



SC4051 Distributed System

Title: Project Report

Slot Number: 40

Time: 13:15 - 13:30

Student:

Choo Jin Cheng (100%)	U2121190C
Chua Woon Su Samuel (100%)	U2122421B
Lee Chun Yang (100%)	U2121175F

Table of contents

Table of contents	2
Background of Requirements	3
System Overview	3
Functionalities Overview	3
Client-Server Architecture	4
System Diagram	4
Request-Reply Design	5
Message Wrapper Format	5
Request Format	6
Reply Format	6
Monitor Format	7
Client Design & Implementation	8
Idempotent Function (Clear File Content)	8
Non- Idempotent Function (Copy File)	8
Client-side Caching	9
Server Design & Implementation	10
Experiments on Invocation Semantics	11
At least once	11
At most once	12

Background of Requirements

System Overview

Our project aims to develop a distributed file system using the User Datagram Protocol (UDP) as the underlying communication protocol. This system facilitates remote file access and management through a client-server architecture, allowing seamless interaction between users and files across a network.

Functionalities Overview

Client Side

Monitor File

- Monitors a certain file for updates for a fixed duration specified by the user. The user will be notified whenever the file gets updated for a specified duration.

Read from File

- Reads a section of a file based on how much offset from the start, and how many bytes to read until.

Insert into File

- Updates a section of a file based on how much the offset from the start with the content that you are going to insert with.

Copy a File (Non-Idempotent)

- The Copy File function is used to make a copy of an existing file on the server.

Get Attributes

- The Get Attributes command gets the last modified time of a file from the server.

Freshness

- Update freshness interval in the client for cache based on the duration specified by the user.

Clear File Content (Idempotent)

- The Clear File Content function is used to clear the file content of an existing file on the server.

Server Side

Alert Client

- Notify the client of the recent updates to their subscribed file by sending an alert message to their side.

Set Invocation Semantics

- Enable server administrators to initiate the server with the flexibility to choose between AtLeastOnce or AtMostOnce invocation semantics.

