Carleton UNIVERSITY

Summer 2017

Title: *FTFP Server-client file transfer system*

Group: 2

Students: Josh Campitelli (101010050)
Ahmed Khattab (100994398)
Dario Luzuriaga (100911067)
Ahmed Sakr (101018695)
Brian Zhang (101008207)

Course: SYSC3303A – Real Time Concurrent Systems
Report Date: June 7, 2017

Table of contents:

Introduction:

This project consists on a client/server system using trivial file transfer protocol (TFTP) with the specification RFC1350. It allows one server to communicate with one or multiple client hosts "simultaneously". A user can then read a file from the server or write a file on the server. The files can be of different lengths and octet types. The project was created in five steps, called "iterations", gradually introducing parts of the functionalities. The project was created to run properly in Eclipse IDE for Java Developers under Microsoft Windows 10 operating system.

Break down of responsibilities:

Iteration 1:

Josh Campitelli: Helped with the design, writing, and debugging of the project, specifically the handling of packets in the Client and Connection classes.

Ahmed Khattab: Worked towards the overall progress of the project including the shutting down of the server and created the UMC and UML diagrams.

Dario Luzuriaga: Helped work towards the completion of the project, the debugging, and the instructions.

Ahmed Sakr: Provided the base code for the project from assignment 1 and helped build and design the protocol and classes of the project.

Brian Zhang: Help build project modularity including constructing the packet class and debugged errors. Worked on logical problems and instructed team members on version control.

Iteration 2:

Josh Campitelli: Contributed with the error handling in the client hub and connection class, helping with the interaction with the error simulations as well.

Ahmed Khattab: Introduced the timing diagrams and helped to develop the verbose mode in the server's side.

Dario Luzuriaga: Contributed to develop the interaction with the user in the error simulator, developed the UML diagrams and wrote documentation like README.txt.

Ahmed Sakr: Introduced the multi-thread capacities in the error simulator, helped to develop methods to add errors in packets and organize their block numbers.

Brian Zhang: Created error packets, helped with error handling on the client and connection classes.

Iteration 3:

Josh Campitelli: Contributed handling errors 1 to 6, categorizing all I/O errors in connection.java, as well as improving the codes in client and server sides.

Ahmed Khattab: Introduced the timing diagrams, specified messages for each error to be displayed to the client's user and helped defining the interaction between Client.java and its user.

Dario Luzuriaga: Contributed to improving the Error Simulator making it easier for users to introduce errors in packets, helped to define new procedures of writing files, and contributed generating documents and UML diagrams.

Ahmed Sakr: Refactored the error simulator, the client, and some of the packet classes. Introduced new documentation across the whole codebase. Wrote many methods for the FileTransfer interface to allow for I/O operations to be completed. Fixed some pre-existing errors.

Brian Zhang: Helped define and integrate all I/O error packets, operated improvements in packet.java and increased the efficiency in handling packets in all instances.

Iteration 4:

Josh Campitelli: Contributed refactoring the client and the connection classes, adding network errors and timeouts on the connection, client and server classes.

Ahmed Khattab: Introduced the timing diagrams, helped to create the errors for simulation as an upgrade of iteration 3, added more comments on programming code.

Dario Luzuriaga: Helped with the interface consistency, and contributed generating internal and external documents and UML diagrams.

Ahmed Sakr: Added network error simulation capability to the error simulator. Fixed all known errors from iteration 2 and iteration 3. Refactored a huge portion of the codebase to be more modular and consistent. Improved I/O methods.

Brian Zhang: Improved upon previous iterations using TA's feedbacks. Helped with network errors.

Iteration 5:

Josh Campitelli: Fixed bugs.

Ahmed Khattab: Fixed bugs.

Dario Luzuriaga: Helped with the documentation/diagrams, guaranteed the computers communication running the system at network level.

Ahmed Sakr: Implemented cross IP TFTP, refactored majority of codebase, ran over 55 test cases, and fixed bugs.

Brian Zhang: Fixed bugs, helped with the documentation.

Detailed set up:

The files contained in the electronic submission of this project should be processed in the following way using Eclipse for Microsoft Windows.

1. Create a new Java project in Eclipse for Java developers.
2. Uncheck *use default location* and choose this project's base folder as the new location.
3. After creating the new project, set up the run configuration for *com.tftp.Server* and *com.tftp.ErrorSimulator* on one computer and *com.tftp.Client* on the same and/or different computers.
    a. To set-up the run configurations, select *run configurations* in the dropdown options beside the run button, right click *Java Application* and make a new configuration. Give the configuration a name and under *Main class* search and select one of the corresponding main classes: *com.tftp.Server*, *com.tftp.ErrorSimulator* or *com.tftp.Client*.
    b. *com.tftp.Server*, *com.tftp.ErrorSimulator* and *com.tftp.Client* are the Server, Error Simulator and Client respectively.
4. Run and follow the command line instructions for setting up the Server and ErrorSimulator first, before running the Client.
    a. The default transfer test files are in data/client or data/server for Client and Server side respectively. They are named after their size (for example, a 16-kilobytes file is named 16KB.txt).
    b. The Error Simulator operates on a mutable queue and *enqueues* and *dequeues* instructions when called upon. This way, you are able to input and load multiple error simulations at once before the need of running the Client.

c. To see the results of a simulated error, read the verbose output of either Client or Server. The interface will display how the system responds to the error. For example, if the modified packet is a data packet with block number 2 and it is an invalid opcode, and the Client sends a write request, then the server will send *Packet type: ERROR*, *Data (as string): Undefined OpCode*, or *Packet* and the Client will print out *Error Packet Received: Error Code: 04, Error Message: Undefined OpCode or Packet*.

Tests:

Several successful tests were run in the system. Some examples are the following ones:

Incorrect Packet Simulation

1. type: WRQ. test: Incorrect mode on WRQ packet (Result: **FAIL**, *RESOLVED*)
2. type: WRQ. test: Incorrect opcode on ACK block. (Result: **PASS**)
3. type: WRQ. test: Incorrect block number on DATA block. (Result: **PASS**)
4. type: WRQ. test: Incorrect block number on ACK block. (Result: **PASS**)
5. type: WRQ. test: Incorrect packet size on DATA block. (Result: **PASS**)
6. type: WRQ. test: Incorrect packet size on ACK block. (Result: **FAIL**, *RESOLVED*)
7. type: RRQ. test: Incorrect mode on RRQ packet (Result: **FAIL**, *RESOLVED*)
8. type: RRQ. test: Incorrect opcode on DATA block. (Result: **PASS**)
9. type: RRQ. test: Incorrect block number on DATA block. (Result: **PASS**)
10. type: RRQ. test: Incorrect block number on ACK block. (Result: **PASS**)
11. type: RRQ. test: Incorrect packet size on DATA block. (Result: **PASS**)
12. type: RRQ. test: Incorrect packet size on ACK block. (Result: **FAIL**, *RESOLVED*)
13. type: RRQ. test: Incorrect packet size on ERROR block. (Result: **FAIL**, *RESOLVED*)
14. type: WRQ. test: Incorrect packet size on ERROR block. (Result: **FAIL**, *RESOLVED*)
15. type: RRQ. test: Incorrect opcode on ERROR block. (Result: **FAIL**, *RESOLVED*)
16. type: WRQ. test: Incorrect opcode on ERROR block. (Result: **FAIL**, *RESOLVED*)
17. type: RRQ. test: Incorrect opcode on RRQ packet. (Result: **PASS**)
18. type: WRQ. test: Incorrect opcode on WRQ packet. (Result: **PASS**)
19. type: RRQ. test: Incorrect packet size on RRQ packet. (Result: **PASS**)
20. type: WRQ. test: Incorrect packet size on WRQ packet. (Result: **PASS**)
21. type: RRQ. test: Incorrect TID on DATA packet. (Result: **PASS**)
22. type: RRQ. test: Incorrect TID on ACK packet. (Result: **PASS**)

23. type: WRQ. test: Incorrect TID on ACK packet. (Result: **PASS**)

Network Error Simulation

24. type: WRQ. test: Delay a DATA packet. (Result: **FAIL**, *RESOLVED*)
25. type: WRQ. test: Delay an ACK packet. (Result: **FAIL**, *RESOLVED*)
26. type: WRQ. test: Delay an ERROR packet caused By I/O error. (Result: **PASS**)
27. type: RRQ. test: Delay a DATA packet. (Result: **FAIL**, *RESOLVED*)
28. type: RRQ. test: Delay an ACK packet. (Result: **FAIL**, *RESOLVED*)
29. type: RRQ. test: Delay an ERROR packet caused By I/O error. (Result: **PASS**)
30. type: WRQ. test: Duplicate a DATA packet. (Result:**FAIL**, *RESOLVED*)
31. type: WRQ. test: Duplicate an ACK packet. (Result: **PASS**)
32. type: WRQ. test: Duplicate an ERROR packet caused By I/O error. (Result: **PASS**)
33. type: RRQ. test: Duplicate a DATA packet. (Result: **FAIL**, *RESOLVED*)
34. type: RRQ. test: Duplicate an ACK packet. (Result: **PASS**)
35. type: RRQ. test: Duplicate an ERROR packet caused By I/O error. (Result: **PASS**)
36. type: WRQ. test: Lose a DATA packet. (Result: **PASS**)
37. type: WRQ. test: Lose an ACK packet. (Result: **FAIL**, *RESOLVED*)
38. type: WRQ. test: Lose an ERROR packet caused By I/O error. (Result: **FAIL**)
39. type: RRQ. test: Lose a DATA packet. (Result: **PASS**)
40. type: RRQ. test: Lose an ACK packet. (Result: **FAIL**, *RESOLVED*)
41. type: RRQ. test: Lose an ERROR packet caused By I/O error. (Result: **FAIL**)

Boundary Conditions Testing

42. type: WRQ. test: transfer 0 bytes file. (Result: **PASS**)
43. type: RRQ. test: transfer 0 bytes file. (Result: **PASS**)
44. type: WRQ. test: transfer 512 bytes file. (Result: **PASS**)
45. type: RRQ. test: transfer 512 bytes file. (Result: **PASS**)

I/O Errors Simulation

46. type: RRQ. test: Read from a file that does not exist. (Result: **FAIL**, *RESOLVED*)
47. type: WRQ. test: Write from a file that does not exist. (Result: **PASS**)
48. type: RRQ. test: Attempt to read from a file that is already in-use. (Result: **PASS**)
49. type: RRQ. test: Lose READ rights on server during a transfer (Result: **PASS**)
50. type: RRQ. test: Lose WRITE rights on client during a transfer (Result: **PASS**)
51. type: WRQ. test: Lose WRITE rights on server during a transfer (Result: **PASS**)
52. type: WRQ. test: Lose READ rights on client during a transfer (Result: **PASS**)
53. type: WRQ. test: Attempt to write a file the client does not have access to its folder. (Result: **PASS**)
54. type: RRQ. test: Attempt to reada file the server does not have access to its folder. (Result: **PASS**)

Programming code parts:

To see the API for the codebase, the reader can extract javadoc.zip and open javadoc/index.html (with some few methods potentially missing for simplification reasons).

