с
15	3, 0.1, 2.0	3 ошибки, 2.0 с, 1	есть	Смешанный сценарий

Требования к отчёту: код клиента, запуск сценария, график, таблица метрик, выводы о роли каждого компонента.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Сравнить четыре конфигурации: (1) только Retry, (2) только CB, (3) Retry+CB, (4) Retry+CB+Fallback на одном и том же сценарии. Измерить все метрики, построить сравнительные столбчатые диаграммы.
- Исследовать влияние параметра half_open_max_requests (1, 2, 3) на скорость восстановления при разных интенсивностях отказов.
- Добавить адаптивный таймаут из ЛР27 в retry-логику (между попытками не просто sleep, а ожидание ответа с таймаутом, который динамически меняется). Сравнить с фиксированным таймаутом.
- Оценить «холодный старт»: поведение клиента при старте, когда fallback-кэш пуст. Предложить и реализовать стратегию дефолтного значения.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Сравнение конфигураций, холодный старт
2	Вариант 2	Адаптивный таймаут, half_open анализ
3	Вариант 3	Сравнение без fallback
4	Вариант 4	Конфигурации, адаптивный таймаут
5	Вариант 5	Холодный старт, half_open
6	Вариант 6	Сравнение, анализ
7	Вариант 7	Адаптивный таймаут, конфигурации
8	Вариант 8	Half_open, холодный старт
9	Вариант 9	Конфигурации, анализ
10	Вариант 10	Адаптивный таймаут
11	Вариант 11	Сравнение, half_open
12	Вариант 12	Конфигурации, адаптивный таймаут
13	Вариант 13	Холодный старт
14	Вариант 14	Адаптивный таймаут, half_open
15	Вариант 15	Все перечисленные дополнительные задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **многопоточного клиента**, где несколько потоков одновременно совершают запросы к общему симулятору сервиса через экземпляр ResilientClient (с потокобезопасным CB и fallback-кэшем). Измерить пропускную способность и стабильность.
- Интегрировать механизм **Health Check** (периодический фоновый запрос к сервису для быстрого обнаружения его восстановления и принудительного закрытия CB без ожидания таймаута). Сравнить с пассивным восстановлением.
- Построить тепловую карту метрики «Retry Cost» в пространстве параметров (max_attempts, initial, max_delay) для типового сценария временного сбоя.
- Сравнить разработанного клиента с использованием готовых библиотек (например, resilience4j эмулировать через похожие параметры или реализовать прототип на tenacity + самодельный CB) и проанализировать разницу в удобстве и предсказуемости.

Вариант	Базовые параметры	Расширенное задание
1	retry 3, init 0.1, max 2	Многопоточность, health check
2	retry 4, init 0.2, max 3	Тепловая карта, сравнение с библиотеками
3	retry 2, init 0.05, max 1	Health check, тепловая карта
4	retry 5, init 0.15, max 4	Многопоточность, сравнение
5	retry 3, init 0.1, max 5	Тепловая карта, health check
6	retry 4, init 0.08, max 2	Многопоточность, библиотека
7	retry 2, init 0.12, max 1.5	Health check
8	retry 3, init 0.1, max 2	Тепловая карта, многопоточность
9	retry 4, init 0.25, max 6	Health check, сравнение
10	retry 3, init 0.06, max 1.2	Многопоточность, тепловая карта

Вариант	Базовые параметры	Расширенное задание
11	retry 5, init 0.2, max 3	Библиотека, health check
12	retry 2, init 0.1, max 2.5	Тепловая карта
13	retry 3, init 0.18, max 5	Многопоточность, health check
14	retry 4, init 0.09, max 2	Сравнение с библиотеками
15	retry 3, init 0.1, max 2	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. Почему недостаточно использовать только Retry или только Circuit Breaker для построения надёжного клиента?
2. Опишите порядок принятия решения в ResilientClient при поступлении запроса.
3. Как наличие fallback-кэша влияет на метрики доступности при полном отказе сервиса?
4. В чём преимущество совместного использования Exponential Backoff и Circuit Breaker по сравнению с каждым по отдельности?
5. Что произойдёт, если в состоянии Half-Open количество пробных запросов превысит half_open_max раньше, чем придёт ответ от сервиса?
6. Какие проблемы могут возникнуть при использовании устаревшего fallback-ответа (stale data)? В каких сценариях это допустимо?
7. Как параметры retry (max_attempts, initial) и CB (failure_threshold, timeout) влияют на компромисс между временем восстановления и нагрузкой на сервис?
8. Почему важно обеспечивать потокобезопасность Circuit Breaker и fallback-кэша в многопоточном клиенте?
9. Что такое Health Check и как он может ускорить восстановление после сбоя?
10. Приведите примеры современных библиотек, реализующих комплексную отказоустойчивость. Какие дополнительные возможности они предоставляют?

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.
3. Теоретическая часть: описание паттернов, их взаимодействия, метрик.
4. Программная реализация: архитектура ResilientClient, листинг ключевых методов.
5. Экспериментальная часть:
    - Визуализация временных рядов (состояние CB, тип ответа клиента).
    - Таблица метрик (время восстановления, retry cost, fallback cover) для каждого сценария.
    - Сравнительные диаграммы для различных конфигураций (средний уровень) или дополнительные графики.
6. Анализ результатов, объяснение вклада каждого компонента.
7. Выводы: рекомендации по выбору параметров под конкретные требования (низкая задержка, высокая доступность, минимизация нагрузки).
8. Приложение с полным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован ResilientClient, объединяющий все три компонента.
- Проведён один заданный сценарий, сняты базовые метрики, построен график состояний CB.
- Отчёт содержит все обязательные разделы, вывод о роли компонентов.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены требования базового уровня.
- Проведено сравнение конфигураций с разным набором компонентов, продемонстрированы преимущества комплексного подхода.
- Исследовано влияние параметров half_open или добавлен адаптивный таймаут.
- Построены дополнительные графики, даны количественные рекомендации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализована многопоточная версия, health check или построение тепловых карт.