Client-Server Architecture

System Diagram

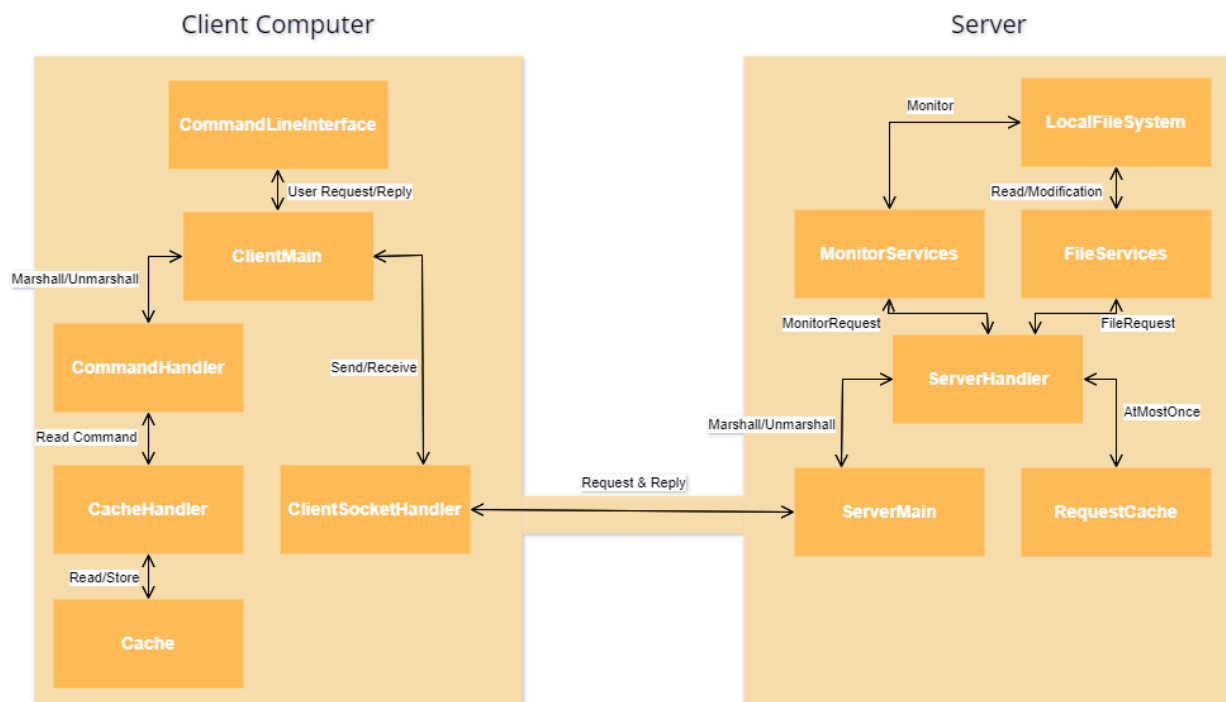


Figure 1: System Overview Diagram

Figure 1 illustrates a general workflow involving various system modules. Initially, the client interacts with the command line interface (CLI) to select their desired function. The inputs are then routed through ClientMain to CommandHandler for processing. In the case of a read command, the system checks if the specified offset and bytes are cached; if found, the data is retrieved and displayed. Otherwise, CommandHandler marshals the request and forwards it to ClientMain, which sends it to ClientSocketHandler for transmission to the server. On the server side, ServerMain receives the request and delegates it to ServerHandler. If employing AtMostOnce invocation semantics, the server verifies the messageID against its history before processing. Upon unmarshalling, the ServerHandler executes the appropriate FileServices function for the read command, generates a reply message upon success, and sends it back to the client via ClientSocketHandler. The reply is then passed through ClientMain and CommandHandler before being displayed on the CLI.

Request-Reply Design

Message Wrapper Format

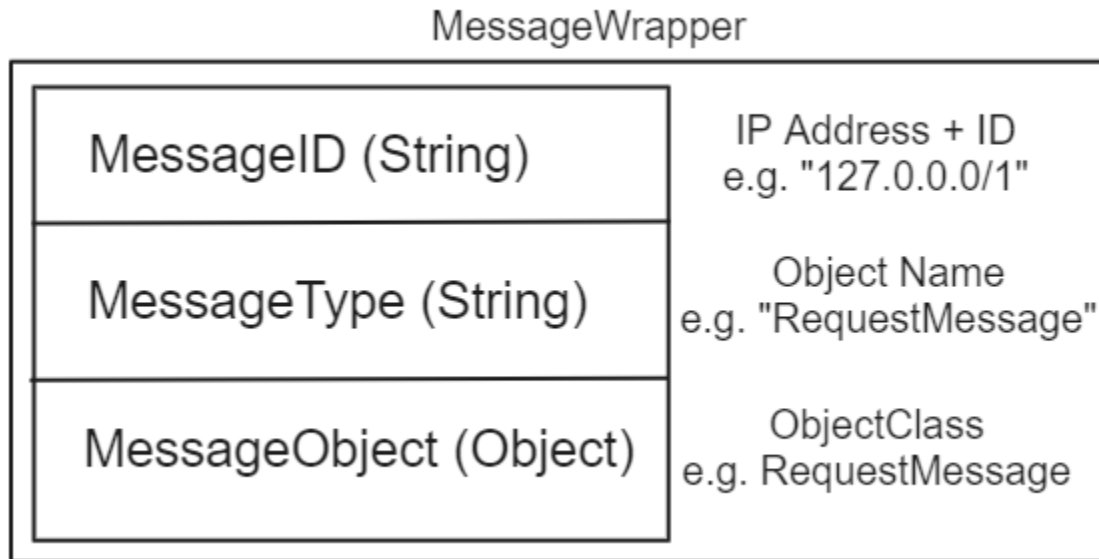


Figure 2: Message Wrapper Format Diagram

The MessageWrapper serves as the cornerstone for communication within our system. It comprises three essential fields: MessageID, MessageType, and MessageObject. MessageID combines the user's IP address with an incremental count for server-side message tracking. This unique identifier allows us to trace individual user requests effectively. MessageType plays a crucial role in identifying the format of the content encapsulated within MessageObject. This distinction enables seamless unmarshalling and marshalling of MessageObject content between clients and servers. MessageObject constitutes the heart of the message, housing the user-requested operations and server-side replies. Its format can be tailored to suit the requirements of either party, provided both agree on the new format's implementation. This design ensures scalability and simplicity. By adhering to a standard format like MessageWrapper, we maintain consistency and flexibility. Even as new operations or needs emerge, we can easily extend the system by incorporating new message formats into MessageObject, thus sidestepping the need for disruptive overhauls. As depicted in Figure 2, the MessageID is structured as a String, combining the user's IP address with a slash and a unique count number. Meanwhile, MessageType corresponds directly to the class name of the ObjectMessage Java class. Within the MessageObject, we encapsulate all the essential fields required to execute specific operations within the system. This design ensures clarity and efficiency in communication, enabling seamless interaction between different components of the system.

