ADS Assignment 1: Services

Implementing a Message Management System with REST and gRPC

Bastien TALEC
Course: Advanced Distributed Systems (ADS)

September 16, 2024

Contents

1 Introduction

This report presents the design, implementation, and evaluation of a message management system built using two different service-oriented technologies: RESTful web services and gRPC. The objective is to explore the interoperability, implementation complexity, and communication performance of each technology.

2 System Design

The system is designed around a service interface called `Messenger`, which allows developers to manage messages in a topic-based subscription model. The key functionalities include storing, retrieving, deleting messages ...

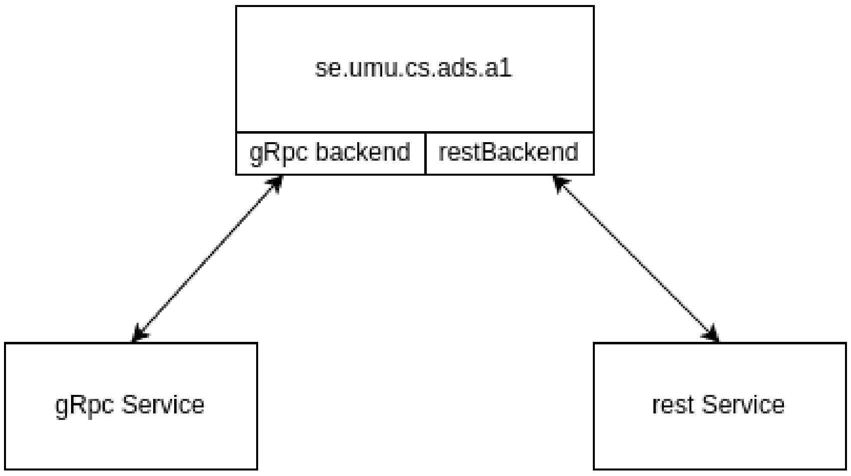

Figure 1: System Architecture of the Message Management System

There are two services that use different protocols to manage the distribution of the messenger data:

- `REST:` The REST service uses HTTP and JSON to transmit data, following a stateless communication model. It is easy to implement and widely supported, but can introduce overhead due to the verbose nature of JSON.
- `gRPC:` The gRPC service uses Protocol Buffers (Protobuf) for efficient, compact data serialization. It supports HTTP/2 and bi-directional streaming.

2.1 REST Service Interface

The REST API provides the following endpoints:

Method	Endpoint	Request Body	Response
POST	`/api/messenger`	`Message`	
POST	`/api/messenger/batch`	`Message[]`	
GET	`/api/messenger/{id}`		`Message`
POST	`/api/messenger/batch/retrieve`	`MessageId[]`	`Message[]`
DELETE	`/api/messenger/delete`	`MessageId`	
DELETE	`/api/messenger/batch/delete`	`MessageId[]`	
POST	`/api/messenger/subscribe`	`SubscriptionRequest`	`Topic[]`
POST	`/api/messenger/unsubscribe`	`SubscriptionRequest`	`Topic[]`
GET	`/api/messenger/users`		`Username[]`
GET	`/api/messenger/topics`		`Topic[]`
POST	`/api/messenger/topics/{username}`	`Username`	`Topic[]`
POST	`/api/messenger/subscribers`	`Topic`	`Username[]`
POST	`/api/messenger/message/user/{username}`	`Username`	`MessageId[]`
POST	`/api/messenger/message/topic`	`Topic`	`MessageId[]`

Table 1: Summary of REST API Endpoints

Notes:

- `{id}` and `{username}` are path variables representing the message ID and username, respectively.
- `SubscriptionRequest` = `{username, topic}`
- All request and response bodies are in JSON format.

