⚡ Практическое занятие

Метрики производительности и CAP-теорема

ФИО студента: Хуснутдинова Венера

Группа: ИМО-426

Дата: 27.10.2025

📊 Основные метрики производительности

Percentiles (Процентили)

- **P50 (медиана):** 50% запросов быстрее этого времени
- **P95:** 95% запросов быстрее
- **P99:** 99% запросов быстрее
- **P99.9:** 99.9% запросов быстрее

⚖️ CAP-теорема

Правило: В распределенной системе можно гарантировать только 2 из 3 свойств:

Практические компромиссы:

- **CP системы:** MongoDB, Redis Cluster (жертвуют доступностью)
- **AP системы:** Cassandra, DynamoDB (жертвуют согласованностью)
- **CA системы:** PostgreSQL, MySQL (не устойчивы к разделению)

🔒 ACID свойства

D - Durability (Долговечность)
Зафиксированные изменения сохраняются навсегда
Пример: После подтверждения транзакции данные не теряются даже при сбое

📈 Availability (Доступность)

Формула: Availability = (Uptime / Total Time) × 100%

SLA уровни:

🧪 Практические задания

Требования к выполнению:

- выполняются на ОС **Linux** (любой дистрибутив)
- пакеты устанавливаются в **виртуальное окружение venv**
- для создания серверов можно использовать **LLM** (желательно Claude Sonnet подключить в IDE)

Задание 1: Установка и использование Apache Bench

Установка Apache Bench на Linux:

```
# Ubuntu/Debian
sudo apt-get install apache2-utils
```

```
# CentOS/RHEL/Fedora
sudo yum install httpd-tools
# или
sudo dnf install httpd-tools

# Проверка установки
ab -V
```

Тестирование производительности:

```
# Запуск 1000 запросов с 10 одновременными соединениями
ab -n 1000 -c 10 http://httpbin.org/get
```

Результаты тестирования:

Время выполнения теста:

61.728 секунд

Requests per second:

16.20 запросов/сек

Time per request (mean):

617.283 мс

Статистика соединений (мс):

	min	mean	median	max
Connect:	1	2	1	9
Processing:	263	608	348	2756

Перцентили времени ответа:

P50: 439 мс

P90: 2093 мс

P95: 2293 мс

P99: 2564 мс

Описание полученных результатов:

Тест прошёл успешно, без ошибок и отказов.

Средняя скорость обработки составила 16.2 запросов в секунду, что указывает на стабильную работу сервера при 10 одновременных соединениях.

Средняя задержка одного запроса — около 617 мс, что является нормальным показателем для удалённого API.

Большинство запросов обслуживались за 300–400 мс, однако в верхних перцентилях наблюдаются задержки до 2.7 секунд, что связано с непостоянством сетевых задержек.

В целом система демонстрирует устойчивую производительность и предсказуемое поведение под нагрузкой.

📊 Подробный анализ результатов Apache Bench

Общая информация о тесте:

Утилита: ApacheBench 2.3
URL: http://httpbin.org/get
Количество запросов: 1000
Уровень параллелизма: 10 одновременных соединений

Параметры сервера:

```
Server Software: awselb/2.0 ← AWS Elastic Load Balancer
Server Hostname: httpbin.org
Server Port: 80 ← HTTP порт
Document Path: /get ← Тестируемый endpoint
Document Length: 162 bytes ← Размер ответа
```

Расшифровка статистики соединений:

- **Connect:** Время установки TCP-соединения
- **Processing:** Время обработки запроса сервером
- **Waiting:** Время от отправки запроса до получения первого байта
- **Total:** Общее время запроса

Анализ качества:

Failed requests: 0 (ошибок нет) Transfer rate: 5.25 Kbytes/sec

Выводы о производительности:

1. Стабильность: Сервер отработал стабильно — все 1000 запросов были успешно выполнены, с нулевым числом ошибок.
2. Производительность: Показатель Requests per second = 16.2 говорит о умеренной производительности для внешнего API, не оптимизированного под высокую нагрузку.
3. Задержки: Средняя задержка на один запрос составила ≈617 мс, медианная — 349 мс. 90% запросов завершались быстрее 2 секунд, однако отдельные пики доходили до 2.7 с, что указывает на колебания времени ответа.
4. Ошибки: Ошибок соединения и передачи данных нет (Failed requests = 0), но присутствуют Non-2xx ответы (1000 шт.), что связано с особенностями тестируемого эндпоинта, а не с проблемой производительности.
5. Масштабируемость: Результаты показывают, что система способна обрабатывать параллельные запросы без сбоев, однако при увеличении числа клиентов время ответа может расти из-за сетевых и серверных ограничений.

