



CZ4013 Distributed Systems Project

Remote File Access System

Name	Matriculation Number	Contribution
Chan Zhao Yi	U2022081F	33.33%
Muhammad Irfan Bin Norizzam	U2021872E	33.33%
Jovian Nursan Ng	U2020037L	33.33%

Tables of Contents

Tables of Contents	1
1. Introduction	2
2. Client-Server Architecture Design	3
2.1. Network Diagram	3
3. Client Implementation	4
3.1. Client.java:	4
3.2. Functions.java:	4
3.3. ClientSideCache.java and CacheEntry.java:	5
4. Server Implementation	6
4.1. Server.java:	6
4.2. Functions.java:	8
5. Marshaling	9
5.1. Client-Side	9
5.2. Server-Side	9
6. Additional Operations	10
6.1. Idempotent Operation	10
6.2. Non-idempotent Operation	11
7. Invocation Semantics	13
7.1. At-least-once	13
7.2. At-most-once	13

1. Introduction

The objective of the project is to design and implement a system for remote file access based on client-server architecture. The system allows multiple clients to request services from the Flight Information server. The UDP Protocol is followed for inter-process communication.

We used Java to code both the client and server.

The requirements of the program include:

- The files are stored on the local disk of the server.
- The server program implements a set of services for clients to access the files remotely.
- The client program provides an interface for users to invoke these services.
- Upon receiving a request input from the user, the client sends the request to the server.
- The server performs the requested service and returns the result to the client.
- The client then presents the result on the console to the user.
- The client-server communication is carried out using UDP.

Functionalities of the program include:

- A service that allows a user to read the content of a file by specifying the file pathname, an offset (in bytes), and the number of bytes to read from the file. The service returns the given number of bytes of the file content starting from the designated offset in the file. If the file does not exist on the server or if the offset exceeds the file length, an error message should be returned.
- A service that allows a user to insert content into a file by specifying the file pathname, an offset (in bytes), and a sequence of bytes to write into the file. The service inserts the sequence of bytes into the file at the designated offset in the file. The original content of the file after the offset is pushed forward. If the file does not exist on the server or if the offset exceeds the file length, an error message should be returned.
- A service that allows a user to monitor updates made to the content of a specified file at the server for a designated time period called monitor interval. To register, the client provides the file pathname and the length of monitor interval to the server. After registration, the Internet address and the port number of the client are recorded by the server. During the monitoring interval, every time an update is made by any client to the content of the file, the updated file content is sent by the server to the registered client(s) through callback. After the expiration of the monitor interval, the client record is removed from the server which will no longer deliver the file content to the client.

