

Term project

SYSC 3303



|  |  |
| --- | --- |
| **Philip Klostermann** | 100983723 |
| **Rui LI** | 100987686 |
| **Yanzhe Zhang** | 100945940 |
| **Amr Gawish** | 100897952 |
| **Loktin Wong** | 100939428 |

April 2, 2016

Carleton university

**TABLE OF CONTENT**

1. **TEAM WORK BREAKDOWN**
   1. Iteration #1
   2. Iteration #2
   3. Iteration #3
   4. Iteration #4
   5. Iteration #5
2. **Diagrams**
   1. Use Case Maps UCM
   2. Timing Diagrams
   3. UML Class Diagrams

1. **Setup & Testing Instructions**
   1. Files
   2. Launching the application
   3. Select a Configuration for Client/Server/Intermediate
   4. Configure the Intermediate Host
   5. Transfer File locations
   6. Error Scenarios
   7. Design Decisions

**1.0) TEAM WORK BREAKDOWN**

1. **Iteration #1**

Amr Gawish: Planning, Diagrams

Loktin Wong: Planning, Client, Intermediate, Server code

Yanzhe Zhang: Planning, Client code, class diagrams

Rui Li: Planning, Client code, class diagrams

Philip Klostermann: Planning, Client code, Server protocol code Class Diagram

1. **Iteration #2**

Amr Gawish: Planning, Timing Diagrams, Client menu

Loktin Wong: Planning, Client error checking & handling

Yanzhe Zhang: Planning, Intermediate Host Menu, Error scenario testing, Class & Timing Diagrams

Rui Li: Planning, Format debugging output

Philip Klostermann: Planning, Server error checking & handling, Intermediate Host error simulation, Class & Timing Diagrams

1. **Iteration #3**

Amr Gawish: Client error / resending

Loktin Wong: Client and server error / resending

Yanzhe Zhang: All the Diagrams

Rui Li: Client and Server error / resending

Philip Klostermann: Intermediate Host, Client Server debugging

1. **Iteration #4**

Amr Gawish: Discussions and Brainstorming, Testing

Loktin Wong: Configuration, Configuration Menu, Error Handling client/server, Testing & Debugging

Yanzhe Zhang: Logger class, Integration Testing, Debugging

Rui Li: Timing Diagrams for IO errors, Discussions and Brainstorming

Philip Klostermann: Server and Client rewrite, Server/Client IO exception handling, debugging, testing

