CSE – 6324: ADV TOPICS SOFTWARE ENGINEERING

Project title: Cloud Based Storage

INSTRUCTOR
Jeff Lei

TEAM - 10

Team members:

Pinnimti Sri Harish	1001865949
Nama Sai Krishna Prateek	1001880903
Mukka Himaneesh	1001861524
Venkataraman Mani Kandan	1001960028

TABLE OF CONTENTS

1. Overview of the Project

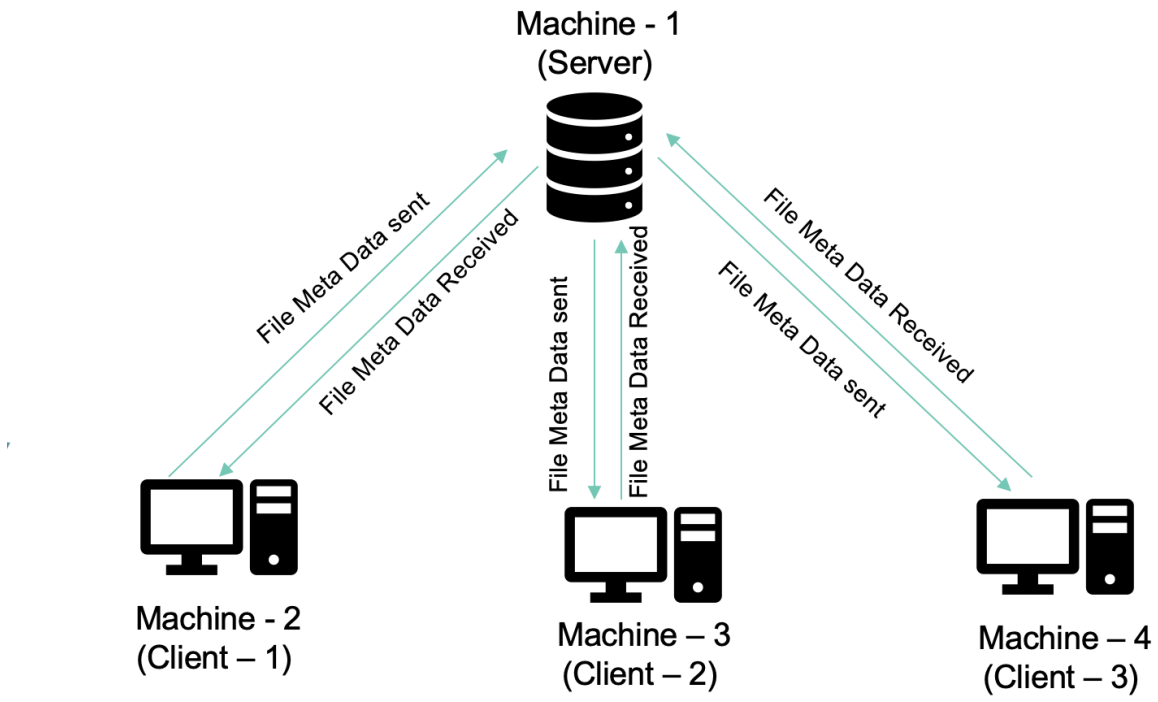
1.1 Introduction:

Our team's objective is to create a client-server application (cloud storage) which helps users to synchronize and manage their files to a server. Users can create, modify, and delete files. In our project we will use multi-threading for concurrency behavior, and UDP file transfer approach for directories synchronization.

1.2 Architecture:

1. **Client**: The client will be using a terminal to connect (Reliable UDP) with the server and be able to perform the operations below.
   - Connect and disconnect from the server.
   - Create, delete, and update operations.

2. **Server**: The server will always be available to users; it also allows users to access files from any device. It receives requests from clients and broadcasts all the changes done by a client on a specific device to all devices to maintain synchronization. The server is capable of handling multi-threading and for every device, a corresponding thread is created.



2. Implementation & Error Handling

2.1 Reliable – UDP:

We wanted to make use of the **FULL DUPLEX** capability of UDP to achieve maximum data throughput i.e., a single UDP socket connection can be simultaneously used for sending and receiving data.