2.2 gRPC Service Interface

The gRPC service `Messenger` provides the following RPC methods:

Méthode	**Requête**	**Réponse**	**Description**
`StoreMessage`	`Message`	`Empty`	Stores a single message.
`Store`	`MessageBatch`	`Empty`	Stores multiple messages in a batch.
`RetrieveMessage`	`MessageId`	`Message`	Retrieves a message by its ID.
`RetrieveMessages`	`MessageIdBatch`	`MessageBatch`	Retrieves multiple messages by their IDs.
`DeleteMessage`	`MessageId`	`Empty`	Deletes a message by its ID.
`DeleteMessages`	`MessageIdBatch`	`Empty`	Deletes multiple messages by their IDs.
`Subscribe`	`SubscribeRequest`	`TopicBatch`	Subscribes a user to a topic. Returns the list of topics subscribed.
`Unsubscribe`	`SubscribeRequest`	`TopicBatch`	Unsubscribes a user from a topic. Returns the list of topics unsubscribed.
`ListUser`	`Empty`	`UsernameBatch`	Retrieves a list of all users.
`ListTopic`	`Empty`	`TopicBatch`	Retrieves a list of all topics.
`ListTopicByUser`	`Username`	`TopicBatch`	Retrieves topics subscribed by a user.
`ListSubscriber`	`Topic`	`UsernameBatch`	Retrieves subscribers of a topic.
`ListMessageByUser`	`Username`	`MessageIdBatch`	Retrieves message IDs sent by a user.
`ListMessageByTopic`	`Topic`	`MessageIdBatch`	Retrieves message IDs in a topic.

Table 2: Summary of gRPC Service Methods

Notes:

- All request and response messages are defined in the `.proto` file.
- The service uses protocol buffers for message serialization.

3 Implementation of Services

3.1 RESTful Web Service Implementation

The RESTful services were implemented using the Spring Boot framework. Each method of the `Messenger` interface was exposed as a REST API endpoint, utilizing HTTP methods such as `GET`, `POST`, `DELETE`, etc. Spring Boot simplifies the process of building RESTful APIs by providing built-in features for routing, request handling and java object serialization. To implement my service, I reused the `Messenger` interface and all the type structures provided by the default framework. Reusing these elements eliminated the need for mapping data between the client and server, allowing for seamless communication and reducing development complexity.

3.2 gRPC Implementation

The gRPC service was implemented using Protocol Buffers (Protobuf) to define the service contract. Each method of the `Messenger` interface was mapped to a corresponding gRPC service method. Reusing the interface and data types generated from the Proto file specification streamlined the process. However, mapping between the client-server data structures is still required for communication.

Below is an example of how the message conversion is handled between the client-server communication:

Listing 1: Message conversion between client and server

```java
private MessengerOuterClass.Message convertMessage(Message message) {
    return MessengerOuterClass.Message.newBuilder()
        .setId(MessengerOuterClass.MessageId.newBuilder()
            .setValue(message.getId().getValue()).build())
        .setTimestamp(MessengerOuterClass.Timestamp.newBuilder()
            .setValue(message.getTimestamp().getValue()).build())
        ...
        .build();
}

private Message convertRpcMessage(MessengerOuterClass.Message message) {
    return new Message(
        new MessageId(message.getId().getValue()),
        new Timestamp(message.getTimestamp().getValue()),
        new Username(message.getUsername().getValue()),
        new Topic(message.getTopic().getValue()),
        new Content(message.getContent().getValue()),
        new Data(message.getData().getValue().toByteArray())
    );
}
```

4 Testing and Performance Evaluation

4.1 Testing Approach

Testing was automated using a test client that evaluates both service logic and communication performance. All services were executed locally on the same machine (localhost) to eliminate network variability and focus on the performance differences between REST and gRPC implementations.

For the performance tests of throughput and bandwidth, statistics were gathered over 10 executions to ensure statistical significance. The performance testing class utilized the `util.CsvMeasurementWriter` class to save the measurements in CSV format in the `measurement` folder.

Subsequently, two Python scripts, `generate_comparison_boxplots.py` and `generate_performance.py`, were used to generate graphs from the collected data for analysis and visualization.

4.2 Logic Test Plans

I have developed automated test plans to ensure the correct logical implementation of the remote functions provided by the `Messenger` interface.