- Проведено сравнение с промышленными библиотеками, даны обоснованные инженерные выводы.
- Отчёт демонстрирует глубокое понимание и способность к архитектурным решениям.

Снижение оценки: некорректная последовательность вызовов (например, fallback вызывается до retry), отсутствие сброса ошибок в CB при успехе, неинтегрированный fallback-кэш, недостаточная визуализация.

Лабораторная работа № 30

Асинхронный инференс через очередь с dynamic batching

---

1. Цель работы

Цель работы — изучить технику **динамического батчирования (dynamic batching)** для повышения пропускной способности систем машинного инференса. Студент реализует асинхронный обработчик с очередью задач, настраиваемым накопителем батчей и имитационной моделью вычислений, проведёт сравнение латентности и пропускной способности (throughput) с синхронным и простым асинхронным подходами.

---

2. Задачи

1. Спроектировать систему асинхронного инференса, состоящую из:
    - генератора запросов (producer), имитирующего API-клиентов;
    - асинхронной очереди задач;
    - воркера (worker), накапливающего запросы в батчи по времени или по количеству;
    - модели машинного обучения, обрабатывающей сформированный батч (эмулируется с помощью задержки, зависящей от размера батча).
2. Реализовать динамическое батчирование с параметрами:
    - max_batch_size — максимальное количество запросов в батче;
    - max_wait_time — максимальное время ожидания накопления батча (таймаут).
3. Реализовать два альтернативных режима для сравнения:
    - **Синхронный инференс** — каждый запрос немедленно вызывает модель (без очереди и батчинга).
    - **Асинхронный без батчинга** — запросы ставятся в очередь и обрабатываются строго по одному.
4. Провести нагрузочное тестирование при различной интенсивности запросов (RPS), измеряя:
    - среднюю и медианную задержку обработки одного запроса;
    - пропускную способность системы (количество обслуженных запросов в секунду);
    - коэффициент использования батчей (средний размер батча).

5. Построить графики зависимости задержки и пропускной способности от RPS для трёх режимов.
6. Проанализировать компромисс между задержкой и пропускной способностью, дать рекомендации по выбору параметров батчирования.

---

3. Теоретическое обоснование

3.1. Проблема эффективности инференса

Модели машинного обучения, особенно глубокие нейронные сети, значительно выигрывают в производительности при обработке нескольких входных примеров одновременно (батчами) благодаря оптимизациям на GPU и использованию векторных инструкций. Одиночный вызов модели (инференс) обычно обладает высокими накладными расходами (фиксированное время подготовки, передачи данных в память GPU), которые амортизируются при увеличении размера батча. Однако в системах реального времени запросы приходят по одному, и ожидание формирования большого батча может увеличить задержку. **Dynamic batching** решает это противоречие, динамически группируя запросы, пришедшие за короткий интервал.

3.2. Принцип динамического батчинга

Воркер постоянно ожидает появления задач в очереди. Когда появляется первая задача, запускается таймер max_wait_time. Воркер продолжает набирать задачи из очереди, пока не будет достигнуто одно из двух условий:

- размер батча достиг max_batch_size;
- истекло время ожидания max_wait_time.

После этого сформированный батч подаётся на модель, а результаты рассылаются соответствующим клиентам (например, через Future). Параметры max_batch_size и max_wait_time позволяют управлять компромиссом между задержкой и пропускной способностью:

- Малое max_wait_time и большой max_batch_size снижают задержку, но могут приводить к малым батчам и неэффективному использованию GPU.
- Большой max_wait_time увеличивает средний размер батча и пропускную способность, но ценой роста задержки для первых запросов в батче.

3.3. Модель времени инференса

Для эмуляции работы модели обычно используют упрощённую линейную модель времени обработки батча размера B:

$$T_{\text{batch}}(B) = T_{\text{fixed}} + (B - 1) \times T_{\text{marginal}},$$

где T_{fixed} — фиксированная составляющая (накладные расходы на вызов), T_{marginal} — добавочное время на каждый дополнительный элемент (обычно значительно меньше, чем T_{fixed}/B при одиночной обработке). Таким образом, среднее время на один запрос $\bar{t} = T_{\text{batch}}(B)/B$ убывает с ростом B, обуславливая рост пропускной способности.

3.4. Сравниваемые режимы

- **Синхронный (без очереди)**: модель вызывается непосредственно в потоке обработчика запроса (или асинхронной функции) с размером батча 1. Время ответа = $T_{\text{batch}}(1)$. Пропускная способность ограничена величиной $1/T_{\text{batch}}(1)$.
- **Асинхронный без батчинга**: запросы помещаются в очередь, воркер извлекает их по одному и подаёт на модель с батчом размера 1. Латентность примерно равна времени ожидания в очереди + $T_{\text{batch}}(1)$. Пропускная способность такая же, как в синхронном режиме (модель используется последовательно).
- **Асинхронный с dynamic batching**: воркер формирует батчи динамически. Латентность возрастает на время ожидания накопления батча, но пропускная способность растёт за счёт уменьшения $\bar{t}$.

