Design and implementation of a file transfer system based on the TFTP specification (RFC 1350). The system consists of TFTP client(s) running on one or several computers, an error simulator, and a multithreaded TFTP server that could run on a different computer.

SYSC 3303 Project Report

TFTP specification Project

Team ID: Group 9

Table of Contents

TEAM STRUCTURE:

Team number: Group 9

Team Members:

Toyin Odujebe ---------- 100855492

Haris Ghauri ---------- 100910613

Sohaib Hefzi ---------- 100867452

Nasir Ali ---------- 100907683

ITERATIONS BREAKDOWN:

Iteration 1:

All team members met and planned the initial design of the whole system. Specifically how the Server side (Server, ConnectionListener, ConnectionManager) will be connected with each other

Specific Contributions:

Haris Ghauri: Designed the foundation classes Client, Error Simulator Server, ConnectionListener and ConnectionManager. Also implemented the methods that make the steady state transfer happen both on the client and server side. Did the README.txt

Toyin Odujebe: worked on Packet.java, added additional methods to verify the kind of requests and data blocks. Also worked on the drawings of UML Class diagrams and Use Case map. Helped Haris in README.txt. Worked on the IO, userInputs from the client, the error simulator and the Server.

Sohaib Hefzi: tested the whole program thoroughly. Added the functionality of dealing with special cases such as the transfer of empty file and the size of 512 bytes. Helped Toyin in the diagrams.

Nasir Ali: worked on byteArray, made sure the conversion of byteArray to ints give the right blockNumber and viceVersa. Helped Toyin in the diagrams.

Iteration 2:

Specific Contributions:

Haris Ghauri: Redesigned error simulator and made it multi-threaded to listen on two opposite ports and send vice versa. Came up with the private messenger class to produce different kinds of network errors.

Toyin Odujebe: Came up with different scenarios for the timing diagrams. What can go wrong? Made decisions on how to deal with the timeouts and how frequent to retransmit. Fixed previous use case diagrams. Worked on Client and Server Side. Implemented the User Input of Error Simulator.

Sohaib Hefzi: tested the whole program thoroughly. Tested different error scenarios, such as delayed, duplicated, and lost packets. Identified and fixed bugs. Made sure that the packets were actually getting delayed, lost or duplicated. Wrote the readme file.

Nasir Ali: Helped Toyin in the new design of the Client and the Server class. Dealt with the timeouts. Helped Haris in coming up with the errors for the messenger class.

Iteration 3:

Specific Contribution:

Haris Ghauri: Updated the Error simulator to include ERROR CODE 4 and 5 with different scenarios for ERROR CODE 4

Toyin Odujebe: Did timing diagram and came up with the different scenarios for ERROR CODE 4

Sohaib Hefzi: updated client and server to be able to handle ERROR CODE 4 and 5

Nasir Ali: Tested the whole system thoroughly and helped Haris with Error Simulator

Iteration 4:

Haris Ghauri: Updated the server including connection manager for error 1, 2, 3

Toyin Odujebe: Drew timing diagrams for the errors, worked on client to handle errors

Sohaib Hefzi: helped with updating server for error 6, and fixed bugs and updated packet class

Nasir Ali: Tested the whole system thoroughly, and helped with fixing bugs.

Iteration 5:

The whole team came together and modified the client and server to reside on different computers

SET UP INSTRUCTIONS (Normal Mode):

1. Open Eclipse and import the project.
2. Open 2 different consoles
3. Run Server (Server.java) first and pin to first console
4. Run Client (Client.java) last and pin to third console
5. You have the option to run in Quiet or Verbose mode in the 2 different consoles.
6. Choose your mode for each console
7. Next select 1 or 2 to run in Normal or test mode respectively in the **Client console**
8. Choose 1 for Normal mode, then the **Client console** then asks if the server is running on the same computer or not choose 1 or 2 respectively.
9. If you select 1 that the Client and Server are on the same computer then you can skip to Step Number 11
10. If you select 2 signifying that the Client and Server are not on the same computers then you have to provide the IP address of the computer that has the server into the **Client console**
11. Select 1 or 2 for read request (RRQ) or 2 for write request (WRQ) respectively in the **Client console**
12. Enter the filename you want to either read or write in the **Client console** and press enter

Important Note:

- Type quit in the **Client console** to end the clients execution at any time when client is taking inputs
- You can shut down the server at any time by typing quit into the **Server console**

SET UP INSTRUCTIONS (Test Mode):

1. Open Eclipse and import the project.
2. Open 3 different consoles
3. Run Server (Server.java) first and pin to first console
4. Run Error Simulator (ErrorSimulator.java which is inside the ErrorSimulator package)second and pin to second console
5. Run Client (Client.java) last and pin to third console
6. You have the option to run in Quiet or Verbose mode in the 3 different consoles.
7. Choose your mode for each console
8. You can quit the server or client at any time by typing quit into the **Server console**
9. Next select 1 or 2 to run in Normal or test mode respectively in the **Client console**
10. Choose 2 for Test mode
11. The **Client console** asks if you want to make a Read Request (RRQ) or Write Request (WRQ) select 1 or 2 respectively
12. The **Error Simulator console** asks if the server is running on the same computer as the client and error simulator. If you select 1 that the Client and Server are on the same computer then you can skip to Step Number 14
13. If you select 2 signifying that the Client and Server are not on the same computers then you have to provide the IP address of the computer that has the server into the **Error Simulator console**
14. On the **Error Simulator console**, you are asked if you want to delay, duplicate or lose packets, choose 1 for Yes and 2 for No
    a. If you choose 1 for Yes, then you should enter the total number of packets you want to manipulate into the **Error Simulator console**. This is not required for the Project but was added as an extra feature. Just put 1 to manipulate only one packet at a time.
    b. Enter the block number of the packet that you want to manipulate into the **Error Simulator console**
    c. Then select what kind of packet you want to manipulate (1 for ACK and 2 for DATA)
    d. Now select the kind of scenario you want to simulate (1 for DELAY, 2 for DUPLICATE and 3 for LOST)
15. Then a menu of errors is shown on the Error simulator that contains a bunch of errors that can be simulated. Select the number that corresponds to the error you want to simulate and 0 if you don't want to simulate any errors
16. Enter the filename you want to either read or write in the Client console and press enter

Important Note:

- Type quit in the Client console to end the clients execution at any time when client is taking inputs
- You can shut down the server at any time by typing quit into the server console

DIAGRAMS:

Iteration 1:

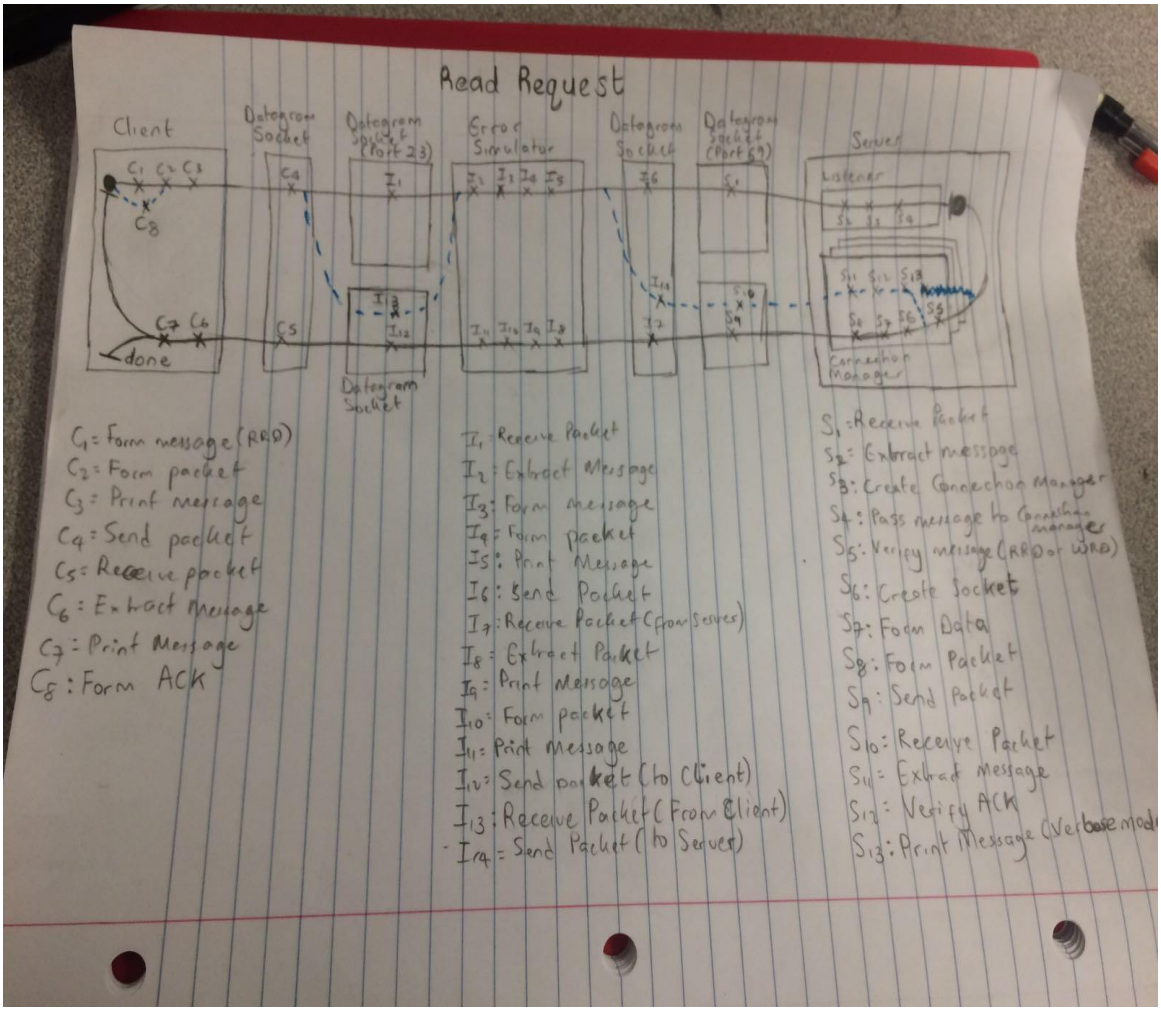

Figure 1. Read request (RRQ) use case map (UCM)