1. **Iteration #5**

Amr Gawish Testing, Documents

Loktin Wong Update client IP modification

Yanzhe Zhang implementing file locking, Testing

Rui Li Intermediate host logging, Testing

Philip Klostermann Updating Intermediate host to remote IP’s, Testing

**2.0) DIAGRAMS**

**2.1) Use Case Map (UCM)**

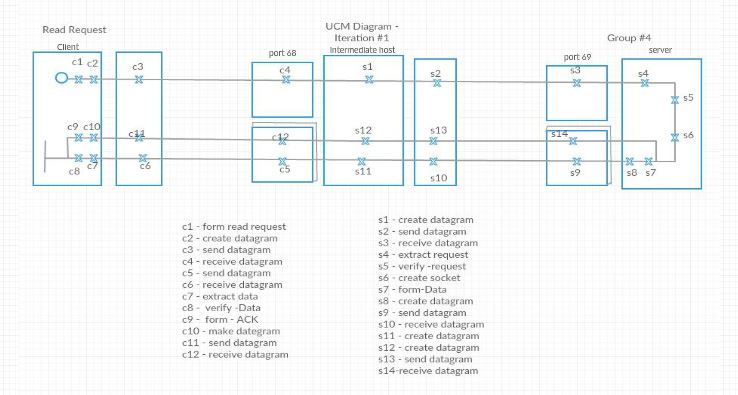


Figure : Read Request UCM

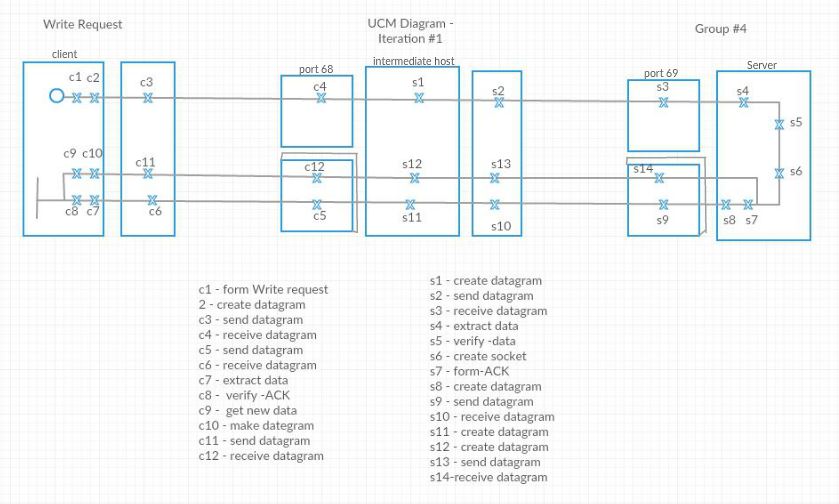


Figure : Write Request UCM

**2.2) Timing Diagrams**

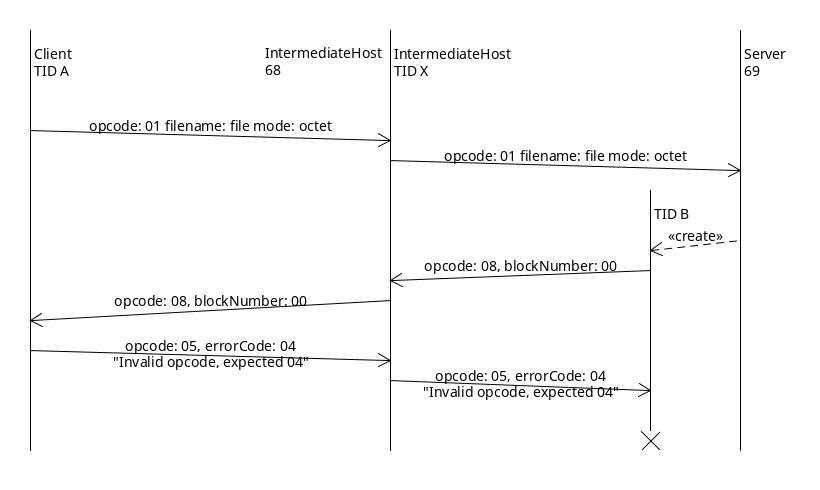


Figure : Invalid ACK – RRQ

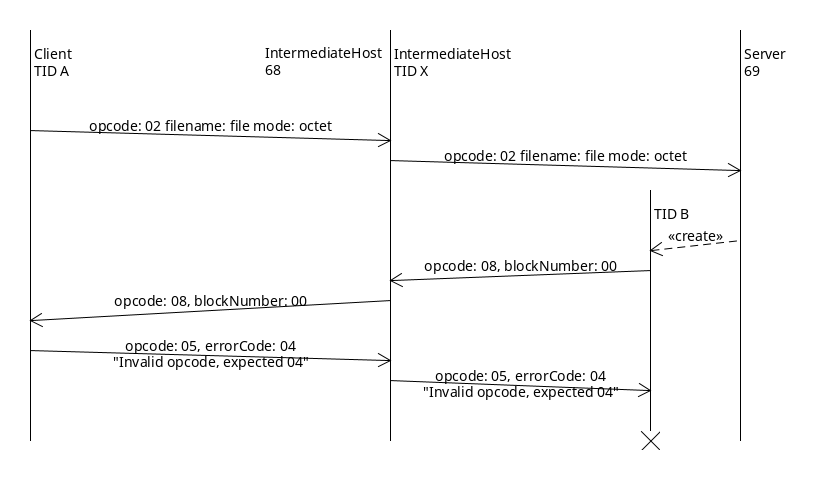


Figure : Invalid ACK – WRQ

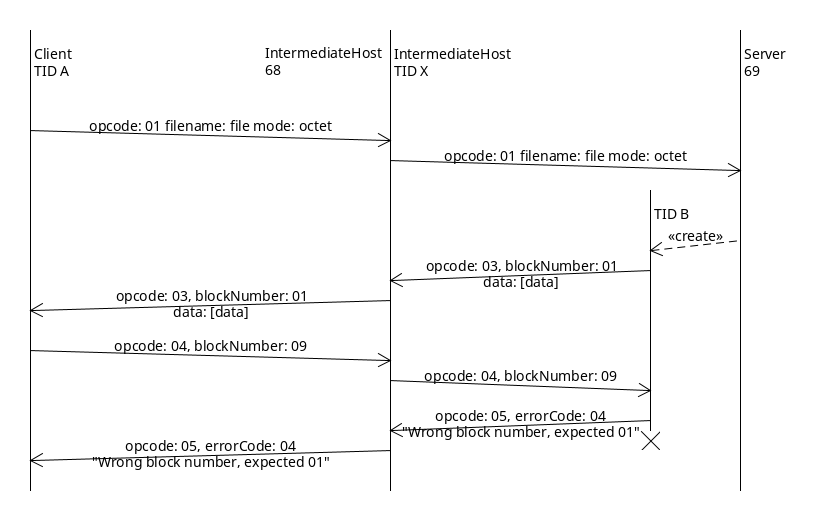


Figure : Invalid Block – ACK

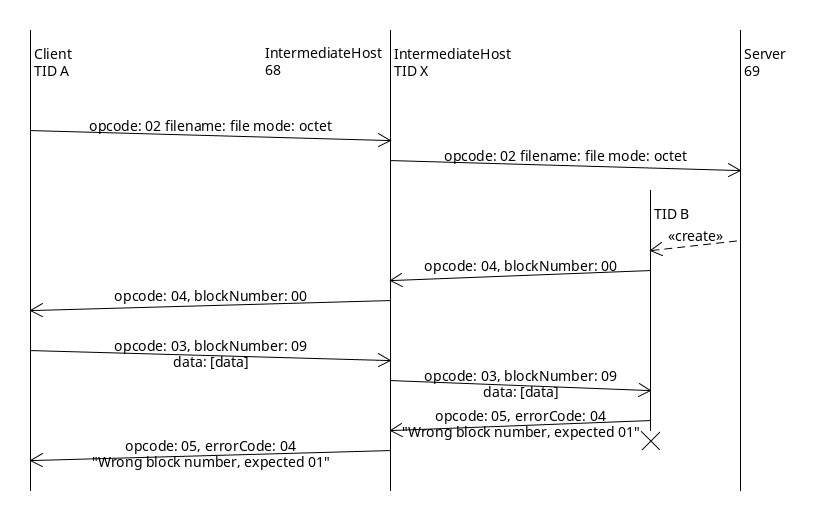


Figure : Invalid Block – DATA

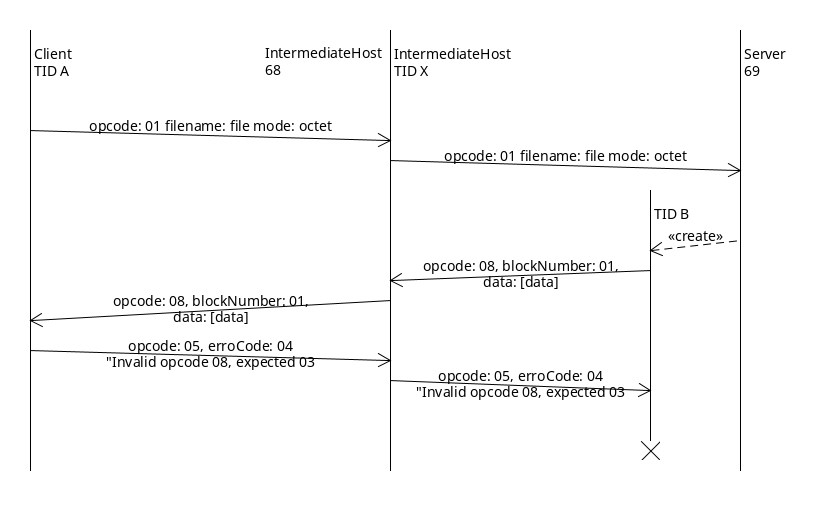


Figure : Invalid Data Packet – RRQ