Request Format

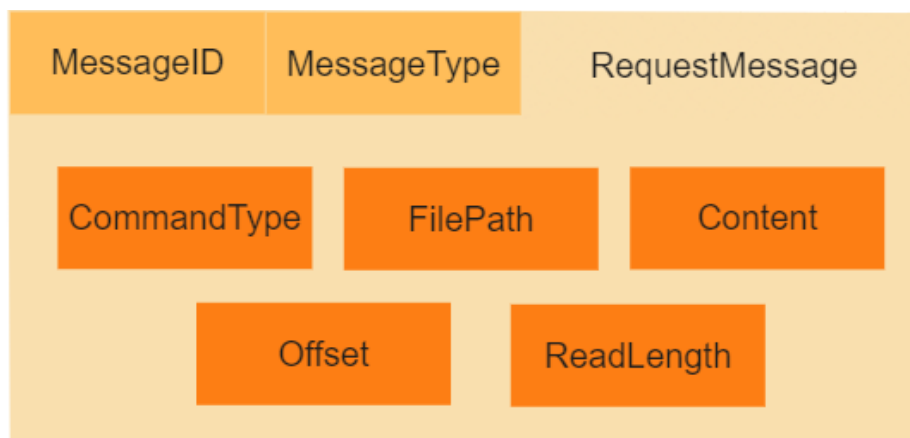


Figure 3: Request Message Format Diagram

As demonstrated in Figure 3, this request format supports most of the fundamental file operations, like reading and inserting etc. Each MessageObject adheres to a standardized format, comprising three essential fields: CommandType, FilePath, and Content. These fields serve as foundational pillars for all file-related operations within the system. CommandType delineates the nature of the operation, FilePath specifies the target file, and Content acts as a container for system messages or file contents.

Reply Format

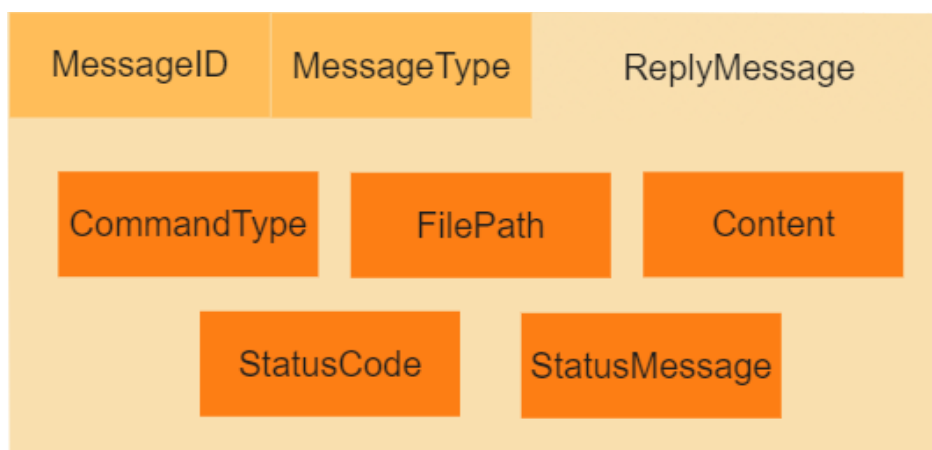


Figure 4: Reply Message Format Diagram

As depicted in Figure 4, our reply format includes two supplementary fields beyond the standard HTTP status code: a status code and a corresponding status message, offering a comprehensive response framework. The status code aligns with conventional HTTP standards, while the status message provides a human-readable explanation of the code's significance. In cases of errors, the content field accommodates detailed error messages, ensuring clarity and precision in communication between the server and client.