In sequential programming, sending, and receiving are blocking operations. So, to make use of full duplex, we are using 2 threads to avoid blocking:

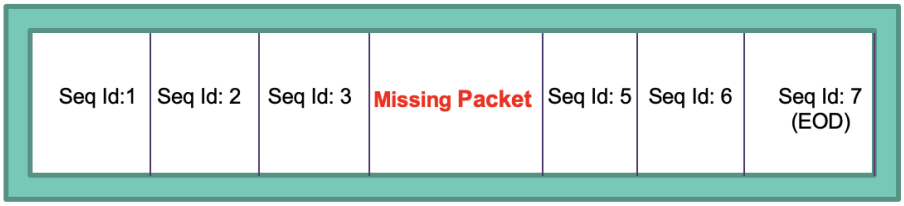
1. Sending the data while receiving.
2. Receiving the data while sending.

We have used asynchronous callback paradigm to achieve **maximum concurrency** while writing **sequential programs** and it makes handling of local data in multithreaded programs much easier. Especially in Client-Server architecture where each client has its own state on server.

We have improved UDP implementation by addressing the issues like Out of order, packet drop, and duplicate packet. Making it more reliable protocol for transferring data.

We have also implemented the **Producer/Consumer** solution to handle the writing & reading operations.

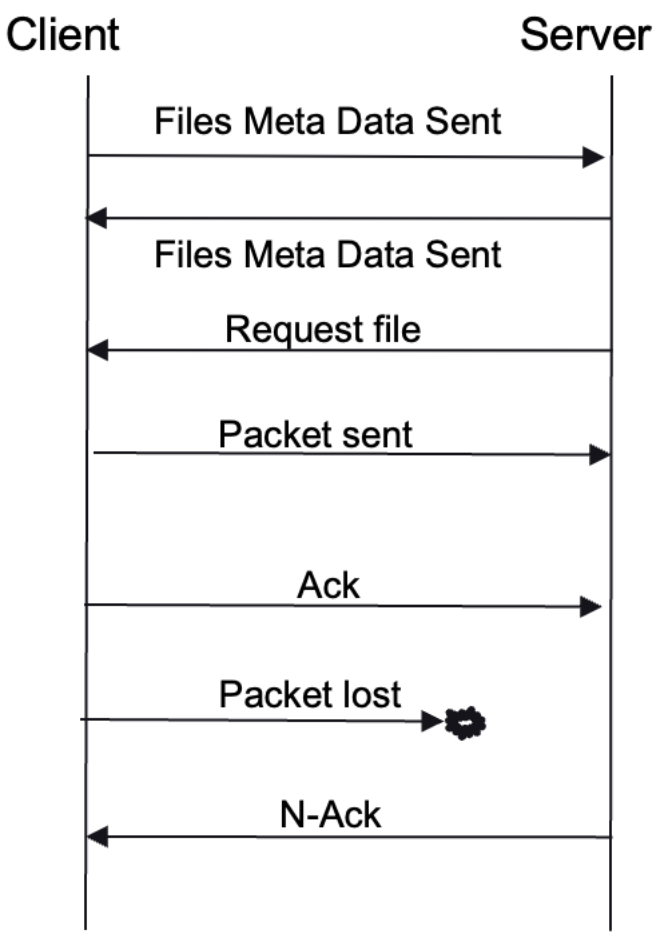
The sender assigns a **SEQUENCE ID** to each packet sequentially before sending it out, the receiver only puts into the data queue if we have all the packets.



Receiver Buffer

2.2 Data flow:

- When client and server connect for the first time. They exchange “Files Meta Data”.
- When a change is detected at client side. It pushes the latest version of the File to the server.
- Server then compares the File Meta Data and sends to all other clients connected.
- When a packet is lost, the server will wait for the packet and after a certain amount of time it will send a Negative acknowledgment to the sender.