To determine what type of packet is being sent, we used two methods from the **Packet** class: *matches()* and *getPacketType(DatagramPacket packet)*. The method *matches()* is used for returning *true* given a certain byte array pattern and opcode. We used *getPacketType(DatagramPacket packet)* for providing the byte array pattern based on TFTP DatagramPacket data regulations and the opcode returning an enumerable named *PacketTypes*.

```
/**
 *
 * Recursively matches a byte array pattern with the provided form as a string where the following letters in the string are important:
 *
 * - c: stands for control and checks for the given byte with the control byte the array provided
 * - x: stands for don't care, used for skipping a dynamic input that terminates once the next pattern in line is found.
 */
protected static boolean matches(byte[] data, int size, String form, byte opcode) {
    return matches(data, 0, size, form, opcode, false);
}

protected static boolean matches(byte[] data, int index, int size, String form, byte opcode, boolean inText) {
    // base case
    if (form.isEmpty() && index == size) {
        return true;
    }

    if (index == size && form.length() == 1 && form.charAt(0) == 'x') {
        return true;
    }

    char letter = 0;
    if (!form.isEmpty()) {
        letter = form.charAt(0);
    }

    if (letter == 'c' && data[index] == opcode) {
        return matches(data, ++index, size, form.substring(1), opcode, false);
    } else if (letter == '0' && data[index] == 0) {
        return matches(data, ++index, size, form.substring(1), opcode, false);
    } else if (letter == 'x') {
        return matches(data, ++index, size, form.substring(1), opcode, true);
    } else if (letter == 'n') {
        return matches(data, ++index, size, form.substring(1), opcode, false);
    } else if (inText) {
        return matches(data, ++index, size, form, opcode, true);
    } else {
        return false;
    }
}
```

```
/**
 *
 * PacketTypes method uses the matches method to determine the type of packet sent to the server
 * then returns the type as an enum temporarily, could have a class with setPacketType() etc.
 */
public static PacketTypes getPacketType(DatagramPacket packet) {
    byte read = 1, write = 2, data = 3, ack = 4, error = 5;

    if (matches(packet.getData(), packet.getLength(), "0cx0x0", read)) {
        return PacketTypes.RRQ;
    } else if (matches(packet.getData(), packet.getLength(), "0cx0x0", write)) {
        return PacketTypes.WRQ;
    } else if (matches(packet.getData(), packet.getLength(), "0cnnx", data)) {
        return PacketTypes.DATA;
    } else if (matches(packet.getData(), packet.getLength(), "0cnn", ack)) {
        return PacketTypes.ACK;
    } else if (matches(packet.getData(), packet.getLength(), "0cx0", error)) {
        return PacketTypes.ERROR;
    } else {
        return PacketTypes.UNKNOWN;
    }
}
```

To verify if the DatagramPacket is a valid packet and not an erroneous one, we used the following three Boolean methods from the **Authenticator** class: *verify(DatagramPacket target, int block)*, *verifyPacketContents(Packet packet, byte[] data, int block)* and *verifyFileEnvironment(Packet packet)*.

```
/**
 * Verifies if the provided target packet is legal and may be consumed.
 * Checks for both errors in the packet itself and the I/O environment.
 *
 * @param target The DatagramPacket in question
 * @param block the actual block number
 *
 * @return  true    if the packet is legal, expected, and may be consumed
 *          false   otherwise
 */
public boolean verify(DatagramPacket target, int block) {
    reset();
    Packet packet = toPacket(target);
    byte[] data = target.getData();

    if (target.getPort() != this.tid) {
        setError(TFTPError.UNKNOWN_TRANSFER_ID);
        setErrorMessage("Incorrect TID");
    } else {
        try {
            if (verifyPacketContents(packet, data, block) && verifyFileEnvironment(packet)) {
                return true;
            }
        } catch (AccessViolationException ex) {
            setError(TFTPError.ACCESS_VIOLATION);
            setErrorMessage("Access Violation: File not accessible.");
            storeResult(target);
            return false;
        }
    }

    storeResult(target);
    return false;
}
```

```
/**
 * Verifies if the provided Packet contains a legal data buffer.
 *
 * @param packet The Packet instance
 * @param data The data buffer of the packet
 * @param block the actual block number
 *
 * @return  true    if the data buffer is legal
 *          false   otherwise
 */
private boolean verifyPacketContents(Packet packet, byte[] data, int block) {
    if (data.length > 516) {
        setError(TFTPError.ILLEGAL_TFTP_OPERATION);
        setErrorMessage("Block greater than 516 bytes");
    } else if (packet == null) {
        setError(TFTPError.ILLEGAL_TFTP_OPERATION);
        setErrorMessage("Undefined OpCode or Packet");
    } else if (mode.equals("reading") && packet.getType() == PacketTypes.DATA) {
        setError(TFTPError.ILLEGAL_TFTP_OPERATION);
        setErrorMessage("Received DATA, Expected ACK");
    } else if (mode.equals("writing") && packet.getType() == PacketTypes.ACK) {
        setError(TFTPError.ILLEGAL_TFTP_OPERATION);
        setErrorMessage("Received ACK, Expected DATA");
    } else if (packet.isBlockNumbered() && BlockNumber.getBlockNumber(packet.getDatagram().getData()) > block) {
        setError(TFTPError.ILLEGAL_TFTP_OPERATION);
        setErrorMessage("Incorrect Block Number");
    } else if (packet.isBlockNumbered() && BlockNumber.getBlockNumber(packet.getDatagram().getData()) < block) {
        setDuplicate(true);
    } else {
        return true;
    }

    return false;
}
```

```
/**
 * Verifies if the environment set up for the packet allows for consumption (i.e. readable/writable).
 *
 * @param packet The Packet instance, used to divert logic flow accordingly
 *
 * @return  true    if the I/O environment is correctly set up
 *          false   otherwise
 */
private boolean verifyFileEnvironment(Packet packet) throws AccessViolationException {
    if (packet.getType() == PacketTypes.RRQ || packet.getType() == PacketTypes.WRQ) {
        String[] parameters = extractTransferParameters(packet.getDatagram());
        this.filename = parameters[0];
        String mode = parameters[1];

        if (!mode.equalsIgnoreCase("octet")) {
            setError(TFTPError.ILLEGAL_TFTP_OPERATION);
            setErrorMessage(String.format("Illegal mode for %s", packet.getType()));
            return false;
        }
    }

    if (packet.getType() == PacketTypes.RRQ && !FileTransfer.isFileExisting(filename)) {
        setError(TFTPError.FILE_NOT_FOUND);
        setErrorMessage(String.format("File Not Found: %s", filename));
    } else if (packet.getType() == PacketTypes.DATA && !FileTransfer.isWritable(filename)) {
        setError(TFTPError.ACCESS_VIOLATION);
        setErrorMessage("Access Violation: File not writable.");
    } else if (packet.getType() == PacketTypes.DATA && FileTransfer.getFreeSpace() < packet.getDatagram().getData().length - 4) {
        setError(TFTPError.DISK_FULL);
        setErrorMessage("Disk Full: Not enough memory to write contents.");
    } else if (packet.getType() == PacketTypes.ACK && !FileTransfer.isReadable(filename)) {
        setError(TFTPError.ACCESS_VIOLATION);
        setErrorMessage("Access Violation: File not readable.");
    } else {
        return true;
    }

    return false;
}
```

To handle and return the appropriate packet, we used a *handlePacket(DatagramPacket received)* method in the **Connection** class. By determining what *PacketTypes* enumerable the Packet is, we assigned the appropriate feedback to the *Packet* variable *response*.

```
private Packet handlePacket(DatagramPacket received) throws UnknownIOModeException, IOException, InvalidPacketException {
    if (!authenticator.verify(received, block)) {
        Packet result = authenticator.getResult();

        if (authenticator.isDuplicate()) {
            return duplicateReceived(result);
        } else if (result.getType() == PacketTypes.ERROR && result.getDatagram().getData()[3] != 5) {
            active = false;
        }

        return result;
    }

    PacketTypes type = Packet.getPacketType(received);
    Packet response;
    if (type == PacketTypes.RRQ) {
        response = rrqReceived(received);
    } else if (type == PacketTypes.WRQ) {
        response = wrqReceived(received);
    } else if (type == PacketTypes.ACK) {
        response = ackReceived(received);
    } else if (type == PacketTypes.DATA) {
        response = dataReceived(received);
    } else if (type == PacketTypes.ERROR) {
        response = errorReceived(received);
    } else {
        throw new InvalidPacketException("Illegal packet parsed!!!");
    }

    if (active) {
        active = fileTransfer != null && !fileTransfer.isComplete();
    }

    return response;
}
```