Monitor Format

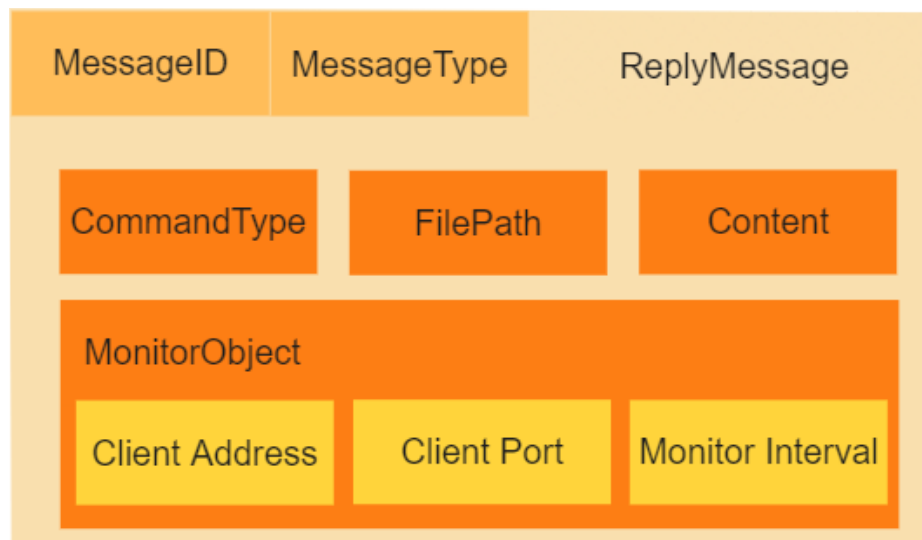


Figure 5: Reply Message Format Diagram

Figure 5 showcases an illustrative example of an add-on designed specifically to cater to additional functionalities, such as monitoring file content changes. This specialized message format necessitates the inclusion of supplementary information to effectively execute the monitoring operation. By extending the capabilities of our system in this manner, we ensure its adaptability to diverse use cases and requirements, thereby enhancing its versatility and utility.

Client Design & Implementation

Idempotent Function (Clear File Content)

```
read test.txt 0 10
[2024-03-31 21:02:16.43] Received from server:
  MessageID:192.168.0.100/2, Command:READ, StatusCode:200, StatusMessage:Success
  Content:hello worl
clear test.txt
[2024-03-31 21:02:45.285] Received from server:
  MessageID:192.168.0.100/3, Command:CLEAR, StatusCode:200, StatusMessage:Success
  Content: File Content Clear successful.
read test.txt 0 10
Cache length: 10, Input Length: 10
[2024-03-31 21:02:53.929] Received from server:
  MessageID:192.168.0.100/4, Command:READ, StatusCode:400, StatusMessage:Bad Request
  Content: Error: There is no content in the file.
read test.txt 0 10
Cache length: 10, Input Length: 10
[2024-03-31 21:05:41.07] Received from server:
  MessageID:192.168.0.100/6, Command:READ, StatusCode:400, StatusMessage:Bad Request
  Content: Error: There is no content in the file.
```

Figure 6: Clearing an Existing File with Content

The Clear File Content function is used to clear the file content of an existing file on the server. It is idempotent as the file content will always be empty regardless of how many times you call this function as shown in Figure 6.

If the file that is being content cleared does not exist, it would just send back an error message saying the file does not exist and no file would be cleared of its content.

Non- Idempotent Function (Copy File)

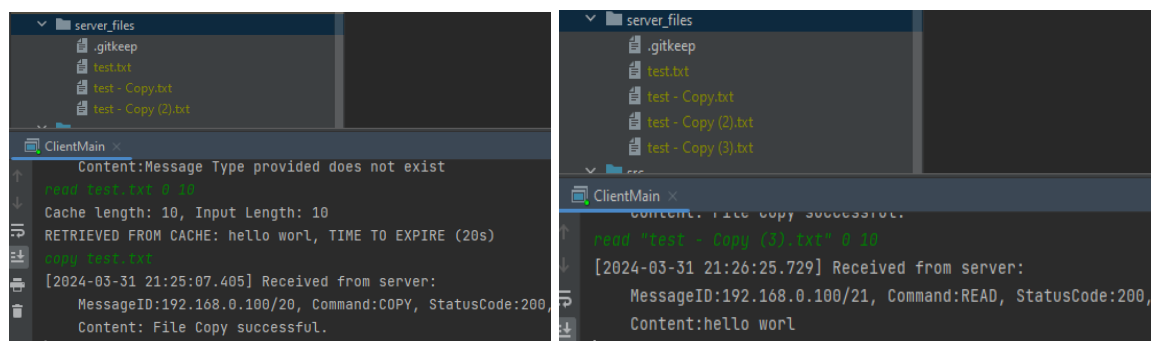


Figure 7: Copying an existing File (Before & After)

The Copy File function is used to make a copy of an existing file on the server. It is non-idempotent as each call to this function will create a copy of the selected file with a different name as shown in Figure 7.