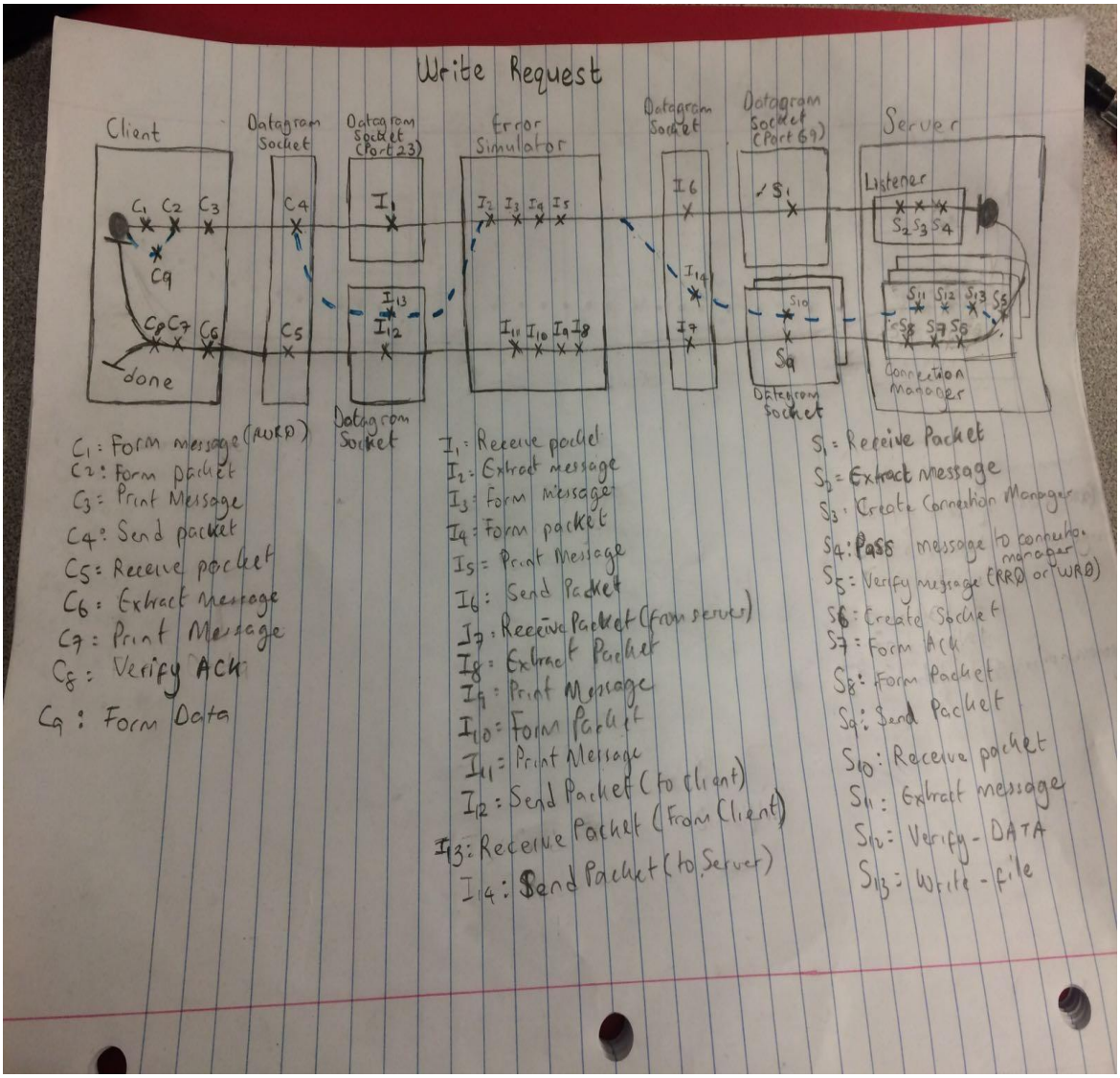

Figure 2. Write request (WRQ) Use case map (UCM)

NOTES:

- The Error simulator in Figure 1 and Figure 2 should not extract and create datagrams, it should only pass the packets as it is. This was fixed in the upcoming iterations.
- Also print message should not be part of Figure 1 and 2 and so it was removed in the upcoming iterations.
- The alternative flow in blue ink will be applied after the initial read or write request.

Iteration 2:

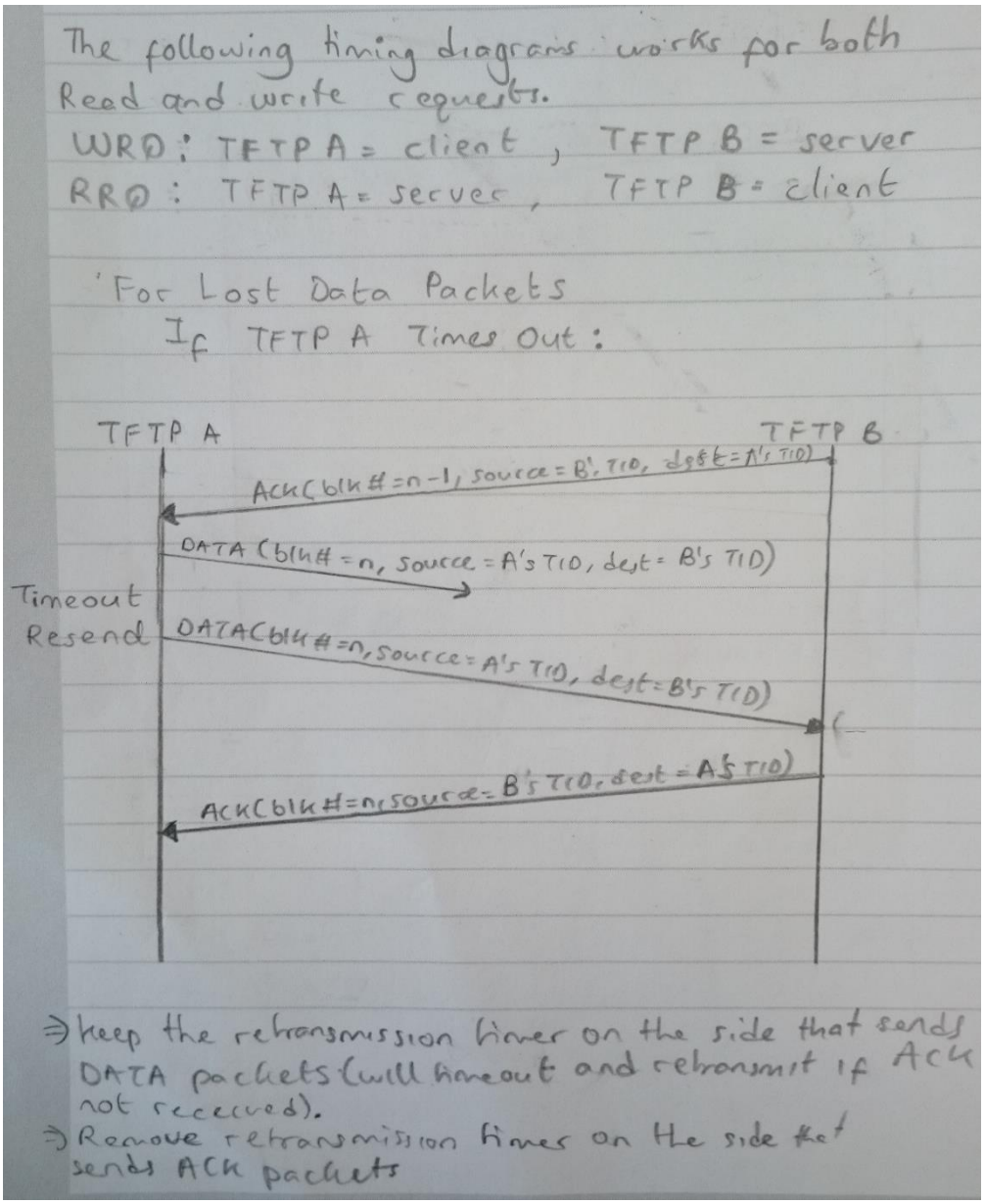

Figure 3. Lost DATA packets timing diagram

NOTES:

- Keep the retransmission timer on the side that sends DATA packets (will timeout and retransmit if ACK packet is not received)
- Remove retransmission timer on the side that sends ACK packets
- WRQ => TFTP A = Client TFTP B = Server
- RRQ => TFTP A = Server TFTP B = Client

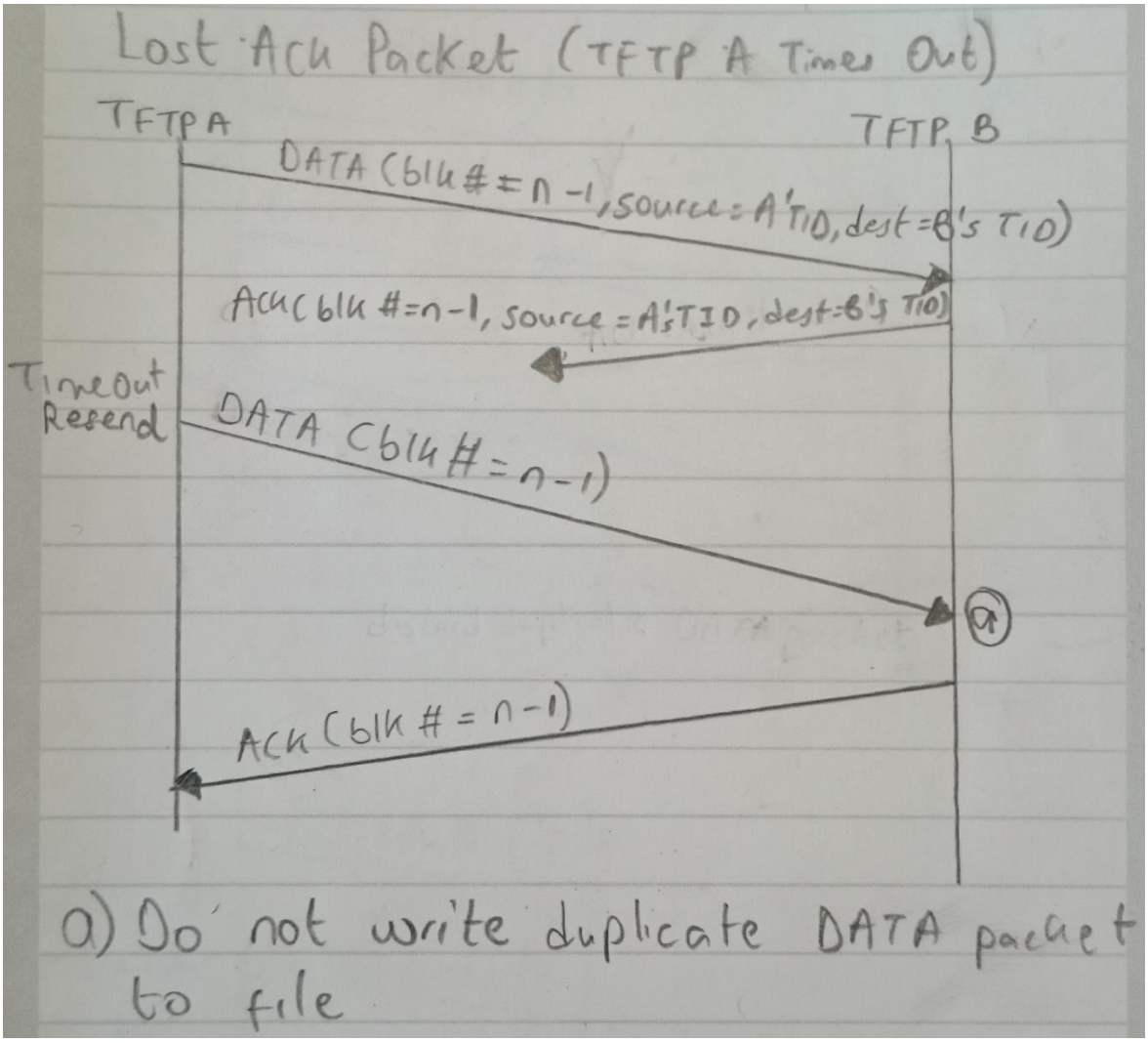

Figure 4. Lost ACK packet timing diagram

NOTES:

a) Do not write duplicate DATA packet to file

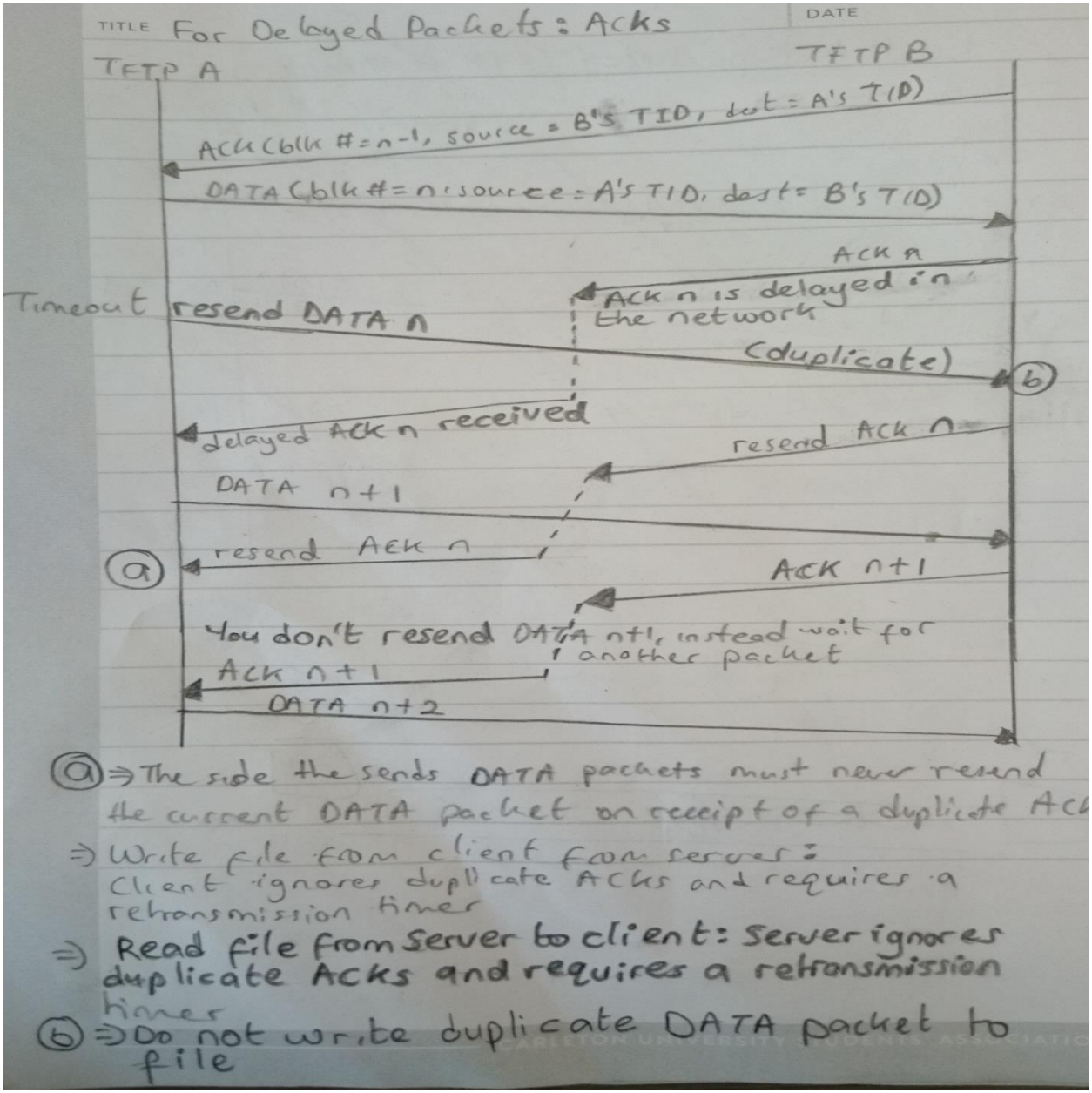

Figure 5. Delayed ACK packets timing diagram

NOTES:

a) The side that sends DATA packets must never resend the current DATA packet on receipt of a duplicate ACK packet
b) Do not write duplicate DATA packet to file

Write file from client to server: client ignores duplicate ACKS and requires a retransmission timer

Read file from server to client: server ignores duplicate ACKS and requires a retransmission timer

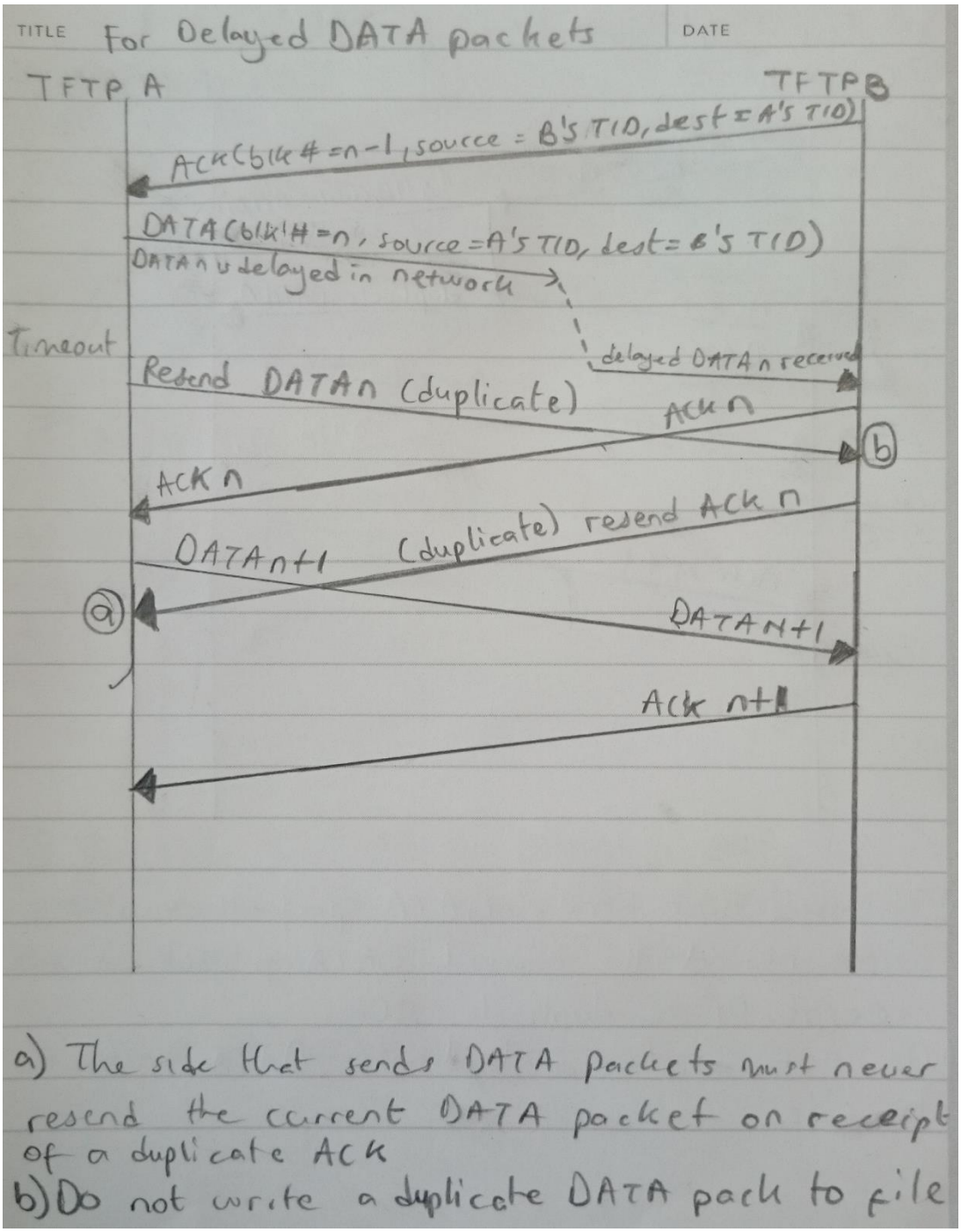

Figure 6. Delayed DATA packets timing diagram

NOTES:

a) The side that sends DATA packets must never resend the current DATA packet on receipt of a duplicate ACK packet
b) Do not write a duplicate DATA packet to file

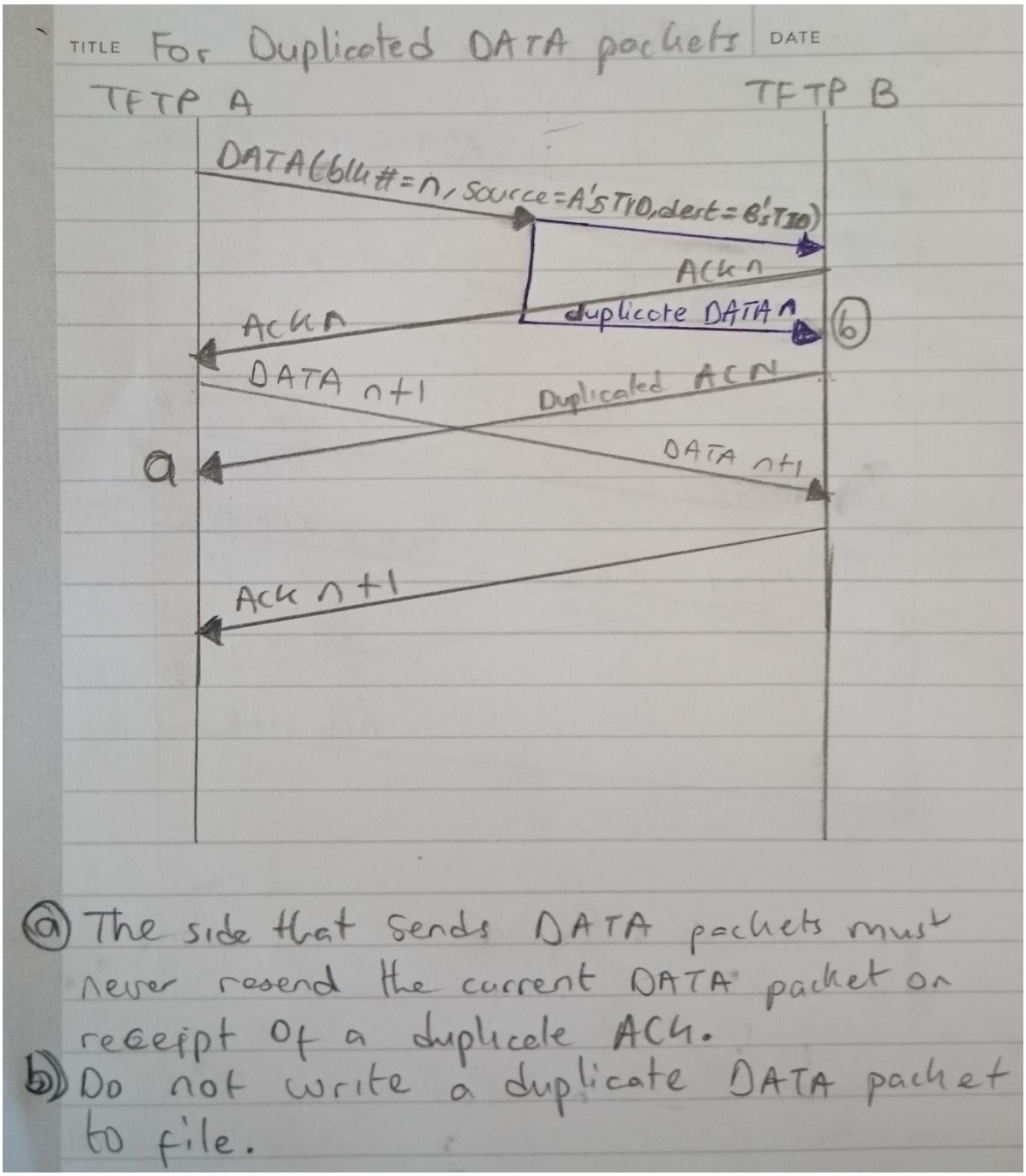

Figure 7. Duplicated DATA packets timing diagram

NOTES:

a) The side that sends DATA packets must never resend the current DATA packet on receipt of a duplicate ACK
b) Do not write a duplicate DATA packet to file