To print all the information in console we used multiple *printf* statements in the method *inform(DatagramPacket packet, String event, boolean extra).*

```
public void inform(DatagramPacket packet, String event, boolean extra) {
    if (!Client.verbose && !Server.verbose) {
        return;
    }

    int len = packet.getLength();
    PacketTypes type = Packet.getPacketType(packet);
    System.out.printf("%s: %s:\n", this.name, event);
    System.out.printf("Packet type: %s.", Packet.getPacketType(packet));

    if (type == PacketTypes.DATA || type == PacketTypes.ACK) {
        System.out.printf(" Block Number: %d\n", BlockNumber.getBlockNumber(packet.getData()));
    } else {
        System.out.printf("\n");
    }

    System.out.printf("%s Host: %s:%d, Length: %d\n",
                    event.toLowerCase().contains("send") ? "To" : "From", packet.getAddress(), packet.getPort(), len);
    System.out.printf("Data (as string): %s\n", new String(packet.getData(), 0, packet.getData().length));

    if (extra) {
        System.out.printf("Data (as bytes): %s\n\n", Arrays.toString(packet.getData()));
    }
}
```

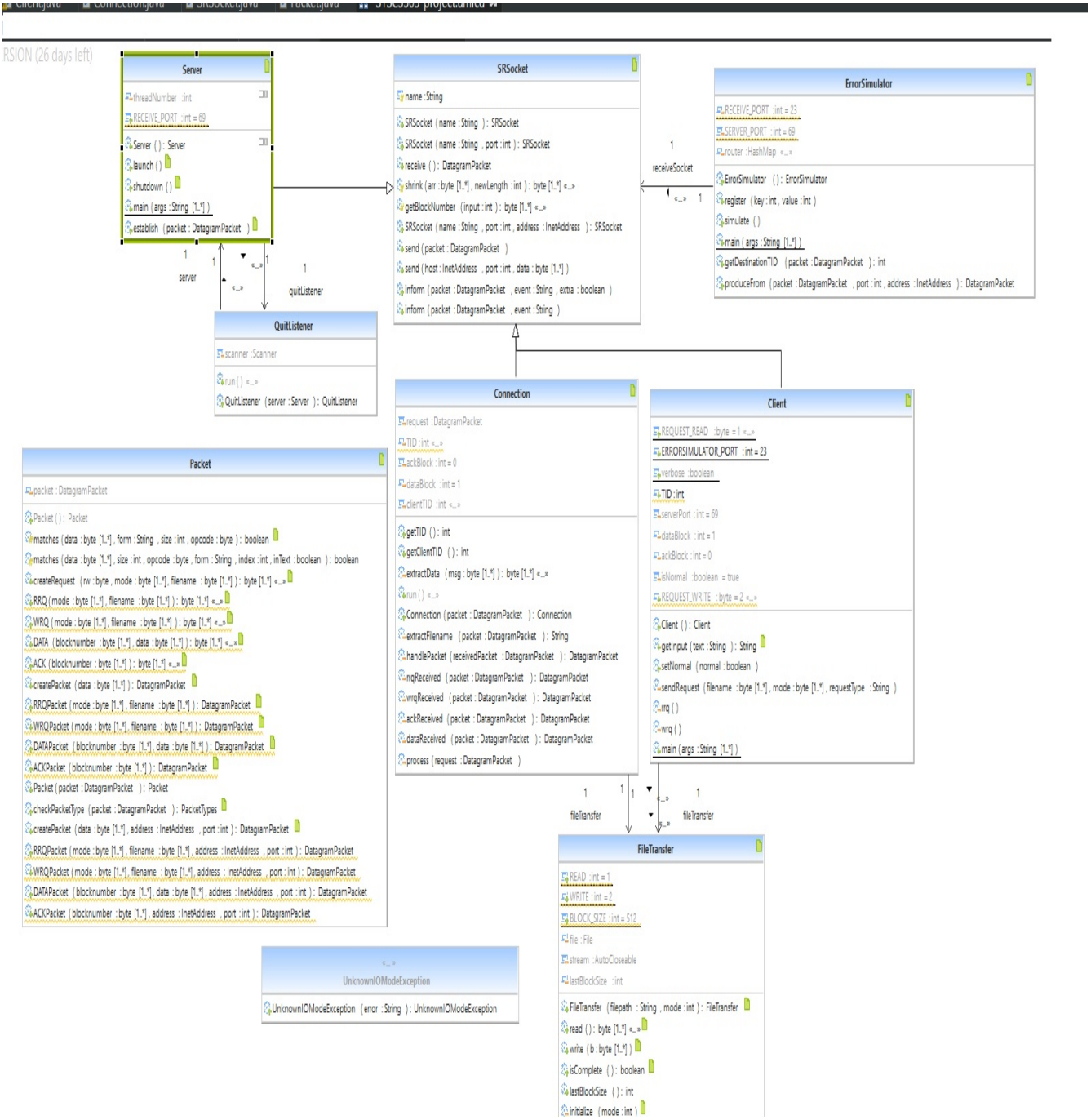
Final remarks:

The system of this project is able to manage the risk of rewriting files, by renaming the new files with a number between brackets. For example, if the file *test(2)*.txt is in the server and the client wants to write a file *test(2).txt* with the same name, the new file that will appear in the server's folder will be *test(3).txt*. However, the client's is not alerted about the change of name, since this information is considered irrelevant for this application.

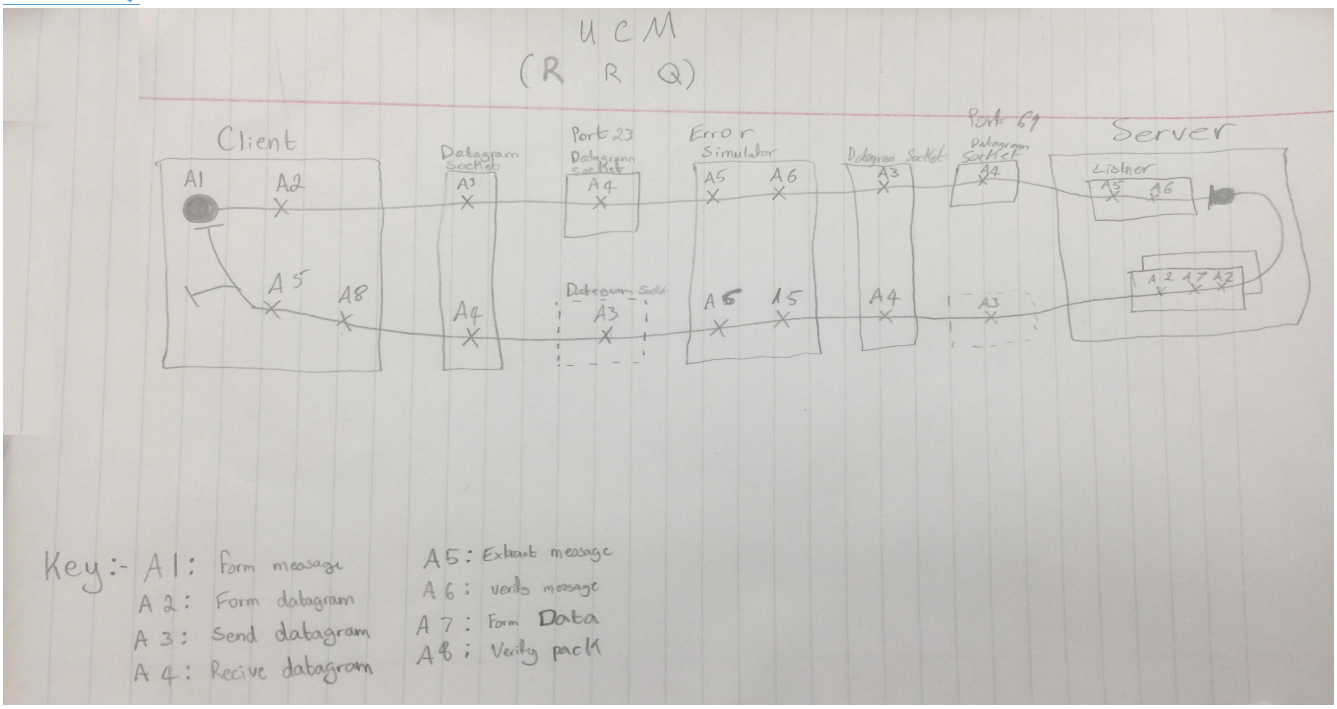
On the other hand, in the case of a duplicated packed, the host sending the duplicated packet is informed about this duplication. The other side of the host just ignores the duplicated packet.

Diagrams for Iteration 1:

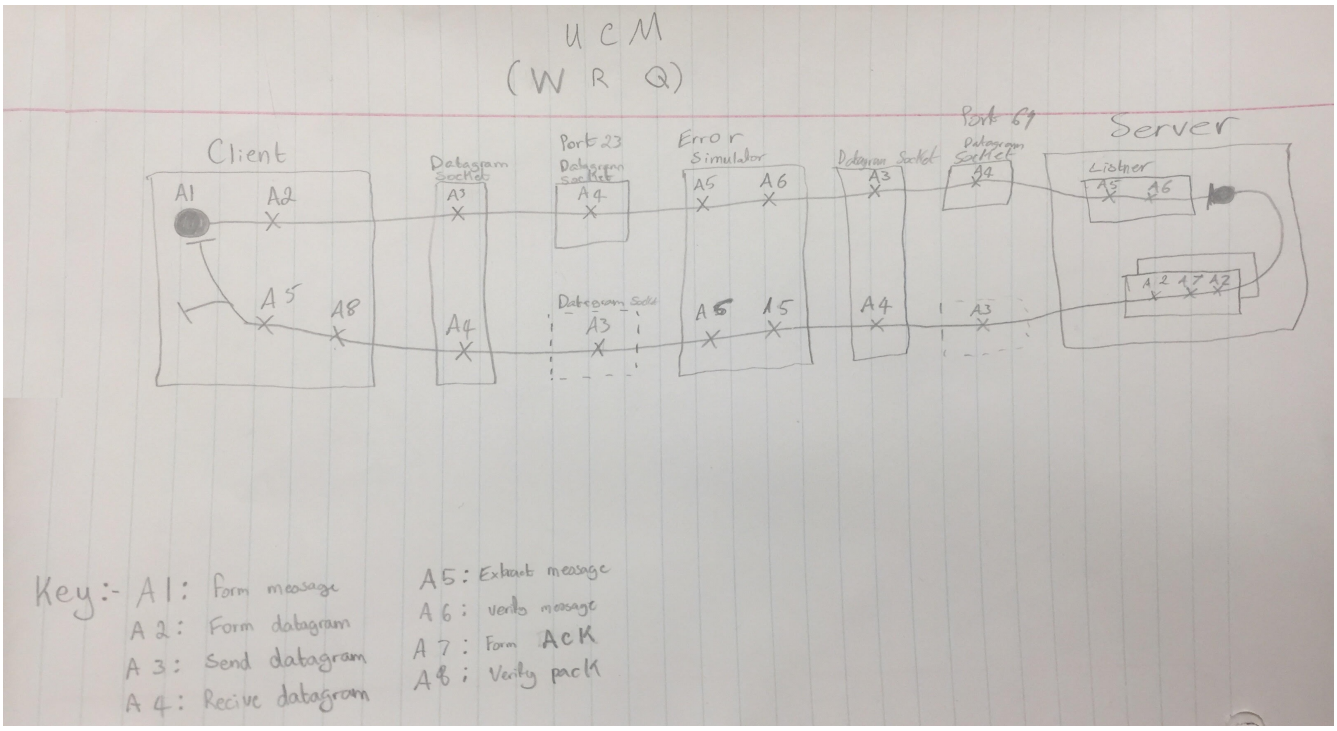
UML class:



UCM RRQ:

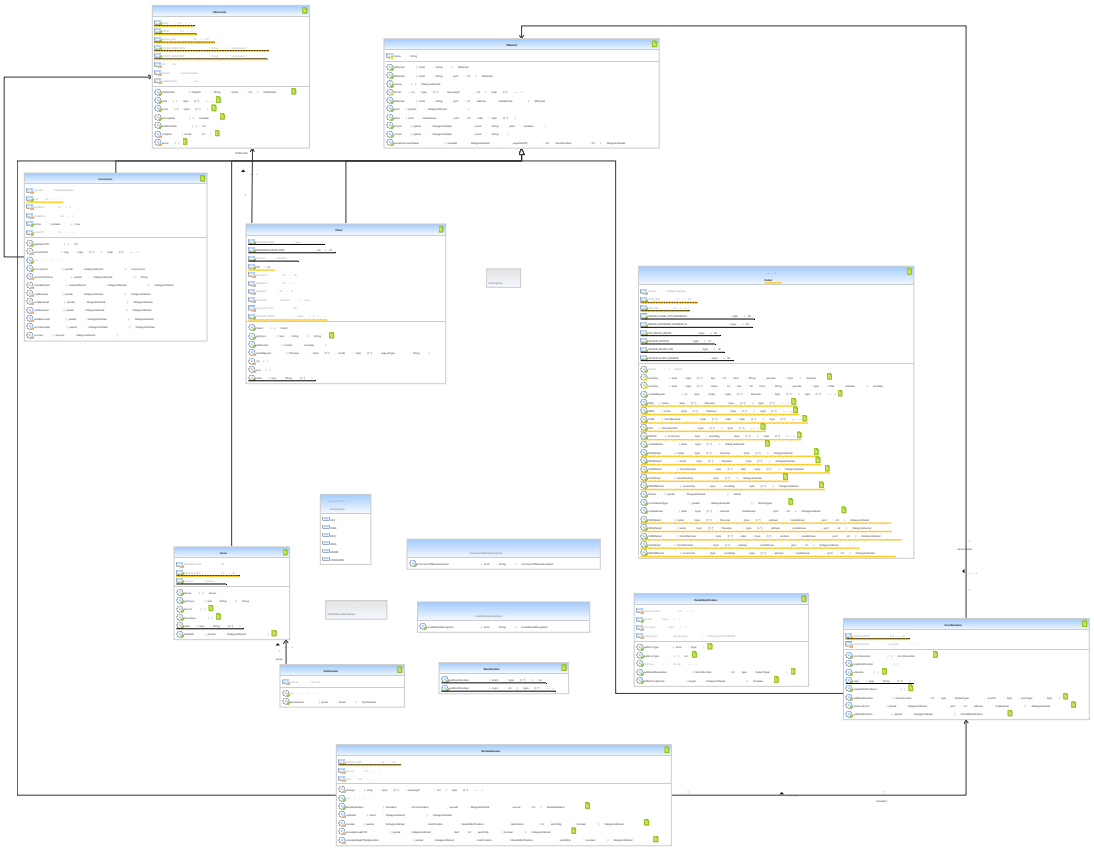


UCM WRQ:

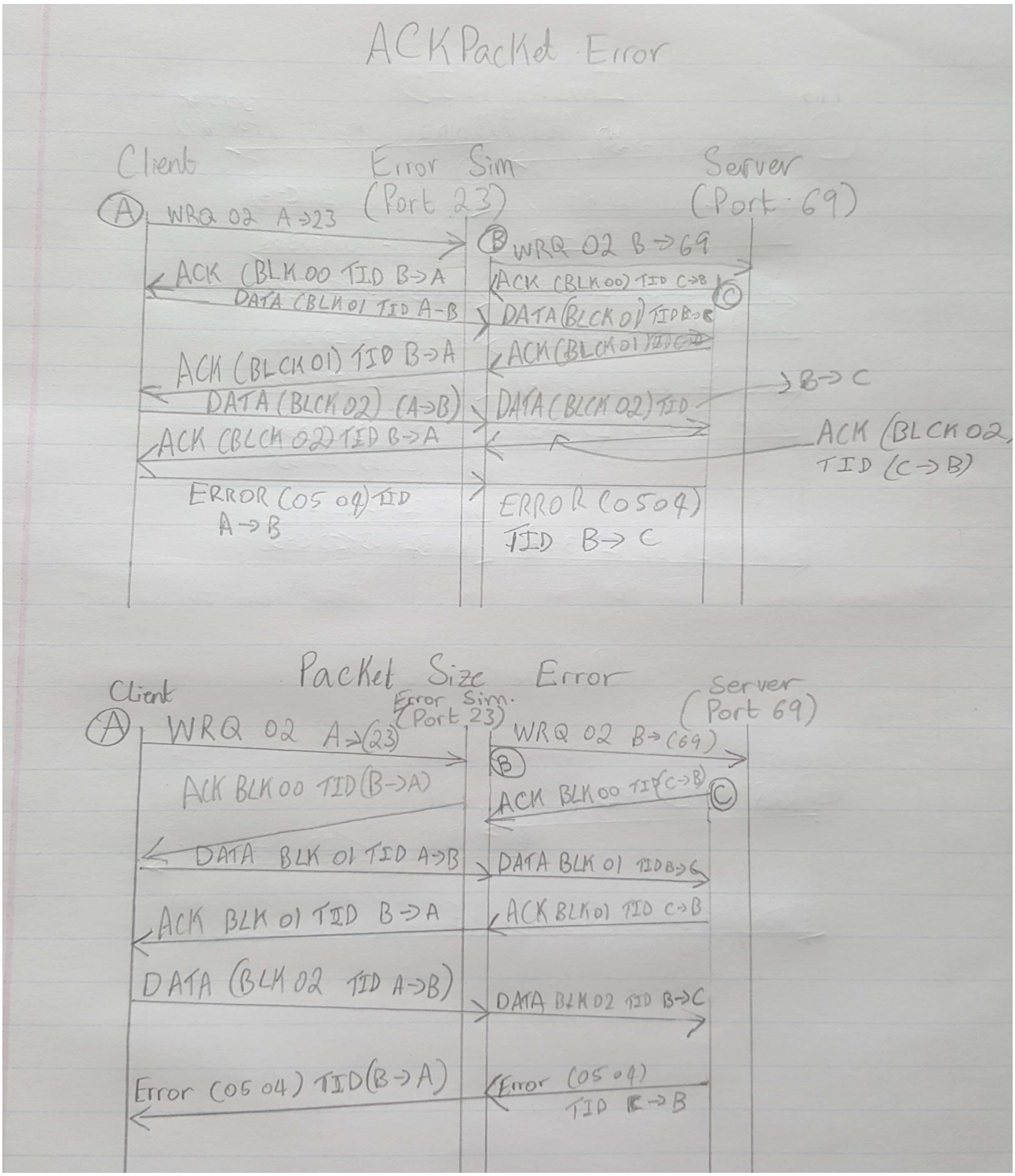


Diagrams for Iteration 2:

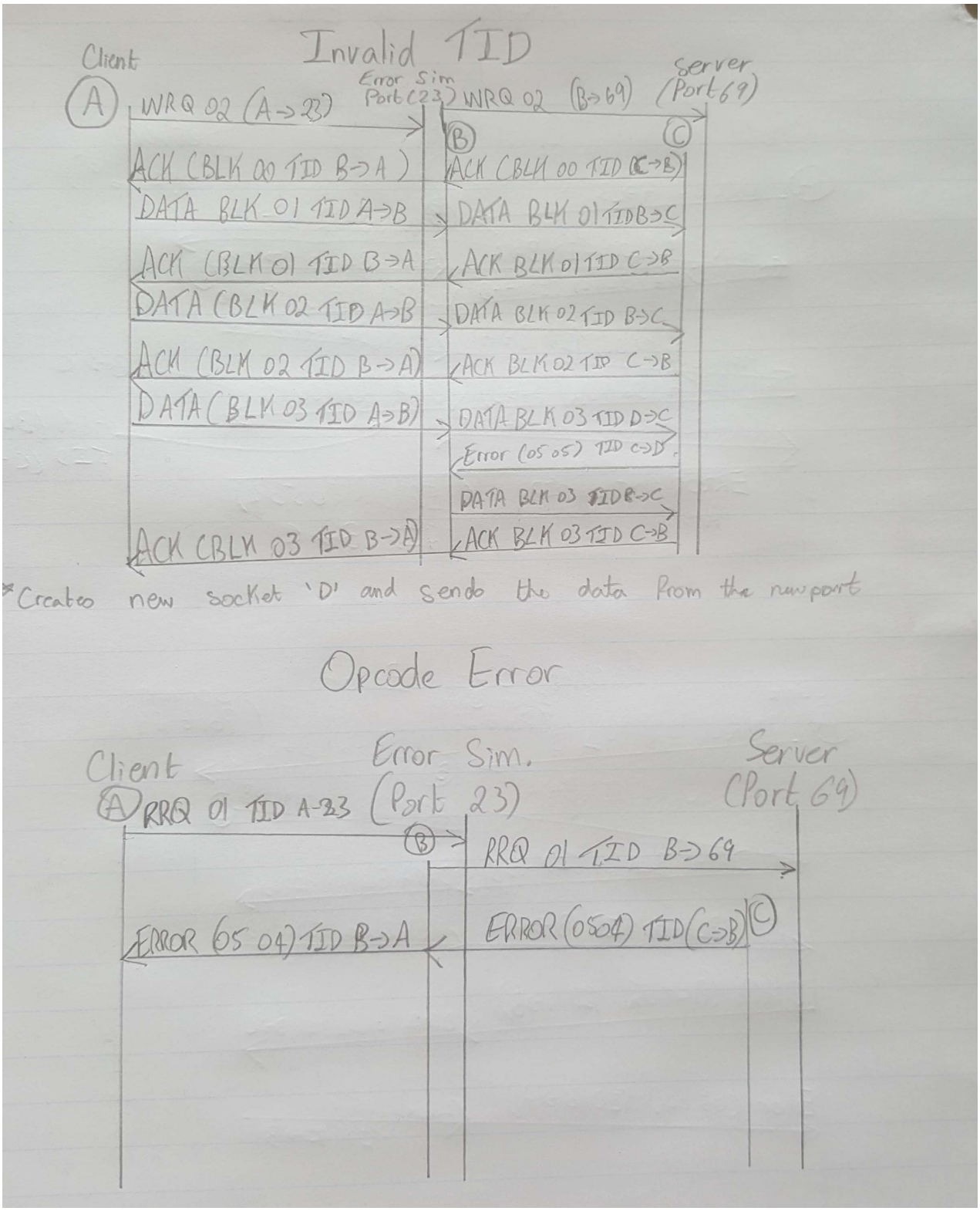
UML class:



Timing diagram 1:

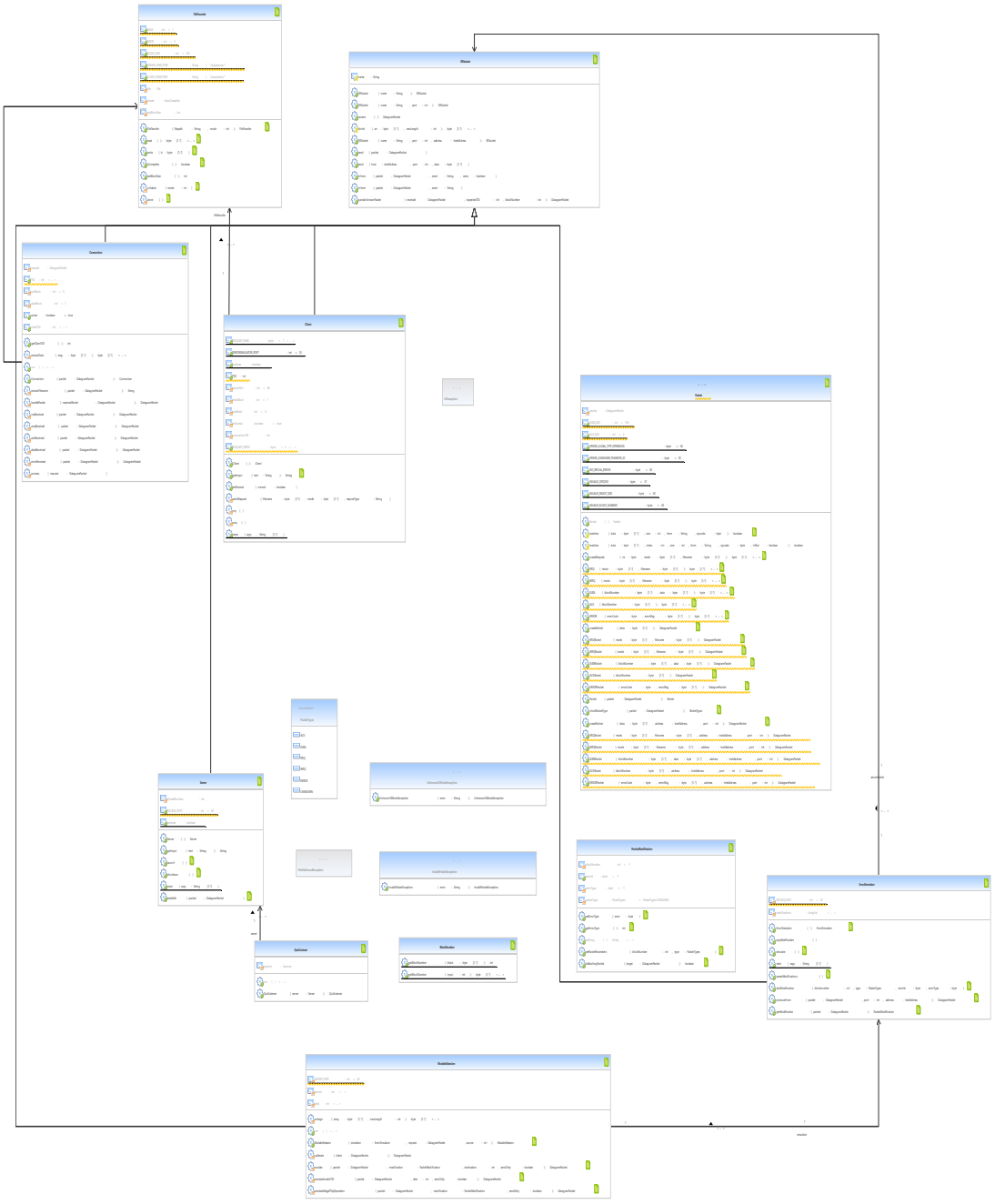


Timing diagram 2:

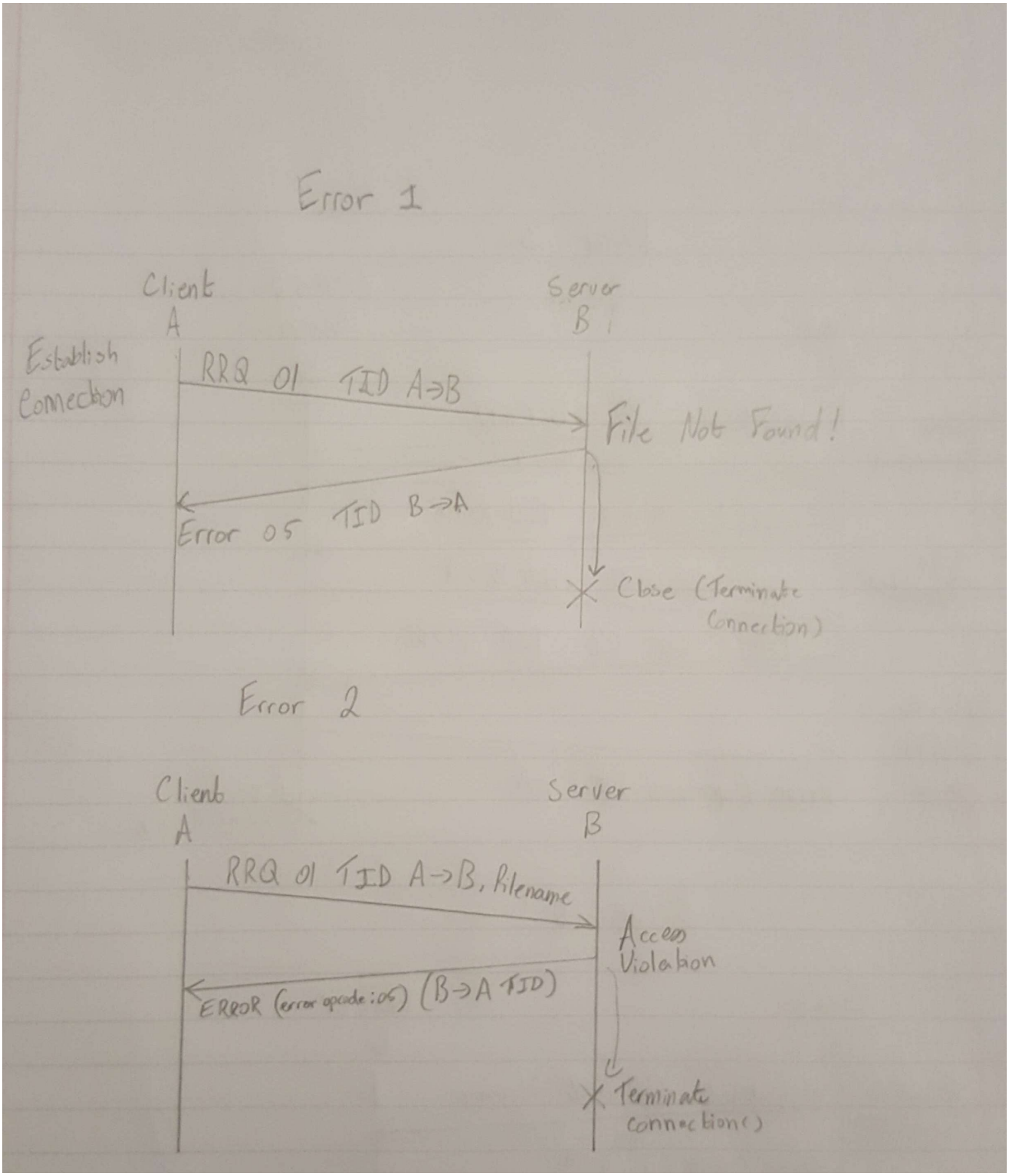


Diagrams for Iteration 3:

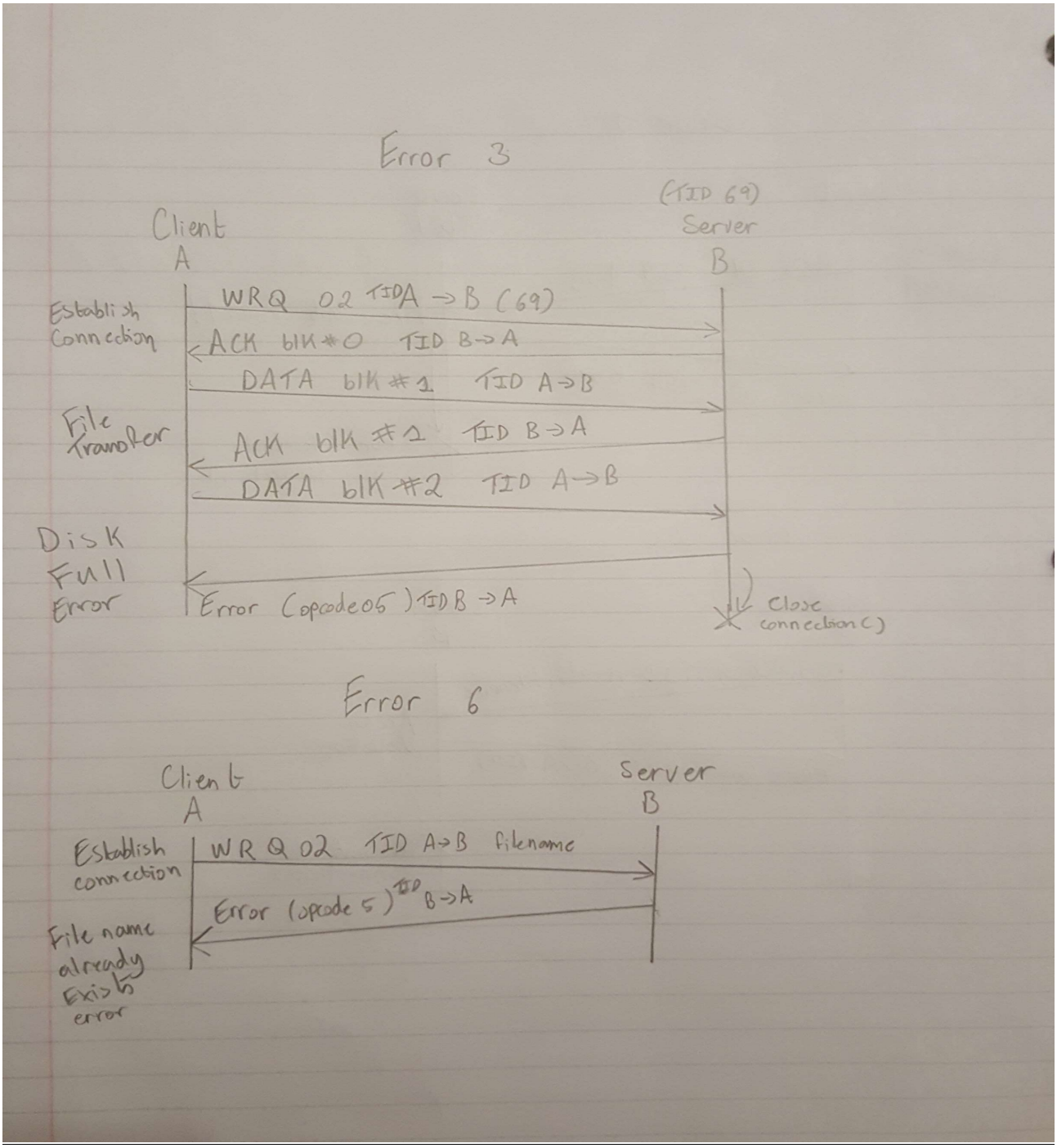
UML class:



Timing diagrams for errors 1 and 2:

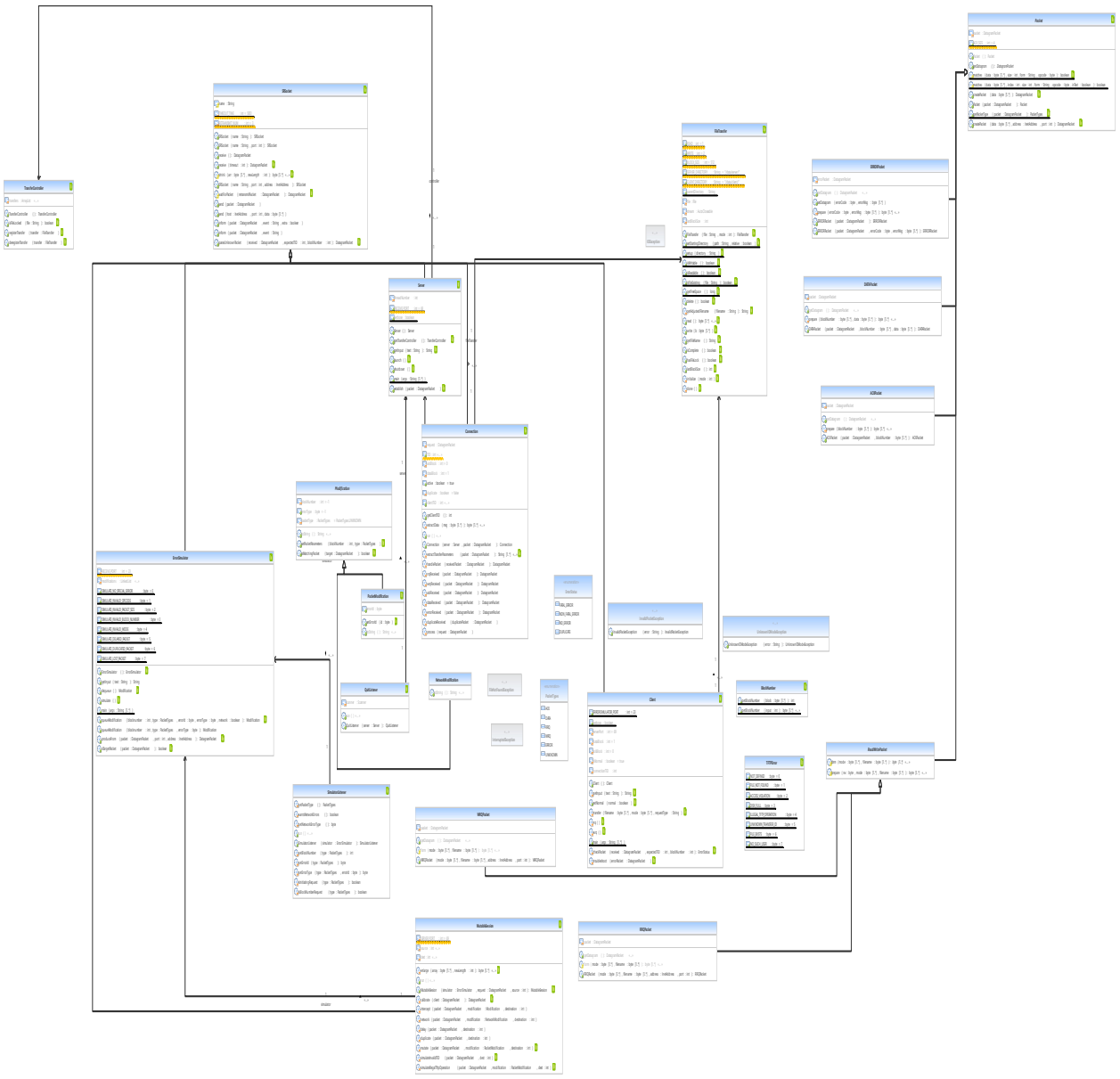


Timing diagrams for errors 3 and 6:

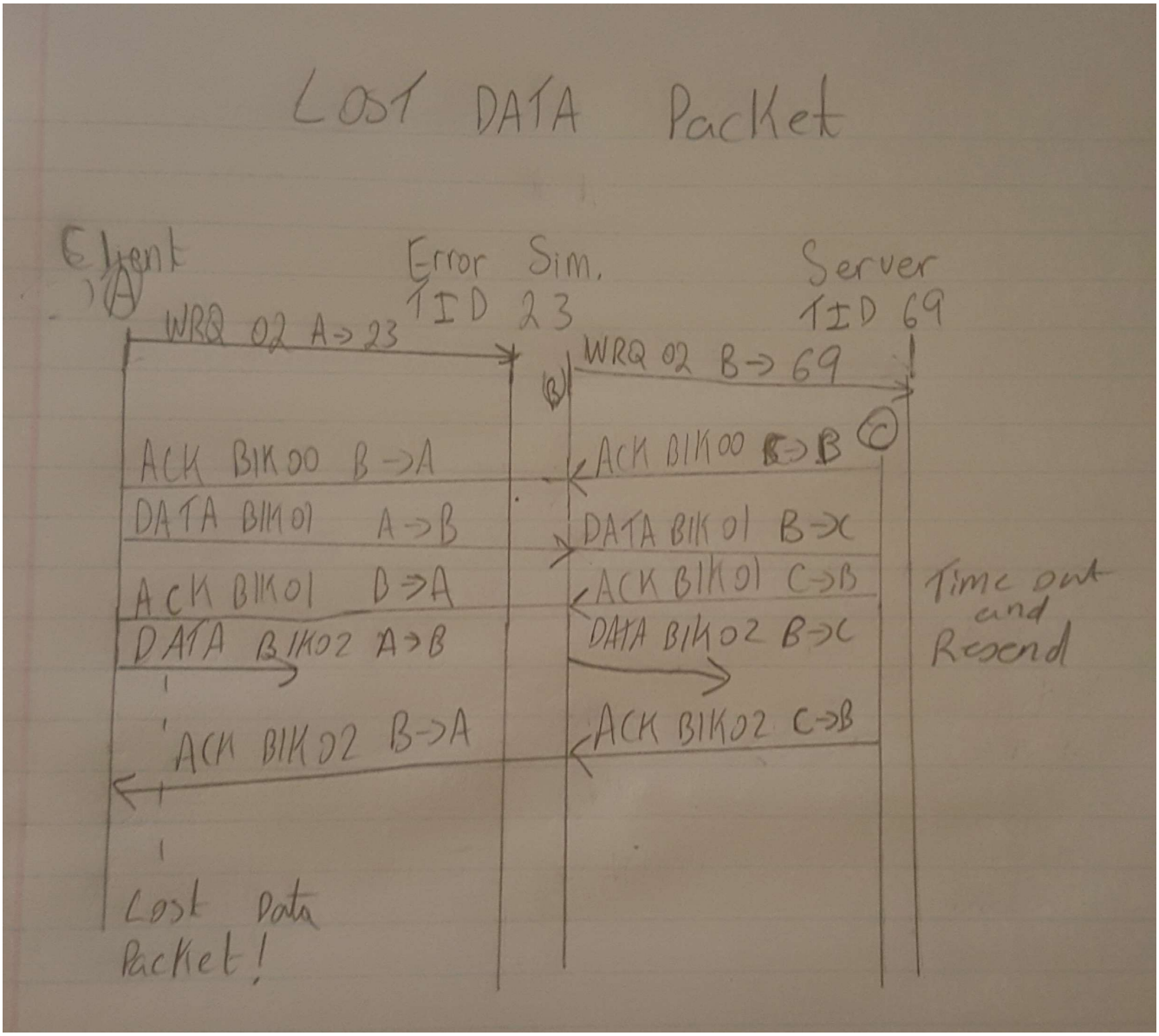


Diagrams for Iteration 4:

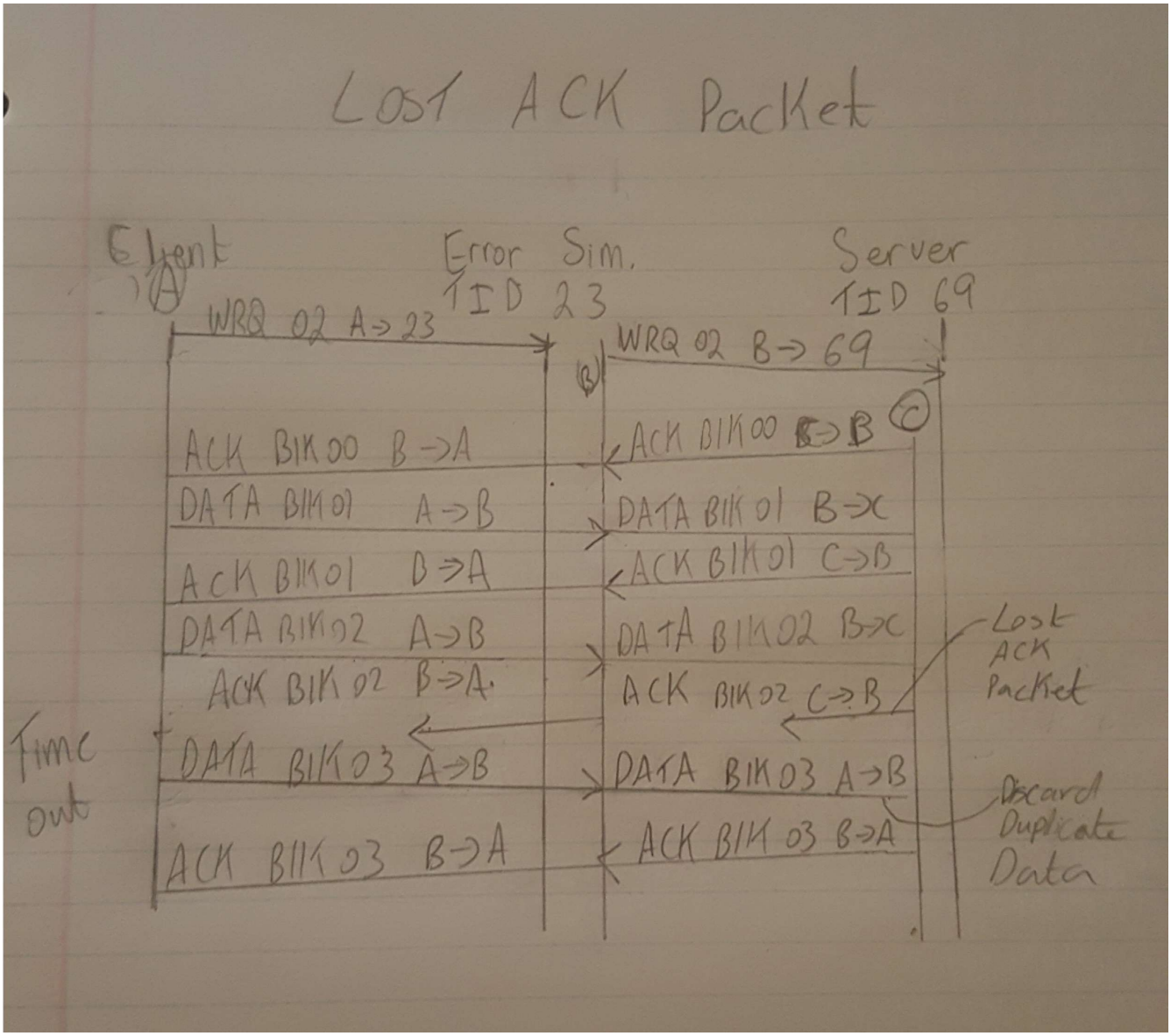
UML class:



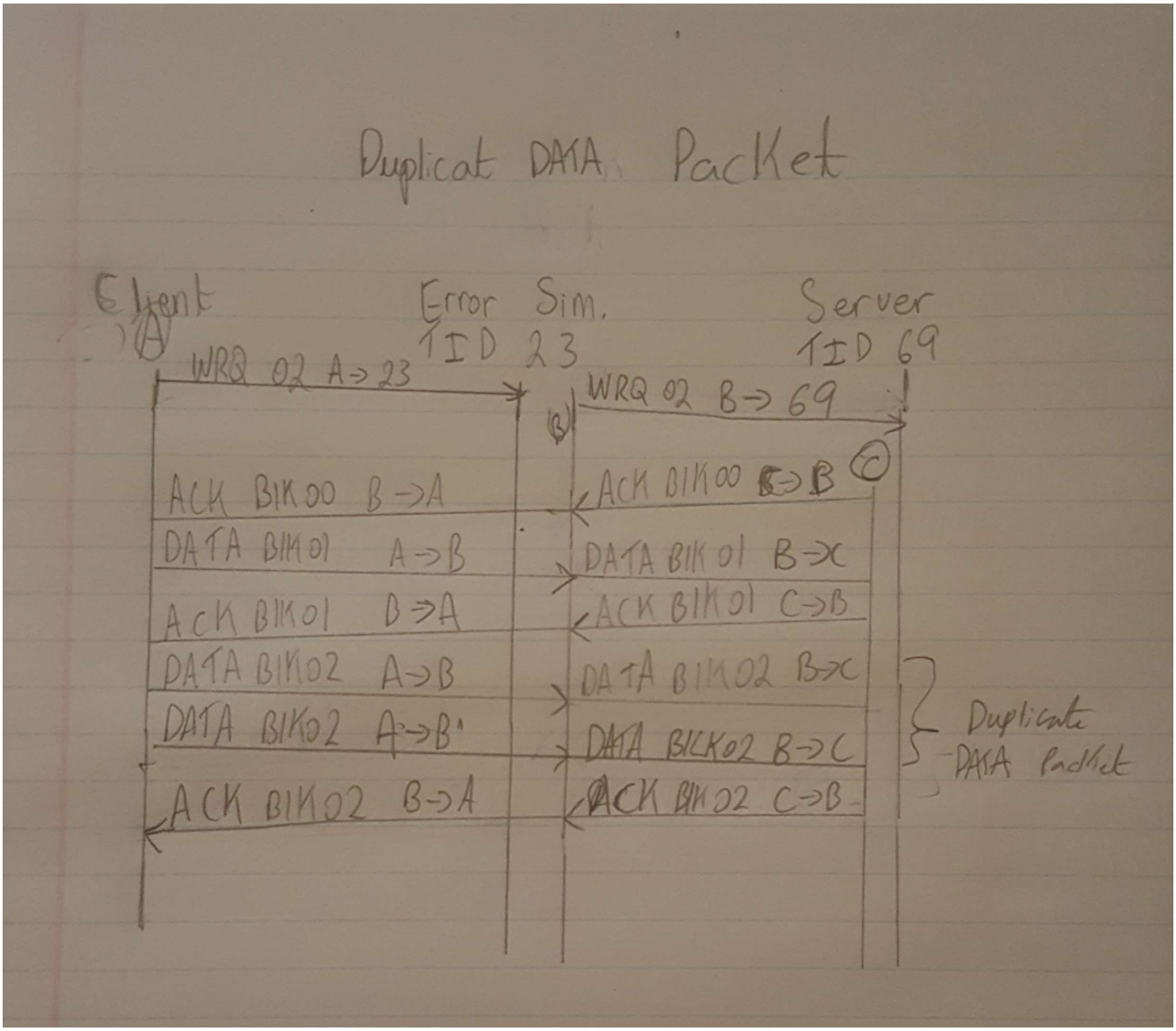
Timing diagram 1:



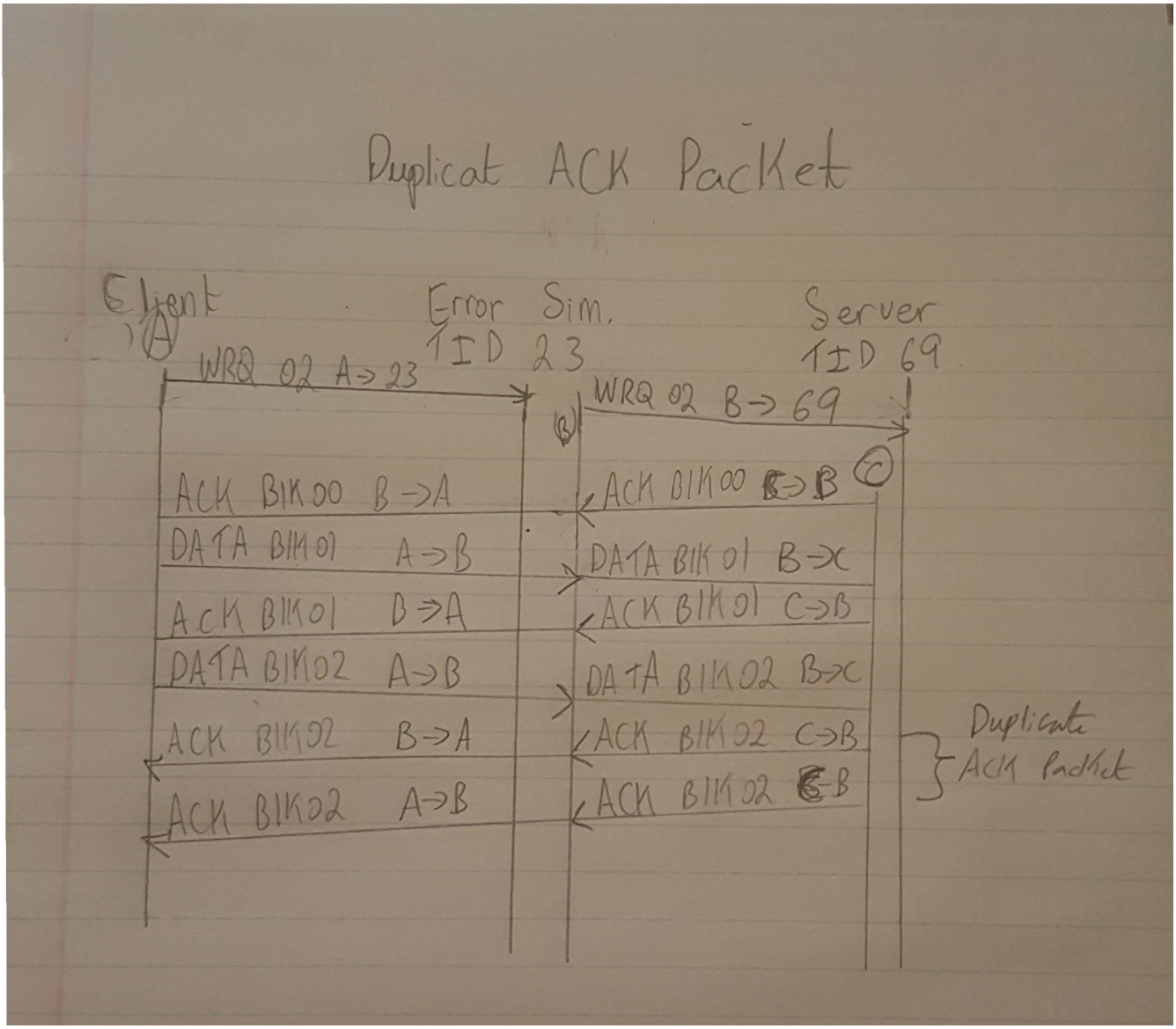
Timing diagram 2:



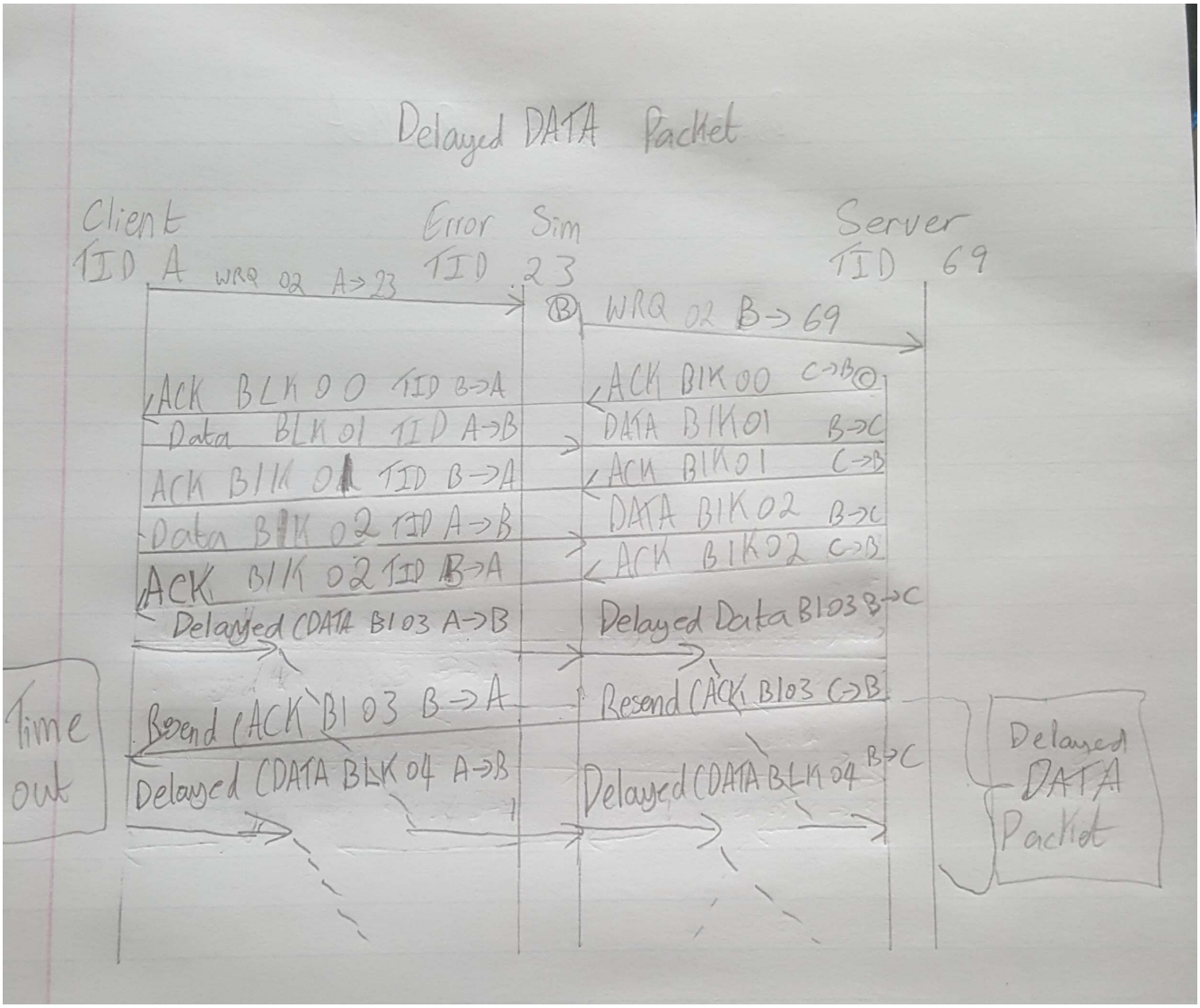
Timing diagram 3:



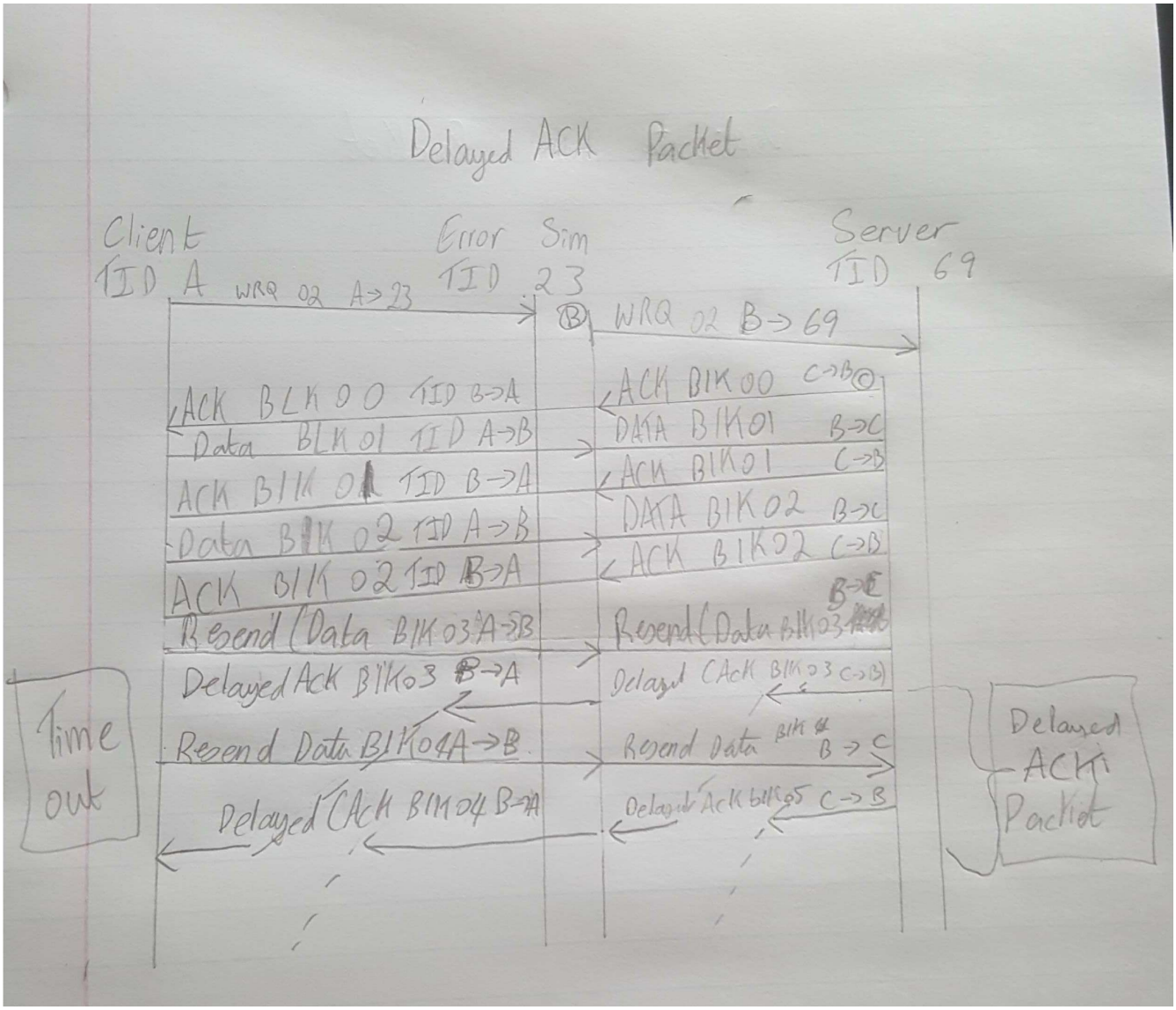
Timing diagram 4:



Timing diagram 5:



Timing diagram 6:



Diagrams for Iteration 5:

UML class: