CSE352: System Analysis and Design

Extracurricular Activity

(Out of 100 marks)

This is individual assignment

Your task **is to implement one of the components of your project** and ensure the response time of chosen component is **always less than 200 msec**.

To be able to do that you need to adopt many **low-latency design patterns**.

Your need to:

1- Choose suitable low latency design patterns for your component and implement these patterns.
2- Show how did you achieved the 200ms constraint.
3- Perform response time testing for your component, and show the percentage for satisfying the constraint, assuming workload is 100 request per sec.
4- Provide a PPT and demo about your work.

You can use tools like **Apache JMeter, Gatling, or k6** to help you in assessing your component performance.

You can refer to the following paper and video for more information.

https://arxiv.org/pdf/2309.04259

https://www.youtube.com/watch?v=q7qKeUVS4Gw&t=9s

if there is more than one student registered for extracurricular and from the same project. Everyone must show a different component.

I know these topics are new to you but do your best.

Professional work and report will be rewarded by more points.

Hints:

To prepare for your testing environment:

Setting up an environment to test for a **p95 < 200 msec** response time with a sustained workload of **100 Requests Per Second (RPS)** requires careful planning in three main areas: the **Test Environment**, the **Load Generation Tool**, and the **Test Configuration**.

1. ⚙ Prepare the Test Environment

The test environment must accurately simulate your production environment to get meaningful results.

A. Component Under Test (CUT)

- **Isolation:** The single component you are testing must be **isolated** from other development or QA traffic. This prevents external factors (like someone else running a database query) from skewing your latency measurements.
- **Parity:** Ensure the CUT's deployment configuration **matches production** as closely as possible, including:
  - **Hardware/VM/Container specs:** CPU, RAM, and network settings must be the same.
  - **Operating System/Runtime:** Same OS, Java/Python/Node.js version, etc.
  - **Dependencies:** Use a **production-like database** (same version, configuration, and critically, a **production-sized dataset** with similar indexing and data distribution). The database is often the source of "tail latency" (P95/P99 issues).

B. Monitoring Infrastructure

- **Observability:** Set up **robust monitoring** (e.g., using Prometheus/Grafana, Datadog, or New Relic) on the CUT server(s). You must correlate response time spikes with system metrics.
- **Key Metrics to Monitor:**

- **Server CPU Utilization:** Should not be pegged at 100%. High CPU often causes latency.
- **Server Memory Usage:** Watch for memory leaks or excessive Garbage Collection (GC) activity, which are major causes of P95 spikes.
- **Network I/O/Latency:** Between the CUT and its dependencies (e.g., database, caching layer).
- **Database Metrics:** Query response times, connection pool saturation, and I/O.

---

2. Choose and Configure the Load Tool

Select a modern load testing tool that can accurately measure percentiles and sustain the required load from the client side.

A. Tool Selection

Popular, reliable tools that support percentile (P95) tracking and high throughput include:

- **k6** (Modern, code-based, great for CI/CD integration)
- **Gatling** (Scala-based, known for high performance)
- **Locust** (Python-based, good for complex user flows)
- **Apache JMeter** (Highly flexible, but requires careful configuration for high load/percentile accuracy)

B. Load Generator Setup

The machine running the load test (the load generator) must be powerful enough not to become the bottleneck.

- **Client Resource Check:** For 100 RPS, a single modern machine is typically sufficient, but you must monitor its CPU and network usage. If the load generator hits 80%+ CPU, its own processing delay will skew your measured latency, an issue called **Coordinated Omission**.

- **Distributed Load:** If you cannot rule out the load generator as a bottleneck, use **multiple distributed load generator agents** in the same region as the CUT to simulate the load.

---

3. 📝 Design and Execute the Test

Your test script must enforce the target workload and define the critical performance threshold.

A. Load Script Design

- **Realistic Flow:** Script the request(s) that represent the most frequent or performance-critical operation on the component. Use dynamic data (e.g., rotating user IDs, unique payload values) to prevent the component from serving only cached responses.
- **Throughput Control:** Configure the tool to maintain a **constant target rate of 100 RPS**. Tools like k6, Gatling, or JMeter's Throughput Shaping Timer allow you to define this rate explicitly, ensuring the load is stable.

B. Defining the Objective (The SLO)

The P95 is the **Service Level Objective (SLO)** you are testing for. This means 95% of all requests in the test run must have a response time of **less than 200 msec**.

C. Setting the Threshold in the Tool

Configure a pass/fail threshold directly in your load test tool:

- **Example (k6):** You would define a threshold like:
JavaScript
thresholds: {
'http_req_duration': ['p(95) < 200'] // 95th percentile response time must be under 200ms
}

- **Ramp-Up/Duration:**
  - **Ramp-Up:** Start with a gradual ramp-up (e.g., over 30 seconds) to 100 RPS to warm up the component and caches.
  - **Sustained Load:** Run the test for a **minimum of 5-10 minutes** at the full 100 RPS to gather sufficient data and detect any degradation.

D. Analysis

The tool's final report will provide the exact P95 value.

- **If P95 < 200 msec:** The test passes for this workload.

If P95 > 200 msec: The test fails. You must then correlate the time of the P95 spikes with your system monitoring data (CPU, GC, DB I/O) to identify the bottleneck.