---

4. Пример практического выполнения

4.1. Эмуляция модели инференса

Объявим функцию, которая имитирует обработку батча с задержкой.

python

```
import asyncio
import time
import random
import numpy as np
import matplotlib.pyplot as plt


# Параметры модели (мс)
FIXED_COST = 10.0       # фиксированное время на вызов
MARGINAL_COST = 2.0     # добавочное время на элемент


async def model_inference(batch):
    """Принимает список входных данных; возвращает список результатов."""
```

```
    batch_size = len(batch)
    # линейная модель задержки
    delay_ms = FIXED_COST + (batch_size - 1) * MARGINAL_COST
    await asyncio.sleep(delay_ms / 1000.0)   # перевод в секунды
    # Возвращаем фиктивный результат (например, умножение на 2)
    return [x * 2 for x in batch]
```

4.2. Воркер с динамическим батчингом

```
python
class DynamicBatcher:
    def __init__(self, model, max_batch_size=32, max_wait_time=0.05):
        self.model = model
        self.max_batch_size = max_batch_size
        self.max_wait_time = max_wait_time
        self.queue = asyncio.Queue()
        self._task = None

    async def start(self):
        self._task = asyncio.create_task(self._worker())

    async def stop(self):
        if self._task:
            self._task.cancel()
            self._task = None

    async def submit(self, item):
        """Отправка запроса; возвращает Future с результатом."""
        future = asyncio.get_event_loop().create_future()
        await self.queue.put((item, future))
        return future
```

```
async def _worker(self):
    while True:
        batch_items = []
        batch_futures = []
        # Ждём первый элемент
        item, future = await self.queue.get()
        batch_items.append(item)
        batch_futures.append(future)
        deadline = asyncio.get_event_loop().time() + self.max_wait_time

        # Собираем батч пока есть время и не превышен размер
        while True:
            remaining = deadline - asyncio.get_event_loop().time()
            if remaining <= 0 or len(batch_items) >= self.max_batch_size:
                break
            try:
                item, future = await asyncio.wait_for(self.queue.get(), timeout=remaining)
                batch_items.append(item)
                batch_futures.append(future)
            except asyncio.TimeoutError:
                break

        # Обрабатываем батч
        try:
            results = await self.model(batch_items)
            for fut, res in zip(batch_futures, results):
                fut.set_result(res)
        except Exception as e:
```

```
            for fut in batch_futures:
                fut.set_exception(e)
```

4.3. Синхронный и простой асинхронный обработчики

python

```
async def synchronous_inference(model, item):
    results = await model([item])
    return results[0]


async def async_no_batching_worker(queue, model):
    """Воркер, обрабатывающий по одному элементу без ожидания накопления."""
    while True:
        item, future = await queue.get()
        try:
            result = await model([item])
            future.set_result(result[0])
        except Exception as e:
            future.set_exception(e)
```

4.4. Генерация трафика и нагрузочное тестирование

Генератор посылает запросы с заданным темпом (RPS). Собираем задержки и количество выполненных запросов.

python

```
async def load_test(handler, rps, duration=5.0, model=None):
    """
    handler: функция-обработчик одного запроса, возвращающая результат.
    Для DynamicBatcher это batcher.submit(x), возвращающая Future.
    Для синхронного — непосредственно вызывает модель.
    """
    latencies = []
    start = asyncio.get_event_loop().time()
```

```
    end = start + duration
    tasks = []
    async def requester(rate):
        interval = 1.0 / rate
        next_time = asyncio.get_event_loop().time()
        while next_time < end:
            # ожидаем до следующего времени отправки
            await asyncio.sleep(max(0, next_time - asyncio.get_event_loop().time()))
            send_time = asyncio.get_event_loop().time()
            # Создаём запрос
            # Для DynamicBatcher
            if isinstance(handler, DynamicBatcher):
                fut = await handler.submit(send_time)  # в качестве данных используем время отправки
                task = asyncio.create_task(measure_latency(fut, send_time, latencies))
            elif asyncio.iscoroutinefunction(handler):
                # синхронный обработчик
                task = asyncio.create_task(single_request(handler, model, send_time, latencies))
            tasks.append(task)
            next_time += interval
        await asyncio.gather(*tasks, return_exceptions=True)


    async def measure_latency(fut, send_time, lat_list):
        try:
            res = await fut
            lat = asyncio.get_event_loop().time() - send_time
            lat_list.append(lat)
        except Exception:
            pass
```

```
    async def single_request(handler, model, send_time, lat_list):
        try:
            res = await handler(model, send_time)  # предполагаем, что handler ожидает (model, item)
            lat = asyncio.get_event_loop().time() - send_time
            lat_list.append(lat)
        except Exception:
            pass

    await requester(rps)
    return latencies
```

4.5. Сравнительный эксперимент

Проведём замеры при разных RPS для каждого метода.

```
python
RPS_VALUES = [10, 20, 50, 100, 200, 500]
# Инициализация
model = model_inference
batcher = DynamicBatcher(model, max_batch_size=32, max_wait_time=0.05)
await batcher.start()

# Сбор результатов
sync_lat = {}
async_lat = {}
batch_lat = {}

for rps in RPS_VALUES:
    # Синхронный (эмулируем, вызывая synchronous_inference в requester)
    sync_lat[rps] = await load_test(synchronous_inference, rps, model=model)
```

```
# Асинхронный без батчинга
queue = asyncio.Queue()
worker = asyncio.create_task(async_no_batching_worker(queue, model))
async_lat[rps] = await load_test(lambda m, it: queue.put((it, asyncio.get_event_loop().create_future())))..., сложнее)
```

Упростим: в load_test будем передавать функцию, которая ставит в очередь и возвращает future. Для async_no_batching используем общую очередь и worker.

python

```
# Для асинхронного без батчинга:
queue = asyncio.Queue(maxsize=0)
worker = asyncio.create_task(async_no_batching_worker(queue, model))
async def submit_async(item):
    fut = asyncio.get_event_loop().create_future()
    await queue.put((item, fut))
    return fut
```

Для dynamic batcher передаём batcher.submit.

Для синхронного: передаём функцию, которая сразу ожидает модель.

Соберём средние задержки и throughput.