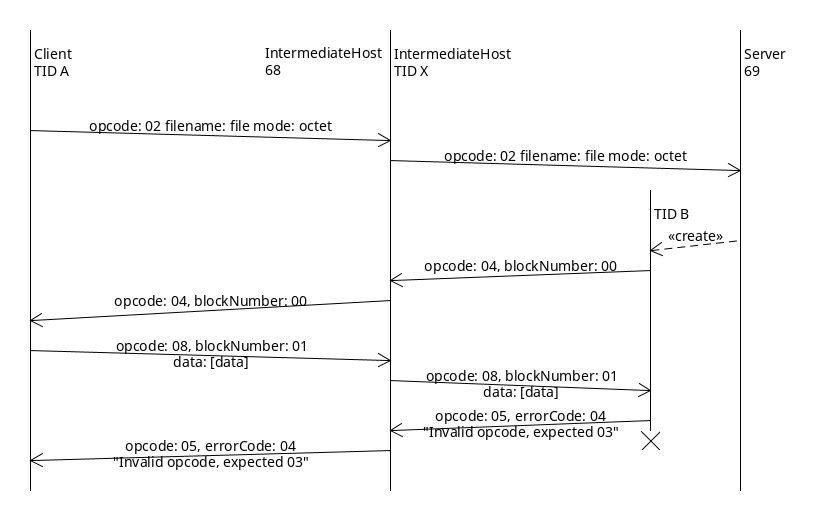


Figure : Invalid Data Packet – WRQ

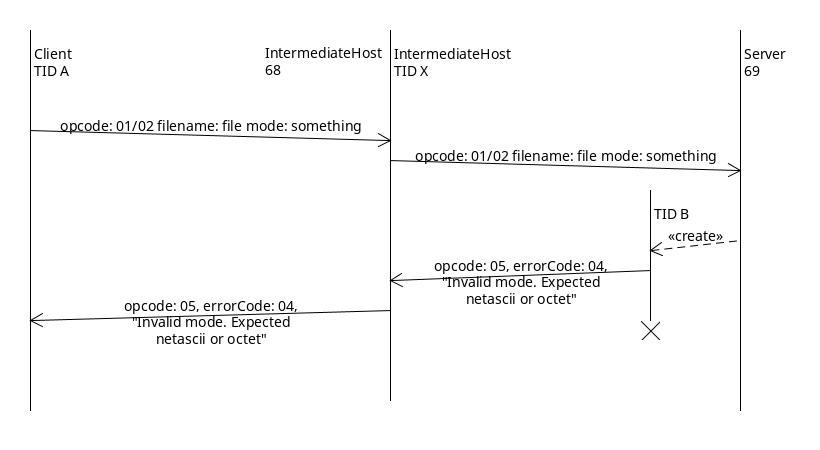


Figure : Invalid Request Mode

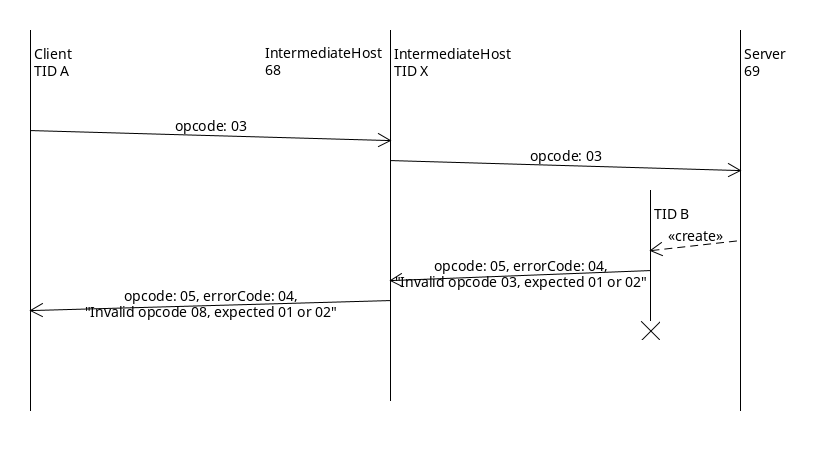


Figure : Timing Invalid Opcode

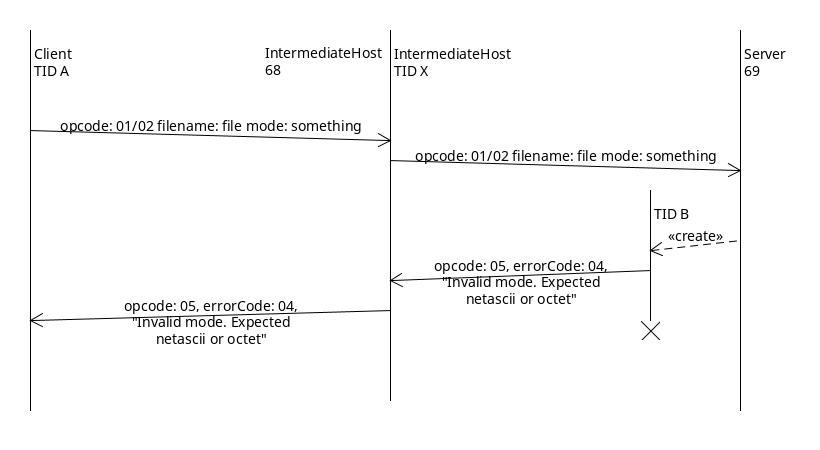


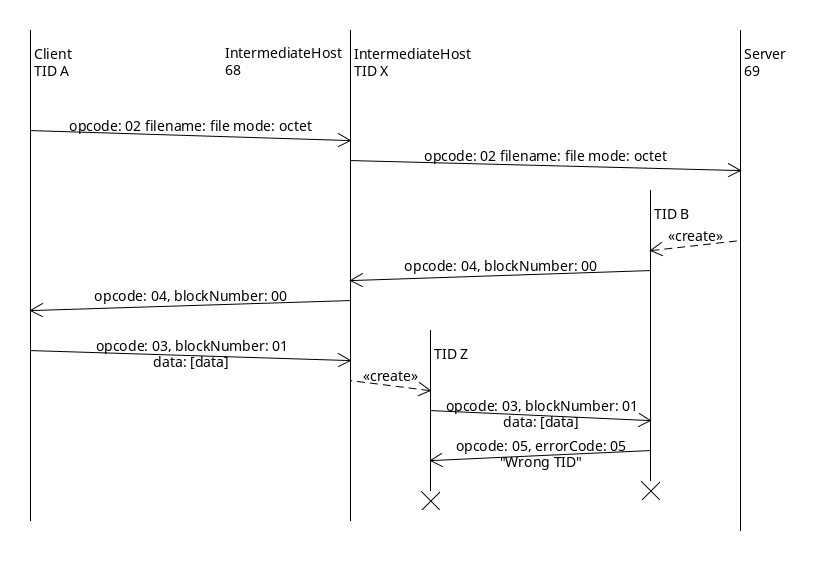
Figure : Timing Invalid Request Mode

Figure : Wrong TID

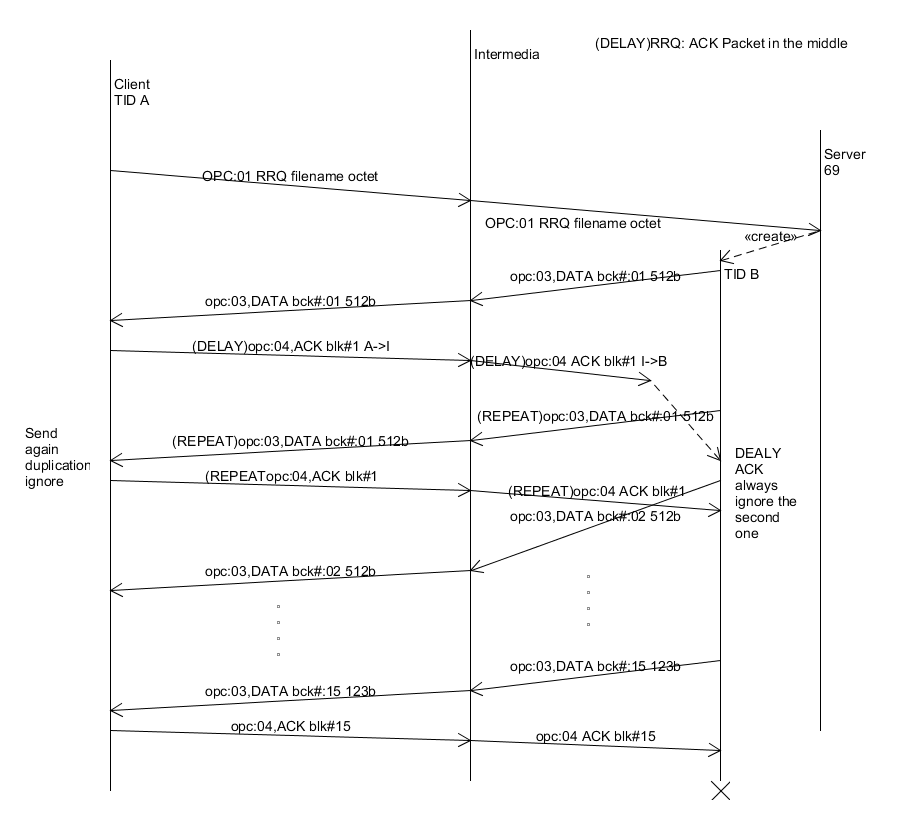


Figure : DEALY ACK Packet in the middle –RRQ

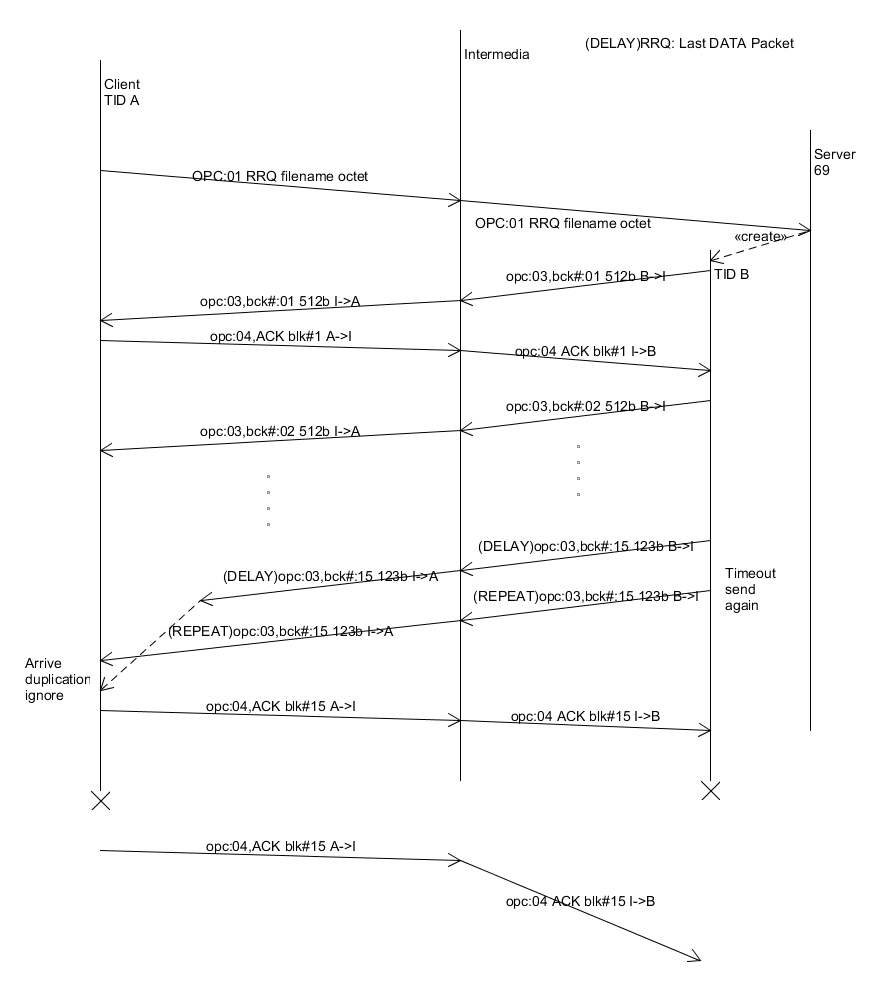


Figure : DELAY Last DATA Packet—RRQ

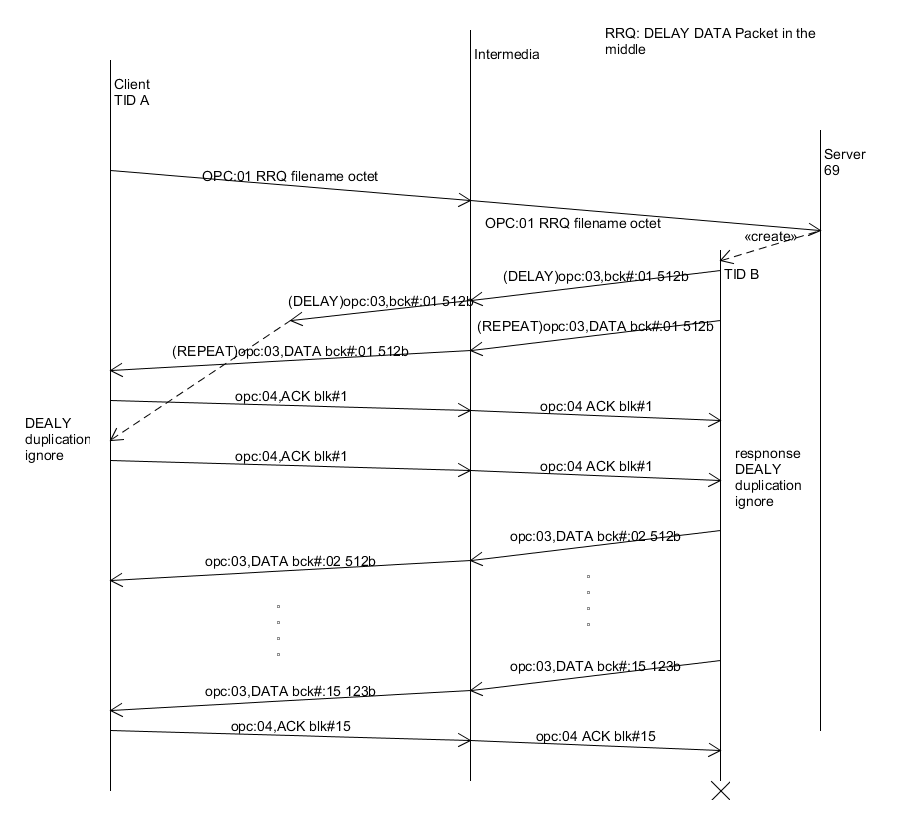


Figure : DELAY DATA Packet in the middle—RRQ

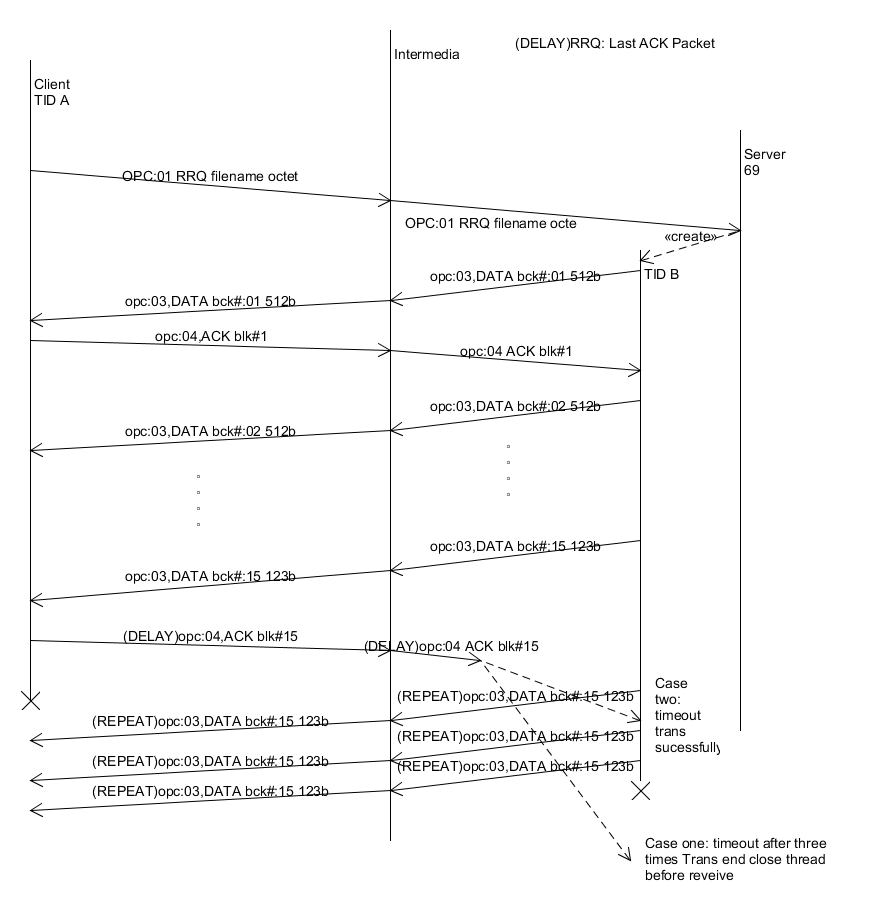


Figure : DELAY Last ACK Packet—RRQ

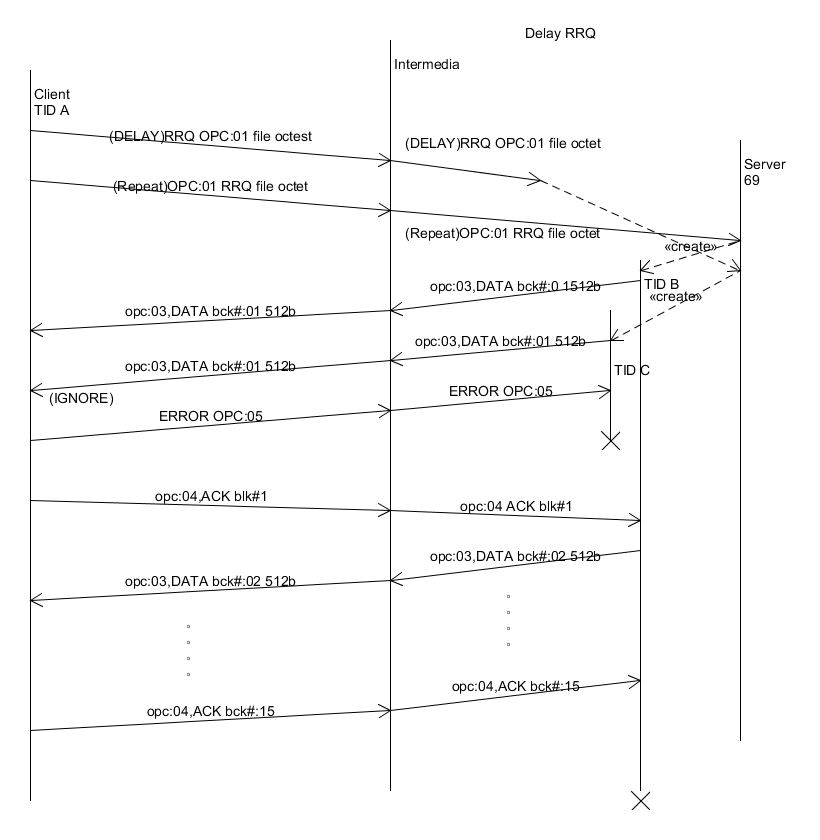


Figure : DELAY RRQ

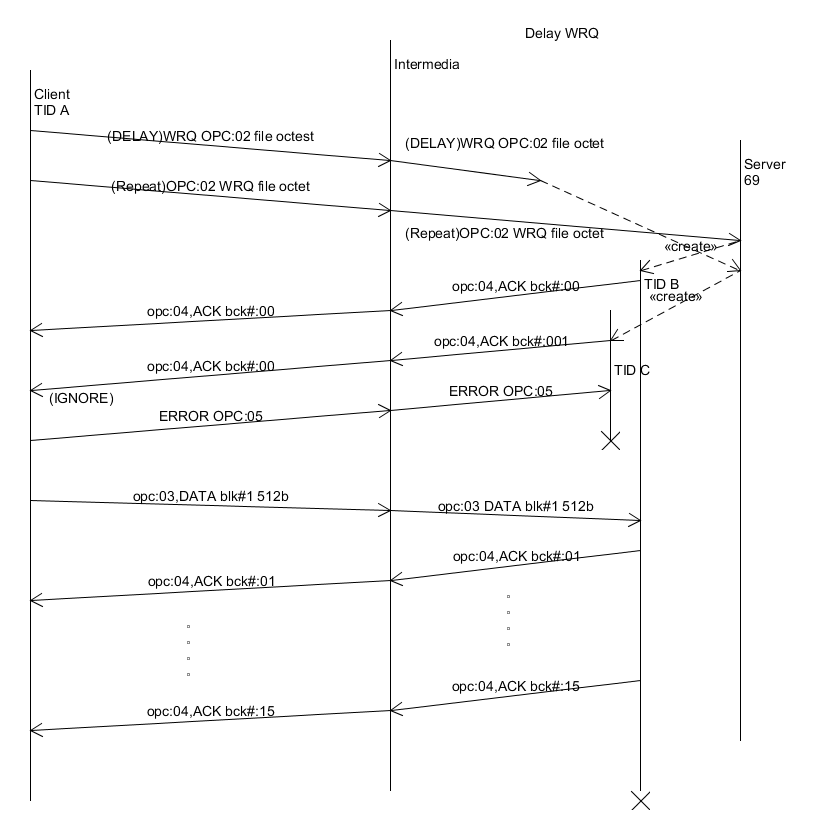


Figure : DeELAY WRQ –WRQ

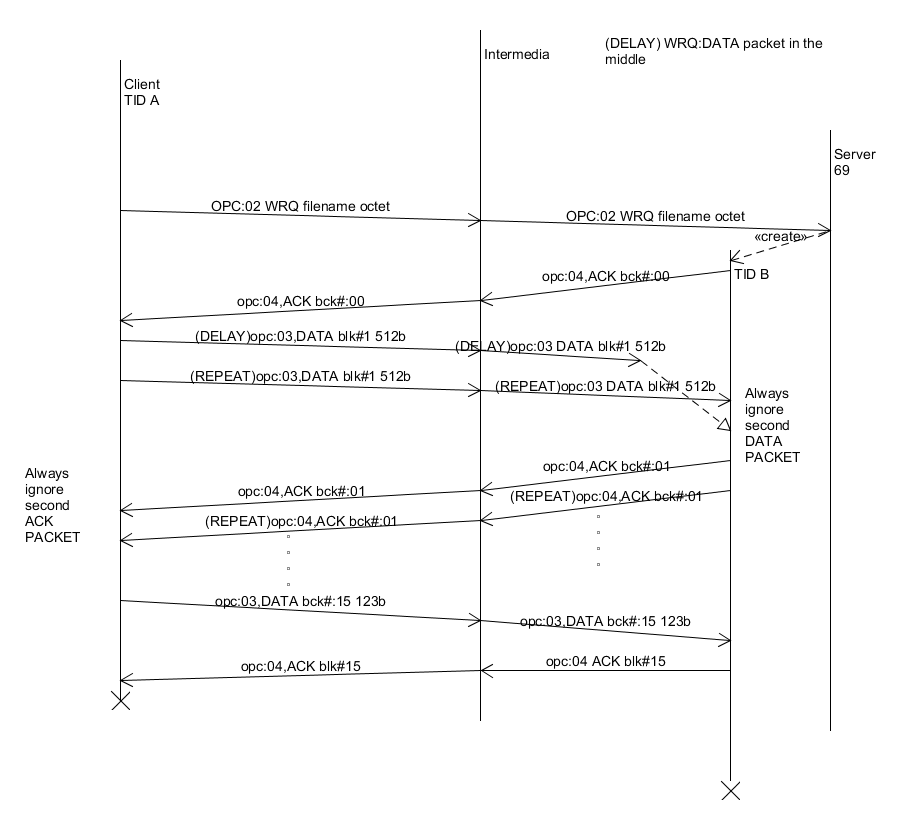
.