2.3 Propagating changes:

1. We have opted for the "**Push**" model instead of the "**Poll**" – this saves the bandwidth and CPU resources
2. When a new change is detected in the **Filesystem**, the **Sender** sends a **METADATA** of the changed files to the **Receiver**. It is the responsibility of the receiver to compare the **METADATA** and request new version files.
3. We don't differentiate between **Server** and **Client**. The codebase and logic are the same. Hence, we use **SENDER** and **RECEIVER**.
4. Whenever a connection happens between **SERVER** and **CLIENT**, concurrently they **share their current METADATA**. They each process it and only request for **OUTDATED** files.

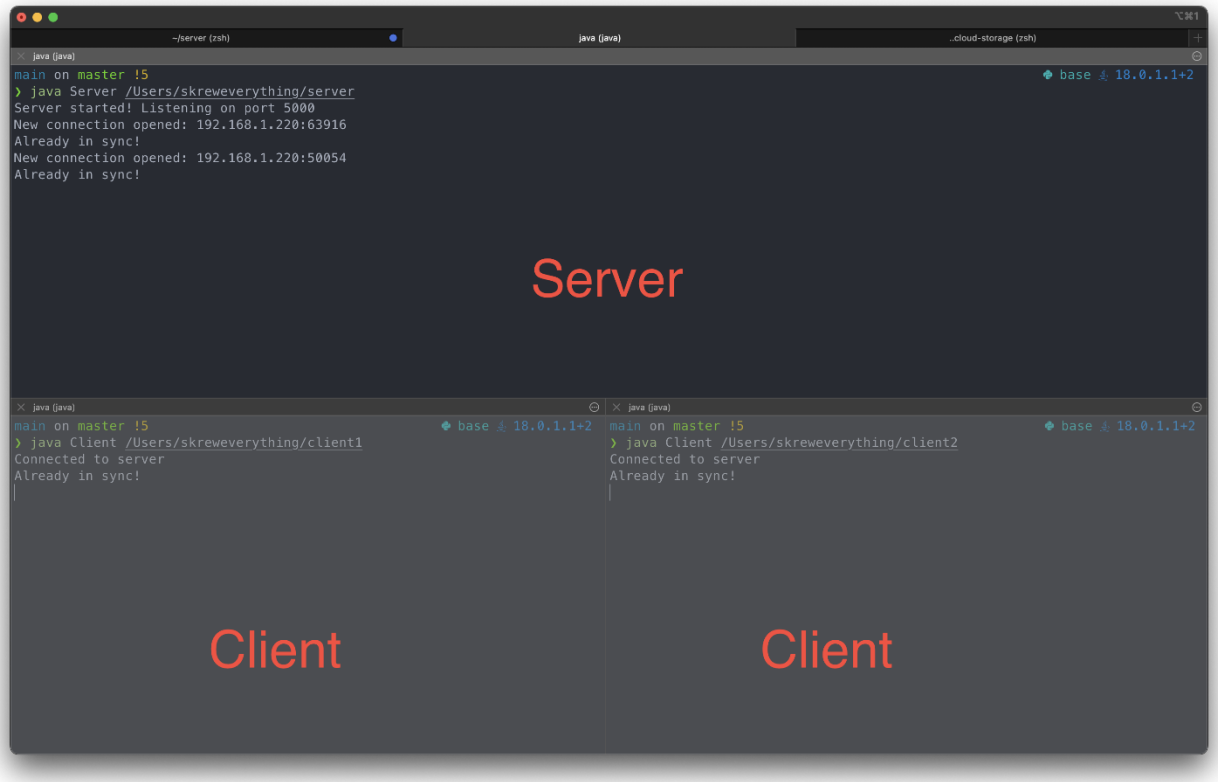
5. Instead of making **SENDER Intelligent**, we are **DELEGATING** the responsibility to the **RECEIVER**. – this helps us economically as we don't need a powerful **SERVER**. The work is being forwarded to the **CLIENTS**.
6. This strategy has helped us to handle **NETWORK ERROR** when disconnection happens during the syncing.