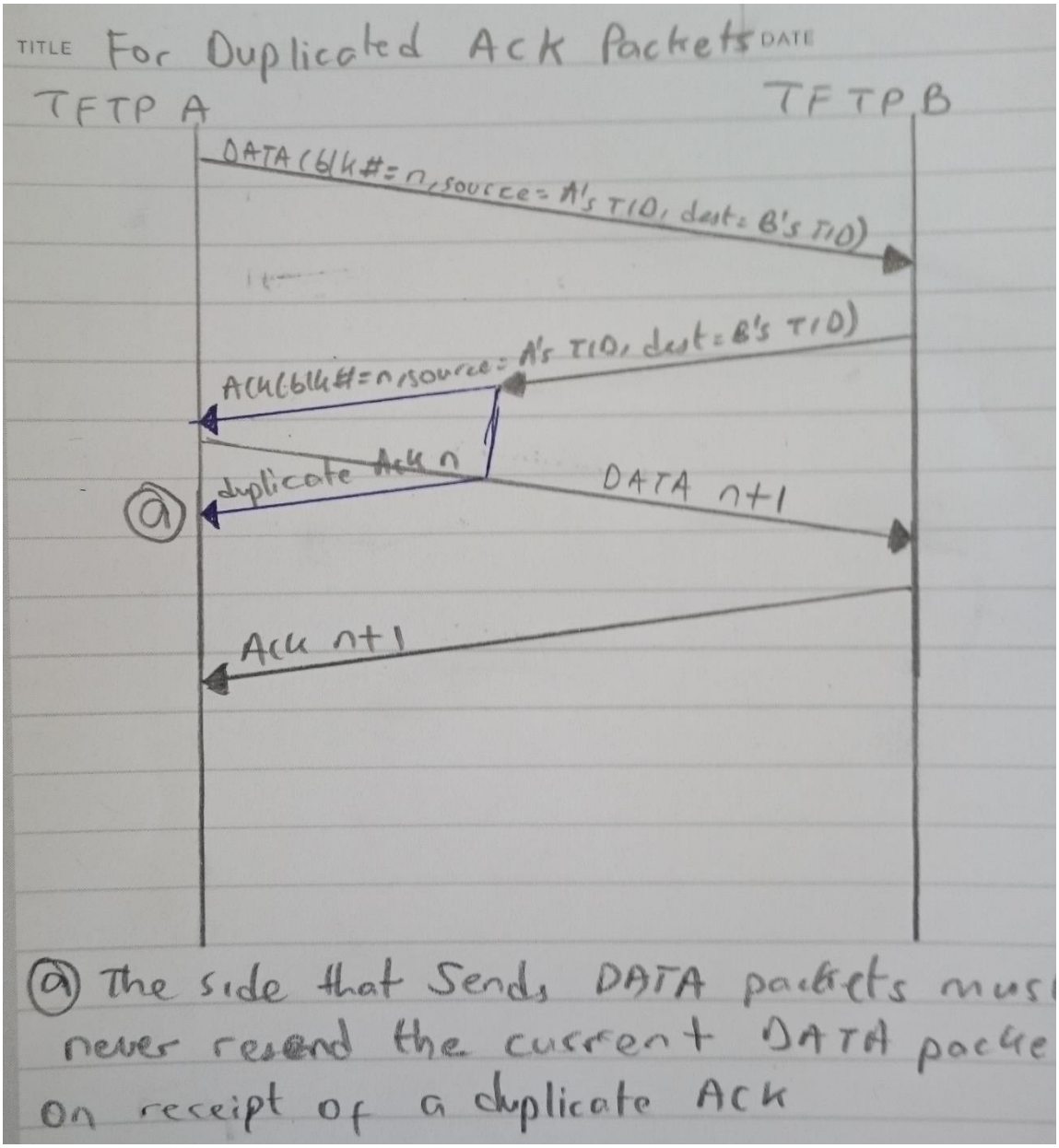

Figure 8. Duplicated ACK packets timing diagram

NOTES:

a) The side that sends DATA packets must never resend the current DATA packet on receipt of a duplicate ACK packet

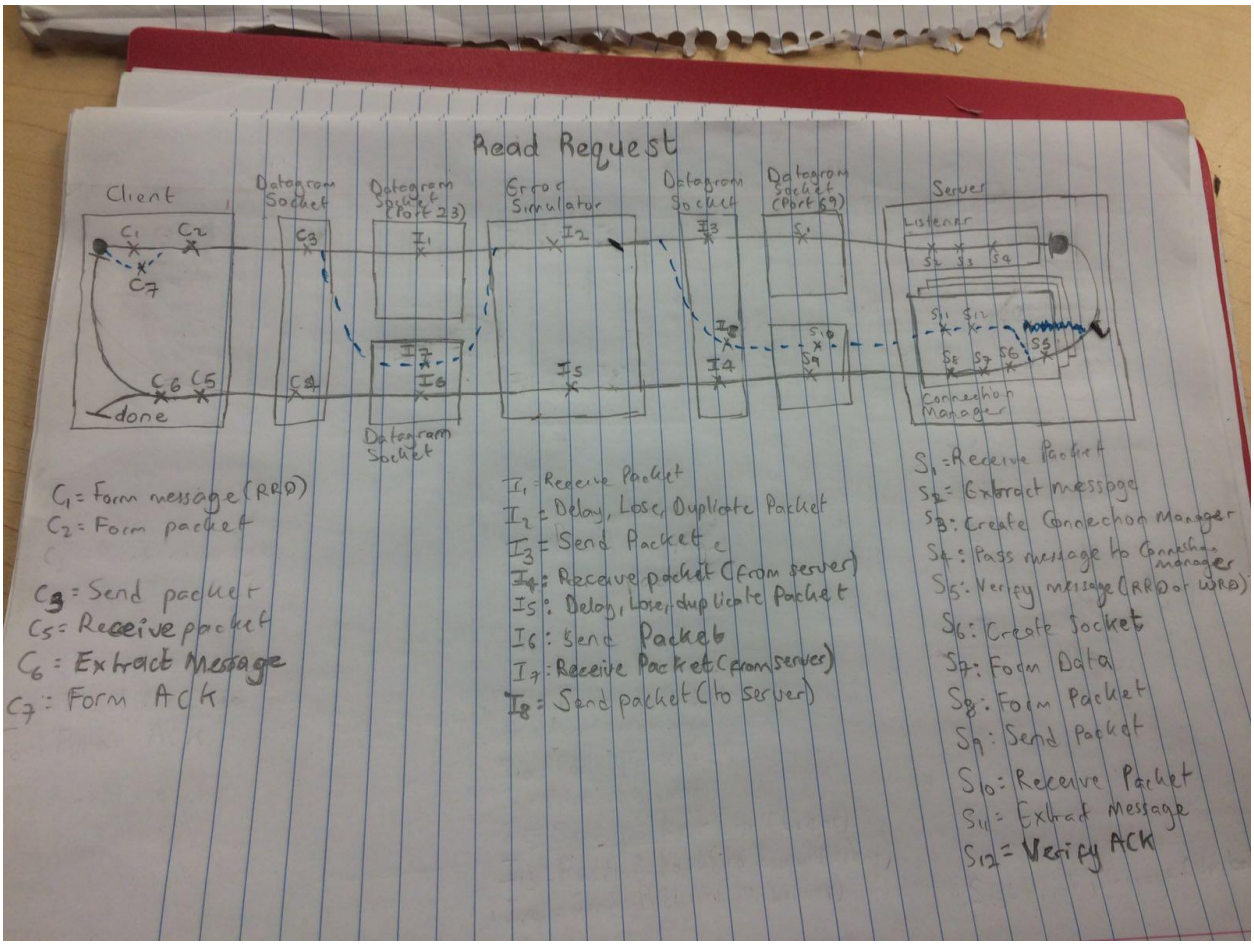

Figure 9. Updated Read request (RRQ) Use case map (UCM) that includes Lost, delayed or duplicated packets

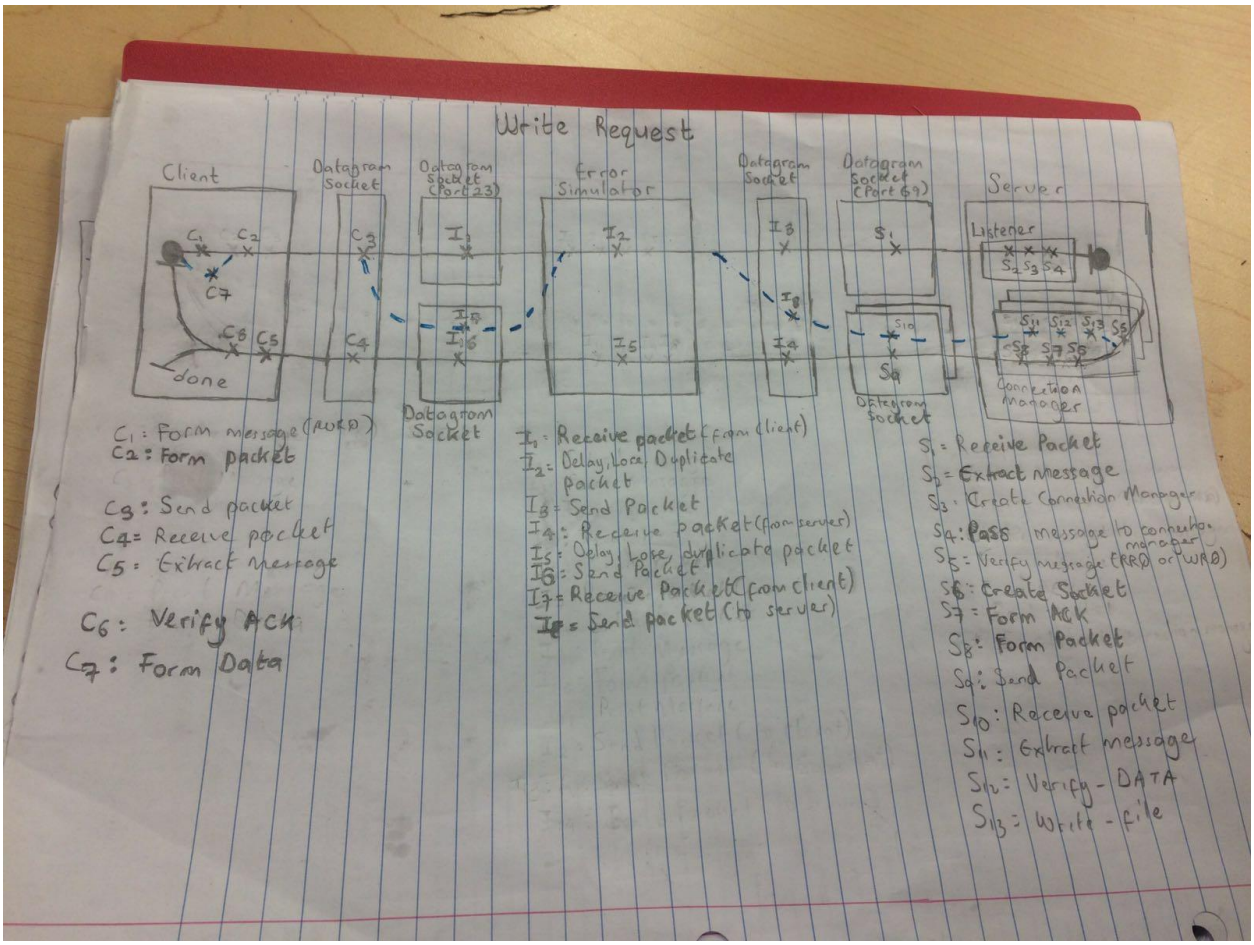

Figure 10. Updated Write Request (WRQ) that includes Lost, delayed or duplicated packets