Figure : DELAY DATA in the middle –WRQ

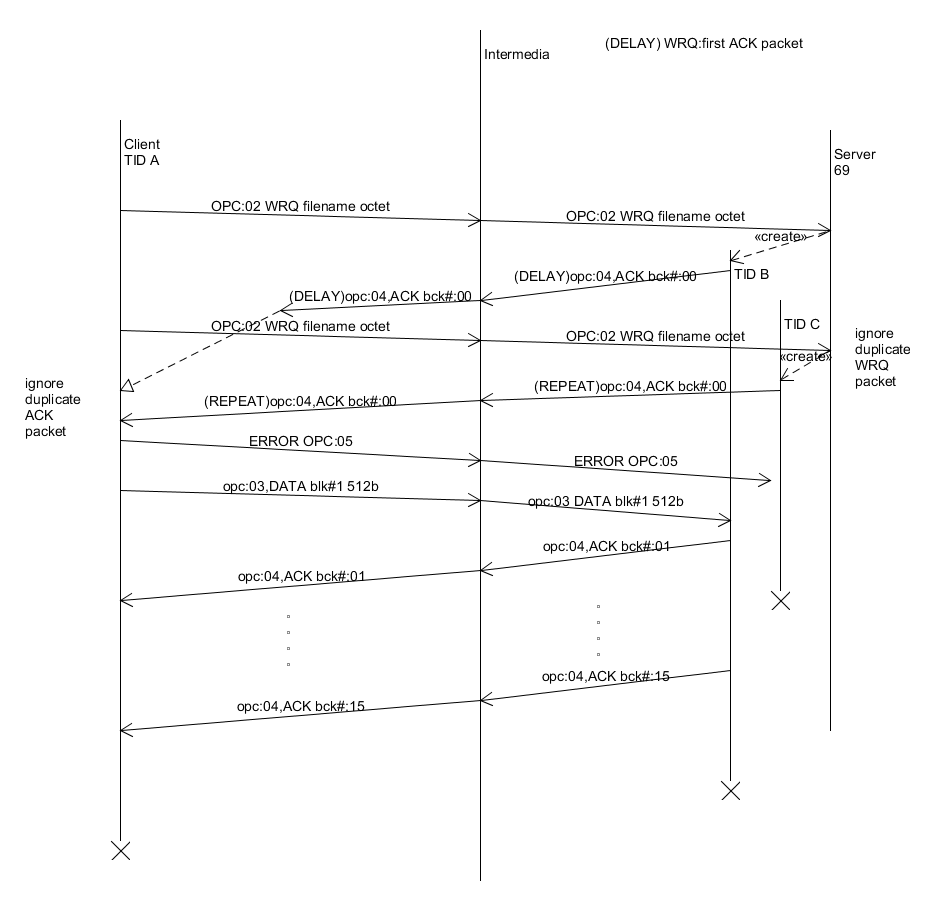


Figure : DELAY first ACK packet –WRQ

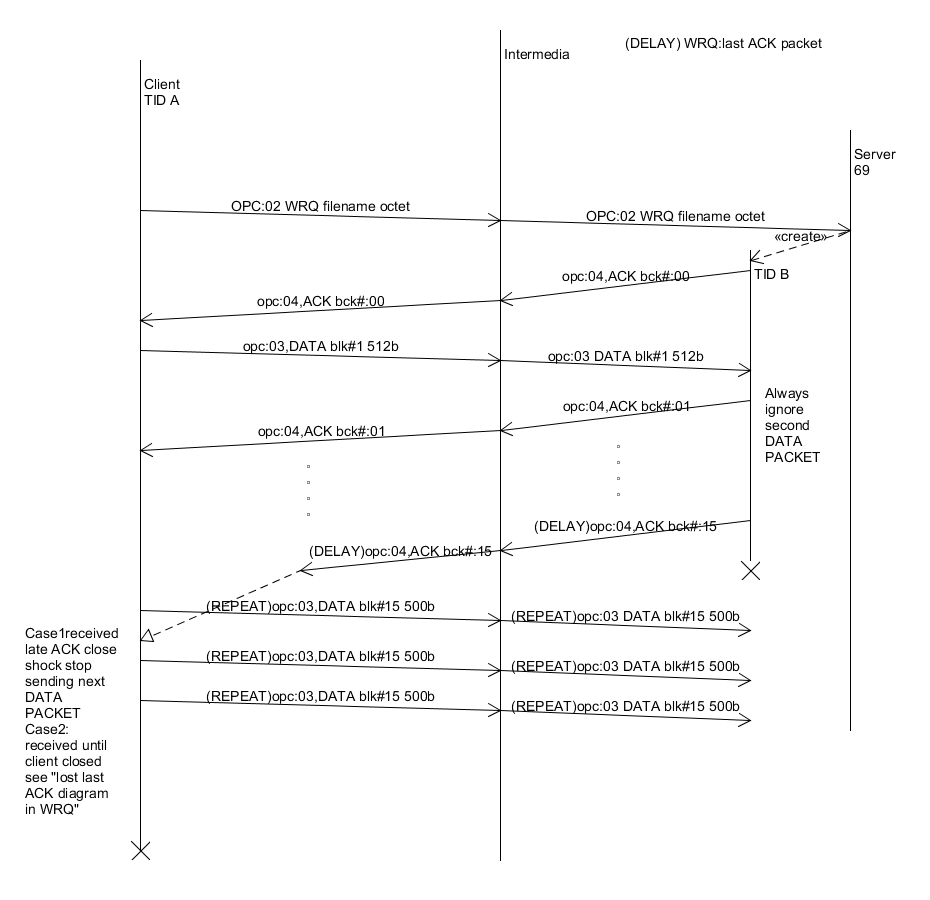


Figure : DELAY first ACK—WRQ

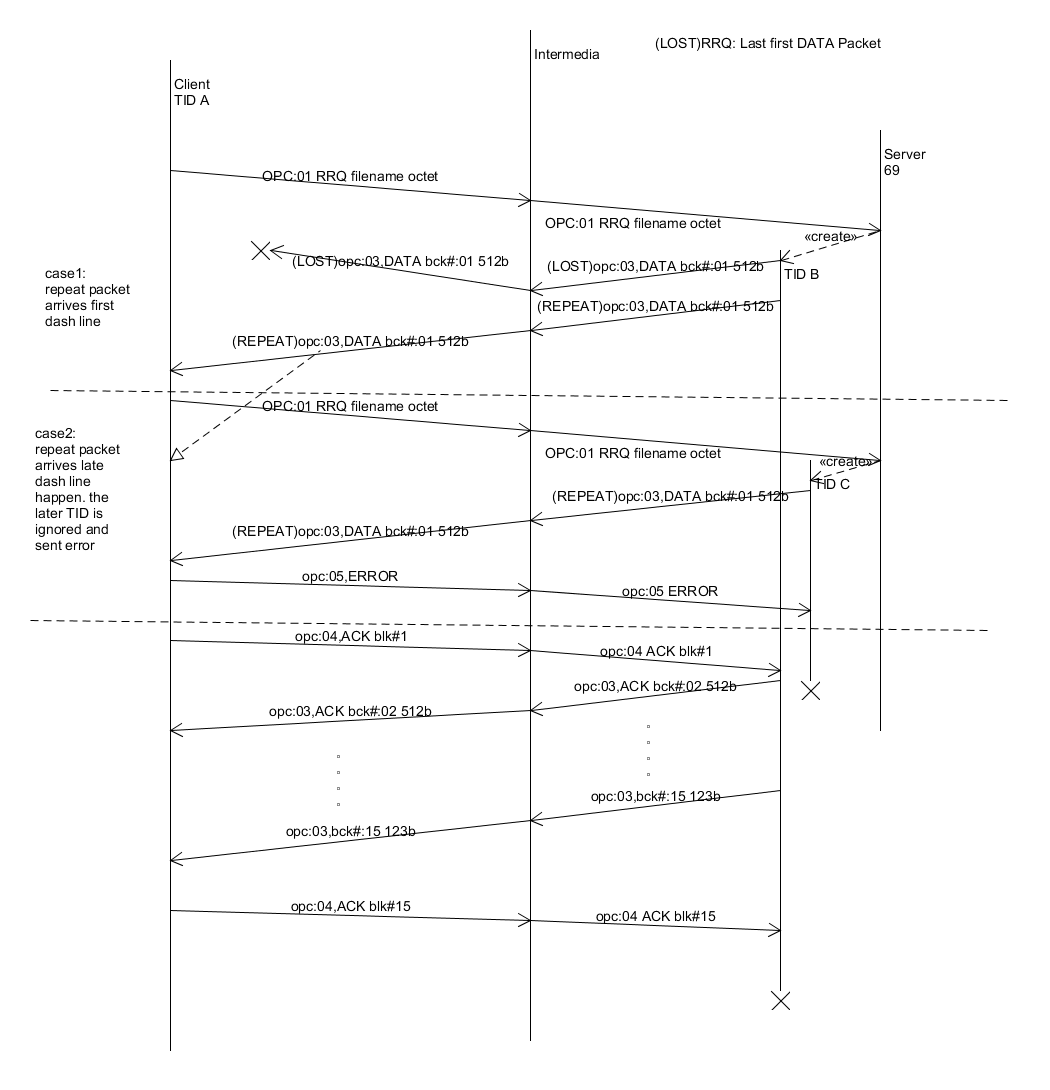


Figure : LOST first DATA packet –RRQ

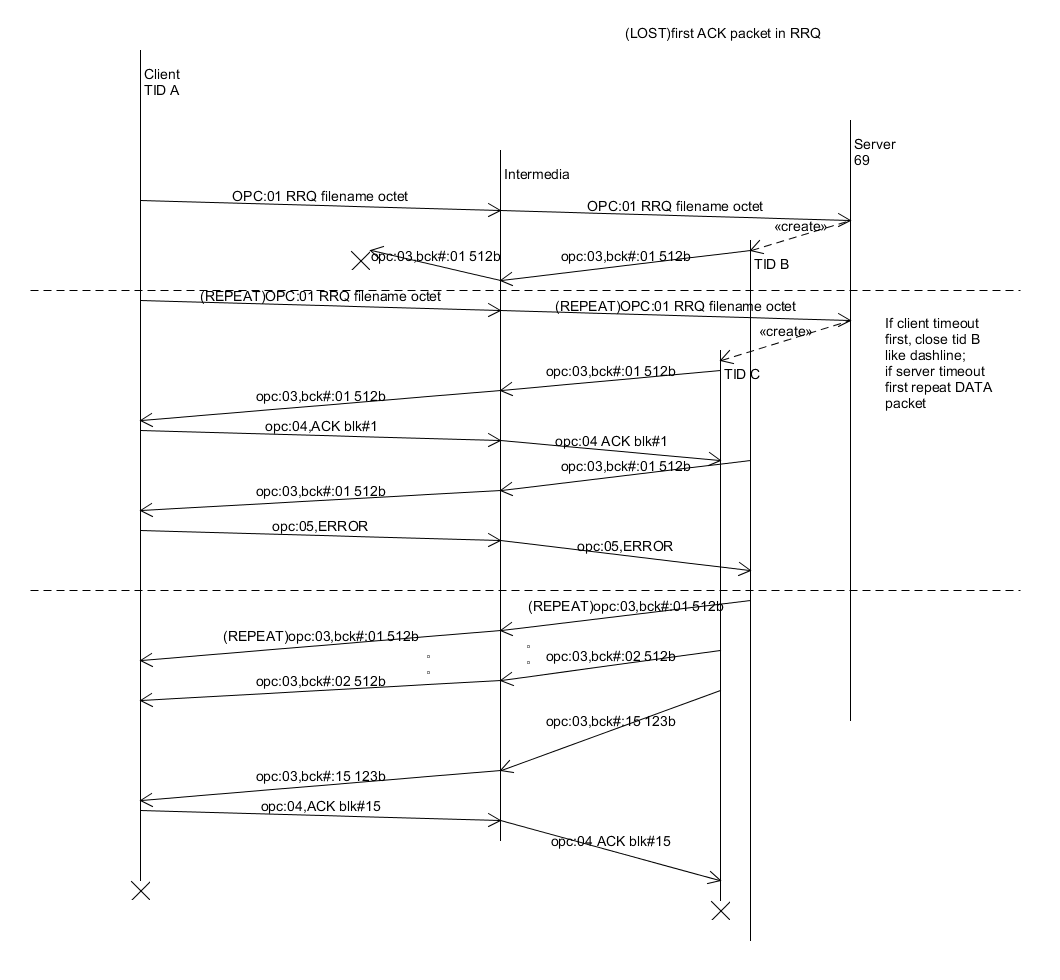


Figure : LOST fist ACK—RRQ

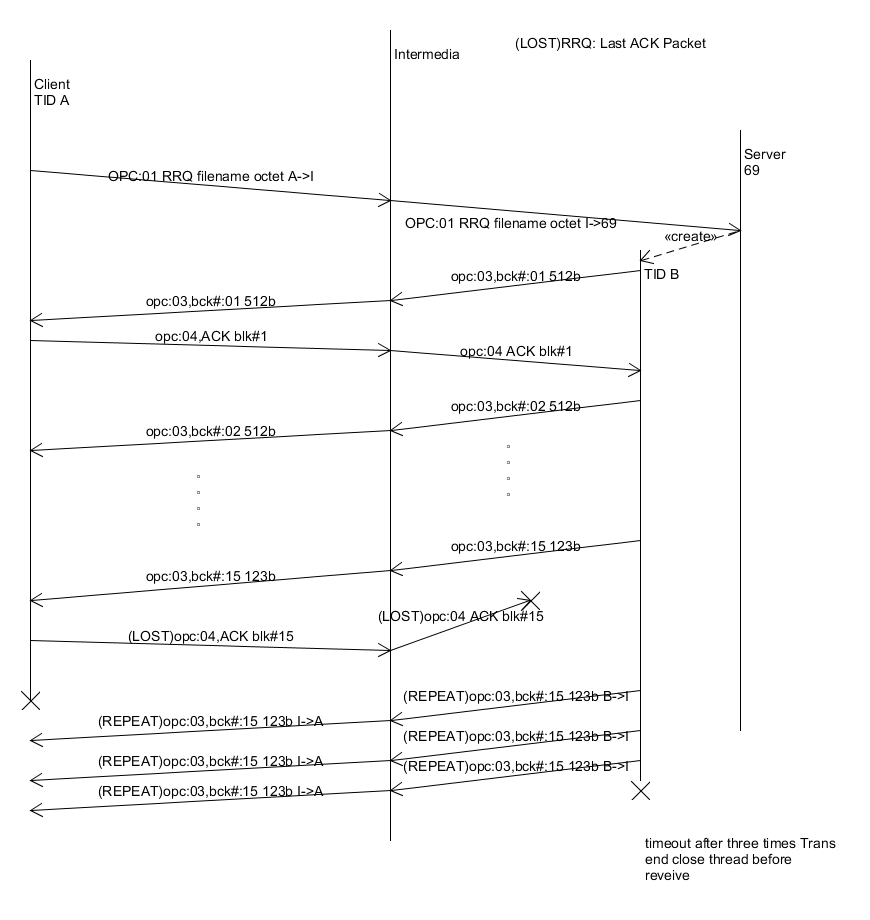


Figure : LOST last ACK packet—RRQ

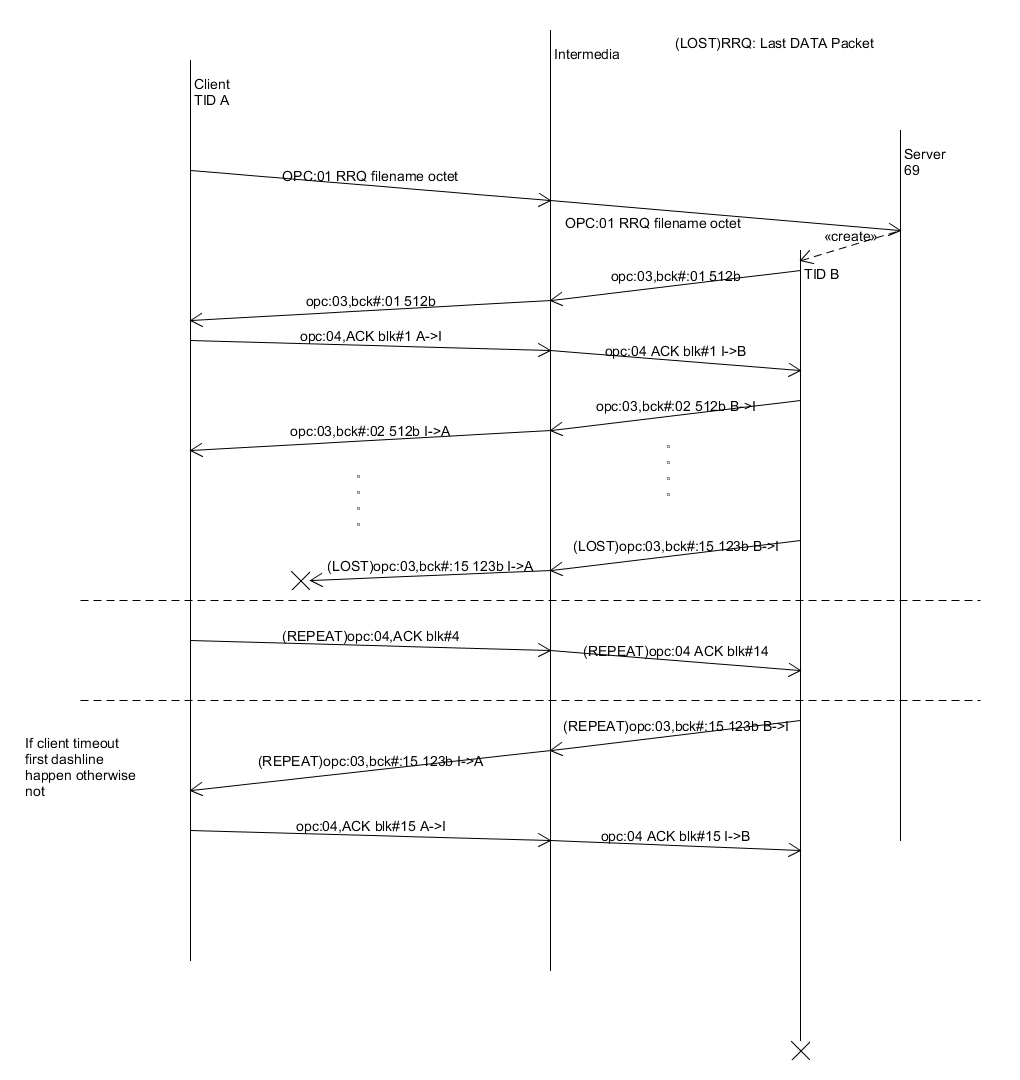


Figure : LOST last DATA packet –RRQ

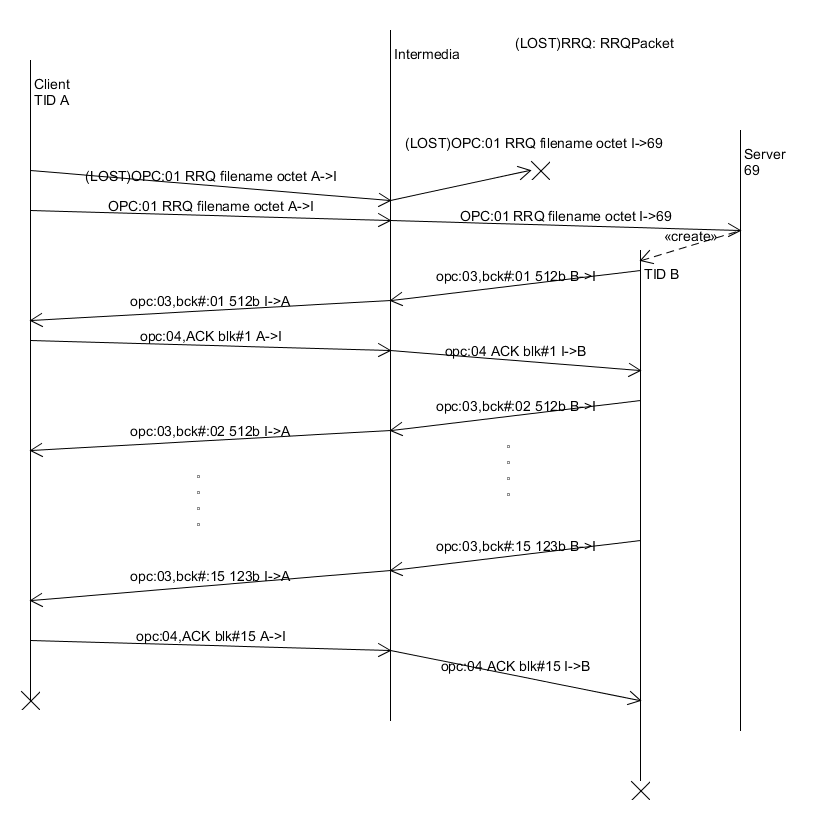


Figure :LOST RRQ

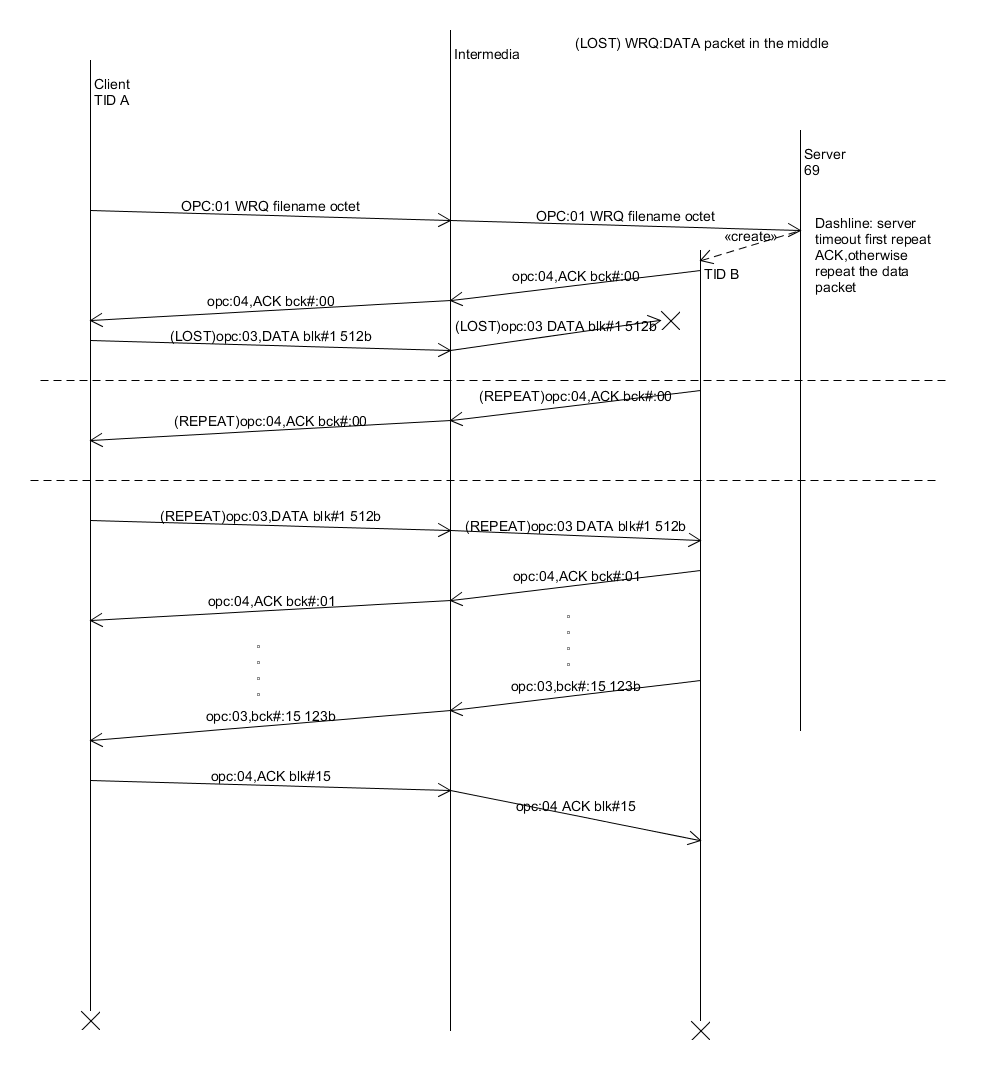


Figure : LOST DATA packet in the middle –WRQ

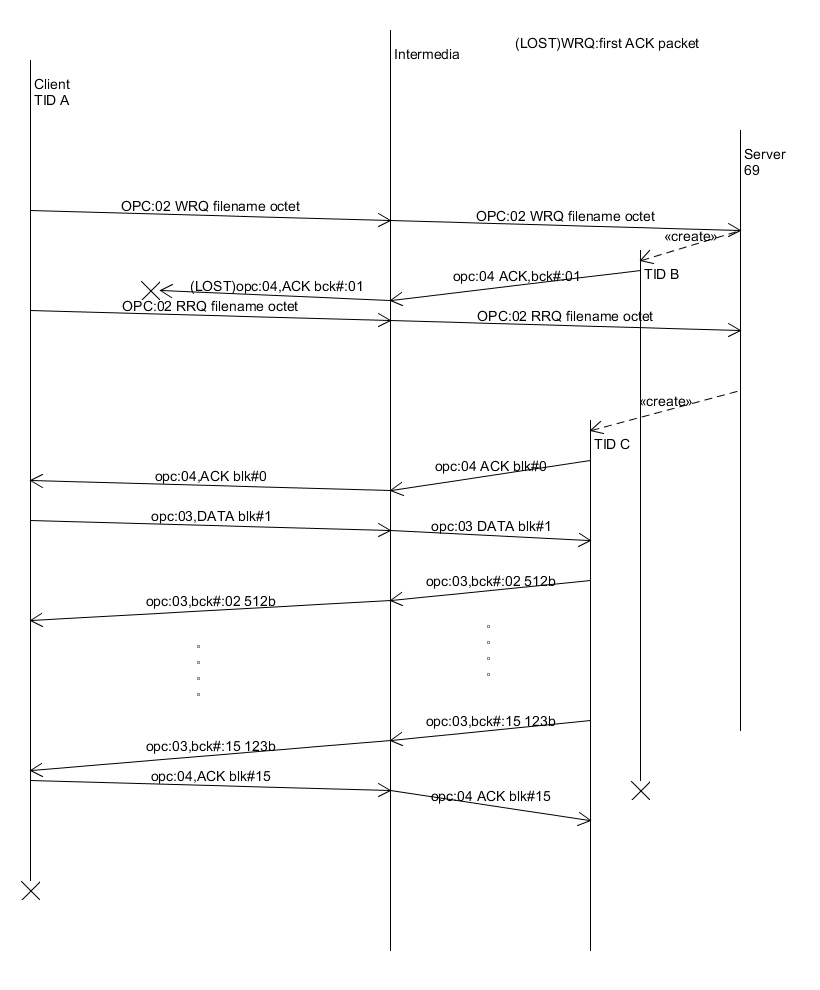


Figure : LOST first ACK—WRQ

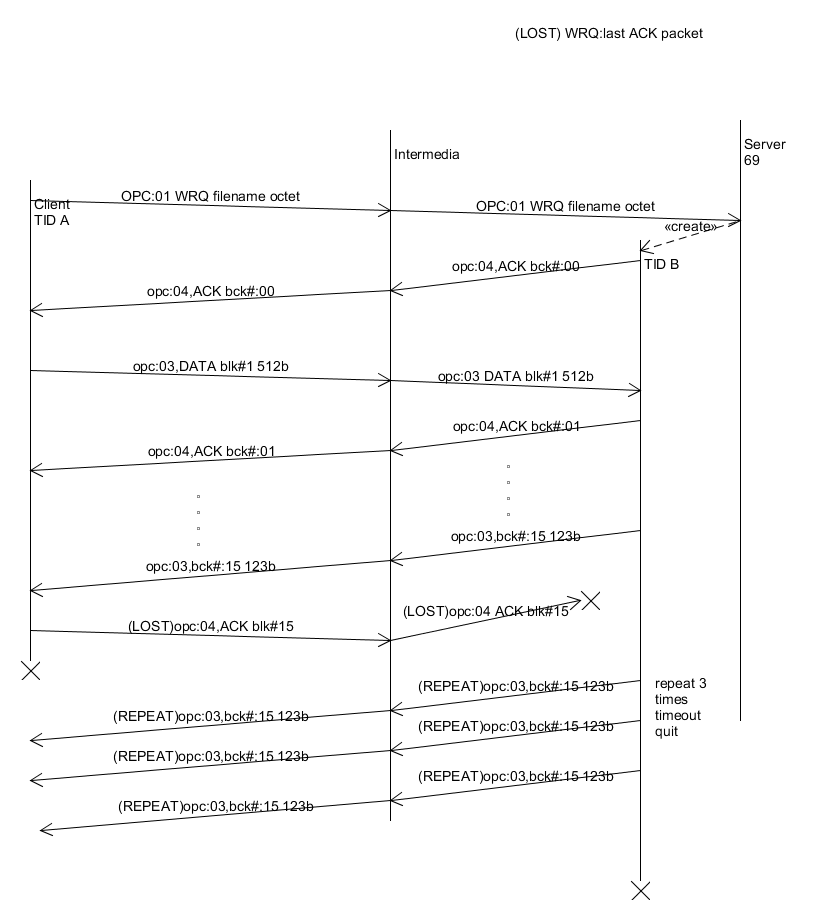


Figure : LOST last ACK—WRQ

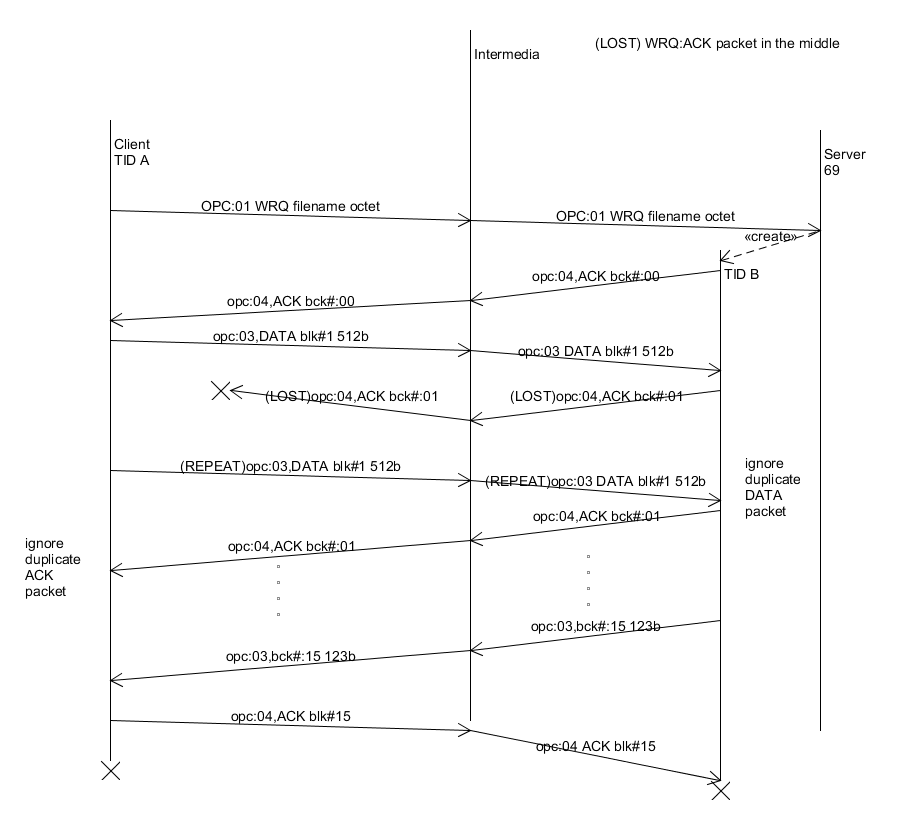


Figure : LOST last DATA packet –RRQ

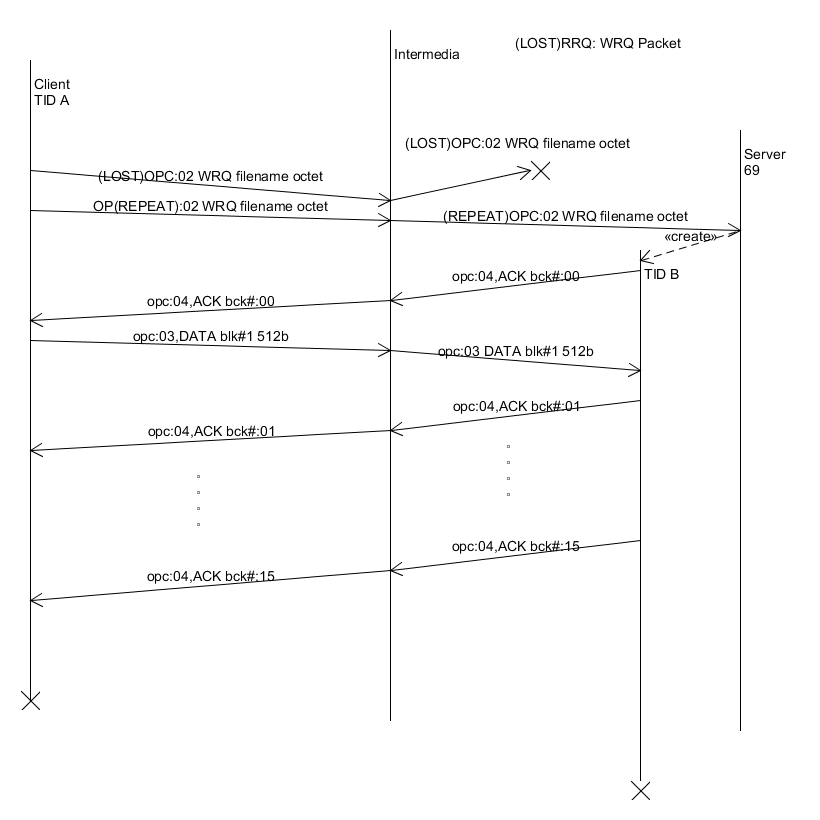


Figure : LOST WRQ –WRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\accessViolation_RRQ.png

Figure : access Violation \_ RRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\accessViolation_WRQ.png

Figure : access Violation \_ WRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\diskFullOrAllocationExceed_RRQ.png

Figure : disk Full Or Allocation Exceed \_ RRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\diskFullOrAllocationExceed_WRQ.png

Figure :disk Full Or Allocation Exceed \_ WRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\fileAlreadyExist_WRQ.png

Figure : file Already Exis t\_ WRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\fileNotFound_RRQ.png

Figure : file Not Found \_ RRQ

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\fileNotFound_RRQ2.png

Figure : file not found \_ RRQ2

C:\Users\PhilipKlostermann\Desktop\iteration05\SYSC3303\Iteration05\Diagrams\fileIO_TimingDiagrams\fileNotFound_WRQ.png

