Fullname: Đào Đình Hiếu
StudentId: 22028221

Final Report: E-Commerce Service Implementations and Performance Evaluation

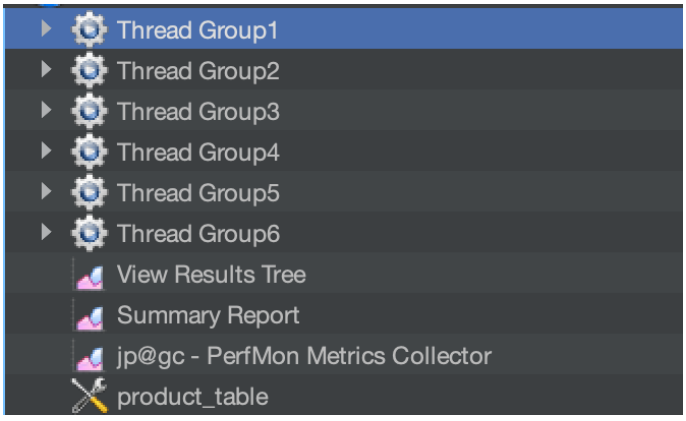
1. Introduction

This report documents the implementation and performance testing of a simple e-commerce service. The service provides a method, **calculateTotal(productId, quantity, processingDelayMs)**, which looks up a static product price from a MySQL database (with 1,500 products), computes the total cost (price × quantity), and returns an order confirmation(ìf the calculation is successful, **returns totalPrice as a double**, rounded to two decimal places. If an error occurs or the product is not found, it returns null). The service was implemented using three different technologies:

- **Java RMI**
- **gRPC**
- **Apache Thrift**

2. Implementation and Test Details

2.1. Test Overview



- **Three Test Plans:**
  Each test plan targets a specific implementation (Java RMI, gRPC, and Apache Thrift).
- **Six Thread Groups per Test Plan:**
  Every test plan contains six thread groups, where each group simulates a distinct level of concurrent calls.

- **View Results Tree**
  This listener displays detailed information about each request and response, including request data, response data, and any errors.
- **Summary Report**
  Another listener that aggregates statistics such as average response time, throughput (requests per second), error rate, and more.
- **jp@gc - PerfMon Metrics Collector**
  A plugin (PerfMon) used to collect server-side resource usage (CPU, memory) in real time. By combining PerfMon data with JMeter's response-time metrics, you can correlate performance bottlenecks with specific resource constraints on the server.
- **product_table.csv**
  A file contains product IDs and prices used by the thread groups. JMeter can reference this file via a CSV Data Set, enabling dynamic input for each request.

2.2 Java RMI Implementation

In the Java RMI solution, we define a remote interface (**OrderService)** that provides **calculateTotal(productId, quantity, processingDelayMs)**. The implementation **(OrderServiceImp)** looks up the product price from MySQL via **ProductService**(method **getPriceById(productid)** implemented in **ProductServiceImp**), applies an optional delay (**Thread.sleep if processingDelayMs > 0**), and returns the total cost (rounded to two decimals). If the product is not found or an error occurs, it returns null. The service is registered in the RMI registry, allowing a Java client (**OrderClient)** to call calculateTotal remotely.

Key Design Decisions and Challenges:

- **Remote Interface Design:** The service interface extende java.rmi.remote, and each method throws RemoteException
- **Serialization Issues:**Class 'Optional' in Java does not support Serializable
- **GitHub:** https://github.com/hieudz2k4/soa/tree/main/lab1/java_rmi

Test Setup: Create class **RMITestSampler** extend **AbstractJavaSamplerClient** in client -> build jar file -> copy to **lib/ext** of jmeter home directory

2.3 gRPC Implementation

For the gRPC implementation, the **order.proto** file was defined to specify the service and message structure. The protoc tool was then used to generate Java code for the client and Go code for the server. The calculateTotal method, which takes three parameters—**productId, quantity, and processingDelayMs**—was implemented in order_server.go. The Go server implements this service by connecting to the MySQL database, calculating the total price, and returning the confirmation. A Java client (generated from proto code) invokes the service.