The implemented test cases are:

- **Test storing and deleting a message** (`testStoreAndDelete(Message message)`):
  - Verify that storing a message increases the total message count by one.
  - Verify that deleting the message decreases the total message count by one.
- **Test storing and deleting multiple messages** (`testStoreAndDelete(Message[] messages)`):
  - Verify that storing multiple messages increases the total message count accordingly.
  - Verify that deleting these messages decreases the total message count accordingly.
- **Test storing and retrieving a message** (`testStoreAndRetrieve(Message message)`):
  - Verify that the stored message can be retrieved correctly.
- **Test storing and retrieving multiple messages** (`testStoreAndRetrieve(Message[] messages)`):
  - Verify that all stored messages can be retrieved correctly.
- **Test subscribing and unsubscribing** (`testSubscribeAndUnsubscribe(Username username, Topic topic)`):
  - Verify that subscribing a user to a topic increases their subscription count.
  - Verify that unsubscribing the user from the topic decreases their subscription count.
- **Test wildcards in topics** (`testWildCard()`):
  - Verify that subscribing to a topic with a wildcard correctly subscribes the user to all matching topics.
  - Verify that unsubscribing works correctly for topics with wildcards.
- **Test clearing all messages of a user** (`testClearAllMessagesOfUser(Username username)`):
  - Verify that all messages from a specific user can be deleted.
- **Test clearing all messages of a topic** (`testClearAllMessageOfTopic(Topic topic)`):

  - Verify that all messages associated with a specific topic can be deleted.

- **Test clearing all messages** (`testClearAllMessages()`):
  - Perform a complete deletion of all messages to ensure the system can be reset to its initial state.

4.3 Performance Test Cases and Results

- **Batch vs. Sequential Message Processing**

  Batch processing demonstrated significant performance gains over sequential processing in both REST and gRPC implementations.