Figure : file Not Found \_ WRQ

**2.3 UML Class Diagram**

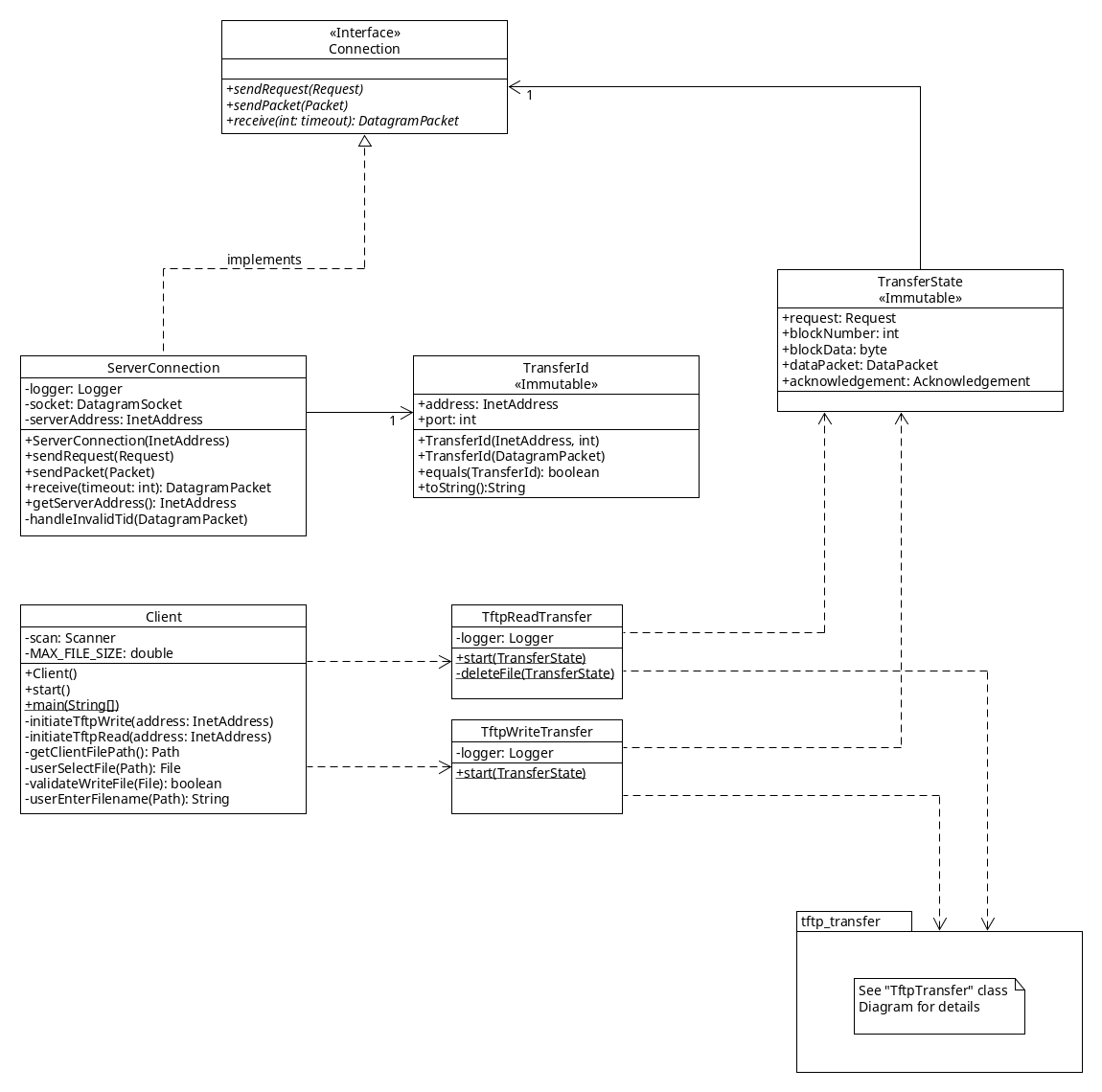


Figure : Client

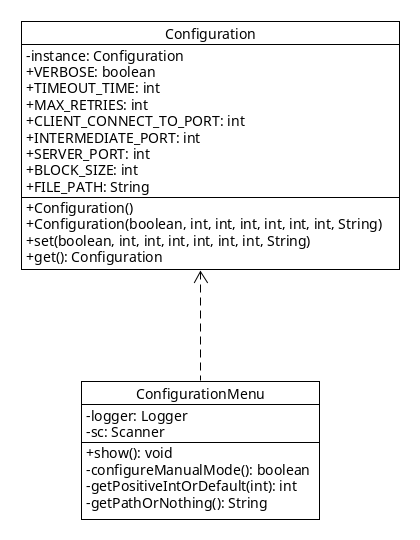


Figure : Configuration

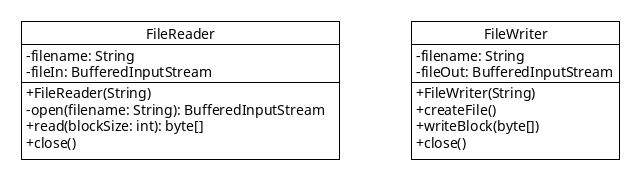


Figure : FileIO

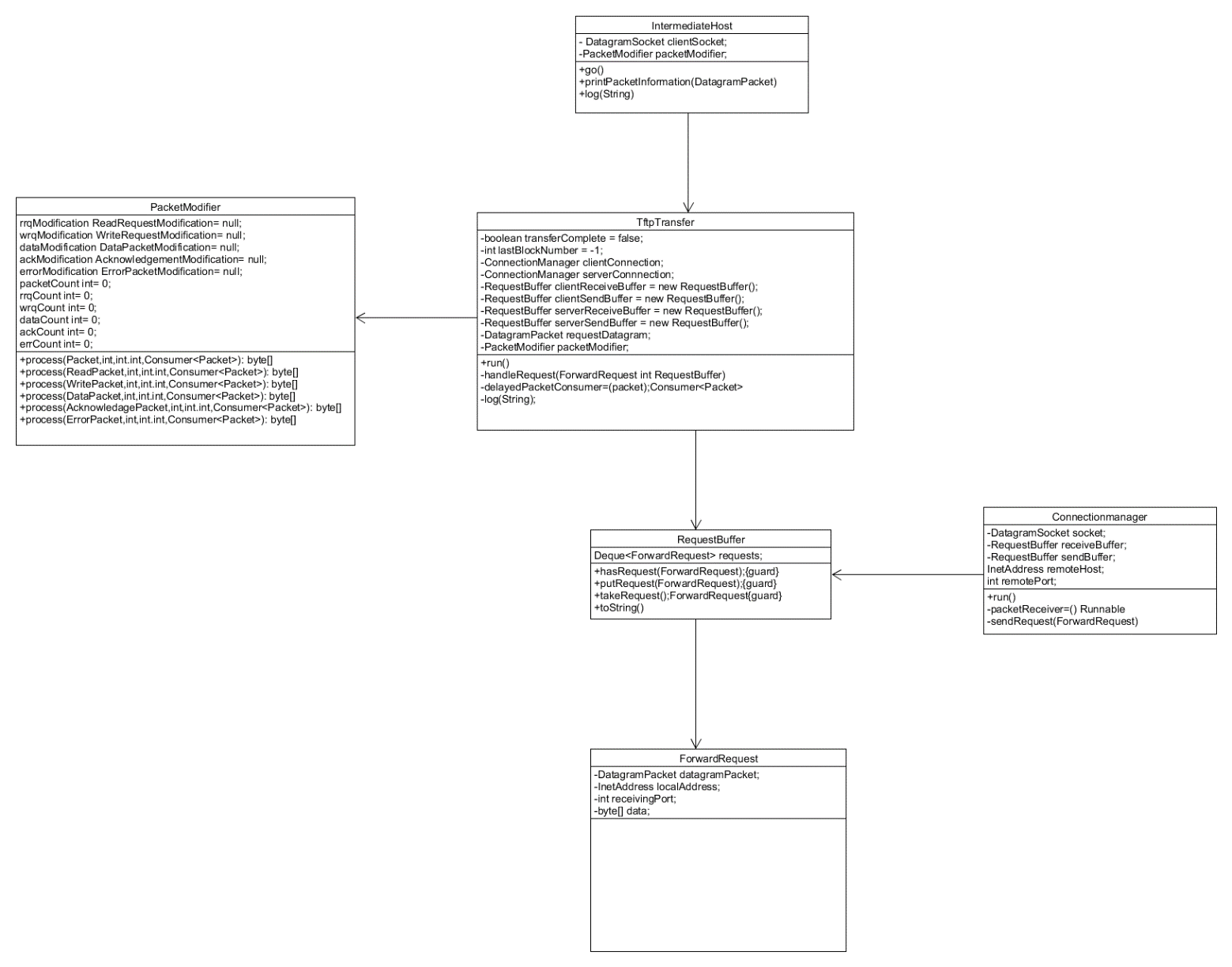


Figure : Intermediate Host

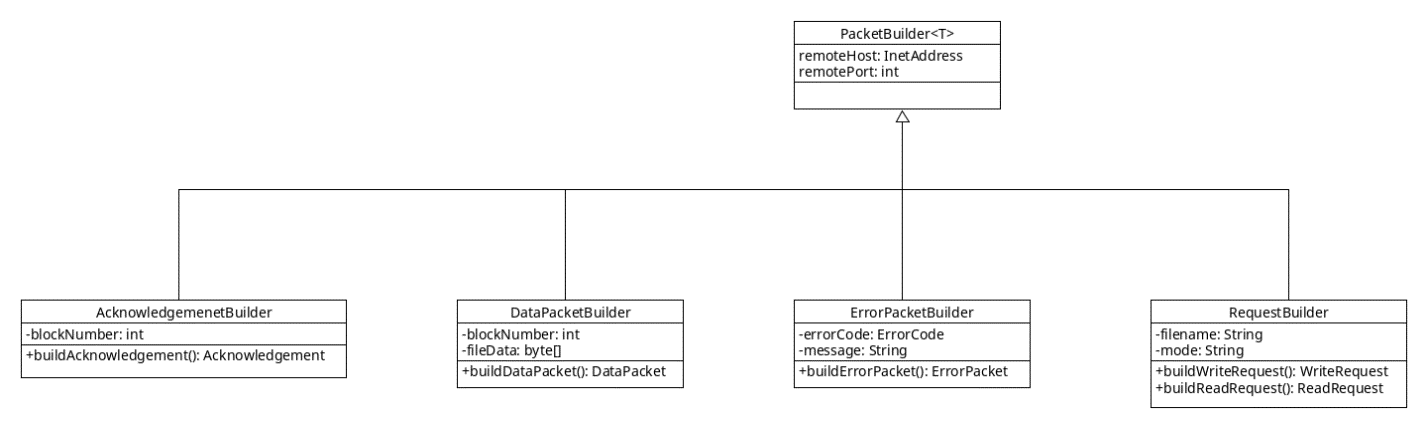


Figure : Packet Builder

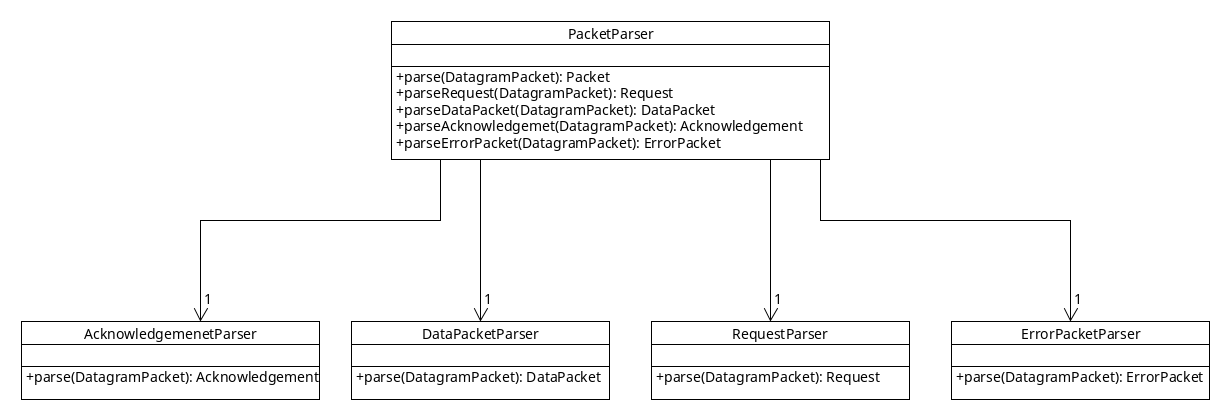


Figure : Packet Parser

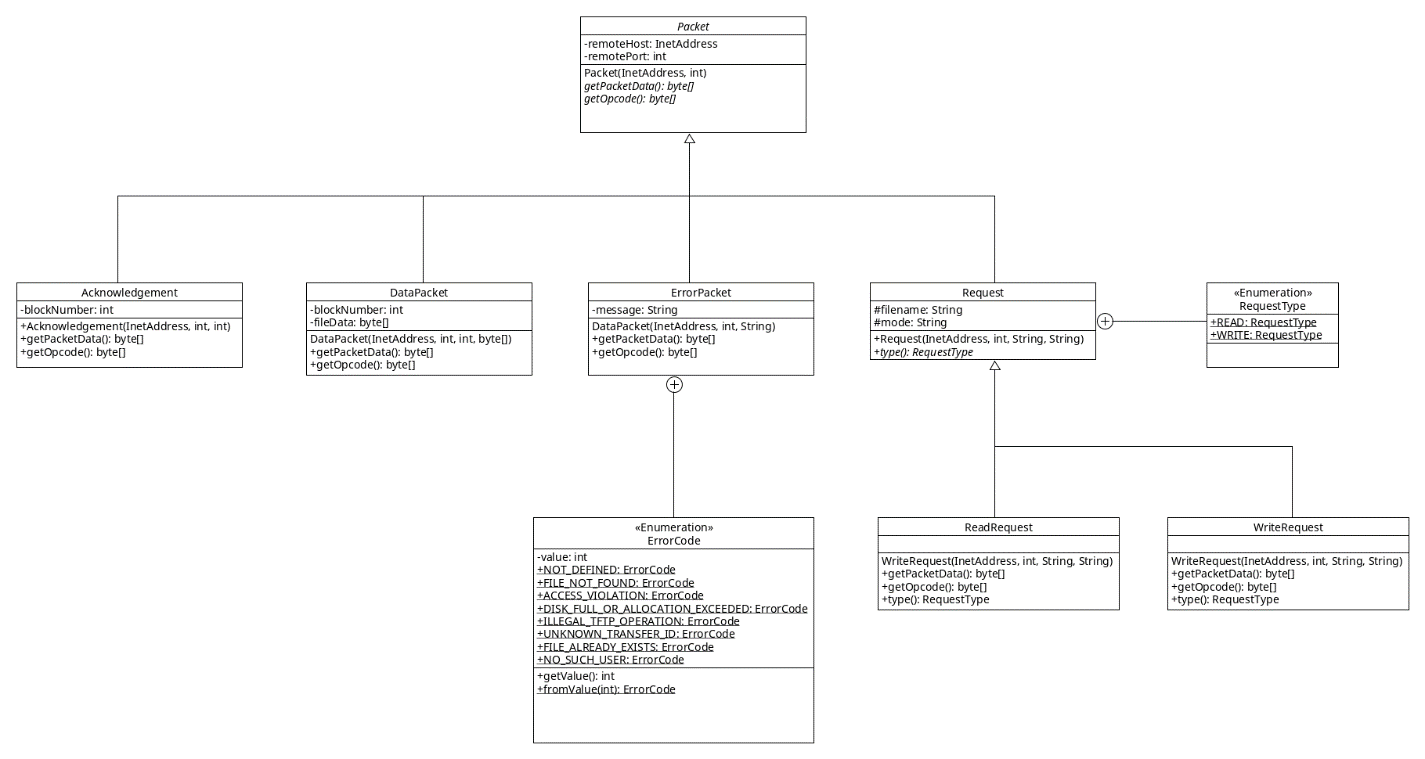


Figure : Packets

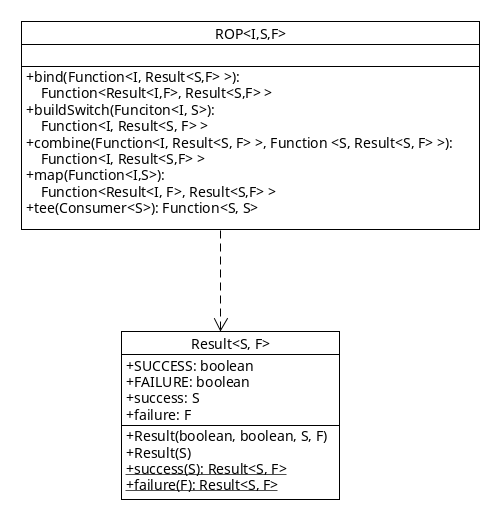


Figure : ROP

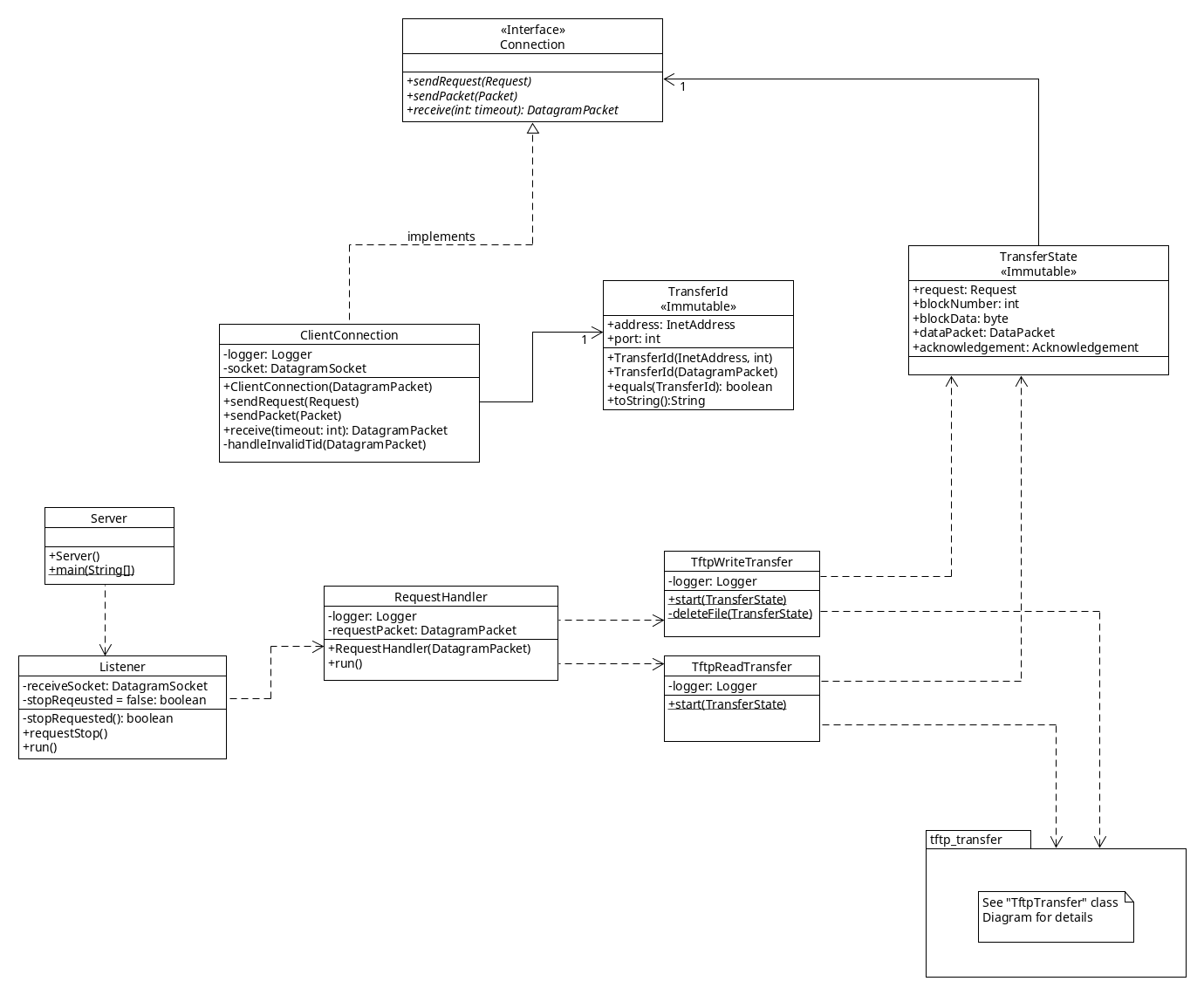


Figure : Server

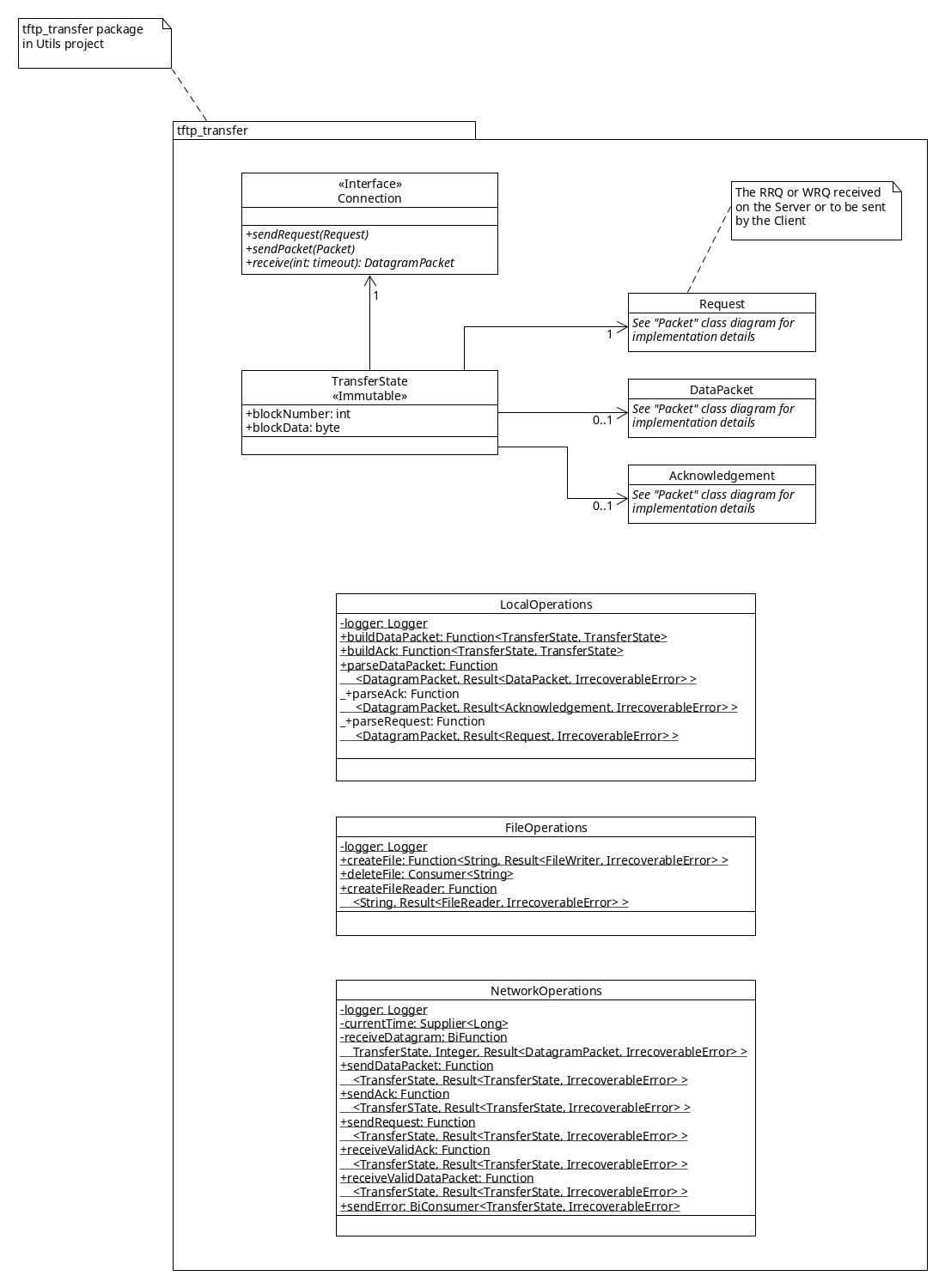


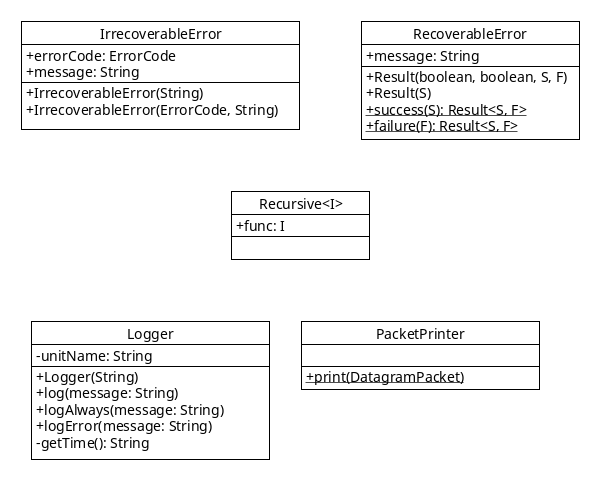
Figure : TFTP Transfer

Figure : Utils

**3.0) SETUP & TESTING INSTRUCTIONS**

**3.1) Files:**

ReadMe.txt - this file

Code/

Client/ - Code for Client

IntermediateHost/ - Code for IntermediateHost

Server/ - Code for Server

Utils/ - Common code between Server, Client, IntermediateHost

Diagrams/

ClassDiagrams/ - Class Diagrams for the system

errors\_4\_5\_TimingDiagrams/ - Error codes 4 & 5 timing diagrams (iteration 2)

lost\_delayed\_TimingDiagrams/ - Lost/Delayed packet timing diagrams (iteration 3)

UCMs/ - Use Case maps (iteration 1)

fileIO\_TimingDiagrams/ - Error codes 1, 2, 3, 6 (iteration 4)

accessViolation\_RRQ.png - Access Violation during RRQ

accessViolation\_WRQ.png - Access Violation during WRQ

diskFullOrAllocationExceed\_RRQ.png - Disk Full during RRQ

diskFullOrAllocationExceed\_WRQ.png - Disk Full during WRQ

fileAlreadyExist\_WRQ.png - File Already exists on WRQ

fileNotFound\_RRQ.png - File not found beginning of RRQ

fileNotFound\_RRQ2.png - File not found during RRQ (file was moved)

fileNotFound\_WRQ.png - File not found during WRQ (file was moved)

**3.2) Launching the application**

Import the projects

1) Open eclipse

2) Select File -> Import...

3) Select General -> "Existing Projects into Workspace"

4) Next to "Select root directory" click "Browse"

5) Select the Code folder of this submission

6) Click "Finish"

Run the program

1) Right click on the "Server" Project and select "Run As" -> "Java Application"

1.1) Select a configuration mode (usually Debug Mode)

2) Right click on the "IntermediateHost" Project and select "Run As" -> "Java Application"

2.1) Select a configuration mode (usually Debug Mode)

2.2) configure the desired error scenario (see below) or "No Modification"

3) Right click on the "Client" Project and select "Run As" -> "Java Application"

3.1) Select a configuration mode (usually Debug Mode)

**3.3) Select a Configuration for Client/Server/Intermediate**

Upon start the Client/Server/IntermediateHost will ask which configuration mode to use. The mode should always be the same between all of them, except when using Manual mode, which allows modification of all options for a particular program.

Debug Mode: Verbose output, use IntermediateHost

- Verbose logging output

- 5000ms socket timeout

- Make 3 attempts to send a packet in case of timeout

- Client connects to port 68 (intermediate port)

- Intermediate listens on port 68

- Server listens on port 69

- File block size is 512 bytes

- File path is empty => use default directory (see below)

Test Mode: Verbose output, ignore IntermediateHost

- Verbose logging output

- 5000ms socket timeout

- Make 3 attempts to send a packet in case of timeout

- Client connects to port 69 (server port)

- Intermediate listens on port 68

- Server listens on port 69

- File block size is 512 bytes

- File path is empty => use default directory (see below)

Quiet Mode: very little logging output, ignore IntermediateHost

- Very little logging

- 5000ms socket timeout

- Make 3 attempts to send a packet in case of timeout

- Client connects to port 69 (server port)

- Intermediate listens on port 68

- Server listens on port 69

- File block size is 512 bytes

- File path is empty => use default directory (see below)

Linux Mode (for testing): verbose output, uses different ports to avoid permission problems

- verbose logging output

- 5000ms socket timeout

- Make 3 attempts to send a packet in case of timeout

- Client connects to port 6900 (server port)

- Intermediate listens on port 6800

- Server listens on port 6900

- File block size is 512 bytes

- File path is empty => use default directory (see below)