Рекомендации для улучшения:

- Оптимизировать можно за счет кеширования
- Рассмотреть CDN для уменьшения задержек
- Увеличить параллелизм для проверки пределов сервера

⏱️ Задание 2: Измерение задержек (latency)

Простой HTTP сервер с задержками (Flask):

```
from flask import Flask
import time
import random

app = Flask(__name__)

@app.route('/fast')
def fast_endpoint():
```

```
    return {'response': 'fast', 'latency': '10ms'}

@app.route('/slow')
def slow_endpoint():
    time.sleep(random.uniform(0.1, 0.5))  # 100-500ms задержка
    return {'response': 'slow', 'latency': '100-500ms'}

if __name__ == '__main__':
    app.run(port=5000)
```

Измерение latency:

```
# Тестирование быстрого endpoint
ab -n 100 -c 5 http://localhost:5000/fast

# Тестирование медленного endpoint
ab -n 100 -c 5 http://localhost:5000/slow
```

Результаты /fast endpoint:

RPS:

310.90

Средняя задержка:

16.08 мс

Результаты /slow endpoint:

RPS:

14.97

Средняя задержка:

333.990 мс

🐍 Сравнение веб-фреймворков

FastAPI (порт 8000)

Код на Python:

```
from fastapi import FastAPI
import time
import random
```

```python
app = FastAPI()

@app.get('/fast')
def fast_endpoint():
    return {'response': 'fast',
        'latency': '10ms'}

@app.get('/slow')
def slow_endpoint():
    time.sleep(random.uniform(0.1, 0.5))
    return {'response': 'fast',
        'latency': '100-500ms'}

if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

448.12

Средняя задержка:

11.16 мс

Результаты /slow endpoint:

RPS:

14.91

Средняя задержка:

335.37 мс

Краткие выводы:

1. Стабильность: высокая, ошибок 0, все запросы успешно обработаны.
2. Производительность: FastAPI показал отличную скорость обработки на быстром эндпоинте (448 RPS), что говорит о высокой оптимизации фреймворка и эффективности асинхронной модели.
3. Задержки: средняя задержка на /fast — около 11 мс, на /slow — около 335 мс, что соответствует добавленной искусственной задержке.
4. Ошибки: отсутствуют, сервер стабильно справился с нагрузкой.
5. Масштабируемость: высокая — FastAPI способен эффективно обрабатывать большое число одновременных запросов.

Django (порт 8001)

Код на Python:

```
api/views.py:

from django.http import JsonResponse
import time
import random

def fast_endpoint(request):
    return JsonResponse({
        'response': 'fast',
        'latency': '10ms'
    })

def slow_endpoint(request):
    time.sleep(random.uniform(0.1, 0.5))
    return JsonResponse({
        'response': 'slow',
        'latency': '100-500ms'
    })

urls.py:

from django.contrib import admin
from django.urls import path
from api.views import fast_endpoint, slow_endpoint

urlpatterns = [
    path('admin/', admin.site.urls),
    path('fast', fast_endpoint),
    path('slow', slow_endpoint),
]
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

267.11

Средняя задержка:

18.7 мс

Результаты /slow endpoint:

RPS:

16.26

Средняя задержка:

307.4 мс

Краткие выводы:

Стабильность: Сервер Django демонстрирует стабильную обработку запросов без ошибок (0 failed).
Производительность: На быстрых запросах Django показывает достойную скорость (267 RPS), но уступает более лёгким фреймворкам вроде FastAPI.
Задержки: Средняя задержка на /fast — около 19 мс, а на /slow — около 300 мс, что соответствует заданным искусственным задержкам.
Ошибки: Отсутствуют.
Масштабируемость: Хорошо обрабатывает до 5 параллельных соединений, но при росте нагрузки ожидаемо увеличится средняя задержка из-за синхронной природы Django.

