CE4013/CZ4013/SC4051 Distributed Systems

Lab Report

Design and Implementation of A System for Remote File Access

Name	Matriculation No.	Contribution	Percentage
Lee Alessandro	U2120619H	Server implementation	50%
Wesley Lim Cher Fong	U2122211K	Client implementation	50%

Table of Contents

1. Introduction

The object of this project is to design and implement a system for remote file access based on a client-server architecture, a foundational model in distributed computing where a client sends requests to a server which processes these requests, and subsequently returns the appropriate responses.

As stated in the project requirements, we have implemented a custom communication protocol atop User Datagram Protocol (UDP), allowing a client to marshal and send requests, while a server receives and unmarshal these requests. Likewise, the server marshals and sends responses, while the client receives and unmarshal these responses. Furthermore, the client is not only able to cache the file's data, but also able to register for callbacks and monitor non-idempotent operations occurring on the server that alter the state of the file of interest. The at-least-once and at-most-once invocation strategies were implemented to alter the server's behaviour depending on the type of invocation, and the UDP design has been modified to simulate packet loss based on user-input probability.

On top of the 3 methods we are tasked to implement ('Read', 'Write by insertion', 'Monitoring Service'), we have implemented 2 additional operations, namely 'Write by deletion' which allows the user to delete specific segments of the file content (non-idempotent), and 'Read File Info' which reads the file metadata (idempotent).

1.1 Generic Assumptions

Detailed assumptions (if any) are provided in later segments.

CLIENT & SERVER CONFIGURATION

Any client or server configuration that requires the user to input from the CLI (Command Line Interface) cannot be changed during run-time. The user is expected to terminate the client or server and reinitialise it to change their settings.

READ OPERATION

For efficiency purposes, the server processes READ operations to return the exact specified range to be read (denoted by offset + bytesToRead). This is preferred over the alternative where the server returns the entire file and the logic to READ the exact segment is handled in the client.