4.6. Визуализация

python

```
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(RPS_VALUES, [np.mean(sync_lat[r])*1000 for r in RPS], 'o-', label='Sync')
plt.plot(RPS_VALUES, [np.mean(async_lat[r])*1000 for r in RPS], 's-', label='Async no batch')
plt.plot(RPS_VALUES, [np.mean(batch_lat[r])*1000 for r in RPS], '^-', label='Dynamic batching')
plt.xlabel('RPS')
plt.ylabel('Mean latency (ms)')
plt.legend()
plt.grid()
```

```
plt.subplot(1,2,2)
# throughput = количество успешных запросов / время теста
plt.plot(RPS_VALUES, [len(sync_lat[r])/5.0 for r in RPS], 'o-', label='Sync')
# ...
plt.show()
```

Ожидаемый результат: Dynamic batching показывает значительно более высокий максимальный throughput, чем синхронный и асинхронный без батчинга, при этом на малых RPS задержка несколько выше из-за ожидания накопления батча. С ростом RPS задержка dynamic batching растёт медленно, а пропускная способность достигает значений существенно выше 1 / T_batch(1).

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать систему асинхронного батчирования с заданными параметрами. Провести тестирование для трёх режимов (синхронный, асинхронный без батчинга, динамический батчинг) при указанной интенсивности RPS. Измерить среднюю задержку и пропускную способность. Построить график зависимости средней задержки от RPS для всех трёх режимов.

Вариант	max_batch_size	max_wait_time (мс)	FIXED_COST (мс)	MARGINAL_COST (мс)	RPS для теста
1	16	50	10	2	10, 25, 50, 100, 150
2	32	30	12	1.5	10, 30, 60, 120, 200

Вариант	max_batch_size	max_wait_time (мс)	FIXED_COST (мс)	MARGINAL_COST (мс)	RPS для теста
3	8	40	8	1	5, 15, 30, 50, 80
4	64	60	15	3	20, 50, 100, 200, 400
5	16	20	10	2	10, 20, 40, 80, 120
6	32	50	9	2.5	10, 25, 50, 100, 180
7	24	35	11	2	15, 30, 60, 100, 160
8	12	45	7	1.2	8, 20, 40, 70, 100
9	48	55	14	2.8	10, 40, 80, 150, 250

Вариант	max_batch_size	max_wait_time (мс)	FIXED_COST (мс)	MARGINAL_COST (мс)	RPS для теста
10	20	25	10	1.8	12, 25, 50, 90, 140
11	16	50	12	1.5	10, 30, 60, 100, 160
12	32	40	8	1	10, 25, 50, 80, 130
13	8	60	9	2.2	5, 15, 30, 45, 70
14	64	30	15	3.5	20, 50, 100, 180, 300
15	24	45	10	2	10, 20, 40, 80, 120

Требования к отчёту: корректная реализация dynamic batcher, графики задержки и throughput, таблица с числовыми значениями, обоснование преимущества батчинга.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому уровню:

- Исследовать влияние параметра max_wait_time на среднюю задержку и пропускную способность при фиксированном RPS. Построить график зависимости среднего размера батча и throughput от max_wait_time (10, 30, 50, 70, 100 мс).

- Сравнить стратегию формирования батча «по количеству ИЛИ по времени» с альтернативной стратегией «только по времени» (т.е. батч отправляется строго по таймеру, даже если батч пуст? или без ограничения размера). Измерить разницу в задержке и throughput.
- Добавить в модель обработки эффект «масштабирования относительно размера батча» — использовать более реалистичные кривые (амортизация GPU: T = T0 + k*B**0.7 и т.п.). Сравнить с линейной моделью.
- Построить кривую зависимости пропускной способности от RPS (через нагрузку) для оптимально подобранных параметров батчинга, используя итеративный подбор.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	Влияние max_wait_time, сравнение стратегий
2	Вариант 2	Нелинейная модель, кривая throughput
3	Вариант 3	Стратегия "только по времени", влияние max_wait_time
4	Вариант 4	Нелинейная модель, сравнение
5	Вариант 5	Влияние max_wait_time, throughput
6	Вариант 6	Сравнение стратегий, нелинейная модель
7	Вариант 7	Влияние max_wait_time, стратегия
8	Вариант 8	Нелинейность, сравнение
9	Вариант 9	Стратегия, max_wait_time
10	Вариант 10	Нелинейная модель, графики
11	Вариант 11	Влияние max_wait_time, стратегия

Вариант	Параметры базового уровня	Дополнительные задания
12	Вариант 12	Нелинейность, throughput
13	Вариант 13	Стратегия, max_wait_time
14	Вариант 14	Нелинейная модель, сравнение
15	Вариант 15	Все перечисленные дополнительные задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Реализовать **адаптивный dynamic batching**: параметр max_wait_time динамически изменяется на основе очереди (например, если очередь растёт, таймаут уменьшается, чтобы ускорить обработку и не копить батчи). Сравнить с фиксированным таймаутом в сценариях переменного RPS.
- Моделировать **множественные воркеры/GPU**: несколько воркеров параллельно разбирают очередь, каждый формирует свои батчи. Исследовать влияние количества воркеров на пропускную способность и задержку.
- Интегрировать реальную ML-модель (например, ONNX-модель или TensorFlow Lite) и сравнить синтетическую задержку с реальной на CPU/GPU.
- Построить трёхмерную диаграмму зависимости среднего времени ответа от RPS и max_batch_size.
- Разработать полноценную систему с обратным вызовом (callback) или WebSocket для возврата результатов, эмулирующую production-архитектуру.