2. Client-Server Architecture Design

2.1. Network Diagram

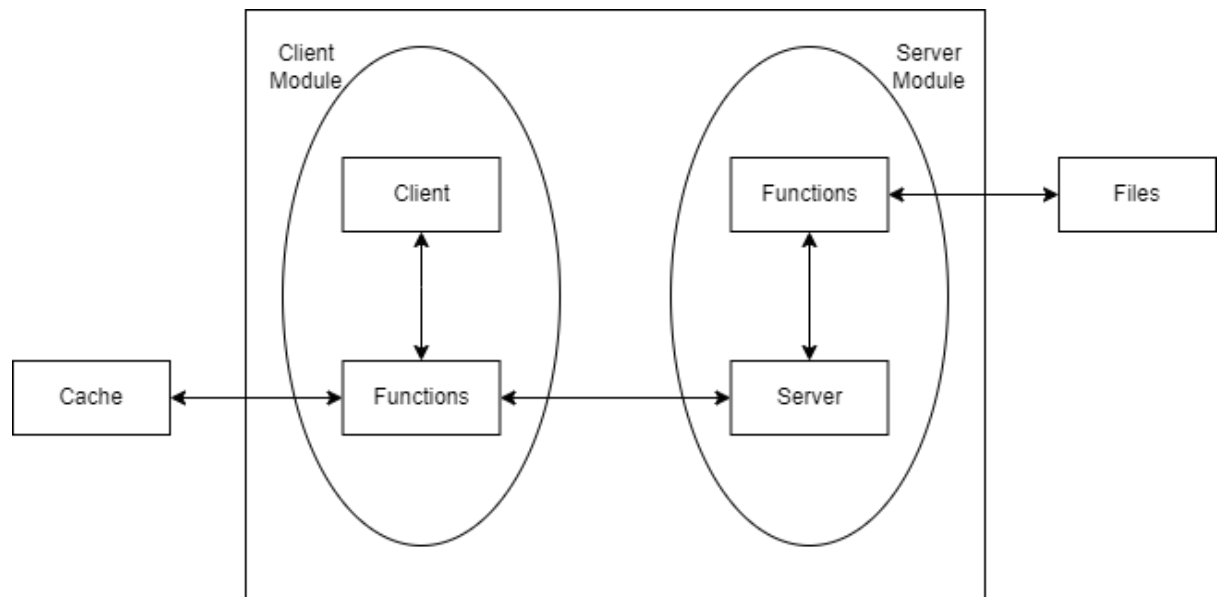


Figure 1: Network Diagram

The above figure depicts the content of a request message from a client to server. The datagram packet content contains an option, fileInput, and if needed additional inputs. These fields are of String type and concatenated using the character '|' as a separator such that the server is able to differentiate them. The option is a unique ID for identifying the request message and differentiating new requests from retransmissions. A fileInput indicates the path of the requested file. Additional input consists of offsetInput, byteInput, and contentInput. These inputs are required for knowing the offset, finding the byte size and changing the content of the file.

In cases where there's a duplicate request made by the client, with a different unique ID, the client will first check if the existing records are in the cache instance. If yes, it will retrieve the information from the cache, speeding up the read operation. Otherwise, it will send a request message to the server to retrieve the information.

After completion of the option operations, the return type of the response varies with respect to the option parameter. In the event of exceptions, the error message will be returned as a String type. The reply content would then be serialised into a byte array before being returned to the server.

3. Client Implementation

The client provides a console-based interface where the user can perform various operations, including creating files, editing files, monitoring files, setting invocation semantics, checking file existence, overwriting existing files and client-side caching. This functionality is implemented across four main files: Client.java, Functions.java, ClientSideCache.java and CacheEntry.java.

3.1. Client.java:

In the Client class, the initial setup involves initializing a DatagramSocket and a Scanner object to manage user input. The client's IP address is retrieved and stored in the serverAddress variable, and a DatagramSocket instance is created to establish communication with the server.

The core functionality of the client lies within a loop where user queries are continuously processed until the user decides to terminate the connection. During each iteration, the class obtains the user's request, marshals it into a byte array, encapsulates it in a DatagramPacket, and sends it to the server for processing. If the user selects the option to monitor files, a message indicating the ongoing monitoring process is sent to the user. Otherwise, the server responds with a message based on the user's query, which is unmarshalled into a ReplyMessage object and printed for the user's review.

For example, if the request is the third option, which is monitoring files, the Client class will send a message to the user indicating that monitoring is in progress. Otherwise, the server sends a reply message based on the user's query, which would be unmarshalled into a ReplyMessage object and printed out for the user to view.

3.2. Functions.java:

The Functions class complements the Client class by encapsulating core functionalities. Each method corresponds to a specific operation, such as reading files, editing files, or checking file existence. These methods orchestrate the marshalling of user inputs into a request message, which is then sent to the server for processing.

Upon receiving a response from the server, the Functions class unmarshals the data, extracting relevant information or error messages. The result is presented to the user via the console, facilitating a seamless user experience.

The client implementation prioritizes modularity and extensibility, allowing for easy integration of additional functionalities in the future. Robust error handling mechanisms are also in place to ensure reliability in the face of unexpected events.

3.3. ClientSideCache.java and CacheEntry.java:

The ClientSideCache.java uses the CacheEntry class as an object to store the information regarding the content stored in the client-side cache. The CacheEntry class stores information such as the name of the file being read from, the offset, bytes to read from the file, the content obtained from the file and the timestamp in which the content is cached inside the cache.

The ClientSideCache class is initialised by providing the freshness interval, the server address and the client socket. The ClientSideCache class focuses more on functions such as inputting the content into the cache, getting content from the cache and obtaining the content from the server if the file is not found in the cache. There are also functions such as informing the server when updates are made to the file content and enabling the client to obtain file content cached in the cache without the need of the server.