2.4 Thread Management:

1. **Detection Thread:** Whenever a change is done in Filesystem, this change is detected by the Filesystem thread.
2. **Client Thread:** For every new incoming client connection. A new thread is created for the respective client to achieve the maximum concurrency.
3. **Receiver Thread:** Like IN UDP, receiving is a blocking operation. So, to avoid that we are creating a centralized dedicated thread to receive all the data.
4. **Cleaner Thread:** The purpose of this thread is to perform clean up by the terminating threads after the client connection is closed. This applies to unexpected client disconnection. This helps us to avoid **memory leaks**.

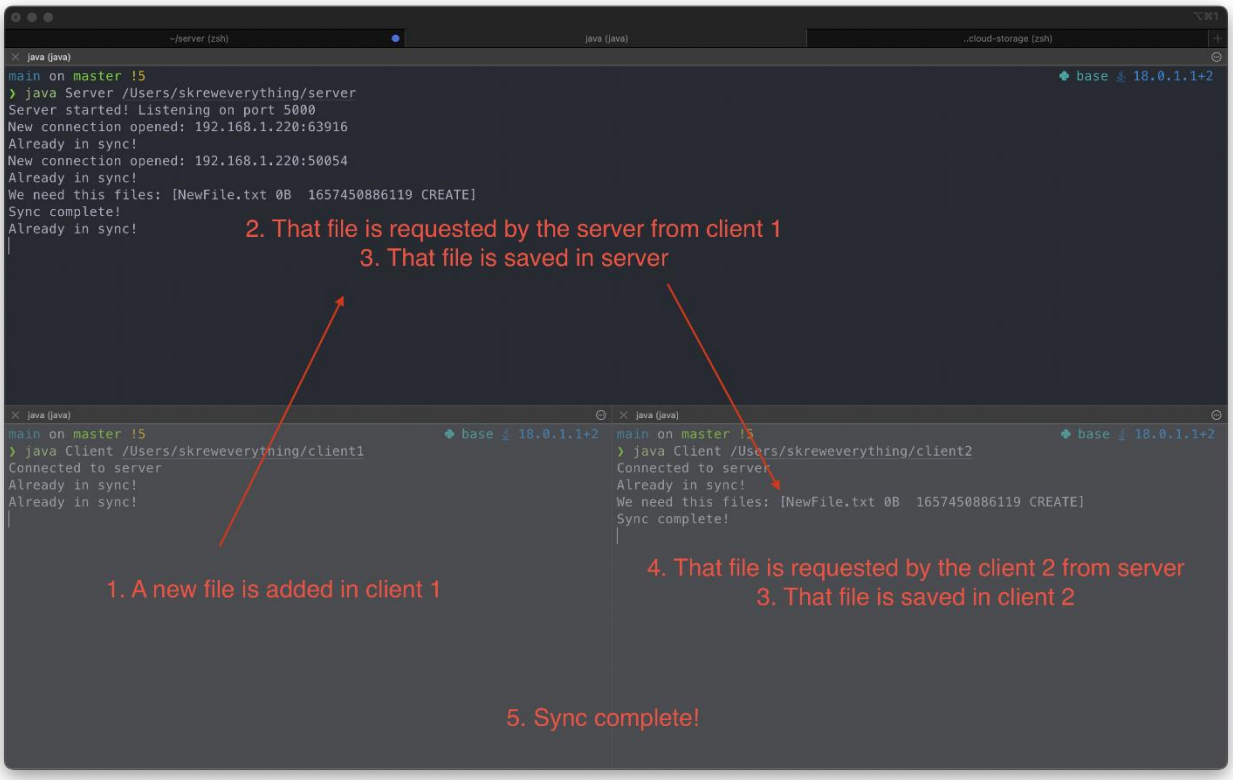
3. USE Cases

3.1 **First Connection:**



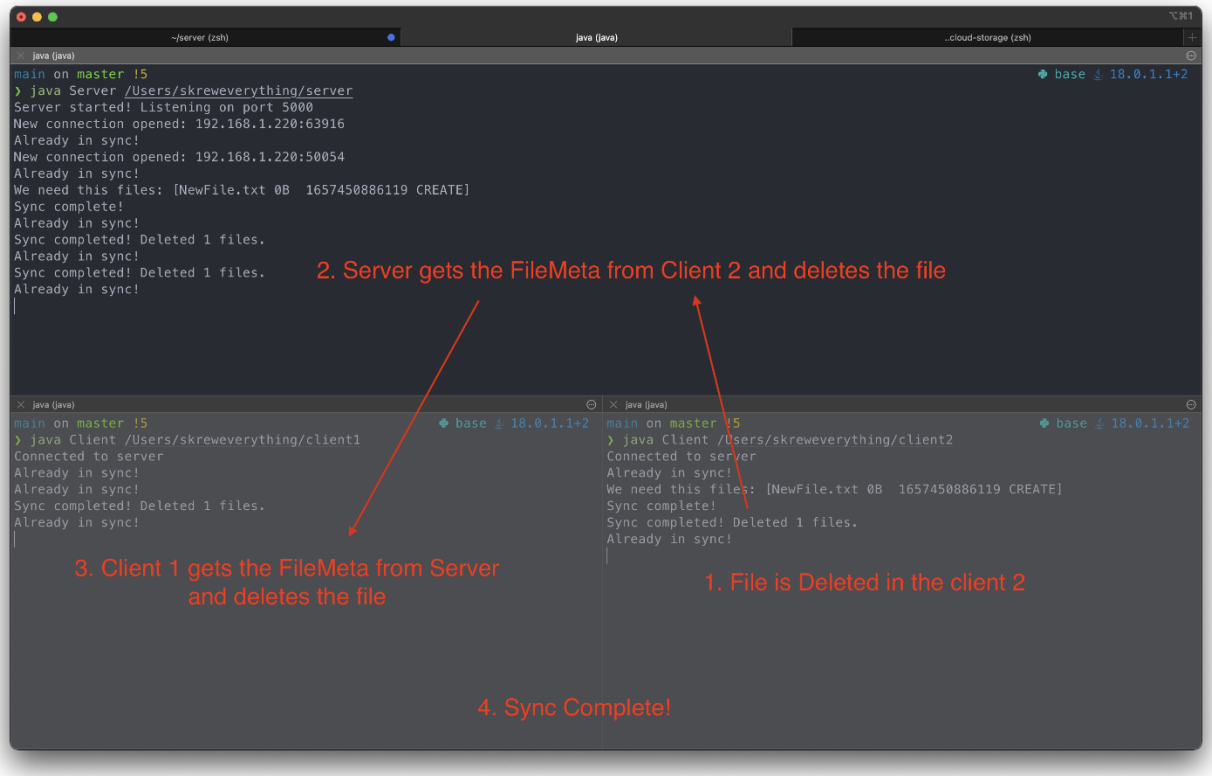
When the client and server first connect, they synchronize i.e., send each other their file meta data. If they are already in sync, then nothing is done. If not, they request the required files to be in sync from other machines.