When it copies “test.txt”, it would append “ - Copy” before “.txt” for the new file name. However, since both “test - Copy.txt” and “test - Copy(2).txt” already exist, the number in the file name would continue to increment until it checks that the file name is unique. Hence “test - Copy(3).txt” is created, with the content from the original file “test.txt” copied over as shown in Figure 7.

If the file that is being copied does not exist, it would just send back an error message saying the file does not exist and no file would be copied.

Client-side Caching

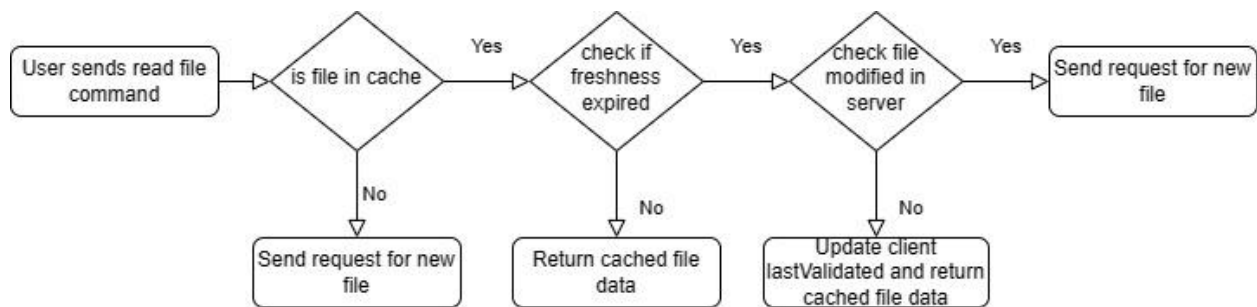


Figure 8: Client Caching Flow

Every time the user sends a read command to the server with the filename, offset, and length parameters, the client will first check the cache to see whether the data already exists in the cache as shown in Figure 8.

1. If the data does not exist in the cache, the client will send a request to the server and store the response text in the cache if the request is successful.
2. If the data exists in the cache, the cache will first check whether the file is still fresh, if it is fresh, it will just return the requested message to the user. If it is not fresh, it will check with the server to see whether the file has been modified since the last validation. If it is modified, the client will send a request for the new file as per normal. If it is not modified, the client will update the last validated field of the file and output the requested message to the user from the cache.

The cache is a hashmap with the key being the filename and the value being a ClientCacheData object. The ClientCacheData has the offset, length, content, serverLastModifiedTime and clientLastValidated fields.

The reason for storing the offset and length is so that we are able to handle requests of the user as long as the offset+length is between or equal to the cached data’s offset+length. This allows the cache to handle most requests of the user as especially if the cached file data length is large.

Server Design & Implementation

The server is structured into three key components: `ServerMain`, `ServerHandler`, and `FileServices`, which facilitate file operations. Its primary function is to listen for incoming requests, decode and route them to the `ServerHandler` for processing, and then encode and dispatch the resulting responses back to the client.

In `ServerMain`, the central hub of UDP socket operations, tasks like port management, message reception, and simulation of network loss scenarios are handled. Incoming messages are captured, encapsulated in datagrams, and converted into byte arrays for further processing by `ServerHandler`. Additionally, `ServerMain` oversees simulations of request and reply message losses, simulating real-world network conditions.

`ServerHandler` takes charge of message processing, encompassing tasks such as marshalling, unmarshalling, message updates, and creation. It adheres to invocation semantics like `AtLeastOnce` or `AtMostOnce`, managing message processing accordingly. To prevent duplicate messages in `AtMostOnce` scenarios, `ServerHandler` maintains a map of received requests, clearing out expired entries periodically. Its central function revolves around identifying message classes, unmarshalling them, and executing the requested operations using the unmarshalled data through `FileServices` on the server's local files. Upon completion, a new reply message class is generated, its data updated, and then it is marshalled back into byte arrays for transmission in `ServerMain`.