The ClientSideCache class is initialised in the Client Functions.java file. The functions of the ClientSideCache are mainly used in the ReadFile function and the EditFile function of Functions.java. After obtaining the specified content from the server through the ReadFile method, the content is stored in the cache, along with the file name, offset, number of bytes read and the timestamp of the content is specified as the system time when the content was received from the server (System.currentTimeMillis()). For the EditFile function, the content in the cache is updated on the offset, where the new content will be inserted in for the file. The timestamp of the cached content is also updated to the current time in which the content was updated.

The ClientSideCache class is also implemented with the freshness interval parameter. The freshness interval is used to determine if the cached content is still fresh and relevant or not. The freshness interval is usually between 3 - 60 seconds. How we determine if the cached content is still fresh or not is by computing the difference between the current system time in which we are trying to access the cached content, and the timestamp when the content was cached, or was last updated. If the time difference is greater than or equal to the freshness interval, the cached content is not considered valid anymore, and the newer content will be obtained from the server.

The approximate one-copy update semantics is also implemented in the ClientSideCache in the update file content function, where there is the option to update the server on the updates the client made on the file content. There is also a function on the Function.java file to print out the contents of the cache.

```
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent operation)
8. Exit
9. Print out cache content
-----
Choice: 9
-----
File: Bob.txt
Content: -----
Read File: Bob.txt offset: 0 bytes: 10
-----
changetime, Last Updated Time: 1712242156774
-----
Content: -----
Read File: Tom.txt offset: 0 bytes: 100
-----
Hi this is tom.
, Last Updated Time: 1712242109277
File: Chupapi.txt
Content: -----
Read File: Chupapi.txt offset: 0 bytes: 20
-----
Chupapi Munayo
, Last Updated Time: 1712242141922
-----
```

This image shows an example of the content in the cache.

4. Server Implementation

The server-side implementation comprises the Server and Functions classes, responsible for handling incoming client requests, processing them, and providing appropriate responses. Below is an overview of the server implementation based on the provided code snippets:

4.1. Server.java:

The Server class serves as the entry point for the server-side application. It initializes a DatagramSocket on a specified port (in this case, port 5001) to listen for incoming client requests. Upon receiving a request, the server processes it and sends back a response to the client.

Within an infinite loop, the Server class continuously listens for incoming DatagramPackets. Upon receiving a packet, it extracts the client's IP address, port number, and the content of the packet. The received data typically contains information about the requested operation and any accompanying parameters.

The server then delegates the processing of the request to the appropriate method in the Functions class based on the operation specified. For example, if the client requests to read a file, the server calls the ReadFile method in the Functions class. Similarly, for other operations such as editing files or checking file existence, corresponding methods in the Functions class are invoked.

Upon processing the request, the server sends back a response to the client. This response contains the result of the requested operation or an error message if the operation encountered any issues.

4.2. Functions.java:

The Functions class encapsulates the core functionalities of the server. Each method within this class corresponds to a specific operation that the server can perform in response to client requests.

For example, the ReadFile method reads the content of a specified file from the server's local disk, while the EditFile method allows clients to modify the content of a file. Similarly, the CheckFileExistence method checks whether a specified file exists on the server.

Upon receiving a request from the Server class, the corresponding method in the Functions class processes the request. This may involve reading or modifying files, checking file existence, or performing other operations as required.

After processing the request, the Functions class prepares a response to be sent back to the client. This response typically contains the result of the requested operation or an error message if the operation encountered any issues.

Overall, the server implementation facilitates efficient communication between clients and the server, enabling clients to perform various file-related operations remotely. The modular design of the server allows for easy maintenance and scalability, making it suitable for handling a wide range of client requests in a distributed environment.

5. Marshaling

5.1. Client-Side

In the client-side implementation, marshaling is handled within the Functions class. When a user invokes an operation, such as reading or editing a file, the method in the Functions class responsible for that operation concatenates user input and other parameters into a string format. Delimiters, such as "|", separate different components of the message. This string is then converted into a byte array using the `getBytes()` method before being sent as a `DatagramPacket` to the server.

5.2. Server-Side

Upon receiving the request message, the server-side implementation in the Functions class unmarshals the data by converting the received byte array back into a string format. This string is then parsed to extract relevant parameters required to execute the requested operation. For instance, in the `ReadFile` method, the received string representing the request message is split using the delimiter "|" to extract parameters such as the file name, offset, and bytes to read.