Manual Mode: Everything can be configured by the user

- configure ports, timeouts, etc.

- File path: Enter an absolute file path using. e.g. E:/testfolder

**3.4) Configure the Intermediate Host**

After selecting a configuration mode, the intermediate host displays an extensive menu that allows for modifications. To delay / duplicate / drop a packet, first select the packet type and then the number of the packet you want to modify (RRQ and WRQ is always the first packet). Then select one of the options (e.g. delay packet). There are a number of test files in both the Server and Client folder, these test files are named according to their size.

**3.5) Transfer File locations**

- By default, files on the server are placed in the Code/Server directory

- By default, files on the client are placed in the Code/Client directory

- It is possible to modify the locations by selecting Manual mode when asked

for a configuration mode when the program first starts.

**3.6) ERROR SCENARIOS**

*NOTE: The Client and Server contain a number of test files in their respective default directories. These files are named after their size (e.g. two-blocks will be slightly smaller than 1024 bytes, two-blocks-exactly will be exactly 1024 bytes big). These files are identical on both the client and server, so you may need to delete them on either side to test successful transmissions.*

**File Not Found:**

There are two possible scenarios in which this error may occur:

1) The file does not exist on the server:

The server received a RRQ but the filename contained in the request cannot be found.

In that case the server sends an error (Code 1) and aborts the transfer.

On the client on a Write Request if the user enters a filename and that file does not exist,

the client will show an error message and NOT send a Write Request, therefore there is no

Error being sent either.

2) The file goes away during transfer (e.g. USB key unplugged)

On the server, if during a Read Transfer, the file we are reading from is (re)moved,

the server cannot continue to read from it and thus will send an Error Code 1, and

abort the transfer.

On the client, if during a Write Transfer, the file we are reading from is (re)moved

the Client sends an Error Code 1, and aborts the transfer.

*NOTE: Since we are using a Buffered Input Stream, so for small files this will not occur, since the entire file will already be in the buffer.*

Testing:

- Client:

- Try to send a file that does not exist in the configured path.

-> Does not send an error packet since non-existent file is discovered before sending WRQ

- Configure the client (Manual Mode) to use a path to a USB key

Then initiate a file transfer and unplug the USB key mid-transfer.

- Server:

- Try to send a RRQ for a file that does not exist on the server.

- Configure the client (Manual Mode) to use a path to a USB key

Then initiate a file transfer and unplug the USB key mid-transfer.

**Access Violation:**

On the Client:

If, on a Write Transfer, the file the user selects cannot be read, the user is informed

with an error message and we DO NOT send a WRQ to the server. Therefore the is no error packet.

On a Read Transfer the Client will first attempt to create a file. Only if that's successful

will it send the RRQ.

On the Server:

If the server receives a RRQ or WRQ, and trying to read or write the file is not possible

because the permissions are insufficient (no read/write access), the server responds with an

Error code 2 and terminates the transfer.

Note that if the permissions change during a transfer, it does not affect the program, since it

already opened the file and has a valid file descriptor. Thus it can continue to read/write and

complete the transfer.

Testing:

- Client:

- Configure the client (Manual Mode) to use a path to C:\Users\[someOtherUser]

Then do a RRQ or WRQ.

-> Does not send and error packet, since missing permissions are discovered before RRQ or WRQ is sent

- Server:

- Configure the server (Manual Mode) to use a path to C:\Users\[someOtherUser]

Then do a RRQ or WRQ from the client.

**Disk Full:**

If trying to write a block to a file on disk fails, because the disk is full, both client

and server will respond in the same way. First an error message (Code 3) is sent and then

the file that had been written to is deleted, since it is incomplete.