ITERATION 3:

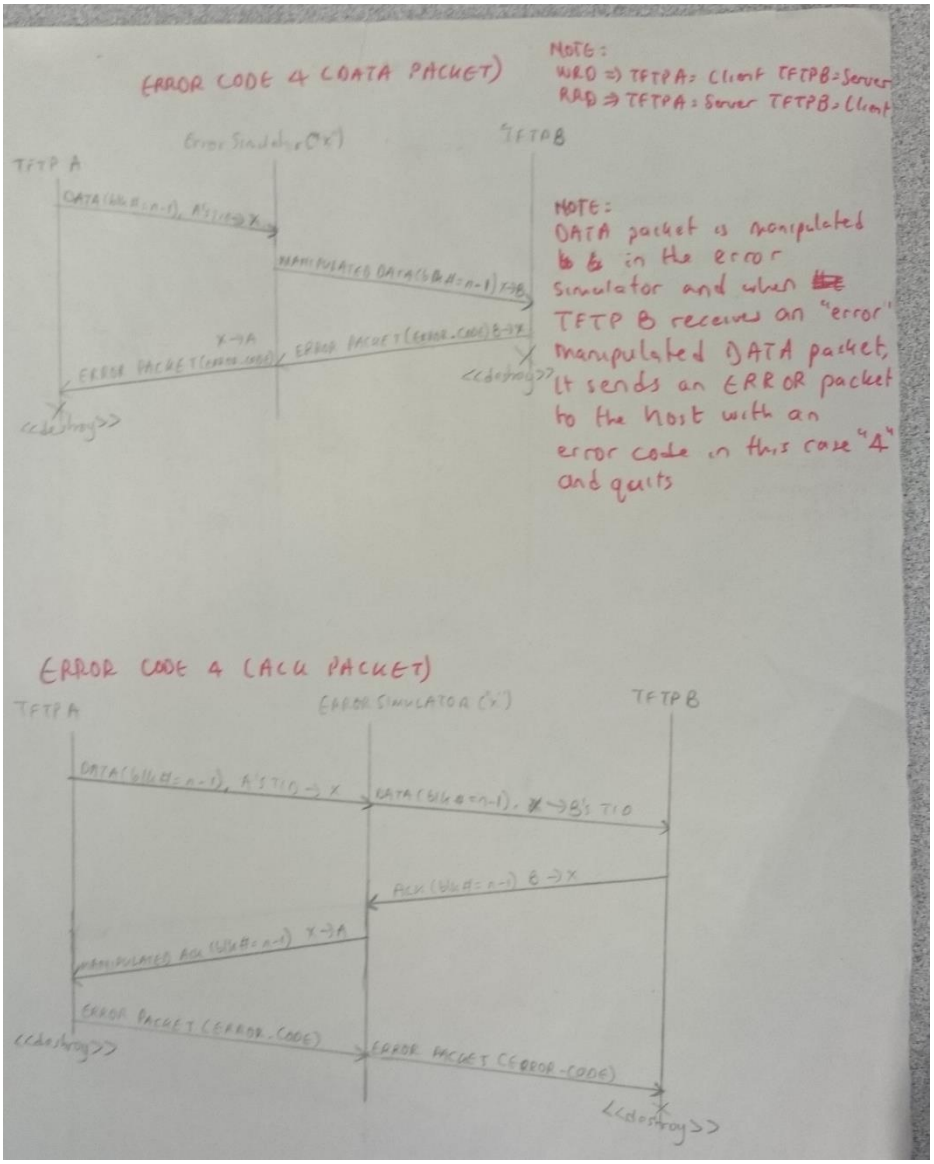

Figure 11. Error Code 4 (with DATA and ACK packets)

NOTES:

WRQ => TFTP A = Client TFTP B = Server

RRQ => TFTP A = Server TFTP B = Client

- DATA packet is manipulated in the error simulator and when TFTP B receives an “error” manipulated DATA packet, it sends an ERROR packet to the host with an error code in this case of “4” and quits

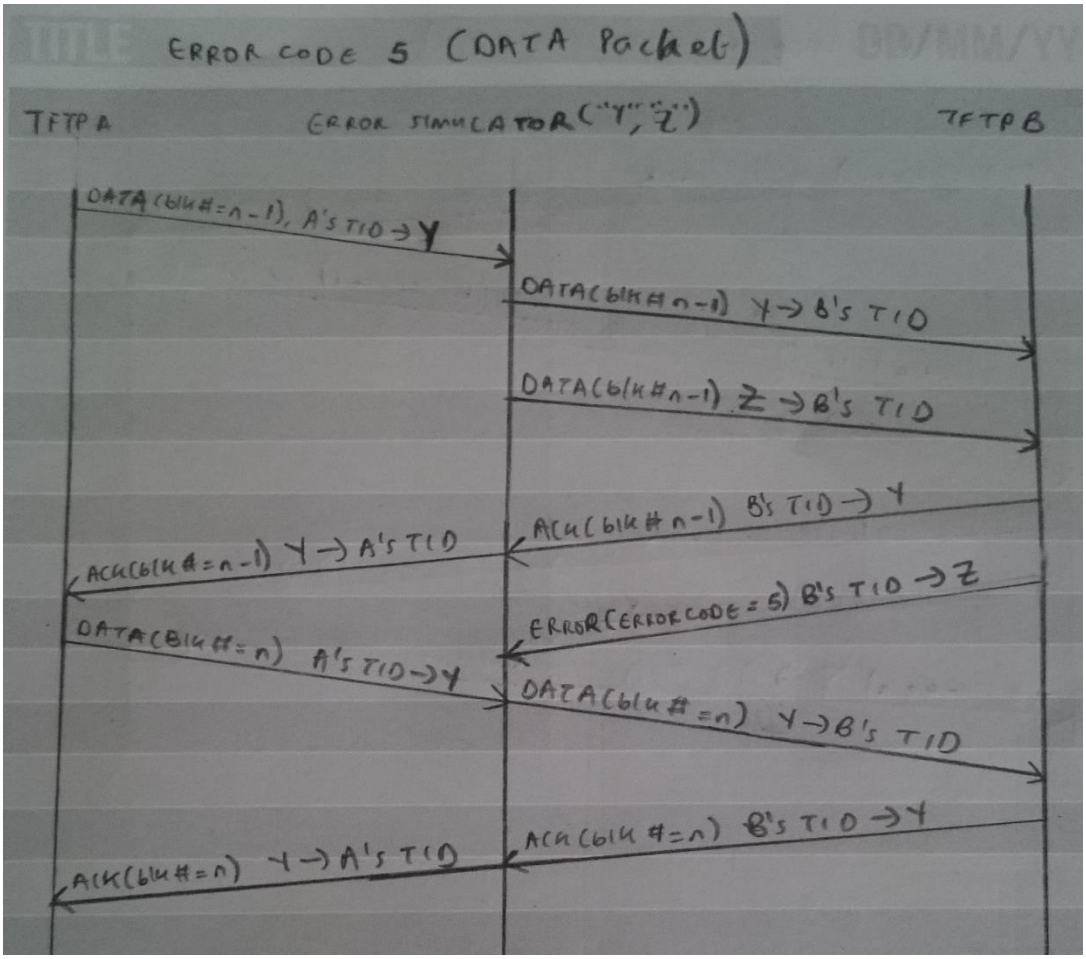

Figure 12. Error Code 5 (DATA Packet) timing diagram

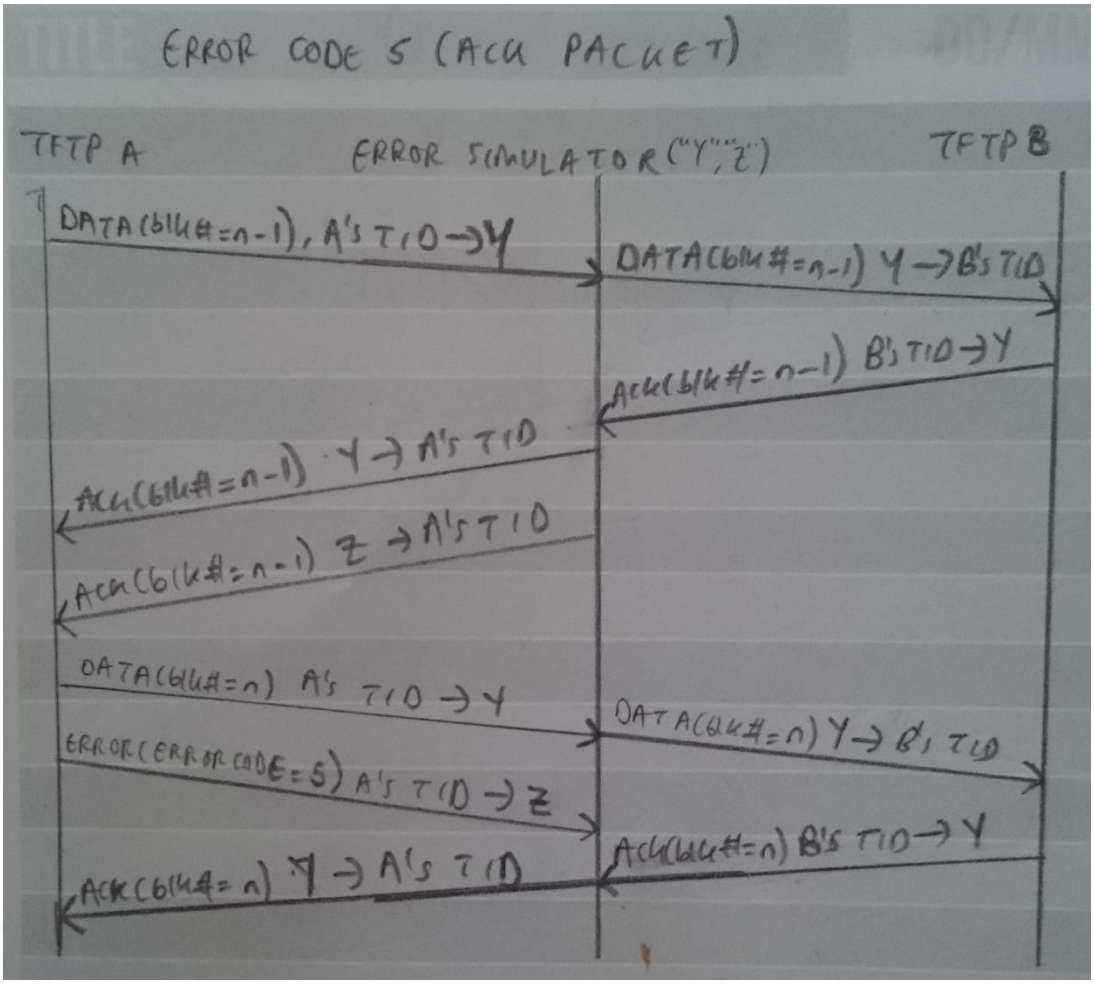

Figure 13. Error Code 5 (ACK Packet) timing diagram

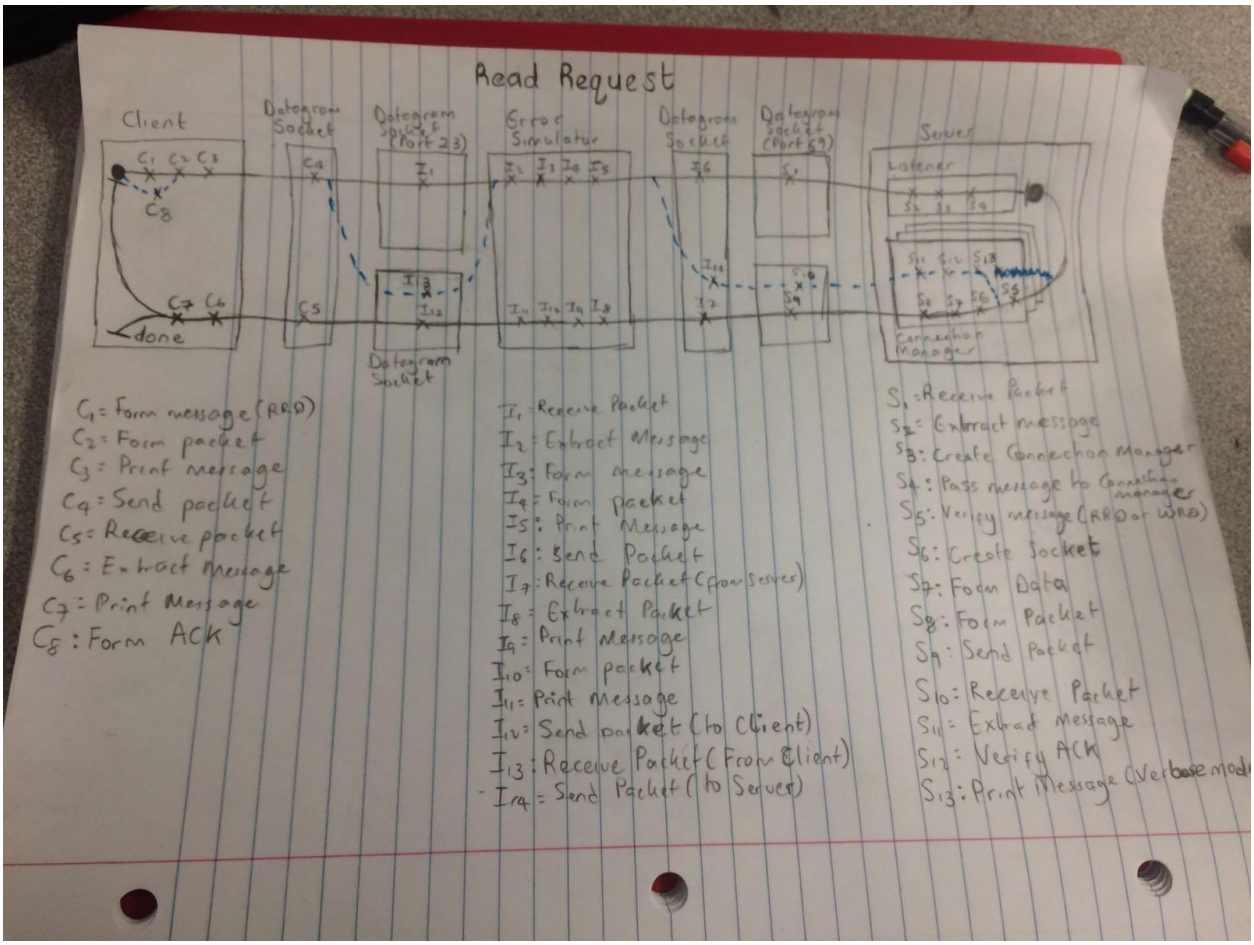

Figure 14. Updated Read Request Use Case Map with errors

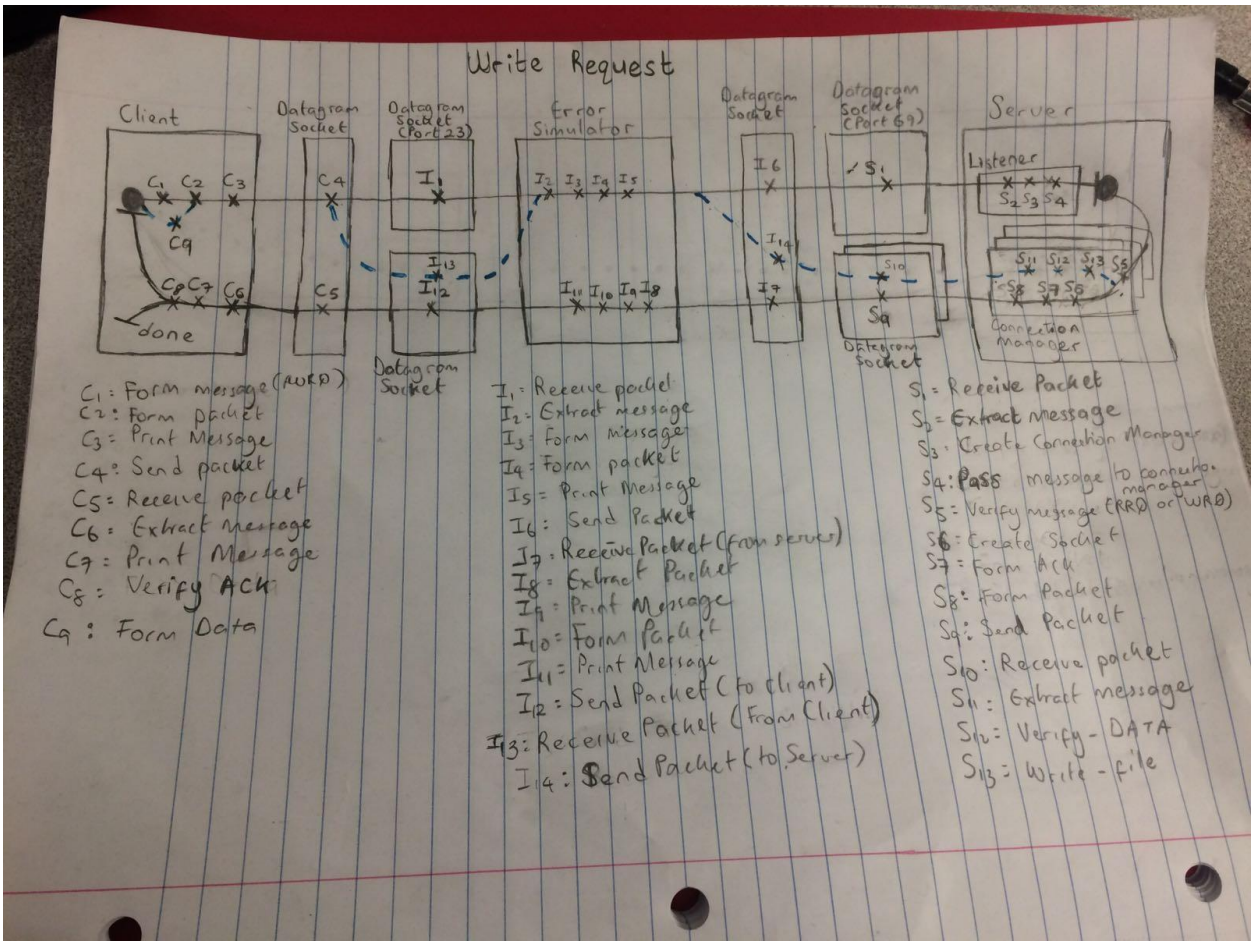

Figure 15. Updated Write Request (WRQ) Use case map (UCM) with errors

ITERATION 4:

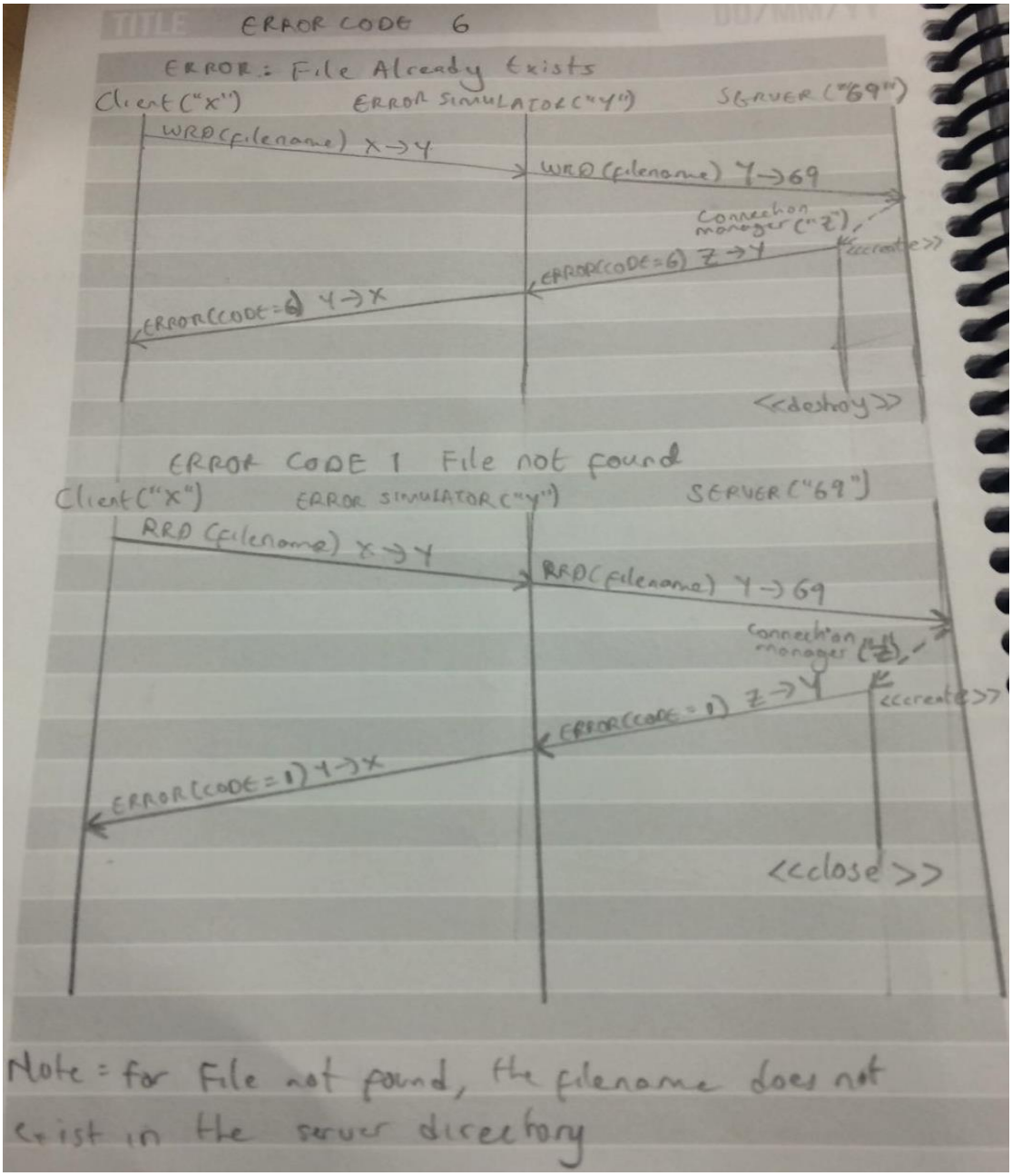

Figure 16. ERROR CODE 6(File Already Exists) and ERROR CODE 1 (File not found) timing diagrams

NOTES:

- For File not found, the filename does not exist in the server directory and can only happen on RRQ
- For File Already Exists, the filename already exists in the server directory and happens when you want to write into the server

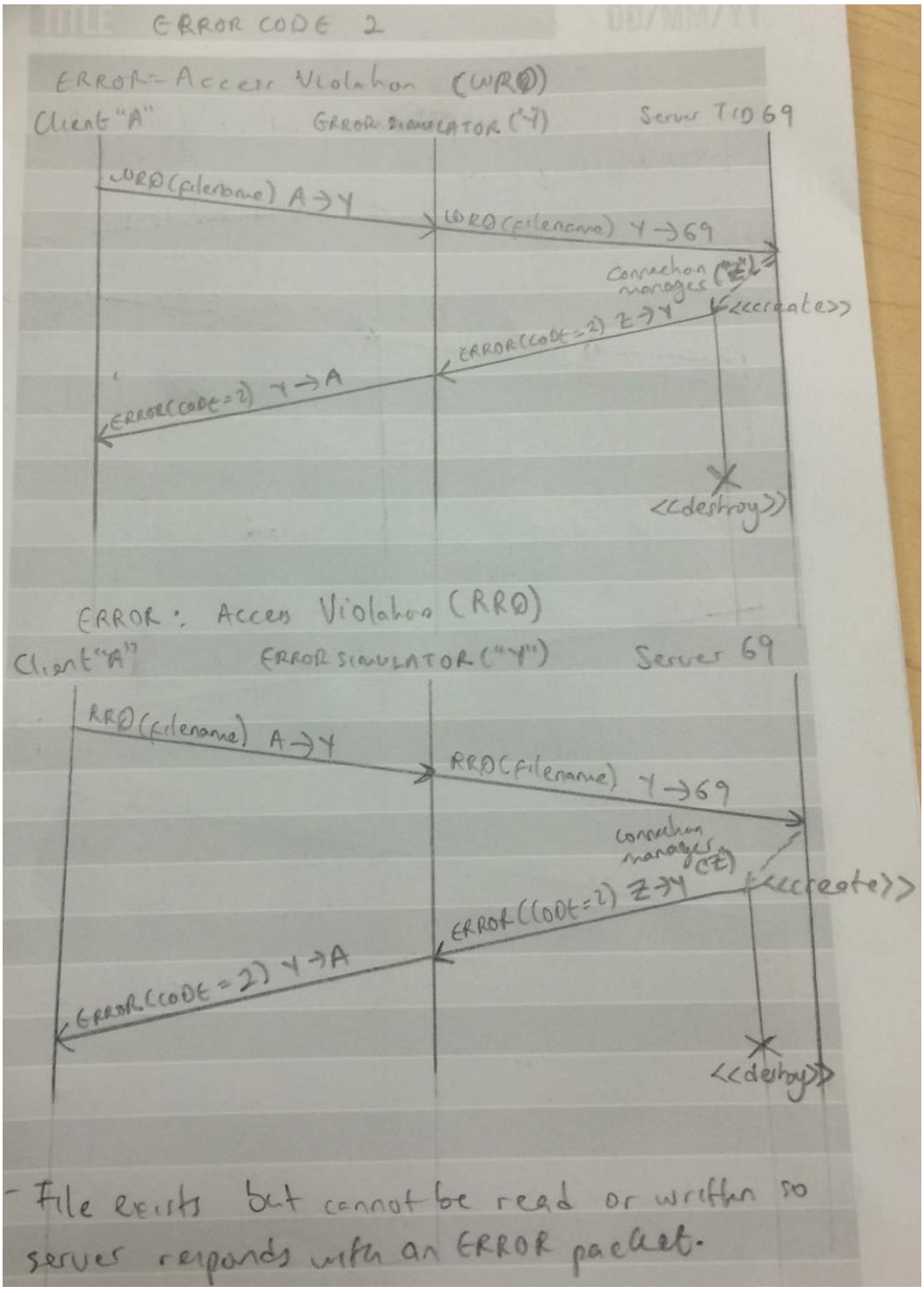

Figure 17. ERROR CODE 2 (Access Violation) timing diagrams for WRQ and RRQ

NOTES:

- File exists but cannot be read or written so server responds with an ERROR packet

