Term Project

Winter 2016 – Group 4 – SYSC 3303

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

Carleton University

April 4th, 2016

Table of Contents

Team Work Breakdown

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

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

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

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

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 for remote IPs, Testing

Diagrams

Use Case Maps (UCMs)

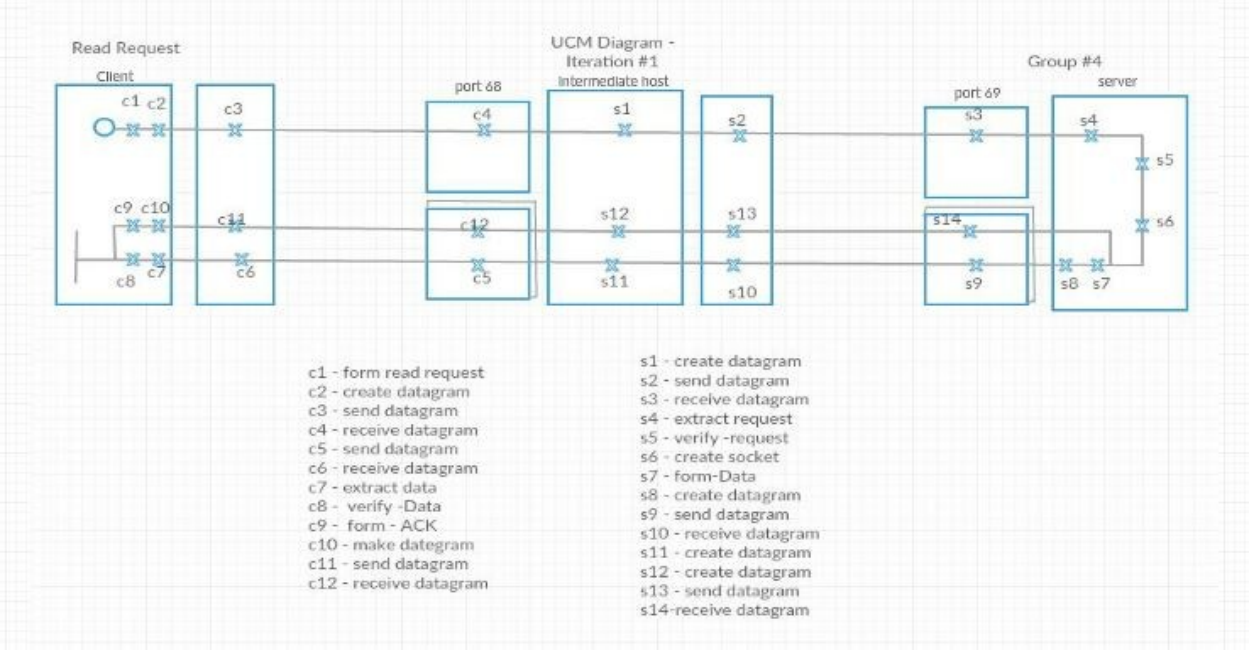


Figure 1: Read Request UCM

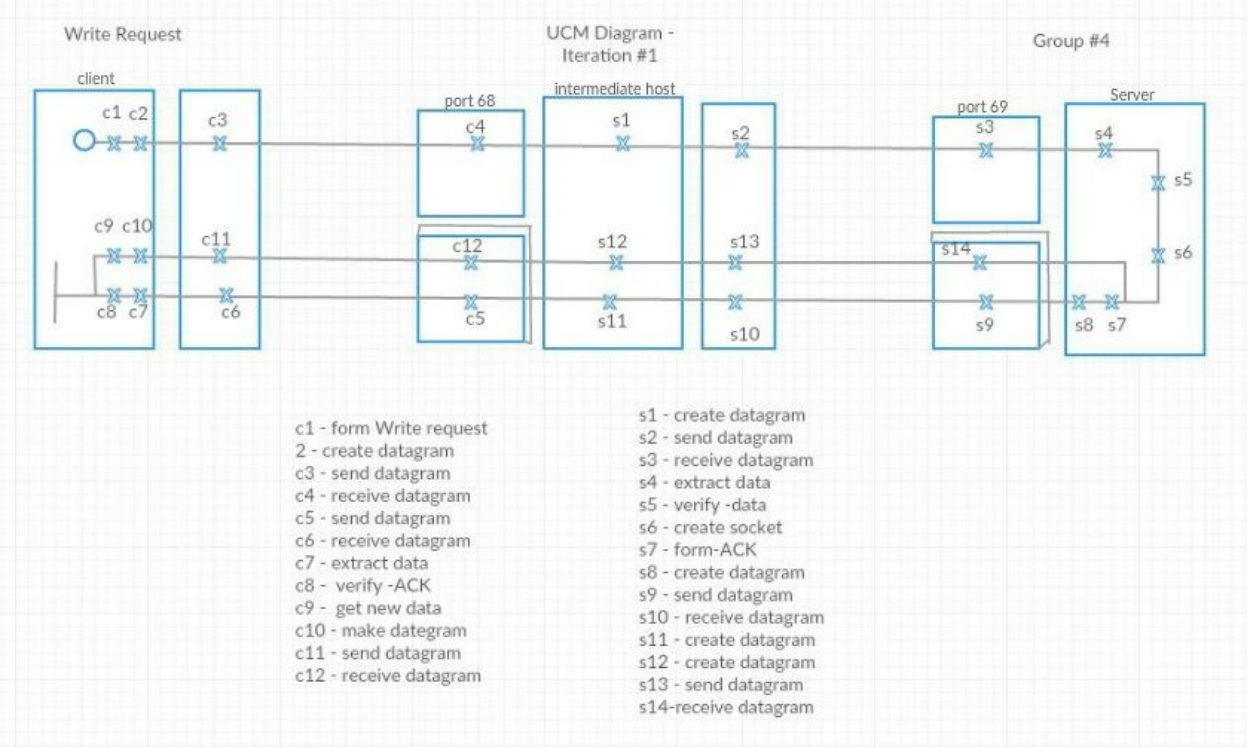


Figure 2: Write Request UCM