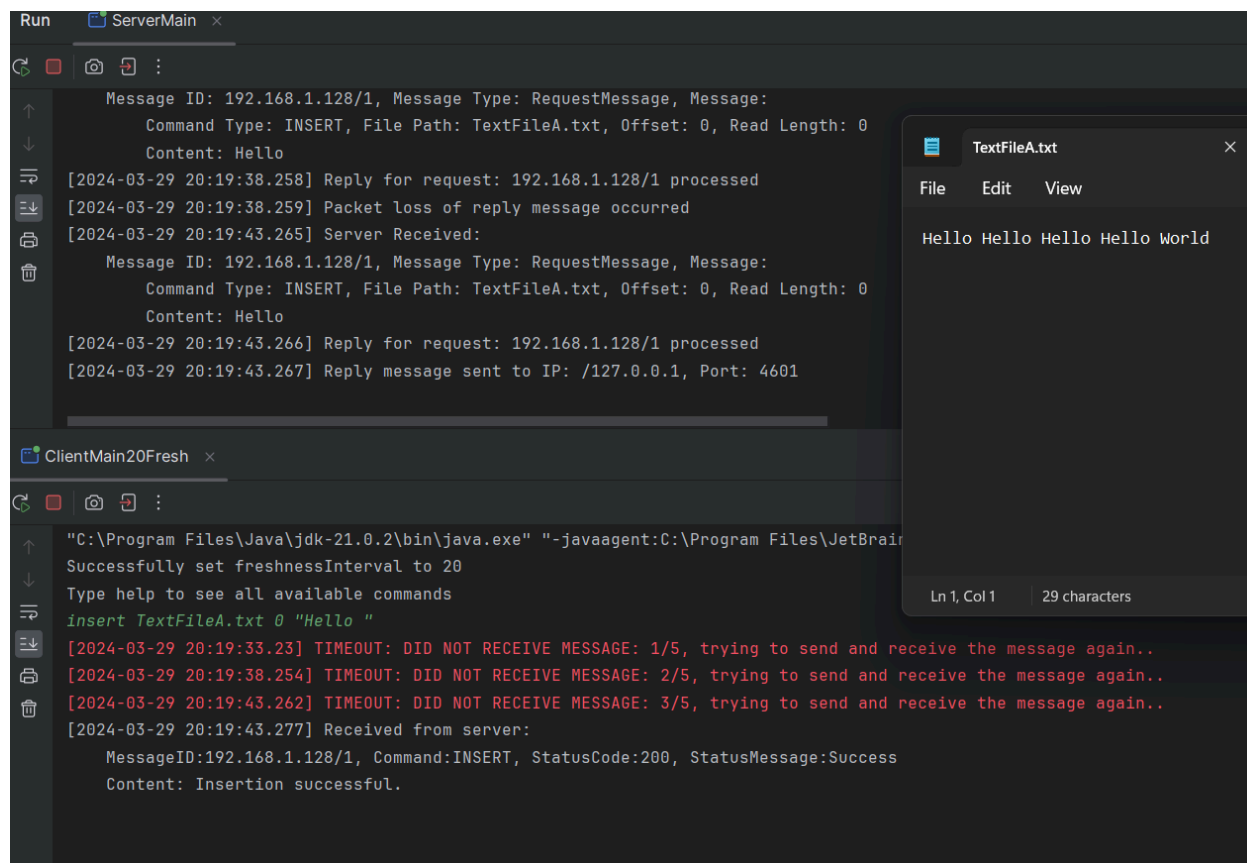
`FileServices`, currently comprising `FileAccessService` and `FileMonitorService`, form the backbone of the system's functionality. `FileAccessService` facilitates basic file operations, enabling remote file access for clients. Meanwhile, `FileMonitorService` monitors specific file changes and dispatches alerts to subscribed clients. These services represent the core operations performed on the system's files, underpinning its functionality and ensuring seamless client-server interaction.

`ServiceHandler` is structured with a dedicated process method for every message class, fostering flexibility for accommodating future system requirements seamlessly. This design ensures that introducing new processes for additional message classes will not disrupt ongoing operations. Moreover, if a new operation reuses an existing message class, implementing it is as simple as integrating the new functionality within the same method associated with that message class.

Likewise, each service class is thoughtfully designed to handle operations exclusively pertinent to its functionality. This meticulous design approach guarantees that services performing independent functions remain decoupled from one another. Consequently, any future enhancements or modifications will exert minimal impact on existing functionalities, fostering a modular and adaptable system architecture.