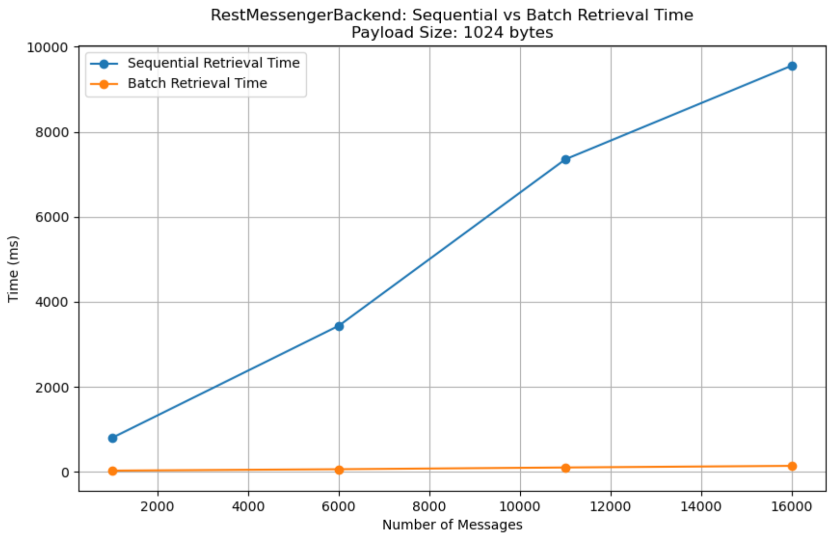

Figure 2: REST: Sequential vs. Batch Retrieval Time

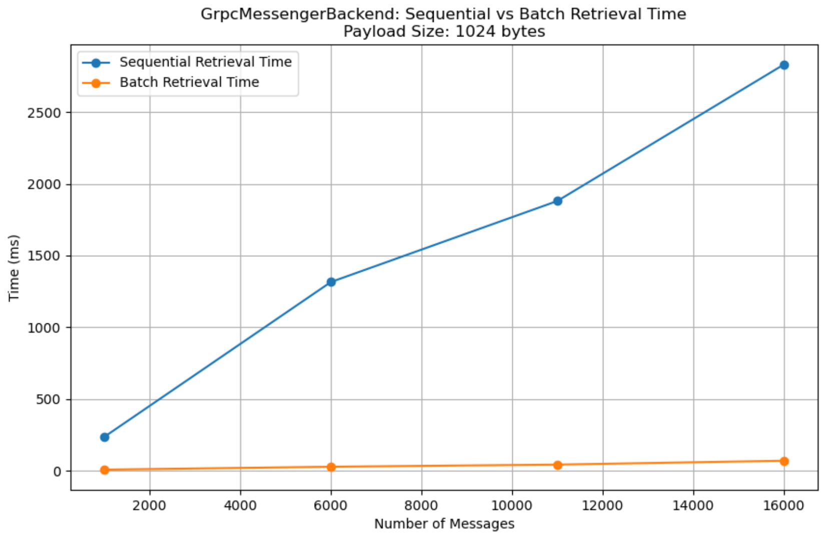

Figure 3: gRPC: Sequential vs. Batch Retrieval Time

Interpretation: Batch processing is faster than sequential processing because it reduces the overhead associated with multiple network requests. In sequential processing, each message retrieval requires a separate request-response cycle, incurring network latency and overhead for each message. Batch processing combines multiple messages into a single request, minimizing the number of network round-trips and reducing total latency. This effect is more pronounced in gRPC due to its support for streaming and efficient binary protocols, which allow for better utilization of network resources compared to REST.

- **Scalability with Number of Messages**

  As the number of messages increased, gRPC maintained better throughput and lower retrieval times compared to REST.