Timing Diagrams

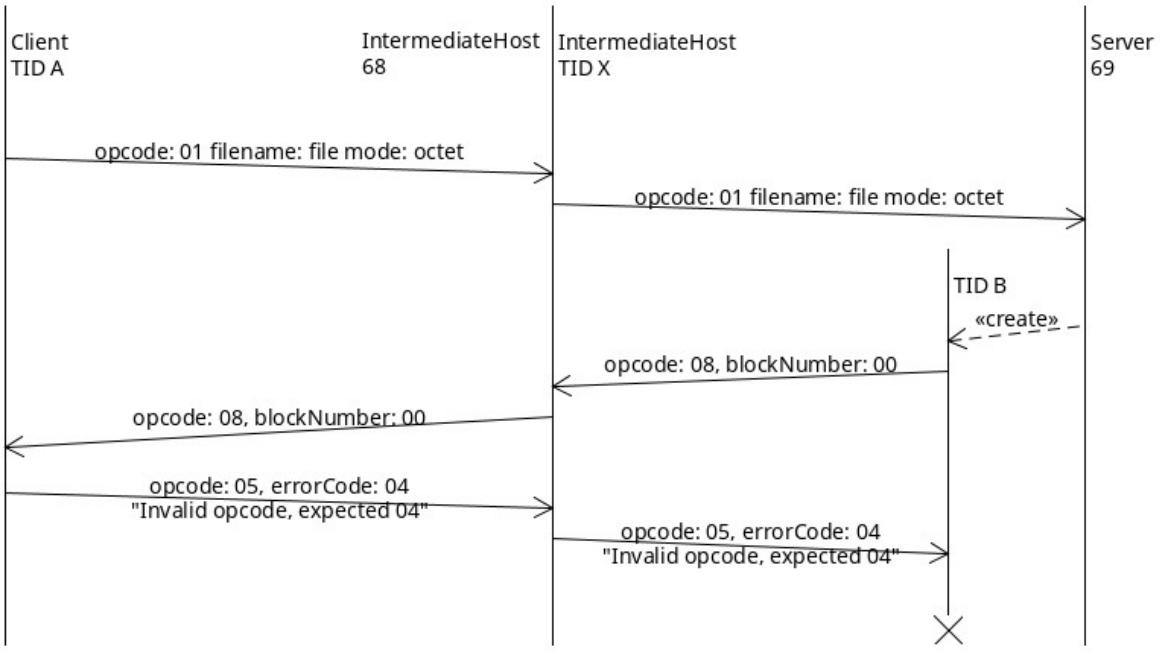


Figure 3: Invalid ACK – RRQ

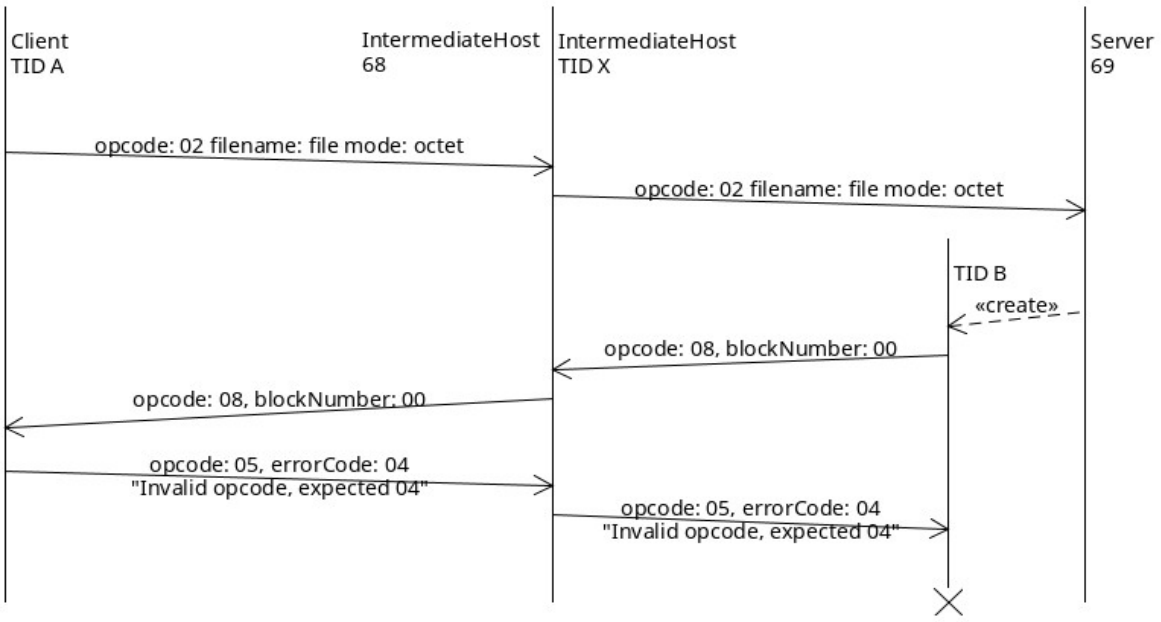


Figure 4: Invalid ACK – WRQ

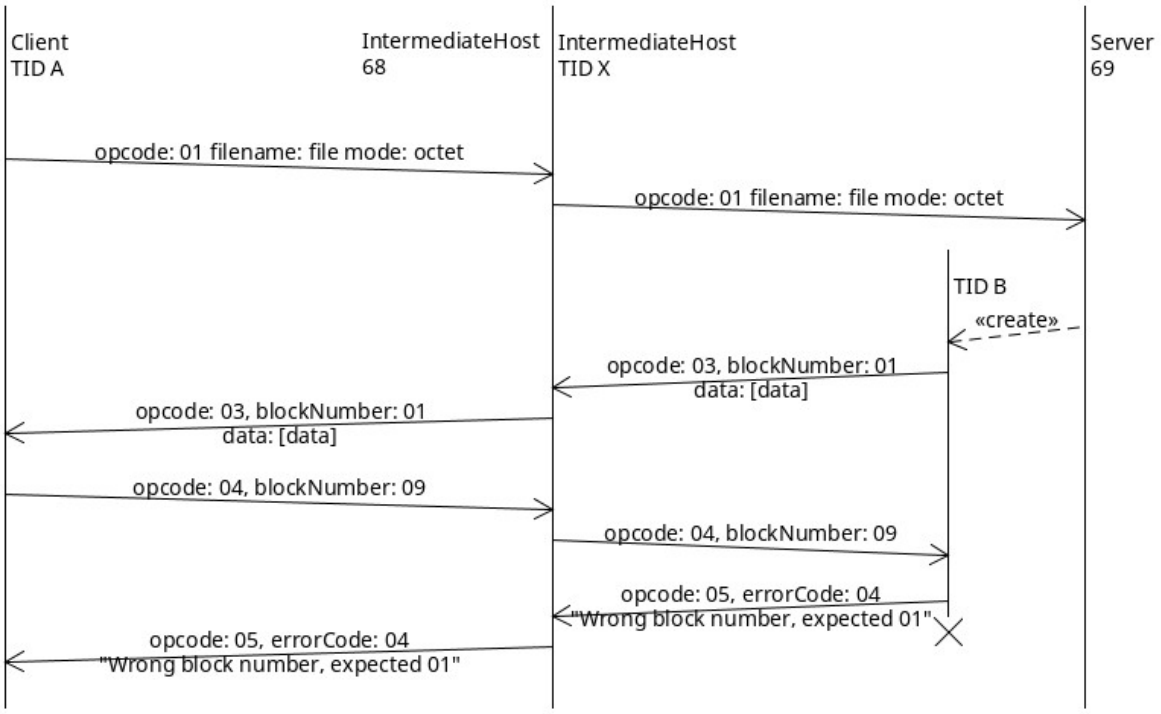


Figure 5: Invalid Block – ACK