Вариант	Базовые параметры	Расширенное задание
1	max_batch=32, wait=0.05	Адаптивный таймаут, множественные воркеры
2	max_batch=16, wait=0.04	Реальная модель, 3D-график

Вариант	Базовые параметры	Расширенное задание
3	max_batch=64, wait=0.06	Адаптивный, production-архитектура с callback
4	max_batch=24, wait=0.05	Множественные воркеры, реальная модель
5	max_batch=32, wait=0.03	Адаптивный таймаут, 3D-график (RPS × batch_size)
6	max_batch=48, wait=0.07	Реальная модель, множественные воркеры
7	max_batch=12, wait=0.02	Production-архитектура, адаптивный
8	max_batch=64, wait=0.08	3D-график, реальная модель
9	max_batch=16, wait=0.05	Адаптивный, множественные воркеры
10	max_batch=32, wait=0.06	Реальная модель, callback
11	max_batch=20, wait=0.04	3D-график, адаптивный
12	max_batch=32, wait=0.05	Множественные воркеры, производственная симуляция
13	max_batch=8, wait=0.03	Адаптивный таймаут, реальная модель

Вариант	Базовые параметры	Расширенное задание
14	max_batch=64, wait=0.07	3D-график, множественные воркеры
15	max_batch=32, wait=0.05	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. Зачем применяется батчирование в системах машинного инференса? Почему одиночные запросы неэффективны?
2. Объясните разницу между динамическим и статическим батчингом.
3. Как параметр max_wait_time влияет на компромисс между задержкой и пропускной способностью?
4. Что произойдёт, если установить max_batch_size очень большим, а max_wait_time очень маленьким при низком RPS?
5. Как асинхронная очередь помогает сгладить пиковые нагрузки?
6. Опишите стратегии возврата результата клиенту после обработки батча (future, callback, polling). Какие из них предпочтительнее в высоконагруженных системах?
7. Как можно адаптировать параметры батчинга в реальном времени в зависимости от длины очереди?
8. Почему при использовании нескольких воркеров (GPU) важно следить за перекосом нагрузки?
9. Какие дополнительные накладные расходы появляются при батчинге (сериализация, десериализация)?
10. Приведите примеры промышленных систем, использующих dynamic batching (TensorFlow Serving, NVIDIA Triton, TorchServe).

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи работы.

3. Теоретическую часть: принципы динамического батчинга, модель времени инференса, метрики.
4. Программную реализацию: основные классы и функции, схема взаимодействия.
5. Экспериментальную часть:
    - Таблицы задержек и пропускной способности для трёх режимов при разных RPS.
    - Графики зависимости средней задержки и throughput от RPS.
    - Для среднего/повышенного уровня – дополнительные графики влияния параметров, адаптивного батчинга, кривые пропускной способности.
6. Анализ результатов, объяснение формы кривых.
7. Выводы о преимуществах и ограничениях метода.
8. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Реализован dynamic batcher с корректной логикой таймаута и размера батча.
- Проведены нагрузочные тесты для трёх режимов, измерены задержки и throughput.
- Построены сравнительные графики, сформулирован вывод о выигрыше от батчинга.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Исследовано влияние параметров, проведено сравнение стратегий, возможно использование нелинейной модели.
- Дополнительные метрики (средний размер батча, использование GPU) визуализированы.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Реализована адаптивная настройка таймаута, множественные воркеры, интеграция с реальной моделью или детальная трёхмерная визуализация.
- Глубокий анализ, практические рекомендации, возможна демонстрация production-окружения.

Снижение оценки: некорректная работа таймера (сбор батча без ожидания), отсутствие сравнения режимов, неверная модель ускорения, неинформативные графики.

Лабораторная работа № 31

Кэширование результатов инференса с TTL

---

1. Цель работы

Цель работы — изучить методы снижения затрат на машинный инференс за счёт гибридного кэша, объединяющего **точное совпадение (exact match)**, **семантическое сходство (similarity)** и **ограничение времени жизни (TTL)**. Студент реализует многоуровневый кэш, проведёт его симуляцию на синтетическом или реальном логе запросов, измерит hit ratio по уровням, экономию вычислительных ресурсов и точность приближённых ответов.

---

2. Задачи

1. Реализовать двухуровневый кэш для ML-инференса:
    - **Уровень 1 (Exact Match Cache)** — хранит пары (входной_запрос, результат) с ограниченным размером и TTL; использует LRU-вытеснение.
    - **Уровень 2 (Similarity Cache)** — хранит (эмбеддинг_запроса, результат); при промахе уровня 1 ищет ближайший эмбеддинг с косинусным расстоянием выше порога.
2. Для каждого уровня задаётся свой **TTL** (время жизни записи), по истечении которого запись удаляется из кэша.
3. Загрузить или сгенерировать лог запросов, имитирующий трафик к модели (например, поиск по изображениям), включающий повторяющиеся и семантически близкие запросы.
4. Симулировать обработку лога через кэш, измеряя:
    - **Hit Ratio** отдельно для Exact Match и Similarity Cache, а также общий hit ratio;
    - **Снижение затрат** (доля запросов, обслуженных кэшем, относительно полного числа вызовов модели);
    - **Точность similarity-ответов** — доля случаев, когда результат, полученный по сходству, совпадает с истинным результатом модели.
5. Исследовать влияние порога сходства, размера кэша Exact Match и TTL на метрики.
6. Построить графики зависимости hit ratio от параметров.

---

3. Теоретическое обоснование

3.1. Проблематика кэширования ML-инференса

Вызов модели машинного обучения — ресурсоёмкая операция, особенно при использовании глубоких нейросетей на GPU. Во многих приложениях (поиск по изображениям, рекомендательные системы) запросы обладают значительной **временно́й и семантической локальностью**: одни и те же или очень похожие входные данные поступают повторно. Кэширование позволяет сохранить результат и избежать повторного инференса. Простой кэш «точного ключа» (exact match) даёт высокий hit ratio только при дословных повторах. Для расширения покрытия применяют **similarity-кэш**, который ищет не точное совпадение, а ближайший запрос в латентном пространстве эмбеддингов.

3.2. Архитектура гибридного кэша

Предлагаемая архитектура включает два уровня:

- **Уровень 1: Exact Match Cache (EMC)**
  Хранит отображение вход → результат. При поступлении запроса сначала проверяется EMC. Если найден и его TTL не истёк — возвращается закэшированный результат. Вытеснение — LRU.