Once the requested operation is executed, the server-side implementation marshals the response data into a string format using appropriate delimiters. This string is then converted into a byte array and sent back to the client as a `DatagramPacket`.

Upon receiving the response packet, the client-side implementation unmarshals the data by converting the byte array back into a string format. The client then parses the string to extract relevant information, such as the result of the operation or any error messages, before presenting it to the user via the console for review.

6. Additional Operations

6.1. Idempotent Operation

In our client-server architecture for remote file access, we have implemented an idempotent operation, namely the "Check File Existence" function. An idempotent operation has the same effect when applied multiple times, regardless of the number of repetitions. In the context of our system, the "Check File Existence" operation satisfies this property because it does not alter the state of the system and provides the same result regardless of how many times it is invoked.

In the client code, the option for the idempotent operation is presented as option 5 in the menu. When selected, the client sends a request to the server to check the existence of a specified file. The server, upon receiving this request, checks whether the file exists on its local disk and sends an acknowledgment message back to the client indicating the file's existence or non-existence. Importantly, this operation does not modify any file contents or system state, ensuring that repeated invocations do not produce different outcomes.

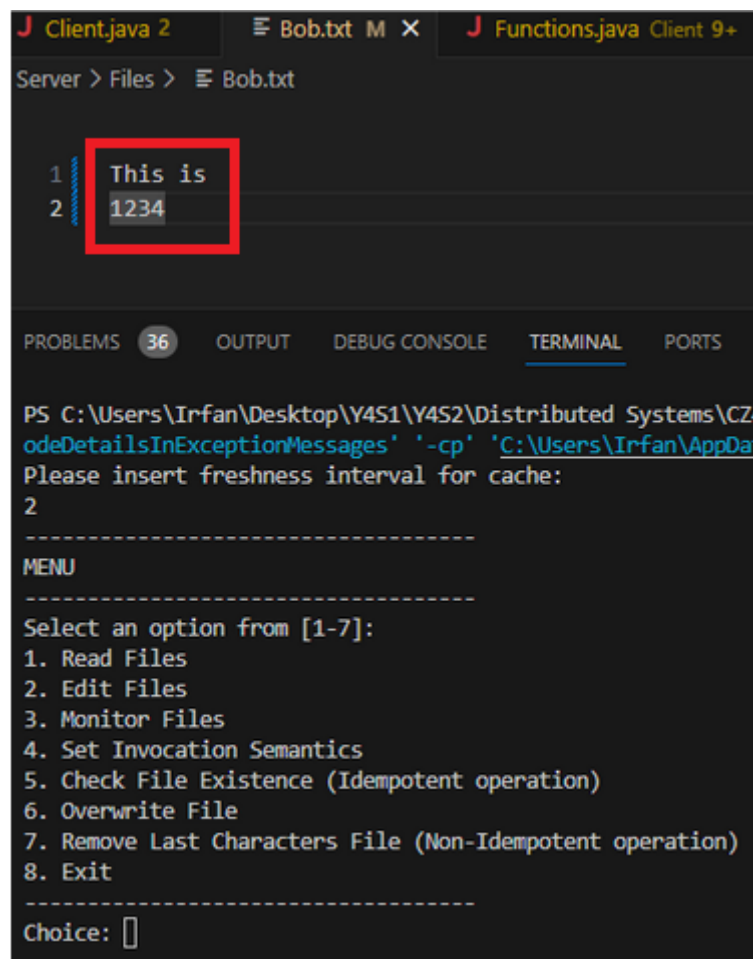
```
odeDetailsInExceptionMessages' '-cp' 'C:\Users\Irfan\AppData\
Please insert freshness interval for cache:
5
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent operation)
8. Exit
-----
Choice: 5
-----
Check File Existence
-----
Enter file name: Bob.txt
ACK: File exists.
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent operation)
8. Exit
-----
Choice: 5
-----
Check File Existence
-----
Enter file name: Bob.txt
ACK: File exists.
```

The image above is the demonstration of our idempotent function.