3.2 **Add Operation:**



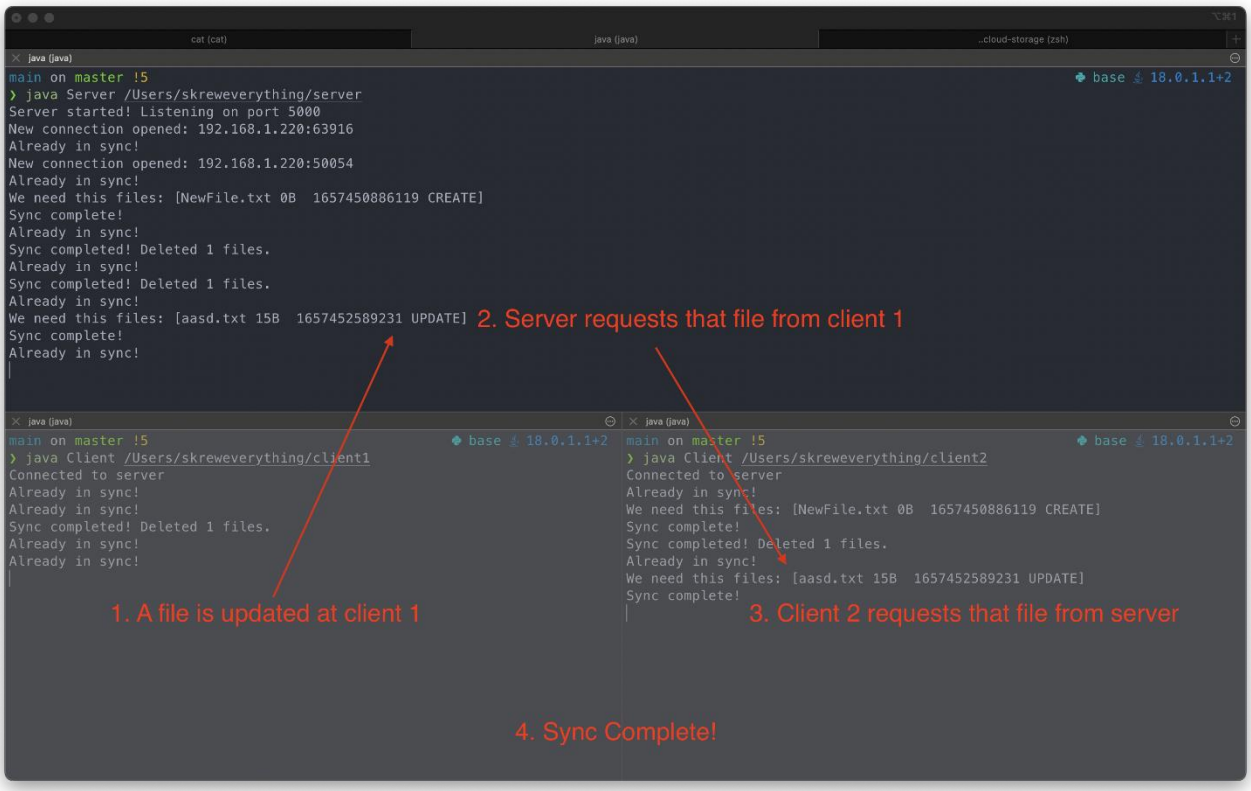
When the service detects a file has been added in the client(sender) system. It will send the updated file meta data to the server(receiver). Server(receiver) being intelligent would request the specific file its missing and server(sender) will broadcast that to all other clients(receivers).

3.3 **Delete Operation:**



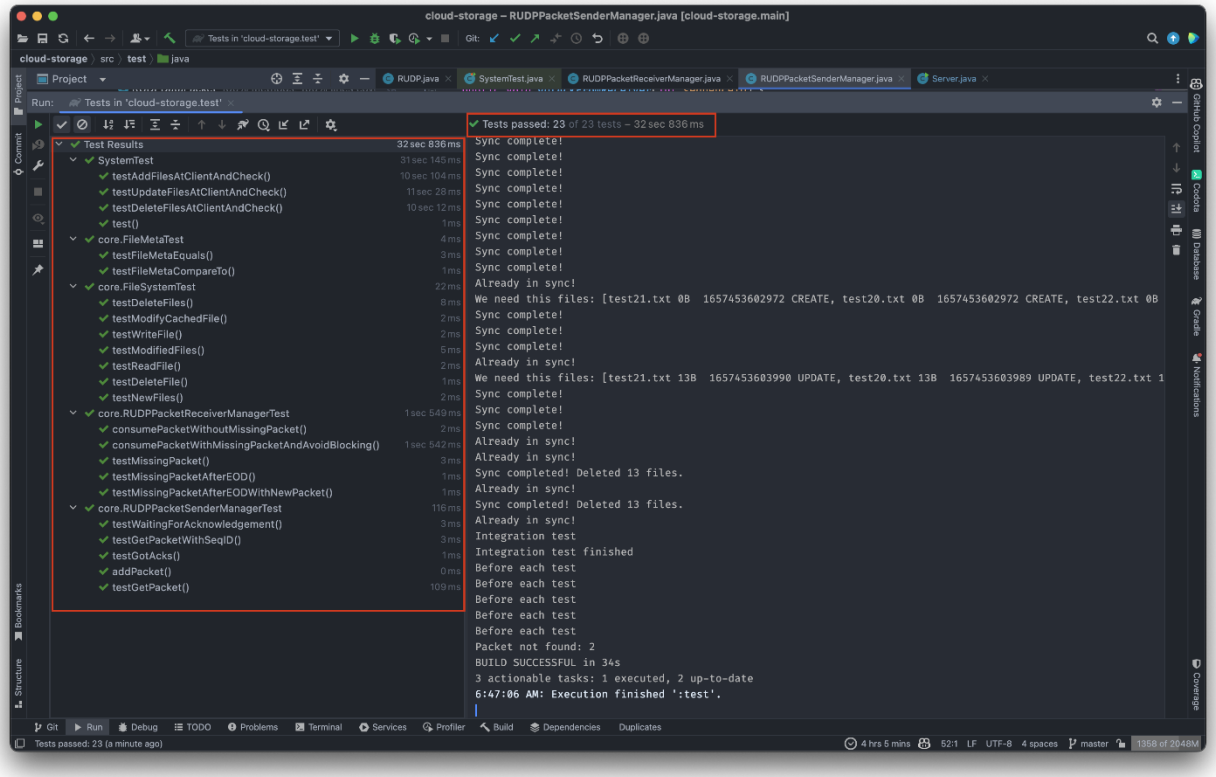
When the service detects a file has been deleted in the client(sender) system. It will send the deleted file meta data to the server(receiver). Server(receiver) deletes the specific file and server(sender) will broadcast that to all other clients(receivers).