Experiments on Invocation Semantics

At least once



```
Run ServerMain x
Message ID: 192.168.1.128/1, Message Type: RequestMessage, Message:
  Command Type: INSERT, File Path: TextFileA.txt, Offset: 0, Read Length: 0
  Content: Hello
[2024-03-29 20:19:38.258] Reply for request: 192.168.1.128/1 processed
[2024-03-29 20:19:38.259] Packet loss of reply message occurred
[2024-03-29 20:19:43.265] Server Received:
  Message ID: 192.168.1.128/1, Message Type: RequestMessage, Message:
    Command Type: INSERT, File Path: TextFileA.txt, Offset: 0, Read Length: 0
    Content: Hello
[2024-03-29 20:19:43.266] Reply for request: 192.168.1.128/1 processed
[2024-03-29 20:19:43.267] Reply message sent to IP: /127.0.0.1, Port: 4601

ClientMain20Fresh x
"C:\Program Files\Java\jdk-21.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrain
Successfully set freshnessInterval to 20
Type help to see all available commands
insert TextFileA.txt 0 "Hello "
[2024-03-29 20:19:33.23] TIMEOUT: DID NOT RECEIVE MESSAGE: 1/5, trying to send and receive the message again..
[2024-03-29 20:19:38.254] TIMEOUT: DID NOT RECEIVE MESSAGE: 2/5, trying to send and receive the message again..
[2024-03-29 20:19:43.262] TIMEOUT: DID NOT RECEIVE MESSAGE: 3/5, trying to send and receive the message again..
[2024-03-29 20:19:43.277] Received from server:
  MessageID:192.168.1.128/1, Command:INSERT, StatusCode:200, StatusMessage:Success
  Content: Insertion successful.
```