6.2. Non-idempotent Operation

In contrast to the idempotent operation, we have also implemented a non-idempotent operation in our system, namely the "Remove Last Character" service. A non-idempotent operation is one where executing the operation multiple times may result in different outcomes or alter the system state with each execution. In the case of our system, the "Remove Last Character" operation is non-idempotent because it modifies the content of the specified file each time it is invoked.

In the client code, the option for the non-idempotent operation is presented as option 7 in the menu. When selected, the client prompts the user to input the filename of the file from which to remove the last character. Upon receiving this request, the server locates the specified file, removes the last character from its content, and sends an acknowledgment message back to the client indicating the successful removal. Subsequent invocations of this operation will result in different file states, demonstrating its non-idempotent nature.



```
J Client.java 2  Bob.txt M X  Functions.java Client 9+
Server > Files > Bob.txt

1 This is
2 1234

PROBLEMS 36 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\Irfan\Desktop\Y4S1\Y4S2\Distributed Systems\CZ
odeDetailsInExceptionMessages' '-cp' 'C:\Users\Irfan\AppData
Please insert freshness interval for cache:
2
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent operation)
8. Exit
-----
Choice: 
```

Image above shows the original status of the Bob.txt File

```
Client.java 2 Bob.txt M X Functions.java Client
Server > Files > Bob.txt

1 This is
2 123

PROBLEMS 36 OUTPUT DEBUG CONSOLE TERMINAL POR
review' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp
Se92598fd295b844849e64b71b582\redhat.java\jdt_ws\CZ401
Please insert freshness interval for cache:
5
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent operati
8. Exit

Choice: 7
-----
Remove Last Character
-----
Enter file name: Bob.txt
ACK: Last character removed successfully.
-----
```

Image above shows status of the Bob.txt File after 1st iteration of non-idempotent function

```
Client.java 2 Bob.txt M X Functions.java
Server > Files > Bob.txt

1 This is
2 12

PROBLEMS 36 OUTPUT DEBUG CONSOLE TERMINAL
Remove Last Character
-----
Enter file name: Bob.txt
ACK: Last character removed successfully.
-----
MENU
-----
Select an option from [1-7]:
1. Read Files
2. Edit Files
3. Monitor Files
4. Set Invocation Semantics
5. Check File Existence (Idempotent operation)
6. Overwrite File
7. Remove Last Characters File (Non-Idempotent op
8. Exit

Choice: 7
-----
Remove Last Character
-----
Enter file name: Bob.txt
ACK: Last character removed successfully.
-----
MENU
```

Image above shows status of the Bob.txt File after 2nd iteration of non-idempotent function

7. Invocation Semantics

7.1. At-least-once

As for at-least-once invocation semantics, duplicate filtering is not needed. Therefore, there's no need to reference the message history of client requests that is stored in the server. To implement at-least-once semantics in our application, the client implements a timeout to resend request messages if no reply is received from the server. The server will always execute the operation upon receiving a request before sending a reply to the client.

In our experimental setup, we are focusing on two non-idempotent functions: inserting and removing content from a file by specifying the file pathname, an offset (in bytes), and a sequence of bytes to be written into or removed from the file. Due to the absence of duplicate filtering under this invocation semantics, our experiments conducted on these two functions will yield wrong results.

For our implementation, it has a server packet loss rate at 50%. Upon server packet loss, the client will repeatedly send requests to the server until it receives a reply. This may cause the functions to be executed more than once on the server. In the “insert” non-idempotent function, the server will insert additional identical content than the intended request from the client. Whereas for the “remove” non-idempotent function, the server will remove additional content than the intended request from the client.

7.2. At-most-once

As for at-most-once invocation semantics, it is imperative to reference the message history of client requests stored on the server, ensuring the filtration of duplicate requests to prevent any operation re-execution. Each client request is assigned a unique identifier, termed as the UID, which is stored in the message history. This enables the server to discern duplicate requests based on matching UID values. Upon receiving a duplicate request, denoted by an identical UID, the server refrains from re-executing the operation and simply resends the reply message to the client. This is very important for non-idempotent functions as they will result in incorrect results if executed more than once. The timeout implemented in the client always triggers request resending if no reply is received from the server, irrespective of the invocation semantics.

For the two non-idempotent functions, the server will always insert/remove the content in the file requested by the client. The experiments conducted on the non-idempotent functions will always yield correct results even if the server receives duplicate requests.