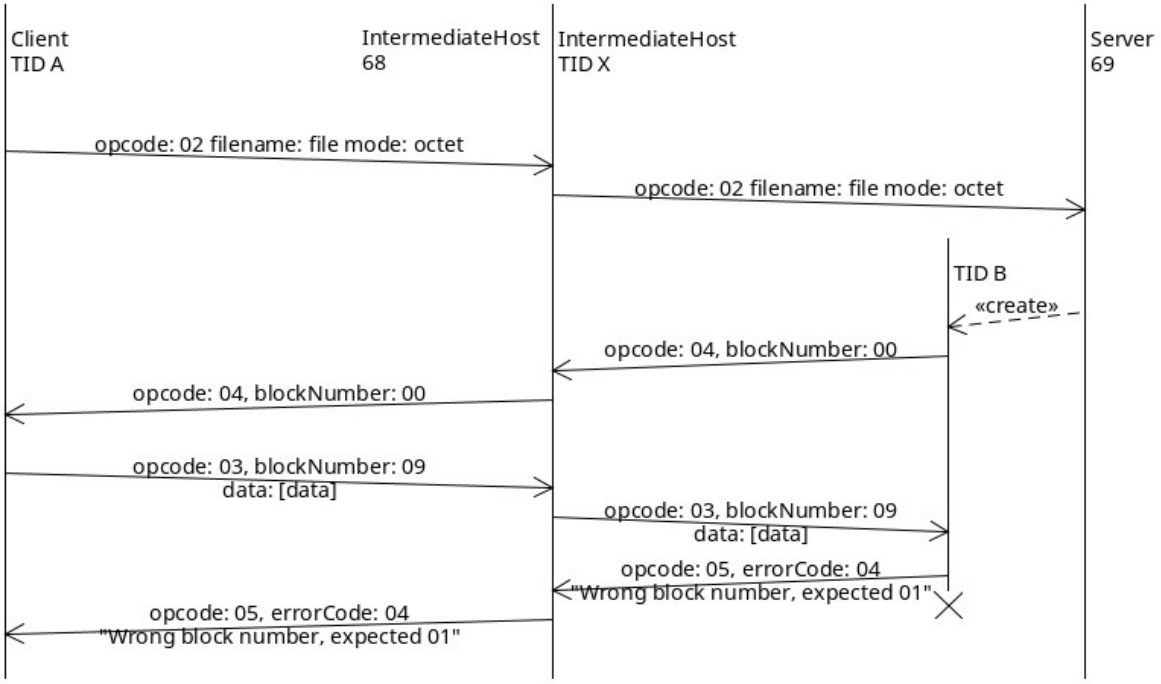


Figure 6: Invalid Block – DATA

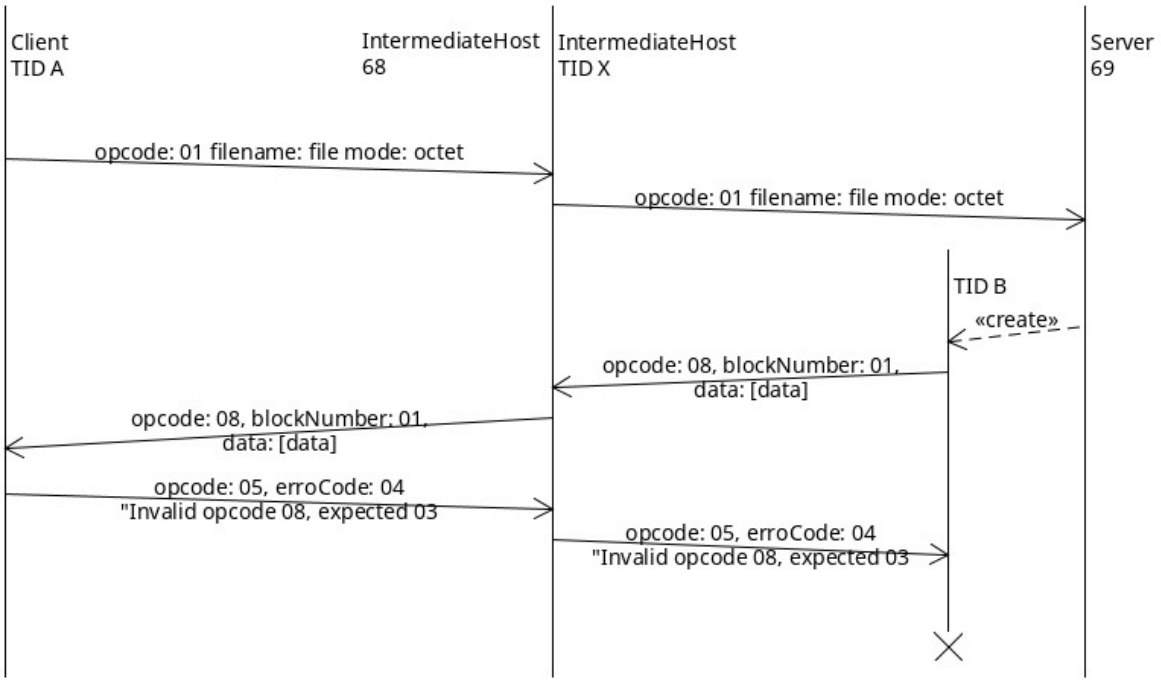


Figure 7: Invalid Data Packet – RRQ

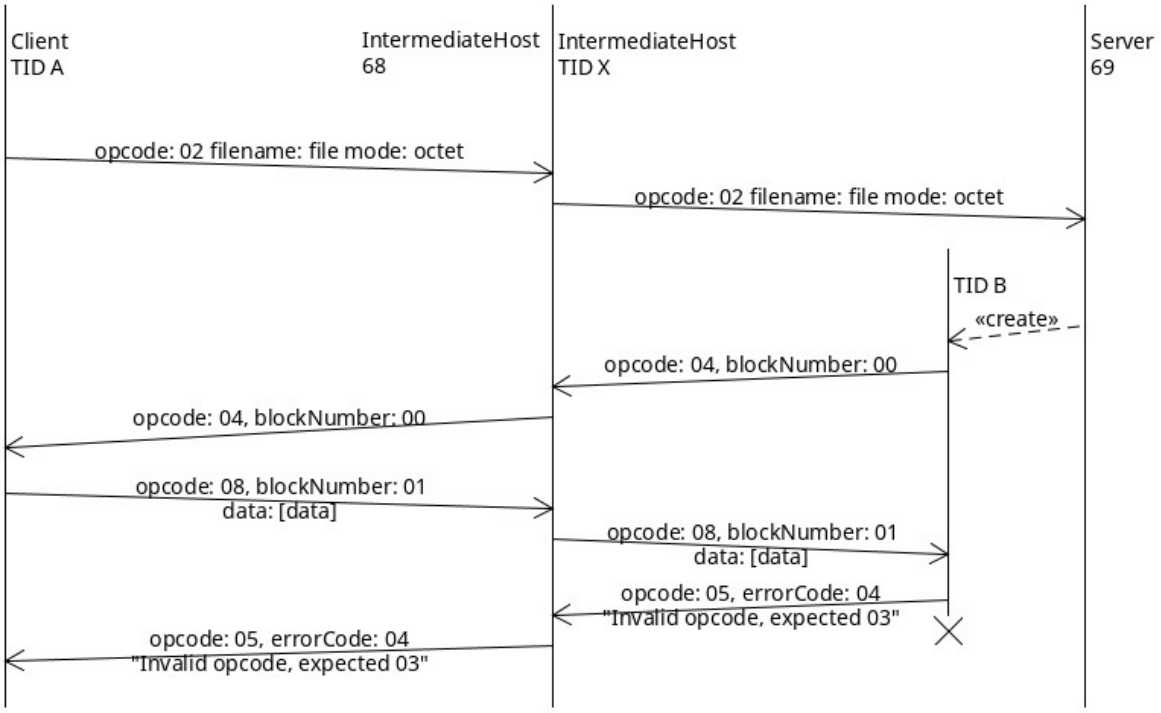