Key Design Decisions and Challenges:

- **Proto File Design:** The proto file was designed with clear message definitions for **OrderRequest** and **OrderResponse** and **OrderService**
- **Interoperability:** The gRPC allowed cross-language support, enabling a Go server with a Java client.
- **GitHub:** **https://github.com/hieudz2k4/soa/tree/main/lab1/gRPC**

Test Setup: Use plugin in jmeter**(Jmeter gRPC Request)**

2.4 Apache Thrift Implementation

The Apache Thrift service was defined in a Thrift IDL file (**order.thrift).** The service, **OrderService**, defines the method calculateTotal that returns an OrderConfirmation (using wrapper types to allow null values if needed). The Go server implements this service by connecting to the MySQL database, calculating the total price, and returning the confirmation. A Java client (generated from Thrift code) invokes the service.

- **Thrift File Design:** The thrift file was designed with clear message definitions for **OrderRequest** and **OrderResponse** and **OrderService**
- **Interoperability:** The Thrift allowed cross-language support, enabling a Go server with a Java client.
- **GitHub:** **https://github.com/hieudz2k4/soa/tree/main/lab1/apache_thrift**

Test Setup: Test Setup: Create class **ThriftTestSampler** extend **AbstractJavaSamplerClient** in client -> build jar file -> copy to **lib/ext** of jmeter home directory

3. Performance Results

Set delayProcessingMs = 1000ms

5 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1074	2.3	15	16
gRPC	885	6.1	14	15
Apache Thrift	891.1	4.7	15.1	14

10 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1057	5.1	25	18.5
gRPC	850	11	24	17.1
Apache Thrift	886	8.1	26.1	16

50 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1030	25	31.5	26
gRPC	841.5	40	30	26
Apache Thrift	870	35.8	31	26.8

100 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1040	49.1	35.5	28
gRPC	1005.4	88.1	31	26.5
Apache Thrift	1035.1	80	34	26.1

500 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1410	112.7	41	36.6
gRPC	1101.1	150.5	35	34
Apache Thrift	1300.5	147.6	38	35

1000 Concurrent Calls

Tech	Latency(ms)	Throughput(TPS)	CPU Usage(%)	Ram Usage(%)
Java RMI	1138	141.7	55	50.8
gRPC	1050	180.1	50	46.5
Apache Thrift	1100.8	175.8	54.1	48

4. Analysis

Java RMI:

- *Advantages:*
  - Straightforward to implement within Java ecosystems.
  - Stateful Remote Objects
  - it fully utilizes all Object-Oriented Programming (OOP) principles, such as **encapsulation, inheritance, and polymorphism**
  - Easily enables remote interaction between remote objects.
- *Disadvantages:*
  - Limited to Java only.
  - Higher overhead due to serialization of Java objects.
  - Higher resource consumption under heavy load.

gRPC:

- *Advantages:*
    - Cross-language support and modern communication protocol (HTTP/2).
    - Lower latency and higher throughput, lower cpu usage than java rmi or apache thrift
    - Built-in support for streaming and robust error handling.
- *Disadvantages:*
    - requires proto file definitions and code generation tool

Apache Thrift:

- *Advantages:*
    - Cross-language support and provides flexible service definitions.
    - Simpler than gRPC in terms of configuration for basic RPC calls.
- *Disadvantages:*
    - Ecosystem and community support is smaller.
    - requires thrift file definitions and code generation tool

Suitability for Different Scenarios:

- **For Java-only environments:** Java RMI is easier to implement but may not scale as well under high loads.
- **For cross-language systems and microservices:** gRPC offers the best performance and modern features.
- **For lightweight RPC requirements with multiple language support:** Apache Thrift is a solid choice with simpler setup than gRPC.

5. Conclusion & Recommendations

Based on our experimental results, **gRPC** demonstrates the best overall performance in terms of latency and throughput, with lower CPU and RAM utilization compared to Java RMI and Apache Thrift. However, if your system is Java-centric and simplicity is key, Java RMI might be sufficient despite its higher resource usage. Apache Thrift provides a good middle ground, especially when you require a lightweight and flexible RPC framework across different languages.

Recommendations:

- **gRPC** is recommended for new, cross-language microservices due to its modern protocol and efficient performance.
- **Apache Thrift** is a viable alternative for projects requiring lightweight, multi-language support.
- **Java RMI** is best used in environments strictly limited to Java and where legacy systems are involved.