3.4 **Update Operation:**

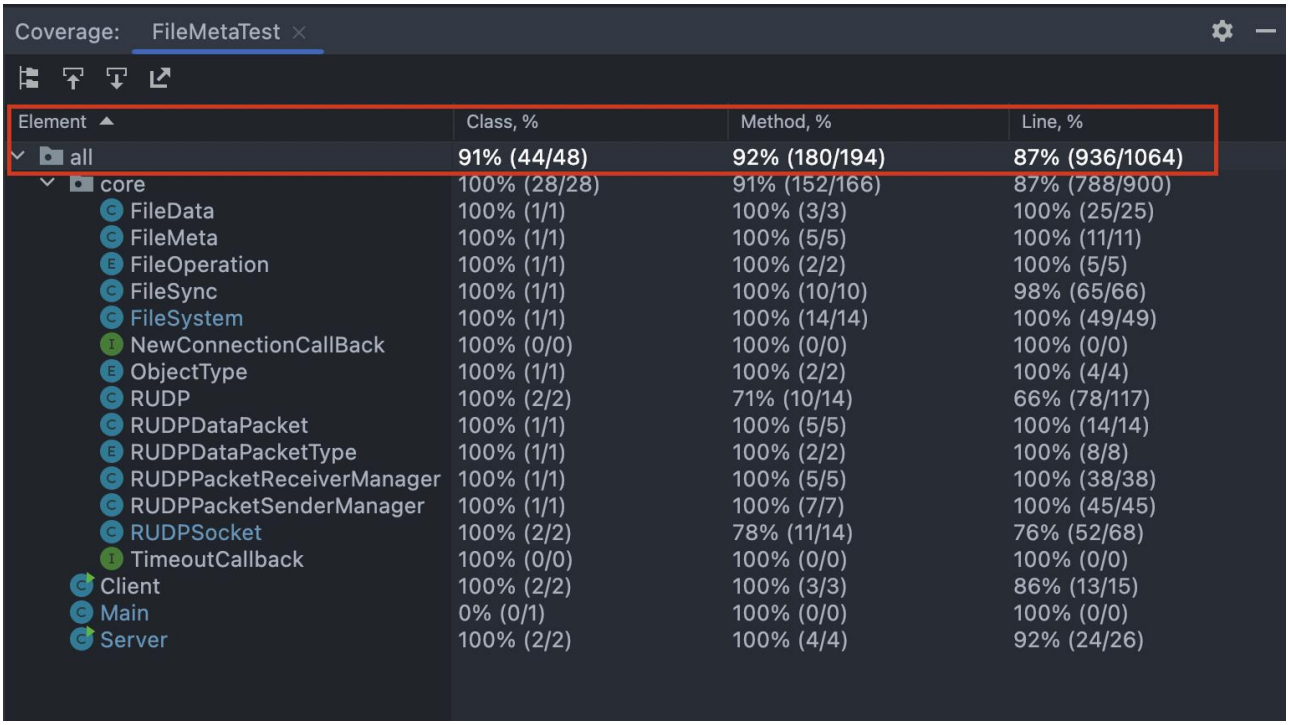


When the service detects a file has been updated in the client(sender) system. It will send the updated file meta data to the server(receiver). Server(receiver) being intelligent would request the specific file if it's the latest version and server(sender) will broadcast that to all other clients(receivers).

4.Testing



We have written a total of 23 core tests that tests its functionalities and all use cases that are mentioned above.



Element	Class, %	Method, %	Line, %
all	91% (44/48)	92% (180/194)	87% (936/1064)
core	100% (28/28)	91% (152/166)	87% (788/900)
FileData	100% (1/1)	100% (3/3)	100% (25/25)
FileMeta	100% (1/1)	100% (5/5)	100% (11/11)
FileOperation	100% (1/1)	100% (2/2)	100% (5/5)
FileSync	100% (1/1)	100% (10/10)	98% (65/66)
FileSystem	100% (1/1)	100% (14/14)	100% (49/49)
NewConnectionCallBack	100% (0/0)	100% (0/0)	100% (0/0)
ObjectType	100% (1/1)	100% (2/2)	100% (4/4)
RUDP	100% (2/2)	71% (10/14)	66% (78/117)
RUDPDataPacket	100% (1/1)	100% (5/5)	100% (14/14)
RUDPDataPacketType	100% (1/1)	100% (2/2)	100% (8/8)
RUDPPacketReceiverManager	100% (1/1)	100% (5/5)	100% (38/38)
RUDPPacketSenderManager	100% (1/1)	100% (7/7)	100% (45/45)
RUDPSocket	100% (2/2)	78% (11/14)	76% (52/68)
TimeoutCallback	100% (0/0)	100% (0/0)	100% (0/0)
Client	100% (2/2)	100% (3/3)	86% (13/15)
Main	0% (0/1)	100% (0/0)	100% (0/0)
Server	100% (2/2)	100% (4/4)	92% (24/26)

We have achieved a classes coverage of **91%,** method testing of **92%** and line coverage of **87%.**

5. Challenges faced

1. **Critical Section Problem:** Initially, we have started with the brute force way of making methods as synchronized to solve critical section issue.

2. **Consumer-Producer Problem:** As UDP Socket's "**receive**" operation is a blocking operation, we encountered a deadlock situation if the client is disconnected. This is where we have thought of asynchronous.

3. **Performance Issues:** Using synchronized methods will block entire access of that object which obviously decreases the performance, so we have instead opted for synchronized blocks to obtain lock only when needed.

6.Member Contributions

Pinnimti Sri Harish: Implemented RUDP and its related test cases.
Nama Sai Krishna Prateek: File system services and its related test cases.
Mukka Himaneesh: File syncing and project overview.
Venkataraman Mani Kandan: Architecture and system testing.

7. Project Schedule

S.No	Dates	Schedule
1	25th – 27th June 2022	Creation of client-side program
2	28th – 1st July 2022	Creation of server-side program
3	2nd July 2022	Integration
4	3rd July 2022	Testing

8. Collaboration Plan

Version control – We are planning to use GitHub.
Planning and discussions – We use Teams for any impromptu meetings and doing in-person meetings thrice a week and planning to spend total 8 hours/week together.
Documentation and status report - Google Docs.

9. Lessons Learned

In the beginning, we struggled to implement the project in JAVA programming as most of us are not familiar in this language but as a team in the end we were able to complete our project by helping each other. Our most challenging part was thread management and error handling as we came across synchronization and deadlocks issues, with the help of professor's lectures, we were able to overcome these issues. We have implemented most of the core functionalities as instructed, given the time we would have enhanced the project with more user-friendly interface.