TextFileA.txt

File Edit View

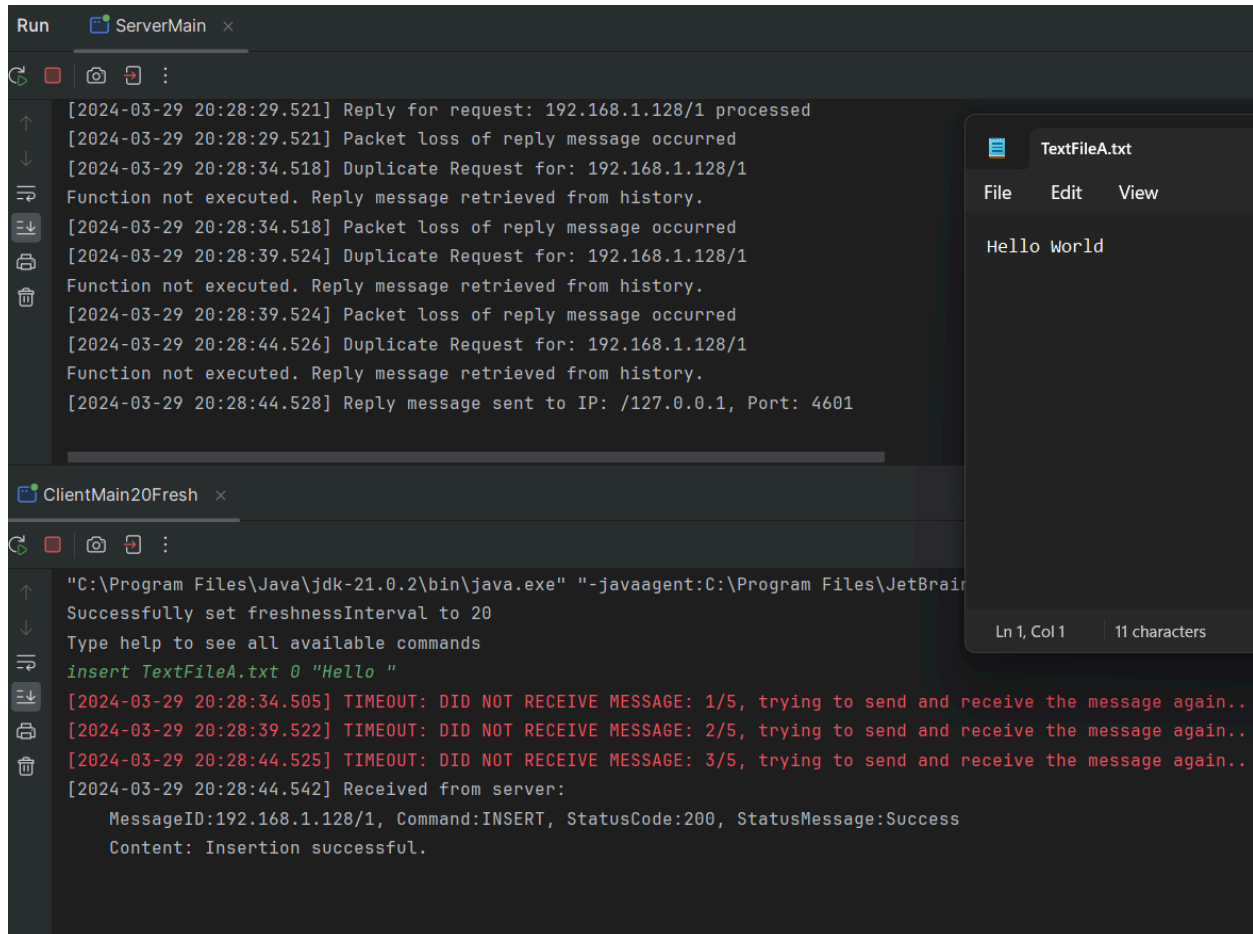
Hello Hello Hello Hello World

Ln 1, Col 1 29 characters

Figure 9: Insert on AtLeastOnce Experiment Diagram

In a system with at least once invocation semantics, idempotent operations like reading from a source will not cause incorrect behaviour since repeated readings yield the same result. However, non-idempotent operations like insertion can lead to unintended outcomes, such as adding extra values to the source if performed multiple times. In this experiment, I inserted "Hello " into a file named TextFileA.txt, which already contained "World" at offset 0, expecting the file to read "Hello World" afterwards. Due to packet loss on the server side, the client's timeout feature will retransmit its request until receiving a response. This resulted in the server processing the request multiple times, altering the file's content to "Hello Hello Hello Hello World," contrary to the intended outcome. Figure 8 illustrates where the client has sent a total of 4 requests before receiving a reply from the server side. Such behaviour aligns with the expected behaviour of non-idempotent operations in a system with at least once invocation semantics, where the client is ensured that its request has been processed at least once upon receiving a reply. It is worth mentioning that in scenarios of request packet loss, no operation occurs on the server, thus avoiding any unintended effects.

At most once



```
[2024-03-29 20:28:29.521] Reply for request: 192.168.1.128/1 processed
[2024-03-29 20:28:29.521] Packet loss of reply message occurred
[2024-03-29 20:28:34.518] Duplicate Request for: 192.168.1.128/1
Function not executed. Reply message retrieved from history.
[2024-03-29 20:28:34.518] Packet loss of reply message occurred
[2024-03-29 20:28:39.524] Duplicate Request for: 192.168.1.128/1
Function not executed. Reply message retrieved from history.
[2024-03-29 20:28:39.524] Packet loss of reply message occurred
[2024-03-29 20:28:44.526] Duplicate Request for: 192.168.1.128/1
Function not executed. Reply message retrieved from history.
[2024-03-29 20:28:44.528] Reply message sent to IP: /127.0.0.1, Port: 4601
```

```
"C:\Program Files\Java\jdk-21.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\lib\idea_rt.jar=12345:C:\Program Files\Java\jdk-21.0.2\bin" -Dfile.encoding=UTF-8
Successfully set freshnessInterval to 20
Type help to see all available commands
insert TextFileA.txt 0 "Hello "
[2024-03-29 20:28:34.505] TIMEOUT: DID NOT RECEIVE MESSAGE: 1/5, trying to send and receive the message again..
[2024-03-29 20:28:39.522] TIMEOUT: DID NOT RECEIVE MESSAGE: 2/5, trying to send and receive the message again..
[2024-03-29 20:28:44.525] TIMEOUT: DID NOT RECEIVE MESSAGE: 3/5, trying to send and receive the message again..
[2024-03-29 20:28:44.542] Received from server:
  MessageID:192.168.1.128/1, Command:INSERT, StatusCode:200, StatusMessage:Success
  Content: Insertion successful.
```

TextFileA.txt

File Edit View

Hello World

Ln 1, Col 1 11 characters

Figure 10: Insert on AtMostOnce Experiment Diagram

In a system with at most once invocation semantics, both idempotent and non-idempotent operations avoid unintended effects because this approach ensures that the client receives confirmation of at most one operation per reply. In this experiment, I replicated a scenario previously tested under at least once semantics to observe potential differences. Figure 9 illustrates the client sending three requests before receiving a server reply due to reply packet loss, yet the file's content remained as expected ("Hello World"). This outcome stems from the server's filtering of duplicate requests, preventing redundant operations. This behaviour aligns with expectations for non-idempotent operations in such a system. Notably, both idempotent and non-idempotent operations, even in scenarios of request or reply packet loss, avoid unintended effects.