Figure 8: Invalid Data Packet – WRQ

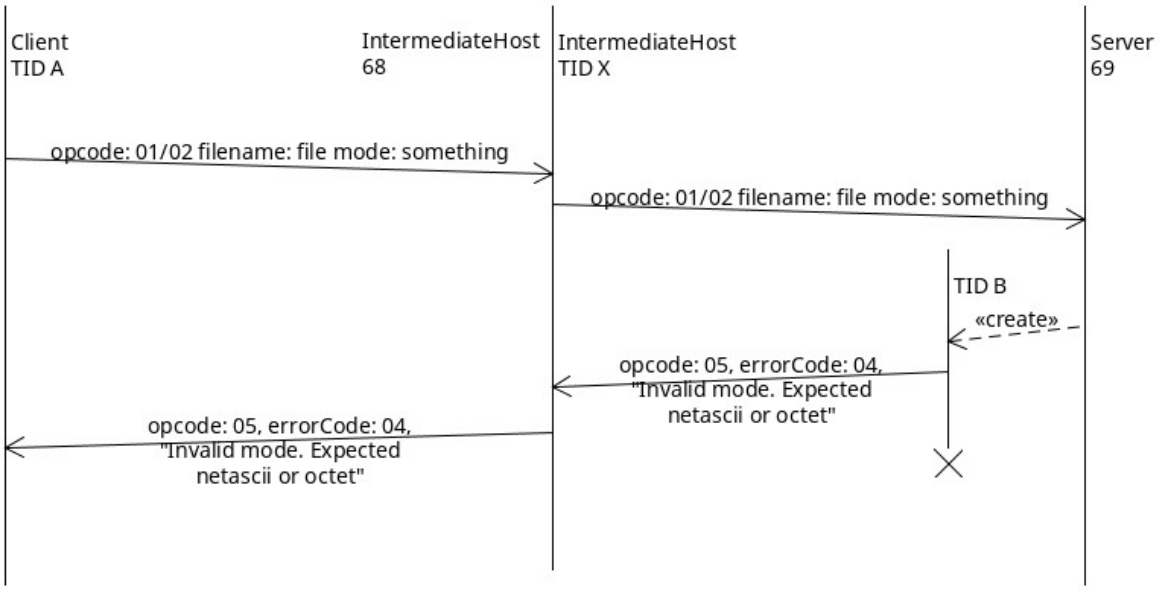


Figure 9: Invalid Request Mode

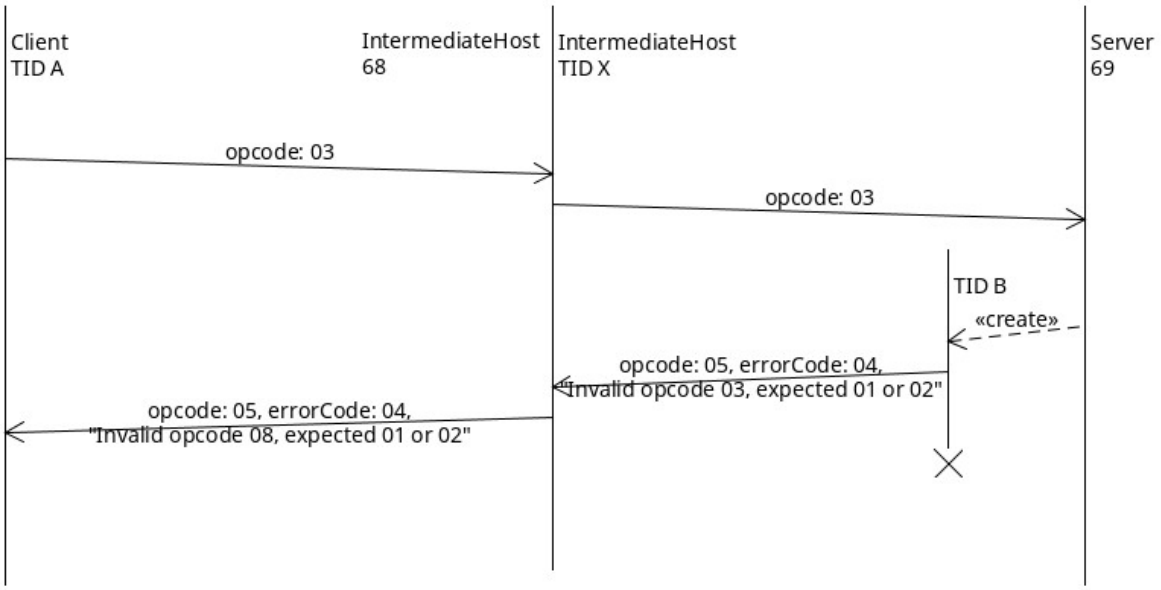


Figure 10: Timing Invalid Opcode

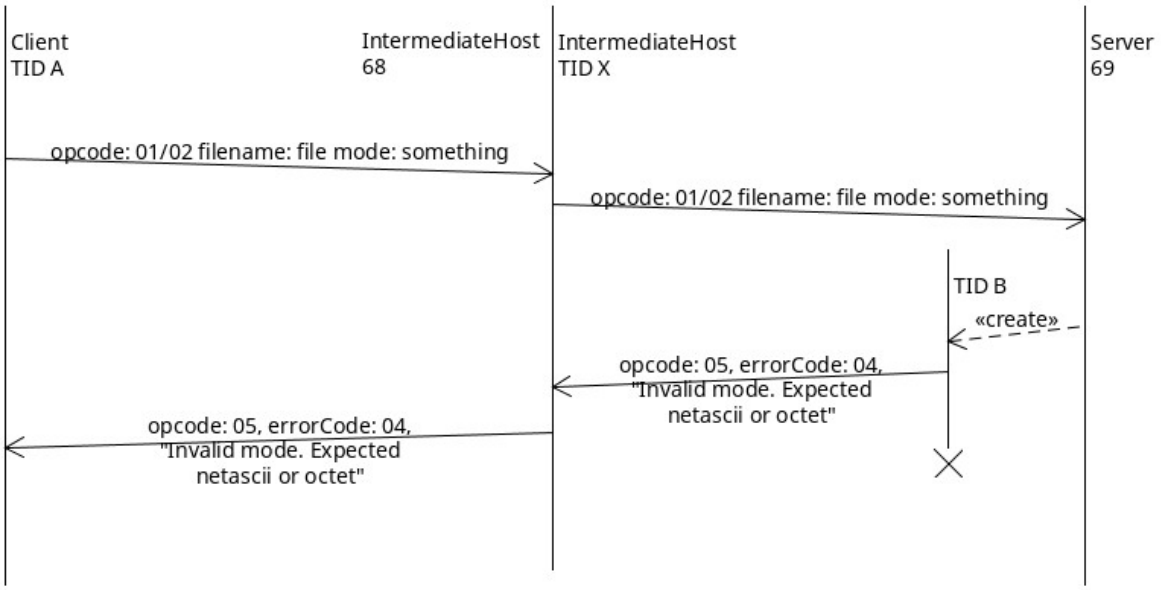