Testing:

- Client:

- Configure the client (Manual Mode) to use a path to a USB key that's full

Then do a RRQ

- Server:

- Configure the server (Manual Mode) to use a path to s USB key that's full

Then do a WRQ

**File Already Exists:**

On the Client:

When performing a Read Transfer, we first check whether a file with the requested filename

already exists on the client. If it does, the user is notified with an error message and

we DO NOT send a RRQ to the server.

On the Server:

When receiving a WRQ the server checks whether the file already exists. If it does we send

a "File already exists" error (Code 6) and abort the transfer.

This also resolves issues if two clients try to write the same file at the same time. Only

one of them (whoever happens to be served first), will be able to write.

Testing:

- Client:

- Try to do a RRQ entering a file that already exists on the client.

-> Does not send an error packet, since existing file is discovered before sending RRQ.

- Server:

- Try to do a WRQ sending a file that already exists on the Server.

**3.7) DESIGN DECISIONS**

Overall Design:

After grappling with many subtle bugs, related to state inconsistencies in previous iterations, we decided to scrap most of the existing code on the Client and Server and start fresh. The goal was to minimize state to only the bare essentials. Thus we introduced a "TransferState" class that encapsulates all state needed for a particular transfer (Read or Write). Furthermore, every instance of TransferState is immutable, which makes it perfectly safe to pass it around to many different methods/functions. Since it cannot be changed along the way, it has become much simpler to keep track and ensure correctness of state.

By removing state completely from the behavior we were able to break out the different steps involved in a transfer (receive an ack, read from file, create data packet, send data packet, etc.) into small simple procedures. We opted to use Java 8's new Lambda implementation for most of them since it allows functions to be treated as values, along with a number of other benefits. Most of these small functions take TransferState as a parameter and return same as a result. By completely decoupling state from behavior, and moving the behavior required for a transfer into small re-usable functions, we were able to break out all of that behavior into its own package.

(Utils/tftp\_transfer).

Clearly, at some point state does have to change, e.g. once we read a block from a file. In that case a function will ALWAYS return a new state object containing new information. Every function is designed to only change a single aspect of the state, thus most of the old state can be cloned and then new values are assigned to the new state object. Creating a new state object for every change may seem inefficient, but as it turns out, Java is pretty efficient in creating objects (who knew!?). Also, the state object is pretty small, containing only a few primitive types, and a number of references to other objects. These references are simply copied over to the new state object and do not require to re-instantiate the objects they refer to. Also, the benefits by far outweigh the negatives as we'll see below.

As it turns out, performing a TFTP Read Transfer on the Server is almost the exact same as performing a TFTP Write Transfer on the Client. The same is true for Write Transfer on the Server and Read Transfer on the Client. Thus, we are now sharing a majority of that behavior between Client and Server. So client and server both run pretty much the same code to facilitate a TFTP Transfer. As such we have reached epic proportions of code reuse and maintainability.

Note on ROP:

-------------

Railway Oriented Programming (ROP) is a new-ish concept used in Functional Programming that can aid in error

handling. Essentially, all functions return a Result type which can either be a SUCCESS or a FAILURE.

When composing functions, as soon as one of them returns a FAILURE result, the rest of them are

bypassed. The error is then handled in a single place afterwards. This makes the code much cleaner

and removes the need for exceptions in that situation.

The Utils/rop package contains some helper functions to facilitate ROP in our project.

Source: http://fsharpforfunandprofit.com/posts/recipe-part2/