2. System Overview

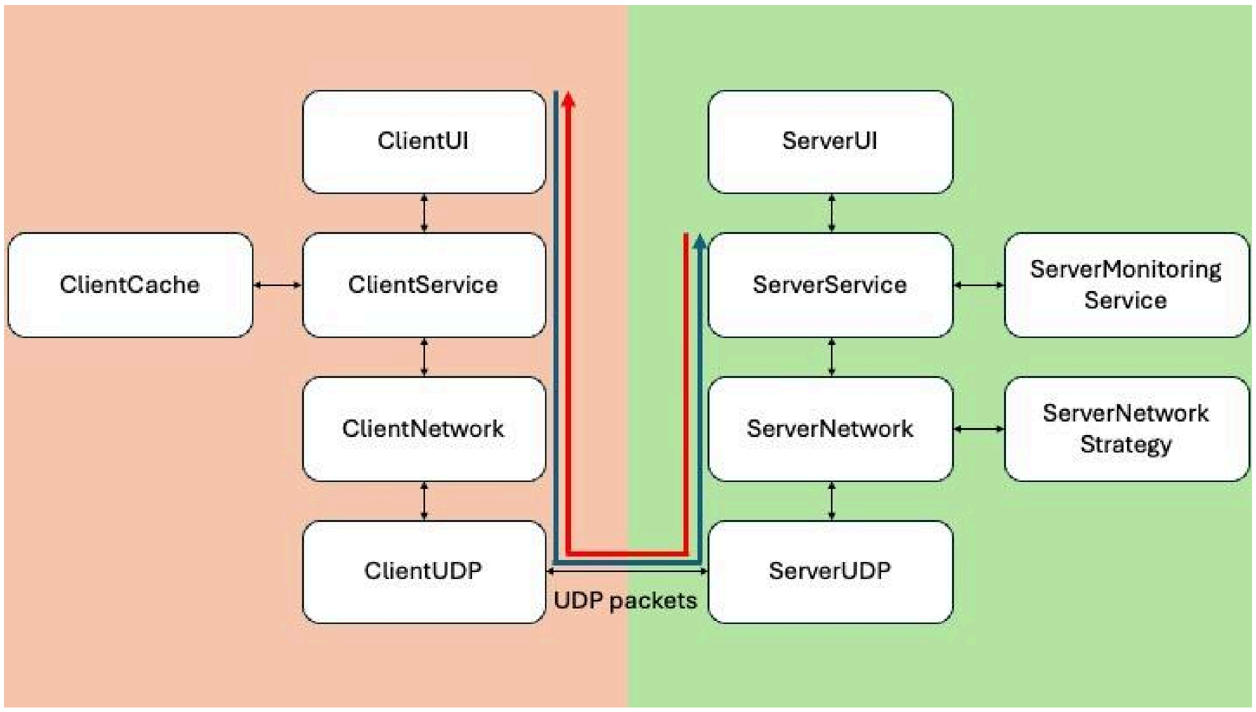

Figure 1: *System Architecture & Communication Flow*

Figure 1 represents the flow of communication in our distributed client-server architecture. The system architecture is designed such that there is a clear client-server architecture that strictly separates the client and server operations. The only communication between the client and server are through datagram packets that are sent via the ClientUDP and ServerUDP.

<u>Maintainability & Scalability</u>
The design is highly maintainable and scalable, as each layer has their defined roles, allowing the debugging process to be simpler since it is more isolated. With clear separation of concerns between each layer, our system is more scalable as we can accommodate the increasing functionality without affecting the entire system.

<u>Communication Flow</u>
Depicted by the blue arrow, when the client/user initiates a request, they are sent over to the server in the form of datagram packets. As shown by the red arrow, the server processes the request in its service layer, and sends back its response, also in the form of datagram packets. This is subsequently processed in the client's service layer before being displayed to the user in the CLI.

3. Design

3.1 Communication Protocol Design

We have adopted the approach of divergent structures for the Request and Response class, where the client makes use of the Request object to solely request for operations to be performed, while the server makes use of the Response object to solely provide results of the operations performed. The distinction in Request and Response structure allows for a clear bidirectional protocol where marshalling and unmarshalling operations are tailored to the specific objects that are being transmitted.

3.1.1 Request Design

The Request class encapsulates the necessary parameters to perform a specific operation on the server's file system. Table 1 clearly depicts the fields and its intended usage.

Attribute Name	**Data Type**	**Justification**	**Byte size**
requestId	long	Unique identifier for each request. Generated by taking the XOR of IP address in bytes and the current system time in nanoseconds.	8
operationType	OperationType enum	To specify the type of operation: READ, WRITE_INSERT, MONITOR, WRITE_DELETE, FILE_INFO	4 (int representation)
filePath	String	Identifies the target file on the server.	Variable (length of path)
bytesToReadOrDelete	long	Number of bytes to read/delete. For operations: READ, WRITE_DELETE	8
offset	long	Position to start read/write/delete. For operations: READ, WRITE_INSERT,	8

		WRITE_DELETE	
data	byte[]	Data to be written. For operation: WRITE_INSERT	Variable (length of data)
monitorDuration	long	Duration for monitoring updates (in milliseconds). For operation: MONITOR	8

Table 1: *Request structure*

3.1.2 Response Design

The Response class is structured such that it returns the results of processing the client's requests. Table 2 clearly depicts the fields and its intended usage.

Attribute Name	**Data Type**	**Justification**	**Byte size**
statusCode	StatusCode Enum	To indicate success or failure for each operation. E.g., READ_SUCCESS, READ_ERROR, etc.	4 (int representation)
data	byte[]	Data returned from the server, if any.	Variable (length of data)
message	String	Informational message about the operation, such as indicating the file length in bytes after each read/write/delete operation.	Variable (length of message)
lastModifiedTimeAtServer	long	Timestamp for client-side cache validation.	8

Table 2: *Response structure*

3.1.3 Marshalling and Unmarshalling Process

Marshalling: Performed by Marshaller.java
Converts Request and Response objects into a byte stream that can be sent over the network (Datagram packets sent between ClientUDP and ServerUDP). Used by client to marshal Request objects, and server to marshal Response objects.

Unmarshalling: Performed by Unmarshaller.java
Reconstructs the original respective Request and Response objects from the byte stream upon reaching the destination (client or server). Used by client to unmarshal Response objects, and server to unmarshal Request objects.

3.2 Client Design

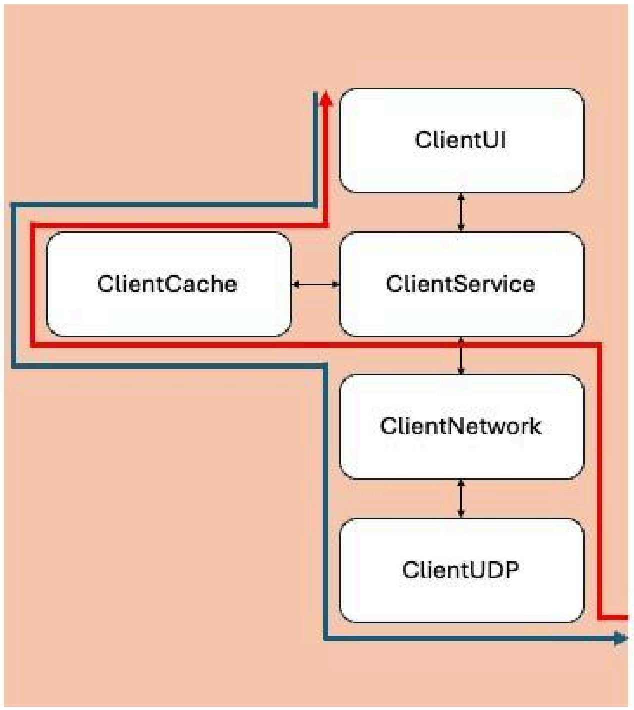

Figure 2: *Client Architecture & Detailed Communication Flow*

ClientMain (not shown in Figure 2)
The main entry point for the client application. It initialises the ClientUI and prompts the user for configurations that creates the necessary client components and starts the client.

Request Sending Phase (Top-down flow): Blue-arrow
ClientUI
Serves as the UI where users interact with the system to initiate several operations.

ClientService
Creates Request objects based on the operation the user initiated, encapsulating operation details. For *READ operation: Interacts with *ClientCache*.

ClientCache *(to be discussed in detail in 3.2.1)*
Check whether the file is cached and valid.

ClientNetwork
Acts as an intermediary between ClientService and ClientUDP. Obtains the Request object from ClientService and marshals it with the help of Marshaller.java. Uses ClientUDP to send the byte array to ServerUDP.

ClientUDP
Responsible for low-level transmission of marshalled Request byte array to the server using the UDP protocol.

Response Processing Phase (Bottom-up flow): Red-arrow
ClientUDP
Waits and receives the marshalled Response byte array from ServerUDP, returning it to ClientNetwork.

ClientNetwork
Unmarshals the received byte array into Response object and performs error-checking, returning the Response object to ClientService.

ClientService
Processes the received Response object by updating the ClientUI with String to be subsequently displayed. Interacts with ClientCache only for *READ operation.

ClientCache
Caches the new file data.

ClientUI
Displays the results of the user initiated operation to the user.

**Note: The 5 operations are READ, WRITE_INSERT, MONITOR, WRITE_DELETE, FILE_INFO.*

3.2.1 Client Cache Design

The ClientCache is designed to optimise only the READ operation by caching the file contents after the initial read, drastically improving response times. Subsequent read requests would be served from the ClientCache.

Cache Validity Checks (CVC)

Freshness Interval Check

Each cache entry has its lastValidationTime field, which is used to calculate its validity:
currentTime - lastValidationTime < freshnessInterval

Range Check *(explained in Partial Caching below)*, is valid if:
(offset >= entry.offset) && [(offset + bytesToRead) <= entry.offset + entry.bytesToRead]

Design Considerations

Concurrency

The ConcurrentHashMap that we are using allows multiple client threads to access the cache concurrently without blocking.

Partial Caching

If the requested read range falls within the range of the cached data, the cache entry returns only the relevant part of the requested read range, instead of the entire cache entry data. If it fails the Range Check in CVC, it will make a READ request to the server.

Update Validation Time of Cache Entry

If the cached entry failed the Freshness Interval Check in CVC but passed the Range Check in CVC, we check the lastModifiedTime of the file in the server, if it is the same as that of the cached entry, we update the lastValidationTime of the cached entry to the current time. Else, if the lastModifiedTime differs, we make a new READ request to the server.

Assumptions/Limitations

1) Assumption is made that the cache will only have 1 cache entry for each file at a time. Subsequent reads of the same file which failed the CVC will overwrite the previous cache entry for that file, ensuring that the cache always contains the most updated content based on the most recent READ.
2) Assumption is made that the user doesn't READ the file beyond its EOF. While READ operations will still be successful, caching is not done because it is not a valid range (see Range Check in CVC).

3.3 Server Design

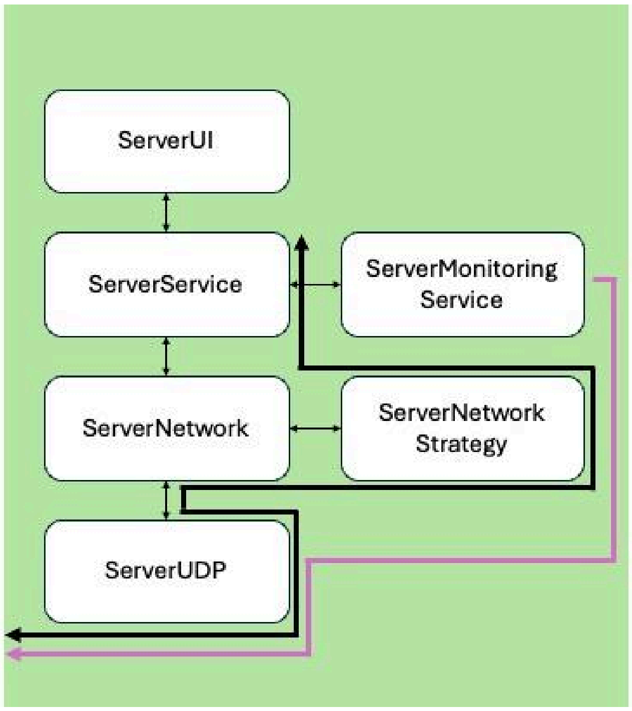

Figure 3: *Server Architecture & Detailed Communication Flow*

<u>ServerMain (not shown in Figure 3)</u>
The main entry point for the server application. It initialises the ServerUI and prompts the user for configurations that creates the necessary server components and starts the server.

Request Processing Phase (Bottom-up flow): Black-arrow (bi-directional)
<u>ServerUDP</u>
Waits and receives the marshalled Request byte array from ClientUDP, returning it to ServerNetwork while expecting a Response object in return. It uses probabilities to simulate network reliability conditions.

<u>ServerNetwork</u>
Unmarshals the received byte array into Request object, delegating the handling of Request to ServerNetworkStrategy by passing the Request object to it.

<u>ServerNetworkStrategy</u> *(to be discussed in detail in 3.3.1)*
Determines how the Request is processed based on the invocation semantics. Calls ServerService to process the Request.

ServerService
The Request is processed and handled here, performing actual file operations such as READ, WRITE_INSERT, WRITE_DELETE. WRITE_MONITOR operation is delegated to ServerMonitoringService. After which, the Response object is constructed.

Response Sending Phase (Bottom-up flow): Black-arrow (bi-directional)
ServerMonitoringService (Independent of the rest of the layers): ***Purple-arrow***
If a Request to monitor the file is received, directly calls ServerUDP to send a callback.

ServerService
Else, returns the constructed Response object to ServerNetworkStrategy.

ServerNetworkStrategy
Returns the newly obtained Response object to ServerNetwork.

ServerNetwork
Marshals the Response object into a byte array and returns it to ServerUDP. Uses ServerUDP to send the byte array to ClientUDP.

ServerUDP
Sends the marshalled Response byte array back to the client using the UDP protocol.

3.3.1 ServerNetwork Strategy Design (Invocation Semantics) & Fault Tolerance

In our server-side architecture, 2 distinct network strategies are employed to handle the At-Most-Once and At-Least-Once invocation semantics of Requests.

Fault Tolerance
Regardless of the type of invocation semantics, the client has a consistent behaviour, where in the event of network failures (and it doesn't receive a response for its request), it will always retransmit the request. Therefore, the same Request object identified by its **requestId** will be continuously sent to the server. The way the server handles these duplicate Request objects it receives depends on the type of invocation semantics. In our project, we have allowed the users to input the maximum number of retries, **maxRetries** (clientUI.selectMaxRetries()), for the client to send the same Request object to the server in the event that no datagram packet (or byte array) is received from the server (marshalled Response objects), before returning an error message. Furthermore, the interval between each retry is determined by the **timeou**t set on the datagram socket (clientUI.selectTimeout()) by the user. Since we have identified that only the server's, not client's, behaviour will vary depending on the type of invocation

semantic, we have decided to introduce the **ability to simulate packet loss using probabilities** under adverse network conditions in our ClientUDP/ServerUDP classes.