Django REST Framework (порт 8002)

Код на Python:

```
api/views.py:

import time
import random
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def fast_endpoint(request):
    return Response({
        'response': 'fast',
        'latency': '10ms'
    })

@api_view(['GET'])
def slow_endpoint(request):
    time.sleep(random.uniform(0.1, 0.5))
    return Response({
        'response': 'slow',
        'latency': '100-500ms'
    })

urls.py:
from django.contrib import admin
from django.urls import path
from api.views import fast_endpoint, slow_endpoint
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('fast', fast_endpoint),
    path('slow', slow_endpoint),
]
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

330.82

Средняя задержка:

15.1 мс

Результаты /slow endpoint:

RPS:

14.99

Средняя задержка:

333.5 мс

Краткие выводы:

Стабильность: сервер стабильно обрабатывает запросы, без ошибок и с равномерным временем отклика.

Производительность: для быстрых запросов (без задержек) RPS ≈ 330 — производительность высокая для Django REST, что указывает на хорошую оптимизацию WSGI.

Задержки: при искусственных задержках (100–500 мс) среднее время ответа выросло до ~333 мс, что ожидаемо для синхронной обработки.

Ошибки: 0% неудачных запросов — сервер корректно обрабатывает нагрузку.

Масштабируемость: подходит для REST API среднего уровня нагрузки; при увеличении одновременных соединений стоит использовать ASGI/uvicorn или gunicorn для повышения пропускной способности.

LiteStar (порт 8003)

Код на Python:

```python
from litestar import Litestar, get
import asyncio
import random

@get("/fast")
async def fast_endpoint() -> dict:
    return {"response": "fast", "latency": "10ms"}

@get("/slow")
async def slow_endpoint() -> dict:
    await asyncio.sleep(random.uniform(0.1, 0.5))
    return {"response": "slow", "latency": "100-500ms"}

app = Litestar(route_handlers=[fast_endpoint, slow_endpoint])
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

420

Средняя задержка:

12.10 мс

Результаты /slow endpoint:

RPS:

18.34

Средняя задержка:

280.2 мс

Краткие выводы:

LiteStar показал высокую производительность на лёгких запросах (/fast), опередив Django и немного превысив FastAPI.

При обработке медленных запросов (/slow) LiteStar также справляется чуть быстрее, демонстрируя хорошую асинхронную масштабируемость.

Ошибок не зафиксировано, сервер стабилен.

Подходит для систем, где важны скорость отклика и асинхронная обработка.

Общие выводы о производительности веб-фреймворков:

FastAPI показал наилучший баланс между скоростью и стабильностью — высокая пропускная способность (RPS) и минимальные задержки при асинхронной обработке.

LiteStar оказался сопоставим по быстродействию с FastAPI, иногда демонстрируя чуть лучшие результаты при высокой нагрузке благодаря оптимизированной архитектуре.

Django REST Framework работает значительно медленнее при тех же сценариях, поскольку использует синхронную модель обработки запросов и более тяжёлую инфраструктуру.

Во всех трёх случаях ошибки отсутствовали, что говорит о стабильности тестируемых конфигураций.
В целом, асинхронные фреймворки (FastAPI, LiteStar) лучше подходят для систем с большим числом запросов, тогда как Django REST целесообразнее использовать для проектов, где важна сложная бизнес-логика и готовая административная часть, а не максимальная скорость.

📝 Домашнее задание

Задача: Доделать задания

Срок сдачи: К следующему занятию

Что нужно сделать:

- Протестировать 3 разных сайта с помощью Apache Bench
- Сравнить их производительность
- Построить график зависимости RPS от количества одновременных соединений
- Создать HTTP серверы на всех указанных фреймворках
- Провести сравнительное тестирование производительности

📊 Критерии оценки

📚 Литература

- Клеппман М. "Высоконагруженные приложения"
- Фаулер М. "Архитектура корпоративных программных приложений"
- Документация Apache Bench
- Официальная документация веб-фреймворков