- **Уровень 2: Similarity Cache (SC)**
  Хранит пары (эмбеддинг_входа, результат). Если EMC дал промах, у запроса вычисляется эмбеддинг (в симуляторе можно использовать предвычисленный или простой хеш-вектор). Затем в SC ищется запись с максимальным косинусным сходством с текущим эмбеддингом. Если сходство превышает заданный порог τ и TTL записи не истёк, возвращается её результат (приближённый ответ). В противном случае выполняется реальный инференс, после чего результат сохраняется в обоих кэшах (точный ключ в EMC, эмбеддинг в SC) с новым TTL.

- **TTL (Time-To-Live)**
  Записи в каждом уровне имеют ограниченный срок жизни. При проверке попадания сначала проверяется, не истекло ли время хранения. Для EMC TTL_emc обычно короче (десятки секунд — минуты), так как точные ответы могут быстро устаревать. Для SC используется более длинный TTL_sc (минуты — часы), поскольку эмбеддинги более стабильны.

3.3. Метрики эффективности

- **Hit Ratio (уровня 1, уровня 2, общий)** — доля запросов, обслуженных каждым уровнем.
- **Снижение затрат (Cost saving)** — доля запросов, для которых **не** выполнялся инференс модели. Эквивалентно общему Hit Ratio.
- **Точность similarity (Similarity Accuracy)** — доля случаев, когда результат из SC совпадает с результатом, который вернула бы модель для данного запроса.

- **Ложные попадания (False Accept)** — когда SC вернул результат, не совпадающий с истинным (приводит к деградации качества).
- **Средняя задержка** — суммарное время ожидания, включая время на вычисление эмбеддинга, поиск по кэшу и (при промахе) время инференса.

3.4. Выбор порога сходства

Порог τ определяет компромисс между покрытием (hit ratio) и точностью. Высокое значение (например, 0.98) даёт почти точные ответы, но низкое покрытие; низкое значение (0.8) увеличивает hit ratio, но может выдавать ошибочные результаты. Оптимальный порог подбирается на основе эталонных данных.

---

4. Пример практического выполнения

4.1. Модели эмбеддингов и инференса

Для простоты в симуляции будем генерировать эмбеддинги как случайные векторы фиксированной размерности, а модель инференса — как детерминированную функцию от входных данных (например, метку класса). В реальной системе здесь может быть вызов TensorFlow Serving.

python

```
import random
import numpy as np
import math
from collections import OrderedDict
import time


class MLModel:
    """Имитация модели машинного обучения."""
    def __init__(self, dim=64):
        self.dim = dim

    def embed(self, input_data):
        """Возвращает эмбеддинг для входных данных. В симуляции — детерминированный на основе хеша."""
        seed = hash(input_data) % 2**31
```

```
        rng = np.random.RandomState(seed)
        emb = rng.randn(self.dim)
        emb /= np.linalg.norm(emb)   # нормализуем
        return emb

    def predict(self, input_data):
        """Имитация инференса: возвращает метку класса на основе входных данных."""
        # Генерируем результат, зависящий от входа.
        seed = hash(input_data) % 2**31
        rng = np.random.RandomState(seed)
        return f"class_{rng.randint(0, 10)}"
```

4.2. Класс гибридного кэша с TTL

Используем OrderedDict для LRU-вытеснения. Для отслеживания TTL храним время вставки.

python

```
class HybridCache:
    def __init__(self, emc_capacity=100, sc_capacity=200,
                 ttl_em=60.0, ttl_sc=300.0, similarity_threshold=0.9):
        # Exact Match Cache: key -> (result, timestamp)
        self.emc = OrderedDict()
        self.emc_capacity = emc_capacity
        self.ttl_em = ttl_em

        # Similarity Cache: список (embedding, result, timestamp)
        self.sc = []   # для простого перебора; для больших размеров лучше FAISS
        self.sc_capacity = sc_capacity
        self.ttl_sc = ttl_sc
        self.threshold = similarity_threshold

        self.model = MLModel()
```

```
    def _cos_sim(self, a, b):
        # a и b нормализованы
        return np.dot(a, b)

    def _cleanup_sc(self):
        """Удаляет просроченные записи из SC."""
        now = time.time()
        self.sc = [rec for rec in self.sc if (now - rec[2]) <= self.ttl_sc]

    def _cleanup_emc(self):
        now = time.time()
        expired = [k for k, (_, ts) in self.emc.items() if (now - ts) > self.ttl_em]
        for k in expired:
            del self.emc[k]

    def query(self, input_data):
        """Возвращает (уровень_попадания, результат).
           Уровень: 'exact', 'similarity', 'model'.
        """
        now = time.time()
        # Удаляем просроченные записи (можно делать периодически или при каждом запросе)
        self._cleanup_emc()
        self._cleanup_sc()

        # Уровень 1: Exact Match
        if input_data in self.emc:
            result, ts = self.emc[input_data]
```

```
    if now - ts <= self.ttl_em:
        # обновить LRU позицию: переместить в конец
        self.emc.move_to_end(input_data)
        return ('exact', result)

# Уровень 2: Similarity Cache
emb = self.model.embed(input_data)
best_sim = -1
best_result = None
for rec in self.sc:
    stored_emb, stored_res, stored_ts = rec
    if now - stored_ts > self.ttl_sc:
        continue
    sim = self._cos_sim(emb, stored_emb)
    if sim > best_sim:
        best_sim = sim
        best_result = stored_res
if best_sim >= self.threshold and best_result is not None:
    # Сохраняем в Exact Match для будущих точных попаданий
    self._save_emc(input_data, best_result)
    return ('similarity', best_result)

# Промах: вызов модели
result = self.model.predict(input_data)
# Сохраняем в EMC
self._save_emc(input_data, result)
# Сохраняем в SC
self._save_sc(emb, result)
return ('model', result)
```

```
def _save_emc(self, key, value):
    if len(self.emc) >= self.emc_capacity:
        self.emc.popitem(last=False)  # удаляем самый старый (LRU)
    self.emc[key] = (value, time.time())


def _save_sc(self, emb, result):
    if len(self.sc) >= self.sc_capacity:
        # удаляем самую старую запись по времени вставки
        self.sc.pop(0)
    self.sc.append((emb, result, time.time()))
```

4.3. Генерация лога запросов

Создадим лог, в котором присутствуют повторяющиеся, слегка изменённые и полностью новые запросы.

python

```
def generate_log(num_requests=5000, repeat_ratio=0.4, similar_ratio=0.3):
    """
    Генерирует список запросов. Часть запросов будет точными повторами предыдущих,
    часть — похожими (с небольшим изменением строки), остальные — случайными.
    """
    base_queries = [f"img_{i}" for i in range(200)]
    log = []
    for _ in range(num_requests):
        r = random.random()
        if r < repeat_ratio and len(log) > 0:
            log.append(random.choice(log))  # точный повтор
        elif r < repeat_ratio + similar_ratio and len(log) > 0:
            # похожий запрос: добавим суффикс к существующему
            proto = random.choice(log)
```

```
            log.append(proto + "_var")
        else:
            log.append(f"img_{random.randint(0, 500)}")
    return log
```

4.4. Симуляция и замер метрик

python

```
def run_simulation(log, cache):
    hit_exact = 0
    hit_sim = 0
    hit_model = 0
    sim_correct = 0
    sim_total = 0

    # Для оценки точности similarity нам нужно знать истинный результат модели.
    # В симуляции можем вычислить его отдельно для всех запросов или доверять model.predict.
    # Будем считать: если similarity вернула результат, сравниваем его с model.predict (без кэша).
    model_no_cache = MLModel()

    for query in log:
        level, result = cache.query(query)
        if level == 'exact':
            hit_exact += 1
        elif level == 'similarity':
            hit_sim += 1
            sim_total += 1
            # проверяем точность
            if result == model_no_cache.predict(query):
                sim_correct += 1
```

```
        else:
            hit_model += 1

    total = len(log)
    cost_saving = (hit_exact + hit_sim) / total
    sim_accuracy = sim_correct / sim_total if sim_total > 0 else 1.0
    return {
        'hit_exact_ratio': hit_exact / total,
        'hit_similarity_ratio': hit_sim / total,
        'overall_hit_ratio': (hit_exact + hit_sim) / total,
        'cost_saving': cost_saving,
        'similarity_accuracy': sim_accuracy,
        'model_calls': hit_model
    }
```

4.5. Визуализация

Построим зависимости hit ratio от порога сходства и размера EMC.

```
python
import matplotlib.pyplot as plt

thresholds = np.linspace(0.8, 1.0, 10)
hit_ratios = []
accuracies = []
for th in thresholds:
    cache = HybridCache(similarity_threshold=th)
    stats = run_simulation(log, cache)
    hit_ratios.append(stats['overall_hit_ratio'])
    accuracies.append(stats['similarity_accuracy'])

plt.figure(figsize=(10,4))
```

```
plt.subplot(1,2,1)
plt.plot(thresholds, hit_ratios, 'o-')
plt.xlabel('Порог сходства')
plt.ylabel('Общий Hit Ratio')
plt.grid()
plt.subplot(1,2,2)
plt.plot(thresholds, accuracies, 's-')
plt.xlabel('Порог сходства')
plt.ylabel('Точность similarity')
plt.grid()
plt.show()
```

---

5. Задания для индивидуального выполнения

Номер варианта определяется как (номер_студента mod 15) + 1.

5.1. Базовый уровень (на оценку «удовлетворительно»)

Реализовать гибридный кэш с LRU-вытеснением и заданными параметрами. Сгенерировать лог запросов по варианту, провести симуляцию, измерить hit ratio по уровням, общий hit ratio и точность similarity-ответов. Построить график зависимости общего hit ratio от порога сходства (варьировать от 0.7 до 1.0 с шагом 0.05).

Вариант	Емкость EMC	Емкость SC	TTL_em (с)	TTL_sc (с)	repeat_ratio	similar_ratio	Длина лога
1	100	200	60	300	0.4	0.3	10 000
2	150	250	120	600	0.35	0.35	8 000
3	80	150	30	150	0.3	0.4	12 000
4	200	300	90	450	0.45	0.25	15 000
5	120	180	45	200	0.5	0.2	7 000

Вариант	Емкость EMC	Емкость SC	TTL_em (c)	TTL_sc (c)	repeat_ratio	similar_ratio	Длина лога
6	100	200	60	300	0.25	0.5	9 000
7	90	220	75	400	0.4	0.3	11 000
8	130	270	50	250	0.38	0.32	10 500
9	70	130	20	100	0.3	0.35	6 000
10	160	240	100	500	0.42	0.28	13 000
11	110	190	55	280	0.33	0.37	8 500
12	140	210	80	350	0.48	0.22	14 000
13	85	160	35	180	0.28	0.42	7 500
14	95	170	65	320	0.36	0.34	9 500
15	105	230	70	380	0.44	0.26	12 500

Требования к отчёту: корректная реализация кэша; таблица с метриками для выбранного порога (например, 0.9); график зависимости hit ratio от порога; вывод о влиянии порога на точность и покрытие.

5.2. Средний уровень (на оценку «хорошо»)

Дополнительно к базовому:

- Реализовать эффективный поиск ближайшего эмбеддинга с использованием библиотеки **FAISS** (или nmslib). Сравнить время поиска с полным перебором и измерить пропускную способность кэша.
- Исследовать влияние TTL на точность и hit ratio: провести эксперименты, варьируя TTL_SC от 60 до 3600 секунд, и построить кривую зависимости hit ratio и точности similarity от TTL_SC.

- Ввести метрику **«Среднее время отклика»** (latency), учитывающую время поиска по кэшу и время инференса при промахе. Построить график средней задержки от порога сходства.
- Сравнить гибридный кэш с раздельными независимыми кэшами (только Exact Match, только Similarity) при одинаковом объеме занимаемой памяти.