Assumptions/Limitations

While we understand that in real-world, packet loss can come from either the client or server's side (or ClientUDP and ServerUDP in our project), we have decided to make only our ServerUDP susceptible to network failures for simplicity. ClientUDP has the probabilities 'requestSendProbability' and 'replyReceiveProbability' each fixed to '1' upon client start-up, indicating 100% possibility of successfully sending a Request object to the server/receiving a Response object from the server respectively. Meanwhile, ServerUDP has its probabilities 'requestReceiveProbability' and 'replySendProbability' vary depending on user input, indicating the probability of successfully receiving a Request object from the client/sending a Response object to the client. This is useful for us to simulate poor network conditions for the experiments and results section below. Since the client only retries sending the same Request object when no Response object is received, varying just the ServerUDP's 'requestReceiveProbability' and 'replySendProbability' is sufficient to determine whether the client side receives a Response object or not.

At-Most-Once invocation semantic

AtMostOnceStrategy.java implements the ServerNetworkStrategy.java interface, where it ensures that each duplicate Request object received from the client doesn't cause ServerService to execute its methods more than once, regardless of network failure or not. Duplicate Request objects are filtered out and the server will not re-execute the operation for a duplicate Request. Instead, the server returns the cached Response, which is only created once for each unique Request.
Each Request, identified by their requestId, is cached using a ConcurrentHashMap<requestId, Response>. Upon receiving a Request object, the map checks whether it contains the key of the Request's requestId. If so, return the cached Response (value), else, invoke the ServerService to obtain the Response object from it, before finally storing the <requestId, Response> pair in the map. Hence, this is friendly to non-idempotent operations.

At-Least-Once invocation semantic

AtLeastOnceStrategy.java implements the ServerNetworkStrategy.java interface, where it ensures that each Request object obtained from the client is executed at least once. In the event of network failures, It may cause multiple executions of the same ServerService methods due to the same Request object being received from the client, resulting in multiple Response objects being created. This is problematic for non-idempotent operations.

4. Experiments and Results

We have fixed the following ***relevant*** variables/user prompts in CLI across the 2 experiments to simulate adverse network conditions and packet loss from server to client.
Client-side CLI:
maxRetries = 3
timeout = 5000 ms
freshnessInterval = 30,000 ms
ClientUDP's 'requestSendProbability' = 1 (Fixed, stated in 'Assumptions/Limitations')
ClientUDP's 'replyReceiveProbability' = 1 (Fixed, stated in 'Assumptions/Limitations')
Server-side CLI:
ServerUDP's 'requestReceiveProbability' = 1 (User input to simulate poor network)
ServerUDP's 'replySendProbability' = 0 (User input to simulate poor network)
*Note, for these experiments, we will focus on the initial vs resultant file state.

Experiment 1: At-Most-Once invocation semantic
(a) Non-idempotent operation
Type of operation performed: WRITE_INSERT
Operation parameters entered: Enter offset=0, Enter data to write=(start)_
Initial file state: Hello from Alex's desktop!
Resultant file state: (start)_Hello from Alex's desktop!
– File content is not modified externally between (a) and (b) –
(b) Idempotent operation
Type of operation performed: READ
Operation parameters entered: Enter offset=0, Enter number of bytes to read = 10
Initial file state: (start)_Hello from Alex's desktop
Resultant file state: (start)_Hello from Alex's desktop
– File content is not modified externally between Experiment 1 & Experiment 2 –
Experiment 2: At-Least-Once invocation semantic
Non-idempotent operation
Type of operation performed: WRITE_INSERT
Operation parameters entered: Enter offset=0, Enter data to write=(start2)_
Initial file state: (start)_Hello from Alex's desktop
Resultant file state: (start2)_(start2)_(start2)_(start)_Hello from Alex's desktop

Therefore, we conclude that At-Least-Once invocation will result in unintended file changes under poor network conditions for non-idempotent operation. However, At-Most-Once invocation will not result in unintended file changes even under poor network conditions.

5. Appendix

Instructions for running the code.

In Client's Command Line Interface:

- Locate the /src directory of the SC4051-RemoteFileAccessSystem project.
- Compile the client with the relevant packages:
  ```
  javac client/*.java common/*.java marshalling/*.java -d ../out
  ```
- Run the client:
  ```
  java -cp ../out client.ClientMain
  ```

In Server's Command Line Interface:

- Locate the /src directory of the SC4051-RemoteFileAccessSystem project.
- Compile the server with the relevant packages:
  ```
  javac server/*.java common/*.java marshalling/*.java strategy/*.java -d ../out
  ```
- Run the server:
  ```
  java -cp ../out server.ServerMain
  ```