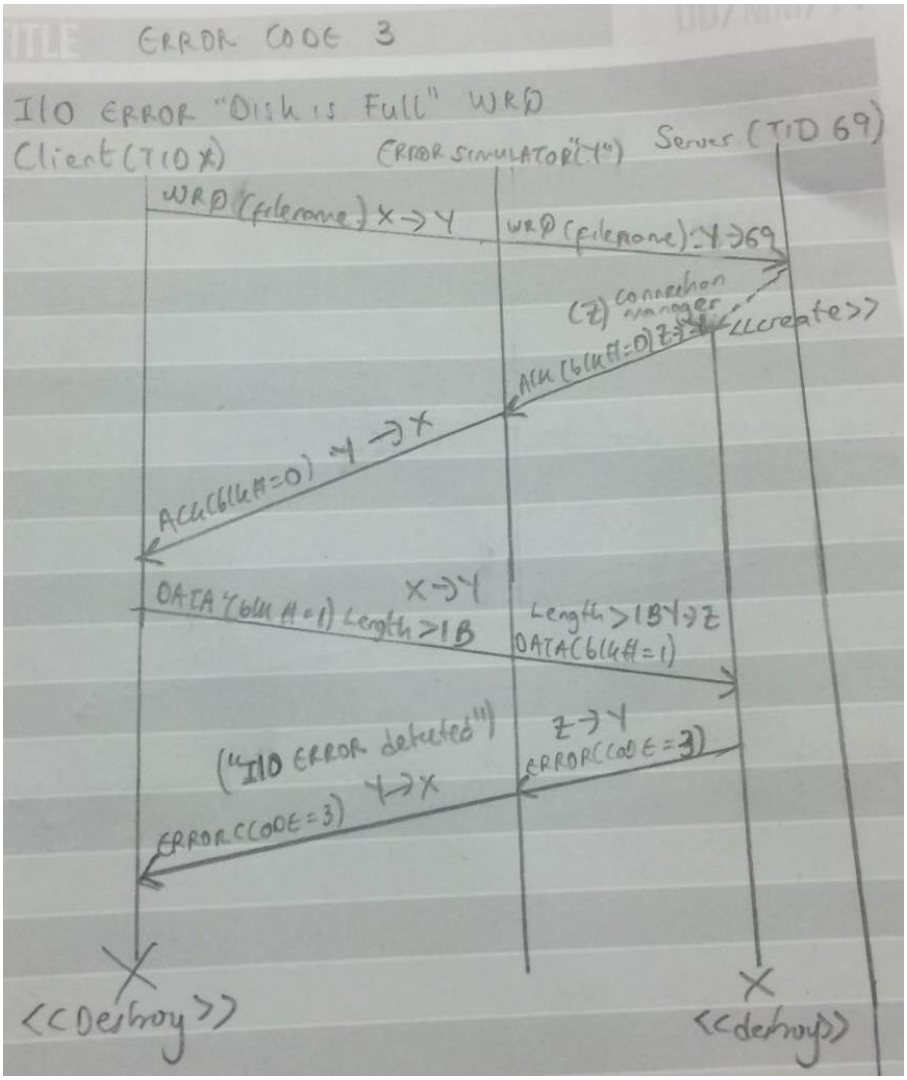

Figure 18. ERROR CODE 3 (Disk is full) on WRQ

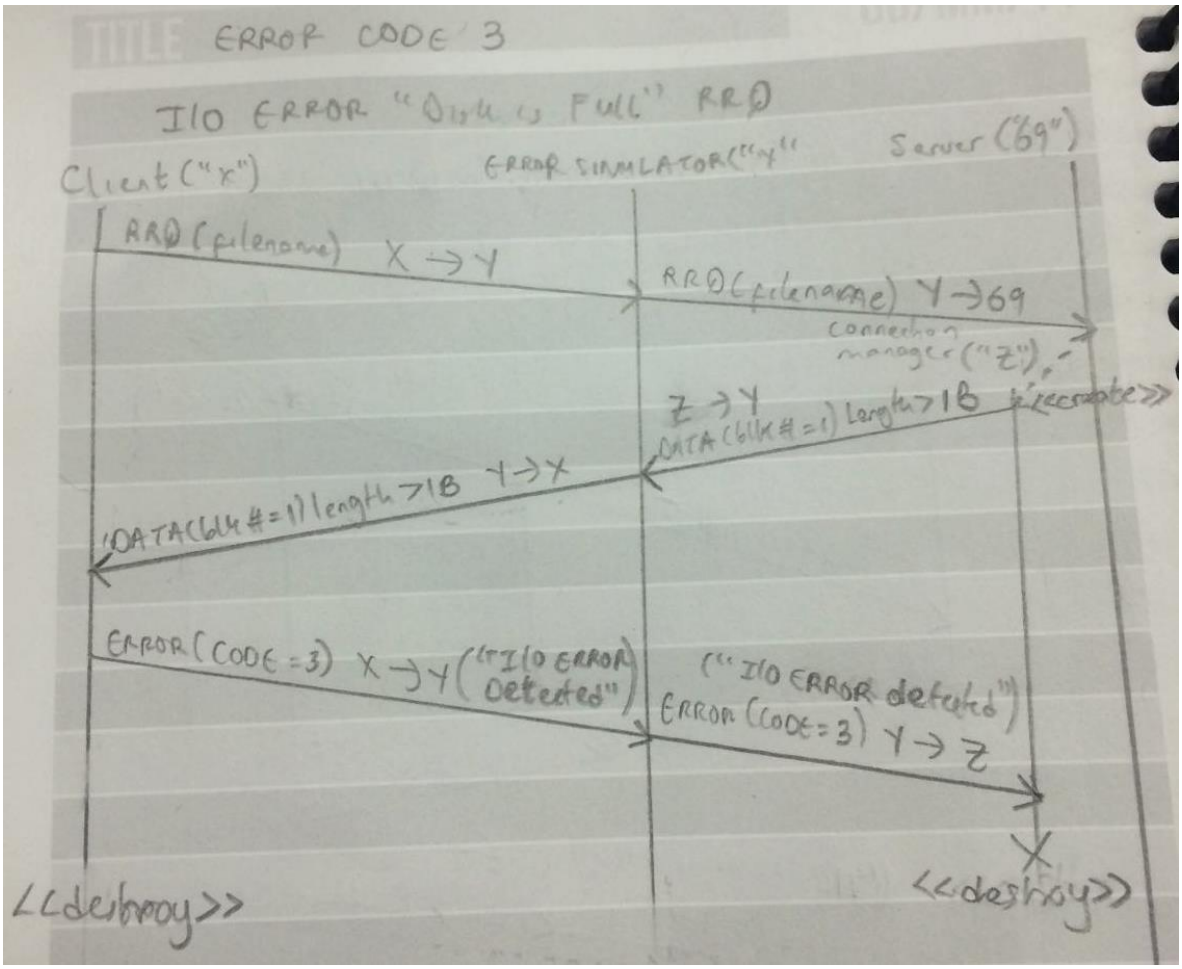

Figure 19. ERROR CODE 3 (Disk is full) on RRQ

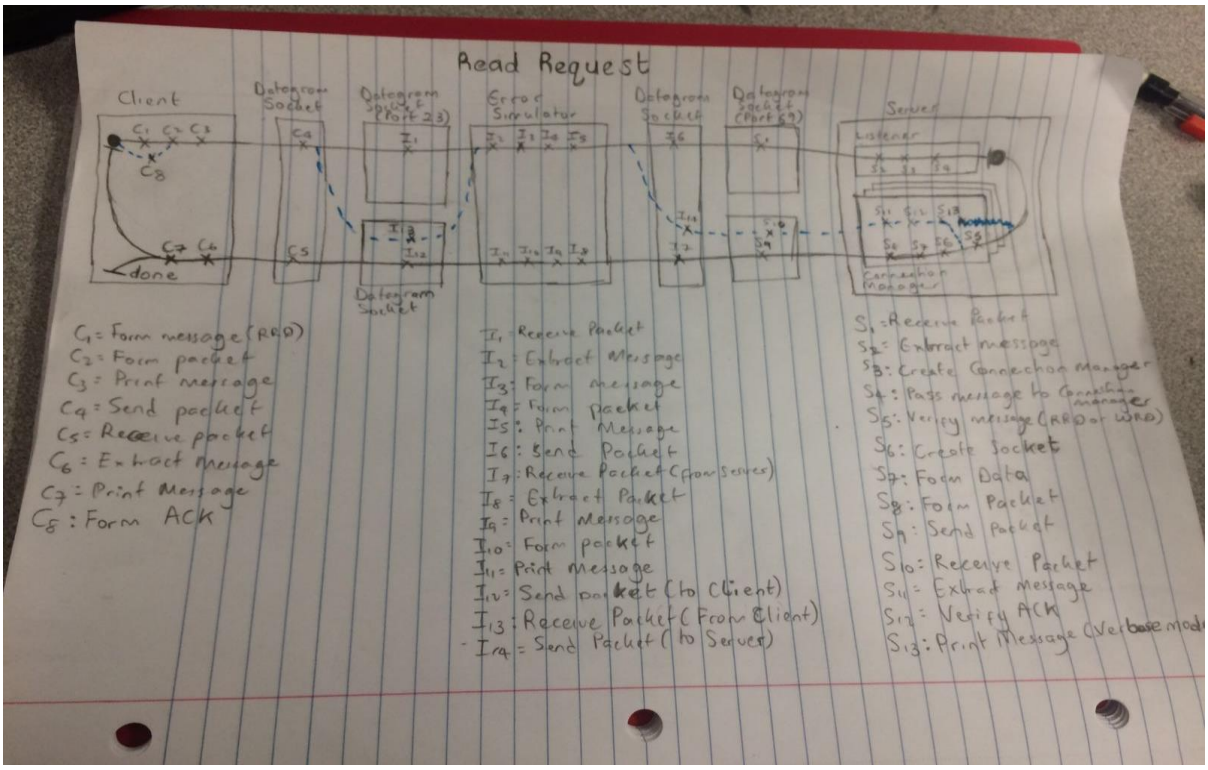

Figure 20. RRQ (Read Request) Use case map

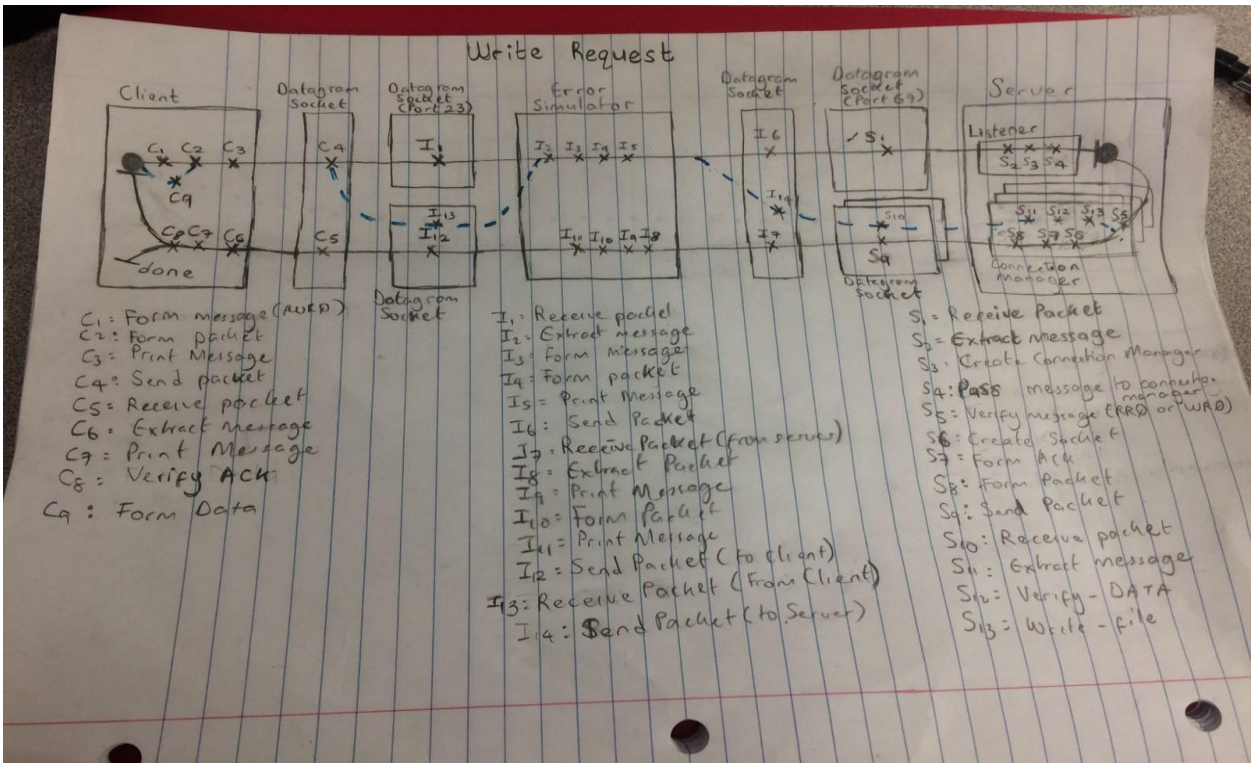

Figure 21. WRQ (Write Request) Use case map

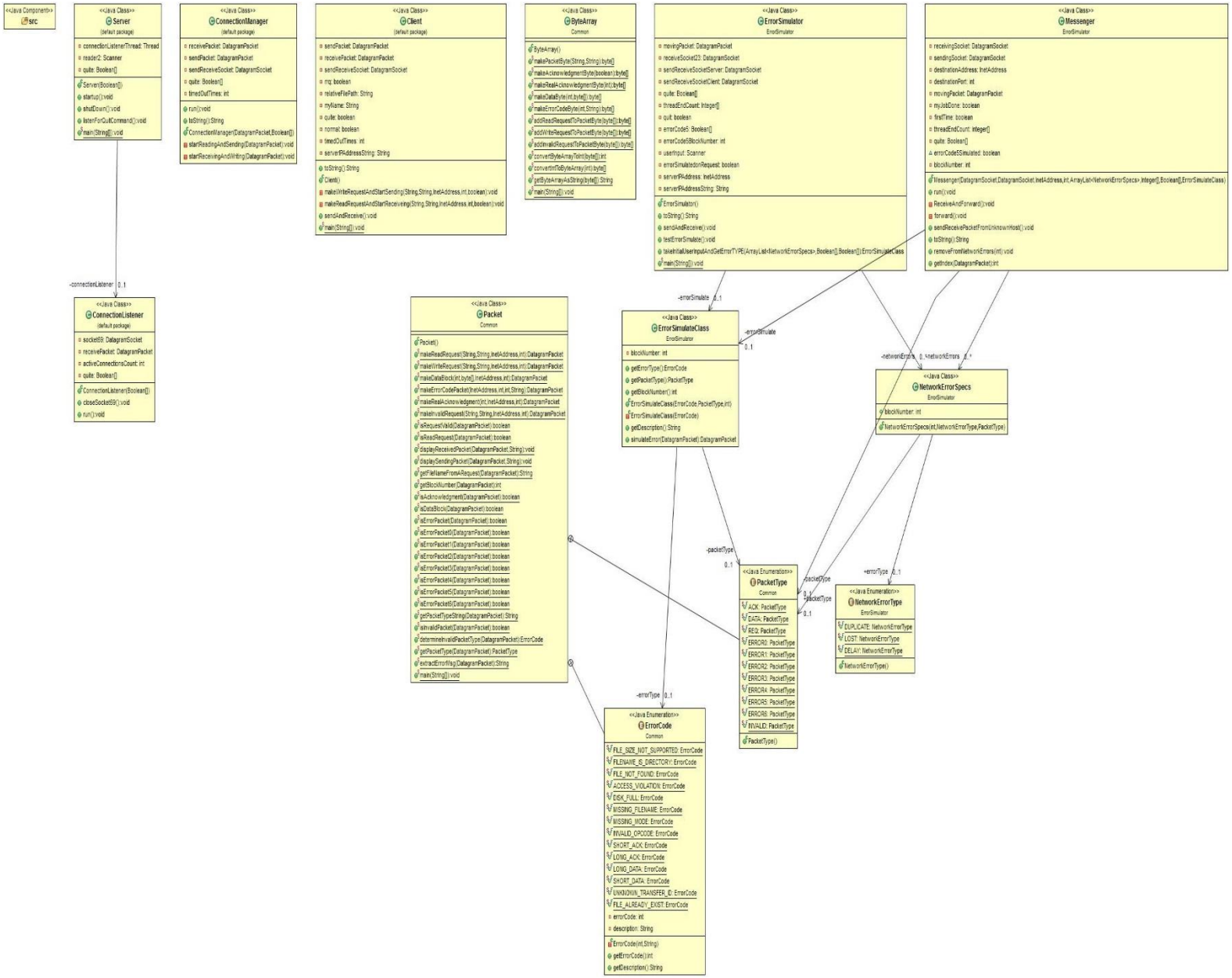

Figure 22. UML Class Diagram of the whole system

FILE DESCRIPTIONS:

Client.java:

This is the client side code. It takes user input as the name of the file you wish to transfer, whether it's a read or a write request, whether it has to operate in either quite or verbose mode, and whether it has to run on normal or test mode. After taking all the user inputs, it makes the request. After the transfer is complete, it asks the user whether the user wants to start another transfer or not. The user has the option to quit or to continue. Another set of user inputs are required if the user wants to make additional transfers.

Server.java:

This is the Server side code. It takes user input whether the program has to be operated in quite or verbose mode. It starts a connectionListenerThread as soon as it starts. It also gives user the option to quit whenever he wants, even during the transfer. To quit, just type quit and push enter into the console; the server will quit and close the ConnectionListener. However all the current requests before quit was requested will be taken care of and completed by the ConnectionManager.

ConnectionListener.java:

This is a thread that is created by the server. It does not take any user inputs. Its only job is to listen on port 69. As soon as it receives a request, it starts a new thread called ConnectionManager that deals with the request.

ConnectionManager.java:

This is a thread created by ConnectionListener. It also does not take in any user input. Its only job is to deal with the requests that is passed on to it by the ConnectionListener. Based on the type of requests it receives, it will make a transfer.

ErrorSimulator.java:

This is the Error Simulator code. It just act as a bridge between the Client and the Server. It takes only one user input initially whether it has to be operated on verbose mode or quite mode. However, after the first transfer of the file. It asks user whether the user wants to quit. A USERINPUT is REQUIRED if you want to make additional transfers.

ByteArray.java:

This is a class with static methods. Its only job is to serve as a list of commonly used methods on ByteArray

Packet.java:

This is also a class with static methods. Its only job is to server as list of commonly used methods on Datagram Packets.

DESIGN DECISIONS

- The Error Simulator was designed as been multithreaded. This was done for simplicity as our Error Simulator can receive packets and sends packets at the same time
- The error simulator and the client are placed on one computer while the server can reside on another computer. If a user decides to run in test mode, then the IP address has to be provided in the Error Simulator.
- The print statements are synchronized. This is done through an Array of Boolean which is shared by all the threads on Server side. The same concept is used on ErrorSimulator.
- Also User has to option to have network errors(duplicate, delay or lose) on multiple packets in one transfer

LIMITATIONS:

The only limitation we have is we cannot put network errors on the initial Read Request (RRQ) or Write Request (WRQ)