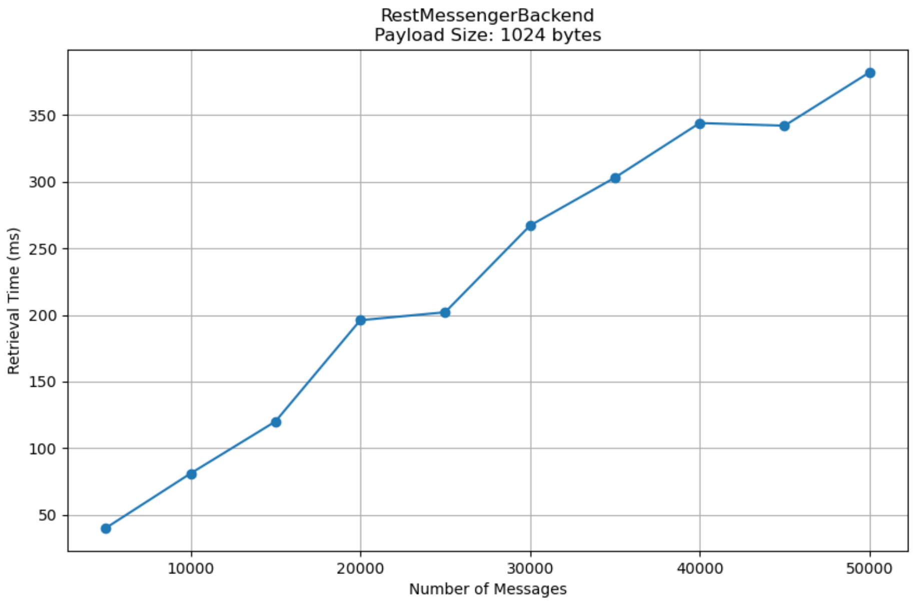

Figure 4: REST: Retrieval Time vs. Number of Messages

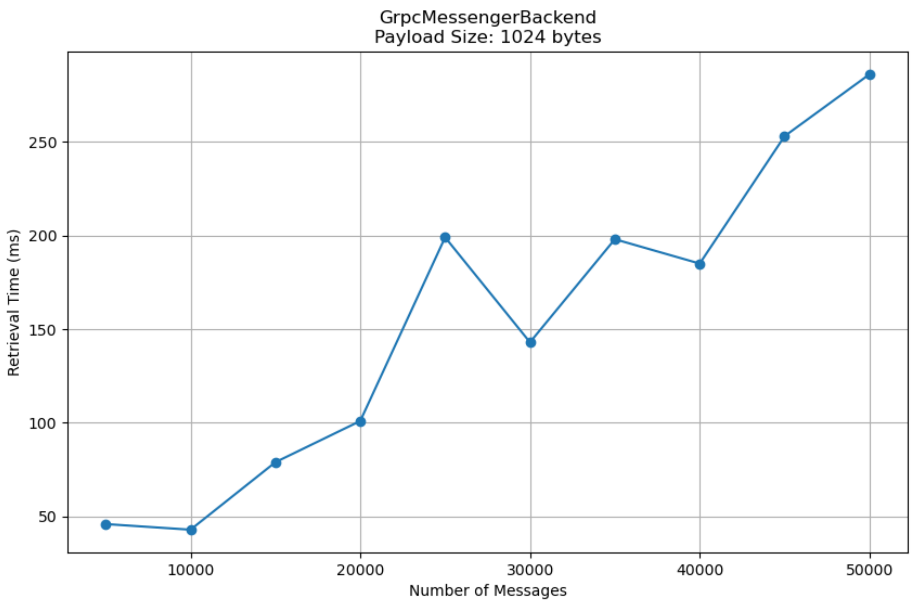

Figure 5: gRPC: Retrieval Time vs. Number of Messages

Interpretation: gRPC outperforms REST in handling a larger number of messages due to its use of HTTP/2, which enables features like multiplexing, header compression, and server push. Multiplexing allows multiple messages to be sent over a single connection simultaneously without interference, reducing connection overhead and latency. REST, typically built on HTTP/1.1, requires establishing new connections or uses less efficient techniques to handle multiple requests, resulting in increased latency as the number of messages grows.

- **Impact of Message Size**

  gRPC showed more efficient handling of larger messages compared to REST, owing to Protocol Buffers' compact binary format.

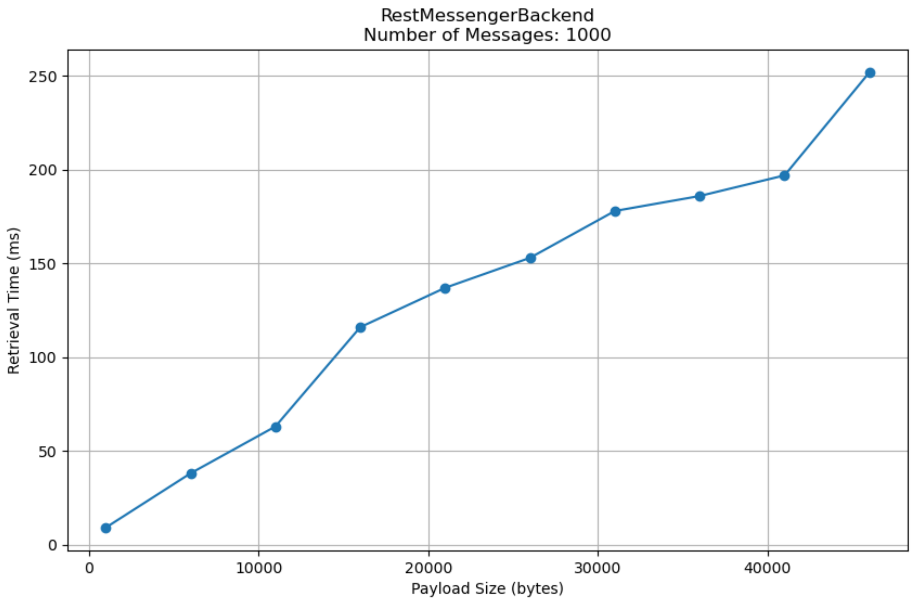

Figure 6: REST: Retrieval Time vs. Payload Size

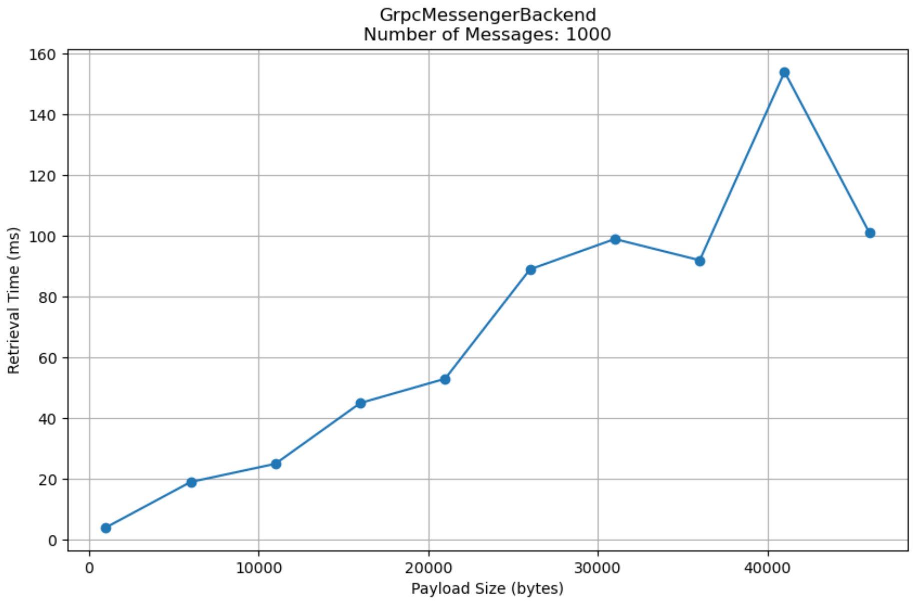

Figure 7: gRPC: Retrieval Time vs. Payload Size

Interpretation: The compact binary serialization used by gRPC (Protocol Buffers) results in smaller message sizes compared to REST's JSON format. Smaller messages consume less bandwidth and require less time for serialization and deserialization, especially noticeable with larger payloads. As message size increases, the efficiency gap widens, with gRPC handling larger messages more efficiently due to reduced parsing overhead and network transmission times.

4.4 Performance Metrics

The key metrics used for evaluation include:

- **Throughput Received Comparison**

  The figure below shows the throughput (number of messages received per second) for REST and gRPC services.

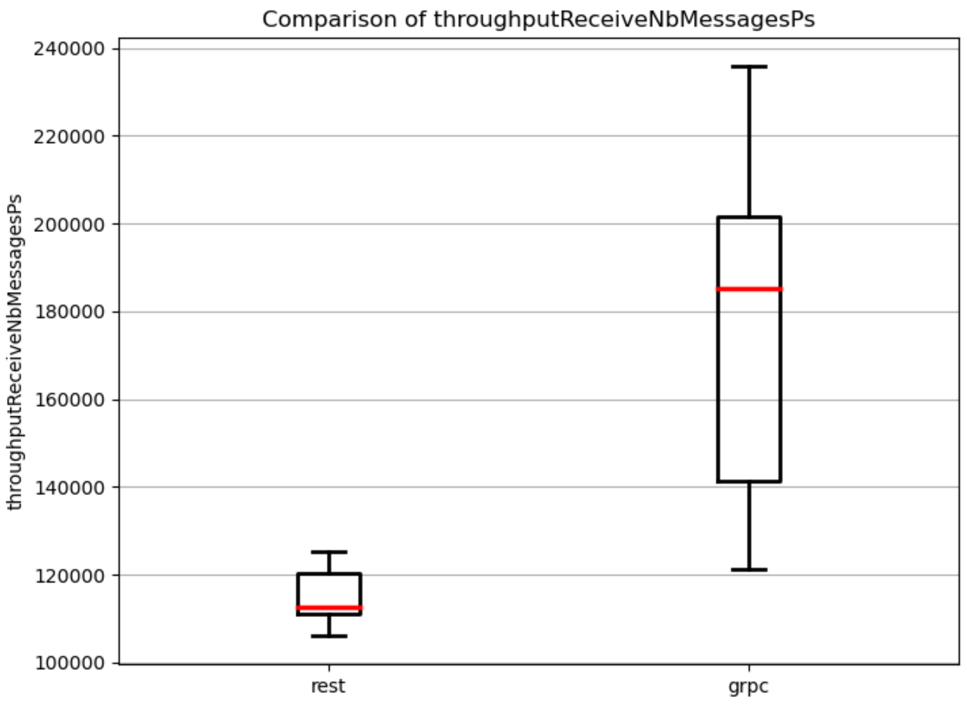

Figure 8: Throughput Received: REST vs. gRPC

  Interpretation: gRPC demonstrates higher throughput compared to REST. The efficient binary serialization and the use of HTTP/2 in gRPC allow for faster data transmission and handling of multiple messages concurrently. REST's reliance on HTTP/1.1 and text-based JSON serialization introduces more overhead, reducing the throughput.

- **Throughput Sent Comparison**

  The following figure illustrates the throughput (number of messages sent per second) when sending messages using REST and gRPC.

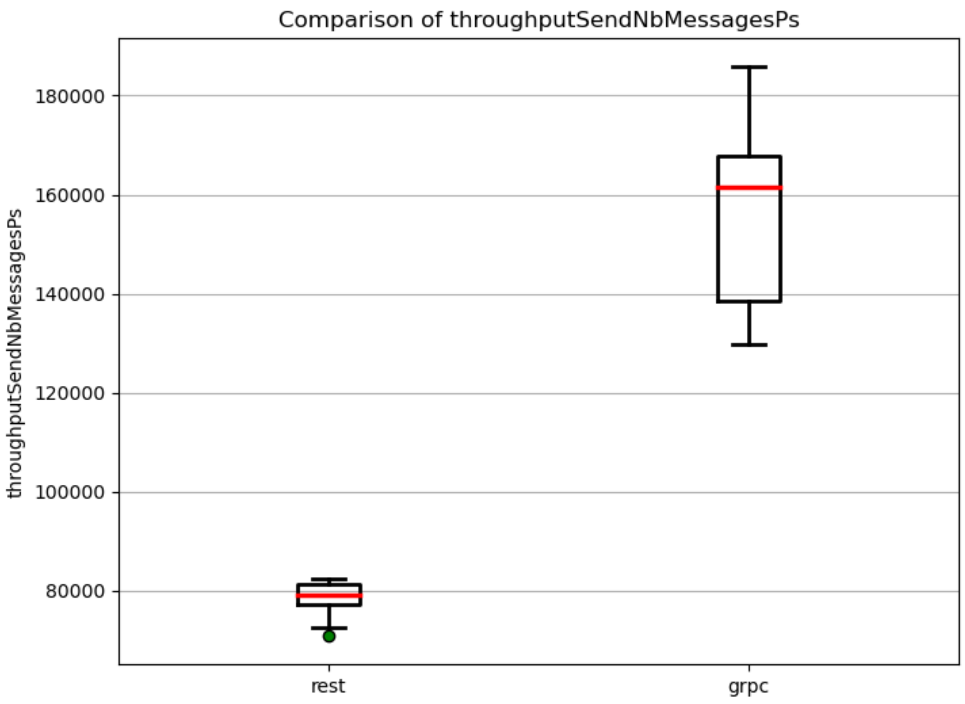

Figure 9: Throughput Sent: REST vs. gRPC

Interpretation: Similar to the received throughput, gRPC achieves higher sending throughput due to its efficient use of network resources and lower serialization overhead. REST's higher overhead results in slower message sending rates.

- **Bandwidth Usage Comparison**

  The following figure compares the bandwidth usage between REST and gRPC during message retrieval operations.

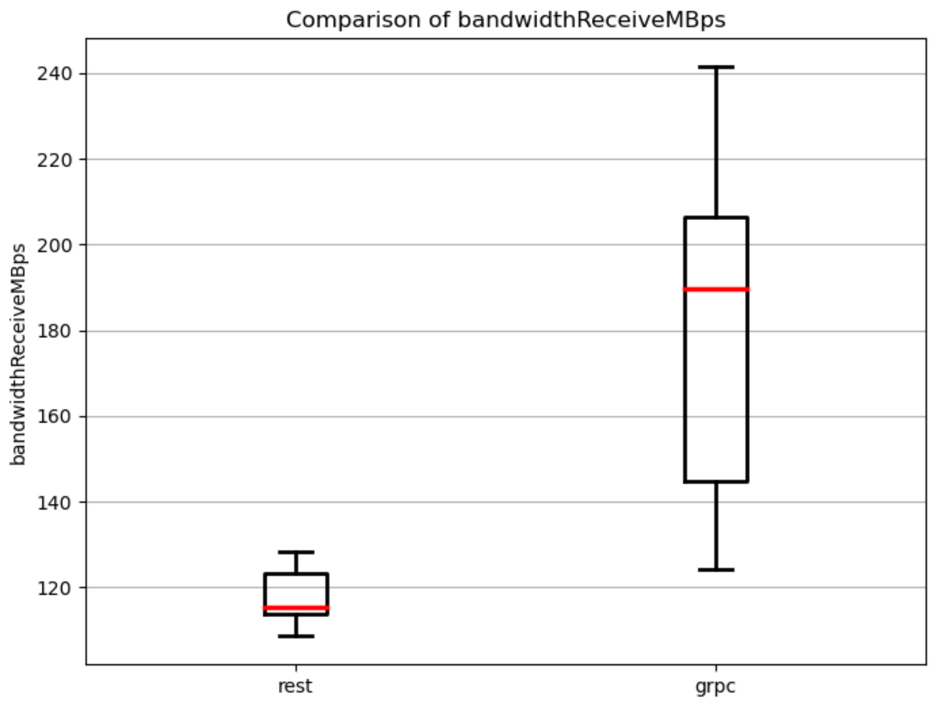

Figure 10: Bandwidth Usage: REST vs. gRPC

Interpretation: gRPC consumes more bandwidth compared to REST. This is because gRPC achieves higher throughput, sending and receiving more messages per second than REST. Even though gRPC uses Protocol Buffers, which serialize data into a compact binary format, the increased number of messages transmitted leads to higher total data transferred over the network.

5 Evaluation and Comparison

5.1 Implementation Complexity

Implementing REST services was straightforward due to the simplicity of the HTTP protocol and the wide availability of tools and libraries. REST uses standard HTTP verbs and JSON formatting, which are familiar to most developers and require no additional code generation steps. In contrast, gRPC required additional setup for Protocol Buffers and the definition of service contracts in the `.proto` files. The necessity to generate code from these proto files can be inconvenient and adds complexity to the development process.

Moreover, gRPC services may encounter issues with firewall traversal since they use HTTP/2 and often operate on ports that may not be open by default. Firewalls configured to allow only HTTP/1.1 traffic on standard ports might block gRPC communication, necessitating additional network configuration.

5.2 Communication Performance

gRPC outperformed REST in terms of throughput and response time, especially in scenarios involving large or numerous messages. The use of HTTP/2 and Protocol Buffers in gRPC contributed to reduced latency and bandwidth usage. Protobuf's efficient binary serialization was evident in the performance gains of gRPC over REST.

6 Conclusion

In this assignment, we implemented a message management system using both REST and gRPC technologies, highlighting the trade-offs between simplicity and performance. REST, with its straightforward setup and reliance on standard HTTP protocols and JSON, offers ease of implementation and wide compatibility. However, it exhibits higher latency and bandwidth usage.

gRPC provides superior communication performance due to its use of HTTP/2 and Protocol Buffers, resulting in lower latency and higher throughput. Nevertheless, it introduces added complexity, requiring code generation from `.proto` files and potentially encountering firewall traversal issues.

The choice between REST and gRPC depends on the specific needs of the project. REST is suitable when simplicity and rapid development are priorities. In contrast, gRPC is advantageous for applications that demand high performance and can accommodate the additional setup complexity. Balancing these factors is essential to selecting the appropriate technology for efficient distributed system communication.

References

[1] REST Web Services, `https://en.wikipedia.org/wiki/REST`

[2] gRPC, `https://en.wikipedia.org/wiki/GRPC`

[3] Protocol Buffers, `https://en.wikipedia.org/wiki/Protocol_Buffers`