Figure 11: Timing Invalid Request Mode

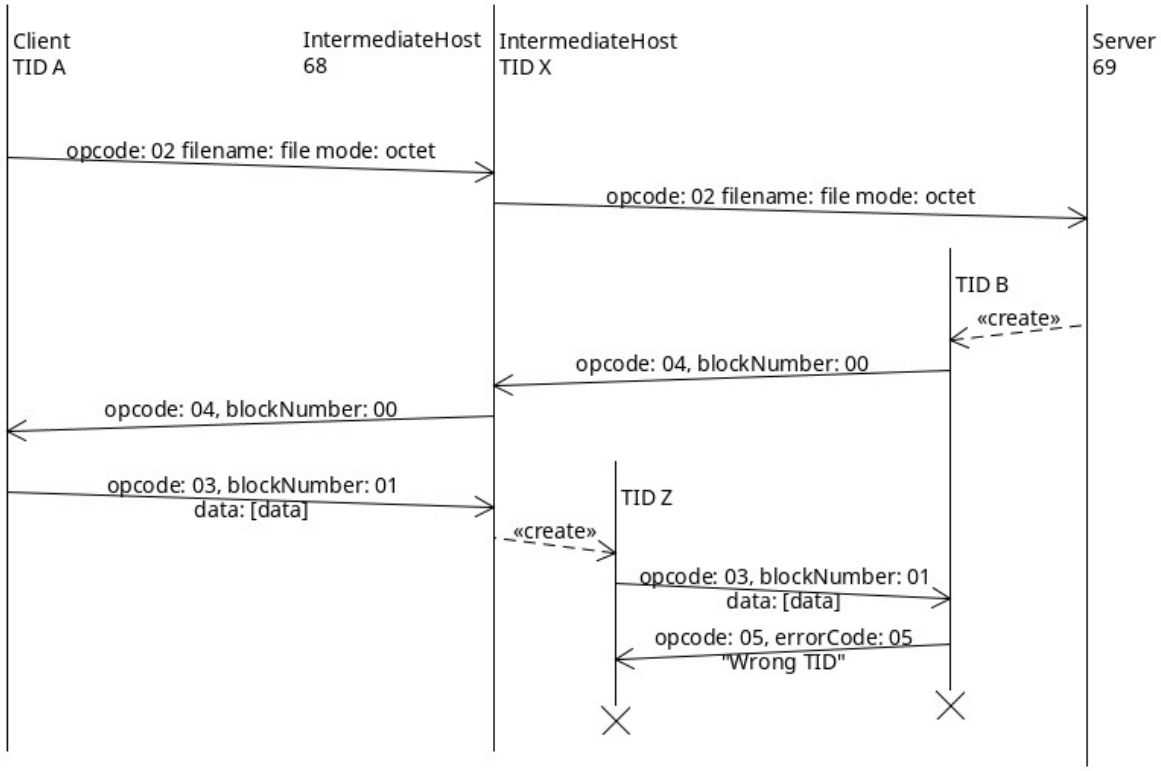


Figure 12: Wrong TID

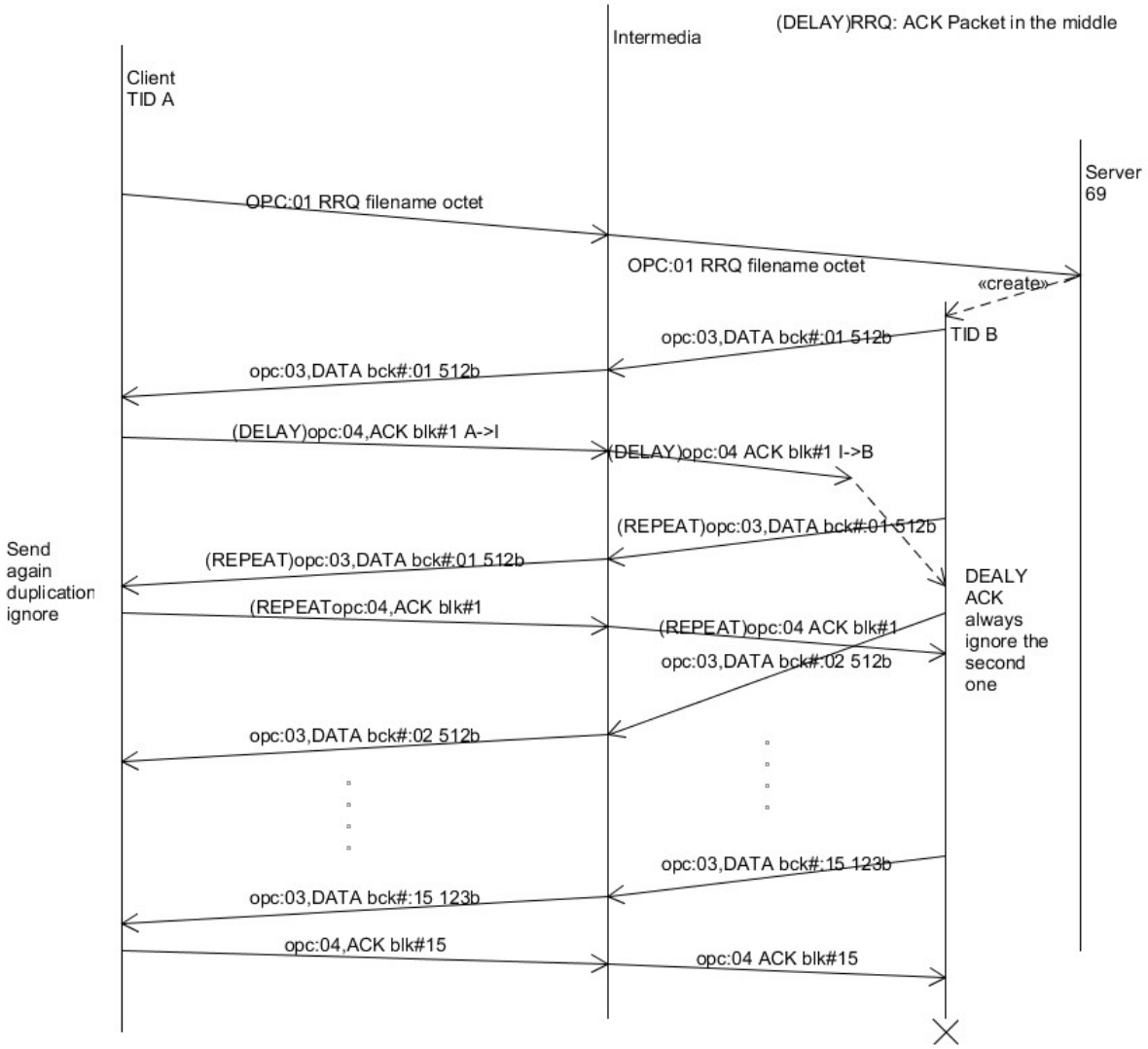


Figure 13: DEALY ACK Packet in the middle –RRQ

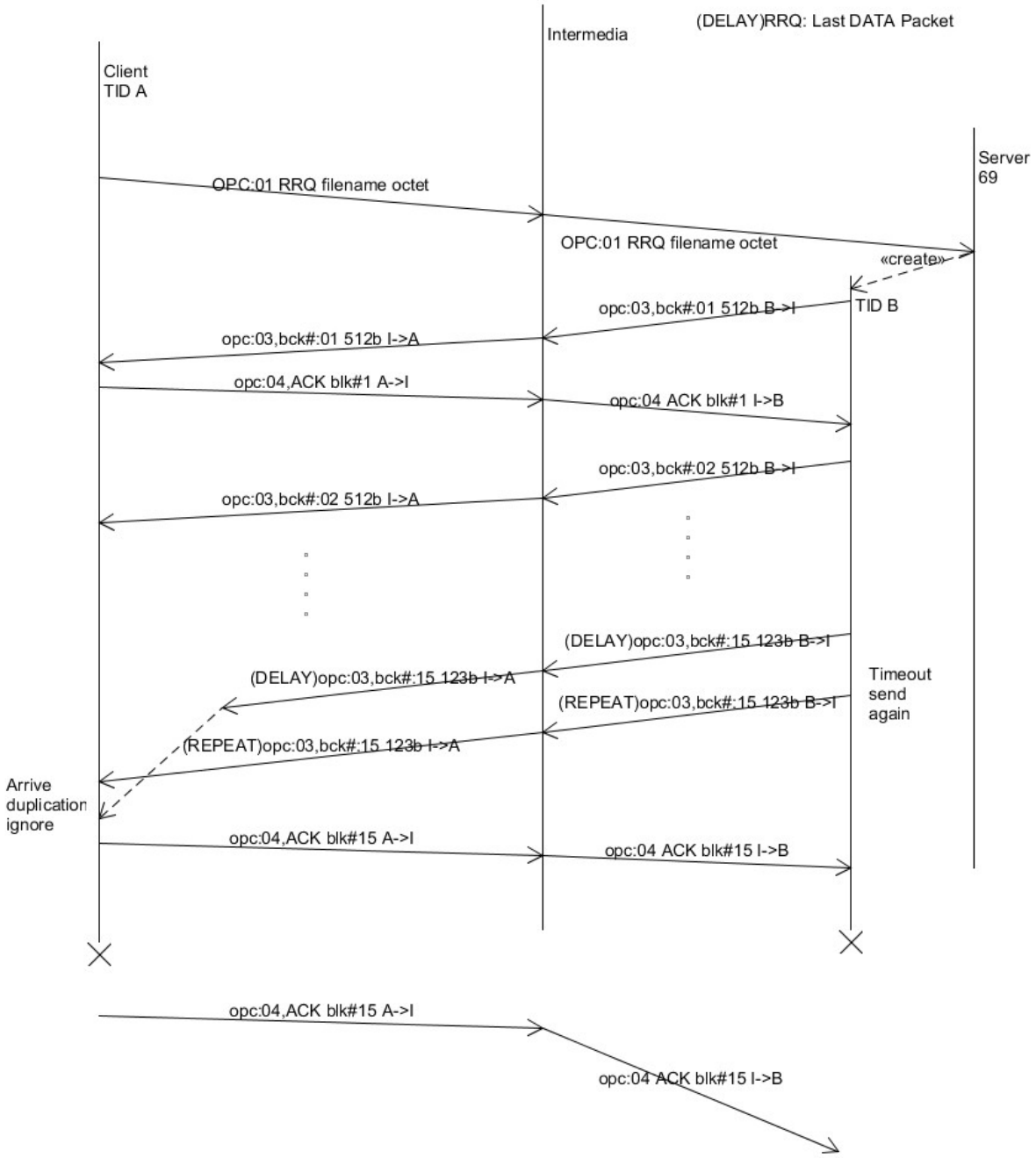