Вариант	Параметры базового уровня	Дополнительные задания
1	Вариант 1	FAISS, влияние TTL, задержка
2	Вариант 2	Сравнение с раздельными кэшами, FAISS
3	Вариант 3	Влияние TTL, задержка
4	Вариант 4	FAISS, сравнение
5	Вариант 5	Задержка, влияние TTL
6	Вариант 6	FAISS, раздельные кэши
7	Вариант 7	Влияние TTL, задержка
8	Вариант 8	FAISS, сравнение
9	Вариант 9	Задержка, TTL
10	Вариант 10	FAISS, раздельные кэши
11	Вариант 11	Влияние TTL, задержка
12	Вариант 12	FAISS, сравнение
13	Вариант 13	Задержка, раздельные кэши
14	Вариант 14	FAISS, влияние TTL

Вариант	Параметры базового уровня	Дополнительные задания
15	Вариант 15	Все перечисленные дополнительные задания

5.3. Повышенный уровень (на оценку «отлично»)

Расширенное исследование:

- Использовать **реальные эмбеддинги** от предобученной модели (например, CLIP, ResNet50) на подмножестве изображений (заранее предоставленном преподавателем или загрузить из открытого датасета). Провести симуляцию с реальными векторами, измерить точность similarity по сравнению с истинными метками.
- Реализовать **динамическую настройку порога сходства** , поддерживающую заданную точность (например, на основе скользящего окна последних точных ответов автоматически подбирать порог, чтобы точность similarity была не ниже 95%).
- Построить **трёхмерный график** зависимости cost saving от размера EMC и порога сходства, выявить оптимальную область параметров.
- Интегрировать кэш в эмуляцию веб-сервера на **Flask/FastAPI** и провести нагрузочное тестирование с помощью wrk или locust, измеряя влияние кэша на задержку и пропускную способность.

Вариант	Базовые параметры	Расширенное задание
1	EMC=100, SC=200, τ=0.9	Реальные эмбеддинги (CLIP), динамический порог
2	EMC=120, SC=250, τ=0.85	Трёхмерный график, веб-сервер
3	EMC=90, SC=180, τ=0.92	Реальные эмбеддинги, динамический порог
4	EMC=150, SC=300, τ=0.88	Веб-сервер, нагрузочный тест
5	EMC=80, SC=150, τ=0.9	Динамический порог, трёхмерный график

Вариант	Базовые параметры	Расширенное задание
6	EMC=110, SC=200, τ=0.87	Реальные эмбеддинги, веб-сервер
7	EMC=130, SC=220, τ=0.91	Трёхмерный график, динамический порог
8	EMC=100, SC=180, τ=0.93	Реальные эмбеддинги, трёхмерный
9	EMC=140, SC=260, τ=0.86	Веб-сервер, динамический порог
10	EMC=70, SC=140, τ=0.89	Реальные эмбеддинги, трёхмерный
11	EMC=160, SC=280, τ=0.94	Динамический порог, веб-сервер
12	EMC=85, SC=170, τ=0.9	Трёхмерный график, реальные эмбеддинги
13	EMC=95, SC=190, τ=0.88	Веб-сервер, нагрузочный тест
14	EMC=105, SC=210, τ=0.92	Динамический порог, трёхмерный
15	EMC=115, SC=240, τ=0.9	Комбинация нескольких расширенных задач

---

6. Контрольные вопросы

1. Зачем нужен уровень similarity-кэша, если уже есть точное совпадение?
2. Как параметр TTL влияет на актуальность результатов и hit ratio?
3. Почему для similarity-кэша обычно устанавливают более длинный TTL, чем для exact match?
4. Какие метрики, помимо hit ratio, важны при оценке качества гибридного кэша?

5. Что произойдёт, если порог сходства установить слишком низким? Слишком высоким?
6. Какие структуры данных и библиотеки можно использовать для эффективного поиска ближайшего эмбеддинга (ANN)?
7. Каким образом можно автоматически обновлять порог сходства, чтобы поддерживать целевую точность ответов?
8. Чем отличается «холодный старт» кэша от стационарного режима? Как его смягчить?
9. Как можно реализовать инвалидацию записей при изменении модели (новые веса)?
10. Приведите примеры реальных систем, использующих similar-кэширование (семантическое кэширование в поиске, GPTCache и т.п.).

---

7. Требования к отчёту

Отчёт должен содержать:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть: описание многоуровневого кэша, TTL, метрик.
4. Программная реализация: основные классы, пояснение алгоритма поиска.
5. Экспериментальная часть:
    - Таблица с метриками (hit ratio, точность, cost saving) для выбранной конфигурации.
    - График зависимости общего hit ratio и точности от порога сходства.
    - (для среднего/повышенного уровня) дополнительные графики влияния TTL, задержки, трёхмерные визуализации.
6. Анализ результатов и выводы.
7. Приложение с исходным кодом.

---

8. Критерии оценивания

8.1. Оценка «удовлетворительно» (базовый уровень)

- Корректно реализованы оба уровня кэша с TTL.
- Проведена симуляция на логе, измерены основные метрики.
- Построен график зависимости hit ratio от порога, сделаны выводы о выборе порога.

8.2. Оценка «хорошо» (средний уровень)

- Выполнены все требования базового уровня.
- Использован эффективный поиск (FAISS) или подробно исследовано влияние TTL и задержки.
- Проведено сравнение с независимыми кэшами, даны рекомендации.

8.3. Оценка «отлично» (повышенный уровень)

- Полностью выполнен средний уровень.
- Применены реальные эмбеддинги, реализован динамический порог, интеграция с веб-сервером или трёхмерные визуализации.
- Отчёт демонстрирует глубокое понимание предмета и содержит практические инженерные выводы.

Снижение оценки: неверная логика работы TTL (записи не удаляются), отсутствие второго уровня, ошибка в вычислении точности similarity, неработающий код.