⚡ Практическое занятие

Метрики производительности и CAP-теорема

ФИО студента:

Патрушева Виолетта Артуровна

Группа:

ИМО-426

Дата:

22.10.2025

📊 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:

243.725 секунд

Requests per second:

4.10 запросов/сек

Time per request (mean):

243.725 мс

Статистика соединений (мс):

	min	mean	median	max
Connect:	0	2	0	207
Processing:	134	2352	1236	20787

Перцентили времени ответа:

P50: 1241 мс P90: 7680 мс P95: 10461 мс P99: 14951 мс

Описание полученных результатов:

Столкнулась с проблемой в Убунту, моя гостевая ОС пинговалась, проверяла доступность сервера по curl, все было окей. Но команда "ab -n 1000 -c 10 http://httpbin.org/get" ломалась на 100-200 запросе, поэтому я воспользовалась -k (keep alive)

ab -n -k 1000 -c 10 http://httpbin.org/get

, что снизило кол-во новых TCP подключений и сработало, но пришлось долго ждать.

📊 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: 5 (ошибок нет) Transfer rate: 1.95 Kbytes/sec

Выводы о производительности:

```
1. Стабильность: сервер отвечает на большинство запросов но с
большим разбросом времени и с использованием -k.
2. Производительность: показатель 4.10 запросов в секунду
показывает, что есть задержки и ограничения API
3. Надежность: ошибки единичны (0.5%)
4. Задержки: присутствуют, время отклика для 1000 запросов НАМНОГО
выше, чем для 100.
5. Причина: сервер httpbin.org ограничивает частоту запросов.
```

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:

224.14

Средняя задержка:

4.461 мс

Результаты /slow endpoint:

RPS:

15.42

Средняя задержка:

64.841 мс

Сравнение веб-фреймворков

FastAPI (порт 8000)

Код на Python:

```
from fastapi import FastAPI
import time
import random

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)) "color: #95a5a6;">#100-500ms
    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:

499.51

Средняя задержка:

2.002 мс

Результаты /slow endpoint:

RPS:

14.13

Средняя задержка:

70.758 мс

Краткие выводы:

FastAPI показывает высокую производительность на /fast,при добавлении искусственных задержок в /slow время отклика возрастает. Все запросы успешно обработаны, что показывается стабильность сервера под нагрузкой.

Django (порт 8001)

Код на Python:

```
from django.http import JsonResponse
from django.urls import path
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)) "color: #95a5a6;">#100-500ms
    return JsonResponse(
        {
            'response': 'slow',
            'latency': '100-500ms'}
            )

urlpatterns = [
    path('fast', fast_endpoint),
    path('slow', slow_endpoint)
]
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

352.11

Средняя задержка:

Результаты /slow endpoint:

RPS:

16.38

Средняя задержка:

2.840 мс

61.035 мс

Краткие выводы:

Мне показалось, что FastAPI быстрее для легких запросов, но Джанго тоже отлично справился. При задержках разница FastAPI и Django минимальная. Все запросы обработаны успешно.

Django REST Framework (порт 8002)

Код на Python:

```
from django.shortcuts import render

"color: #95a5a6;"># Create your views here.
"color: #95a5a6;"># from django.http import JsonResponse
"color: #95a5a6;"># from django.urls import path
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'}
            )


"color: #95a5a6;"># urls.py
```

```
from django.contrib import admin
from django.urls import path
"color: #95a5a6;"># 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:

380.53

Средняя задержка:

2.628 мс

Результаты /slow endpoint:

RPS:

14.59

Средняя задержка:

68.53 мс

Краткие выводы:

Все реквесты успешные, 380 запросов за секунду это быстро; при /slow задержки есть, но производительность не падает. Быстрее джанго, но медленнее FastAPI.

LiteStar (порт 8003)

Код на Python:

```
from liteserver import LiteStar
import time
import random


app = LiteStar()

@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)) "color: #95a5a6;">#100-500ms
    return {'response': 'fast',
            'latency': '100-500ms'}

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8003)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

400

Средняя задержка:

12 мс

Результаты /slow endpoint:

RPS:

15.03

Средняя задержка:

65.45 мс

Краткие выводы:

Все запросы обработаны без ошибок, сервер показывает высокую стабильность при умеренной нагрузке.

Общие выводы о производительности веб-фреймворков:

```
FastAPI остается самым быстрым для легких запросов.
Django и DRF показывают большую задержку, но удобны.
LiteStar имеет достаточно хорошую скорость и стабильность, но хуже чем перечисленные выше, думаю, он подходит для мини-сервисов.
```

📝 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