⚡ Практическое занятие

Метрики производительности и CAP-теорема

ФИО студента:

Фациевич Дмитрий Иванович

Группа:

426

Дата:

26.10.2025

📊 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:

315.863 секунд

Requests per second:

3.17 запросов/сек

Time per request (mean):

315.863 мс

Статистика соединений (мс):

	min	mean	median	max
Connect:	133	144	136	1136
Processing:	139	2979	1423	25444

Перцентили времени ответа:

P50: 1565 мс P90: 9331 мс P95: 13300 мс P99: 19097 мс

Описание полученных результатов:

Анализируя результаты нагрузочного тестирования, можно сделать выводы о производительности сервера. Система испытывает серьезные трудности под относительно небольшой нагрузкой. Всего тысяча запросов при десяти одновременных соединениях для сервера уже являются серьезной 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: 13 (ошибок нет) Transfer rate: 1.48 Kbytes/sec

Выводы о производительности:

1. Система не стабильно. Высокое стандартное отклонение времени ответа (более 4 секунд) и большой разброс между минимальным и максимальным временем обработки говорят о непредсказуемости системы. Периодические серьезные просадки производительности (что 10% запросов обрабатывались дольше 0

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):

```python
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:

```bash
# 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:

376.57

Средняя задержка:

13 мс

Результаты /slow endpoint:

RPS:

15.44

Средняя задержка:

311 мс

🐍 Сравнение веб-фреймворков

FastAPI (порт 8000)

Код на Python:

```
from fastapi import FastAPI
import time
import random

app = FastAPI()

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

@app.get('/slow')
async def slow_endpoint():
    time.sleep(random.uniform(0.1, 0.5))  "color: #95a5a6;"># 100-500ms задержка
    return {'response': 'slow', 'latency': '100-500ms'}

if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app, host='0.0.0.0', port=5000)
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

3495.53

Средняя задержка:

1 мс

Результаты /slow endpoint:

RPS:

3.61

Средняя задержка:

1359 мс

Краткие выводы:

Лучший результат на быстрых запросах благодаря асинхронности и uvicorn. Однако медленные операции (блокирующие) сильно тормозят обработку, если их не вынести в отдельные процессы или потоки.

Django (порт 8001)

Код на Python:

```
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'})

from django.urls import path
from . import views

urlpatterns = [
    path('fast/', views.fast_endpoint, name='fast'),
    path('slow/', views.slow_endpoint, name='slow'),
]
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('api.urls')),  "color: #95a5a6;"># Include your app's URLs
]
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'api',  "color: #95a5a6;"># <-- add this
]
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

1541.71

Средняя задержка:

3 мс

Результаты /slow endpoint:

RPS:

15.82

Средняя задержка:

293 мс

Краткие выводы:

Производительность средняя, но стабильная. Django хорошо держится под нагрузкой и обеспечивает надёжность, хоть и не рекордные RPS. Хорош для комплексных проектов с большим функционалом.

Django REST Framework (порт 8002)

Код на Python:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',  "color: #95a5a6;"># <-- add this
    'api',             "color: #95a5a6;"># <-- your app
]
from rest_framework.decorators import api_view
from rest_framework.response import Response
import time
import random

@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'})
from django.urls import path
from . import views

urlpatterns = [
    path('fast/', views.fast_endpoint, name='fast'),
    path('slow/', views.slow_endpoint, name='slow'),
]
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),  "color: #95a5a6;"># Add this
]
```

Результаты тестирования:

Результаты /fast endpoint:

RPS:

Результаты /slow endpoint:

RPS:

1450.37

15.69

Средняя задержка:

3 мс

Средняя задержка:

302 мс

Краткие выводы:

Немного медленнее обычного Django из-за дополнительного слоя абстракции (сериализаторы, middleware). Однако DRF обеспечивает удобство и масштабируемость при работе с API.

LiteStar (порт 8003)

Код на Python:

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

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

@get("/slow")
def slow_endpoint() -> dict[str, str]:
    time.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:

3086.61

Средняя задержка:

1 мс

Результаты /slow endpoint:

RPS:

3.58

Средняя задержка:

1361 мс

Краткие выводы:

По скорости близок к FastAPI (оба асинхронные, на Starlette/ASGI). Отлично справляется с быстрыми запросами, но при блокирующих операциях также резко падает производительность.

Общие выводы о производительности веб-фреймворков:

Асинхронные фреймворки (FastAPI, Litestar) показывают наилучшие результаты по быстродействию на лёгких (“fast”) запросах — до (примерно) 3500 RPS, с минимальной задержкой 1 мс.
Однако при блокирующих вызовах (например, time.sleep)

📝 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