**Assignment 4a – Client Design**

## **1. Overview**

The **LoadTester** is an HTTP client that simulates high-volume concurrent requests to a given server endpoint. The program executes in two phases: an **initialization phase** and a **main testing phase**, supporting configurable thread groups, request volumes, and delays between executions.

## **2. Design Principles & Best Practices**

The following design principles are employed in the **client**:

### **a. Modularity and Maintainability**

* The code is structured into **distinct methods**, such as runLoadTest(), sendPostRequest(), sendGetRequest(), and sendWithRetries(), improving **readability**and**reusability.**
* Configuration values (e.g., thread count, request count, retries) are stored as **constants**, making it easy to modify and scale.
* Uses **helper methods** for request sending and retries to **reduce duplication** and enhance maintainability.

### **b. Concurrency and Performance Optimization**

* Utilizes a **thread pool** (ExecutorService) to efficiently manage concurrent execution of HTTP requests.
* Implements **futures (**Future<Void**>)** to track the completion of each thread, ensuring efficient task management.
* Uses **non-blocking execution** for request dispatching while allowing the main thread to continue execution.

### **c. Resilience and Fault Tolerance**

* Implements **automatic retries** (MAX\_RETRIES = 5) with exponential backoff for failed requests to **increase reliability**.
* Catches and handles exceptions such as:
  + **IOException** (file read issues)
  + **InterruptedException** (thread interruptions)
  + **ExecutionException & TimeoutException** (failed async tasks)
* Uses **graceful shutdown** of the thread pool to avoid resource leaks (executor.shutdown() and awaitTermination()).

### **d. Scalability and Configurability**

* Accepts **command-line arguments** to configure:
  + **Thread group size**
  + **Number of thread groups**
  + **Delay between thread groups**
  + **Base URL for testing**
* Uses **constants for default values** but allows dynamic modifications at runtime.
* Employs **multi-threading** to simulate large-scale concurrent loads efficiently.

### **e. Correctness and Standard Compliance**

* Adheres to **HTTP standards** (GET, POST, multipart form-data).
* Constructs **valid HTTP requests** with appropriate headers.
* Uses **UUID-based boundary generation** for ensuring proper multipart request formatting.
* Reads and sends **binary file data correctly** (ensuring the correct handling of image uploads).

## **3. Implementation Details**

### **a. Initialization Phase**

* Creates an **initial burst** of HTTP requests using INIT\_THREADS and INIT\_REQUESTS\_PER\_THREAD to **warm up**the server.
* Runs a batch of GET and POST requests to check basic responsiveness.

### **b. Main Load Testing Phase**

* Creates **configurable thread groups** to generate sustained load.
* Each thread executes a **fixed number of requests** (REQUESTS\_PER\_THREAD).
* Introduces a **delay** between thread groups (Thread.sleep(delay \* 1000L)) to simulate periodic spikes.

### **c. Request Execution**

* **GET Request**: Fetches an album’s metadata.
* **POST Request**: Uploads an album image using **multipart/form-data**.
* Requests are **sent asynchronously** with retries on failure.

### **d. Performance Measurement**

* Calculates **total wall time** from the test start to completion.
* Computes **throughput** (requests per second).
* Outputs a **summary report** with:
  + Execution time
  + Total requests sent
  + Effective throughput

## **4. Assumptions & Limitations**

* The load test assumes **uniform request distribution**, which might not reflect real-world usage patterns.
* The current **error-handling strategy** retries failed requests but does not implement advanced logging.