Figure 14: DELAY Last DATA Packet—RRQ

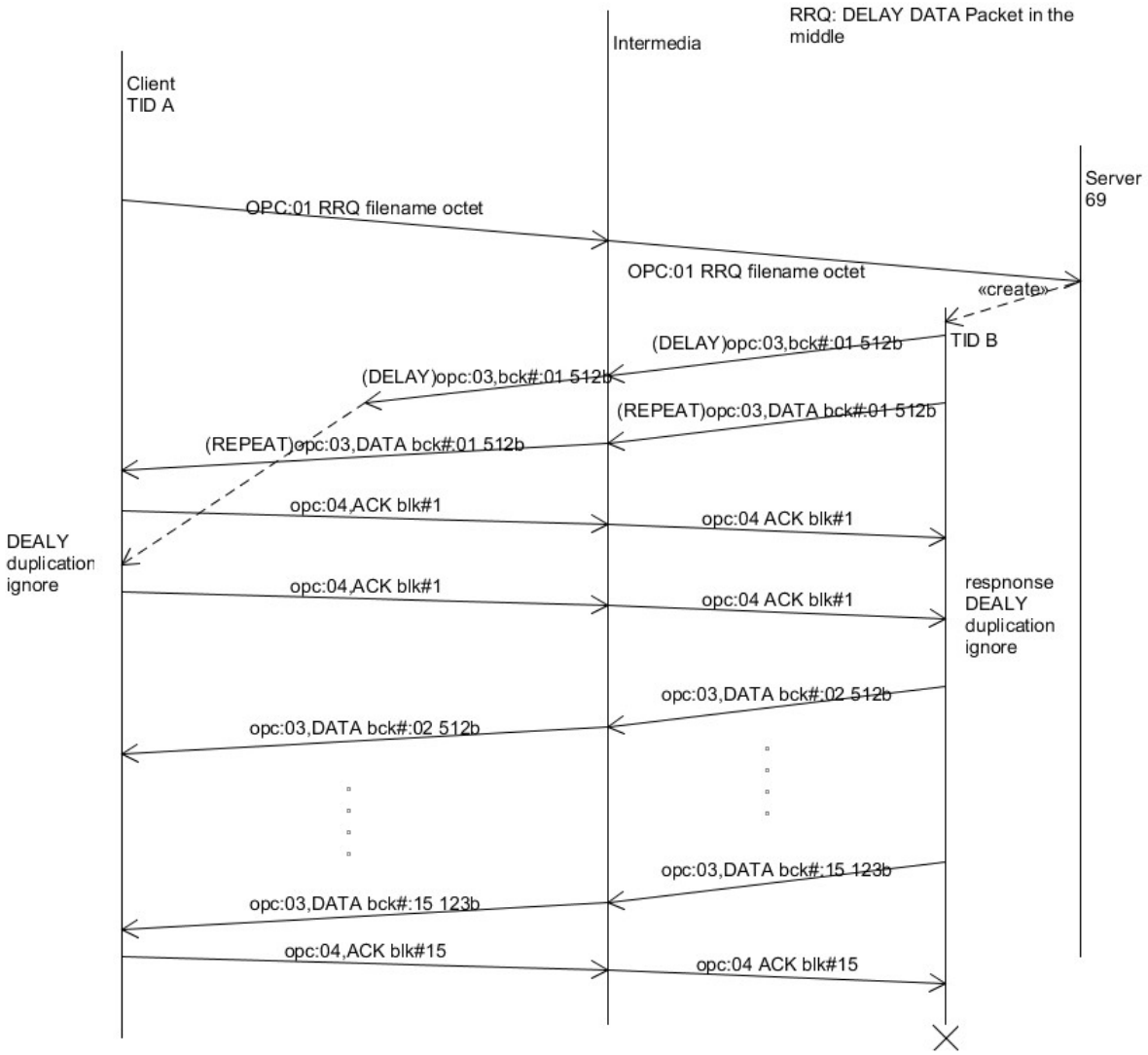


Figure 15: DELAY DATA Packet in the middle—RRQ

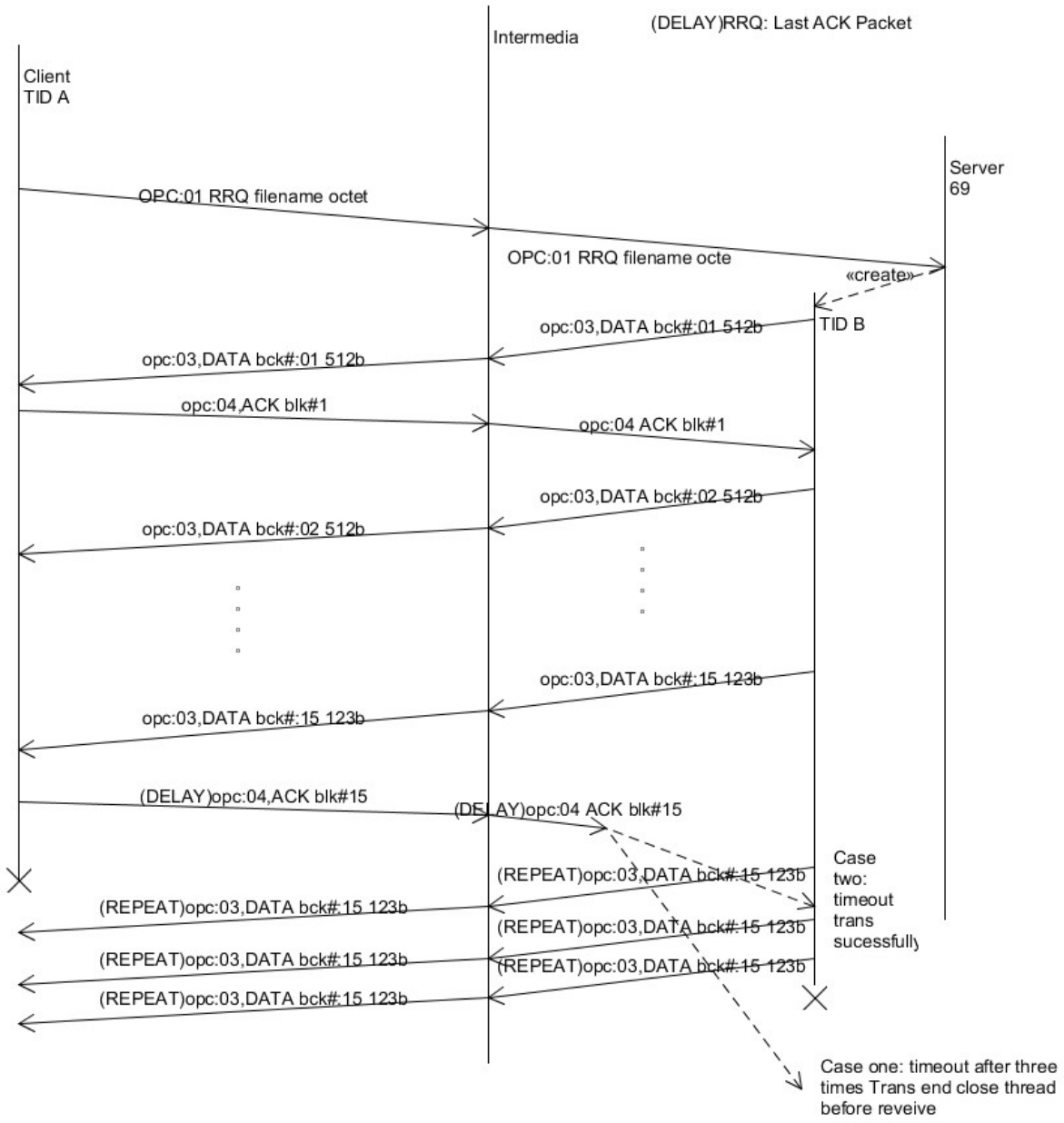


Figure 16: DELAY Last ACK Packet—RRQ

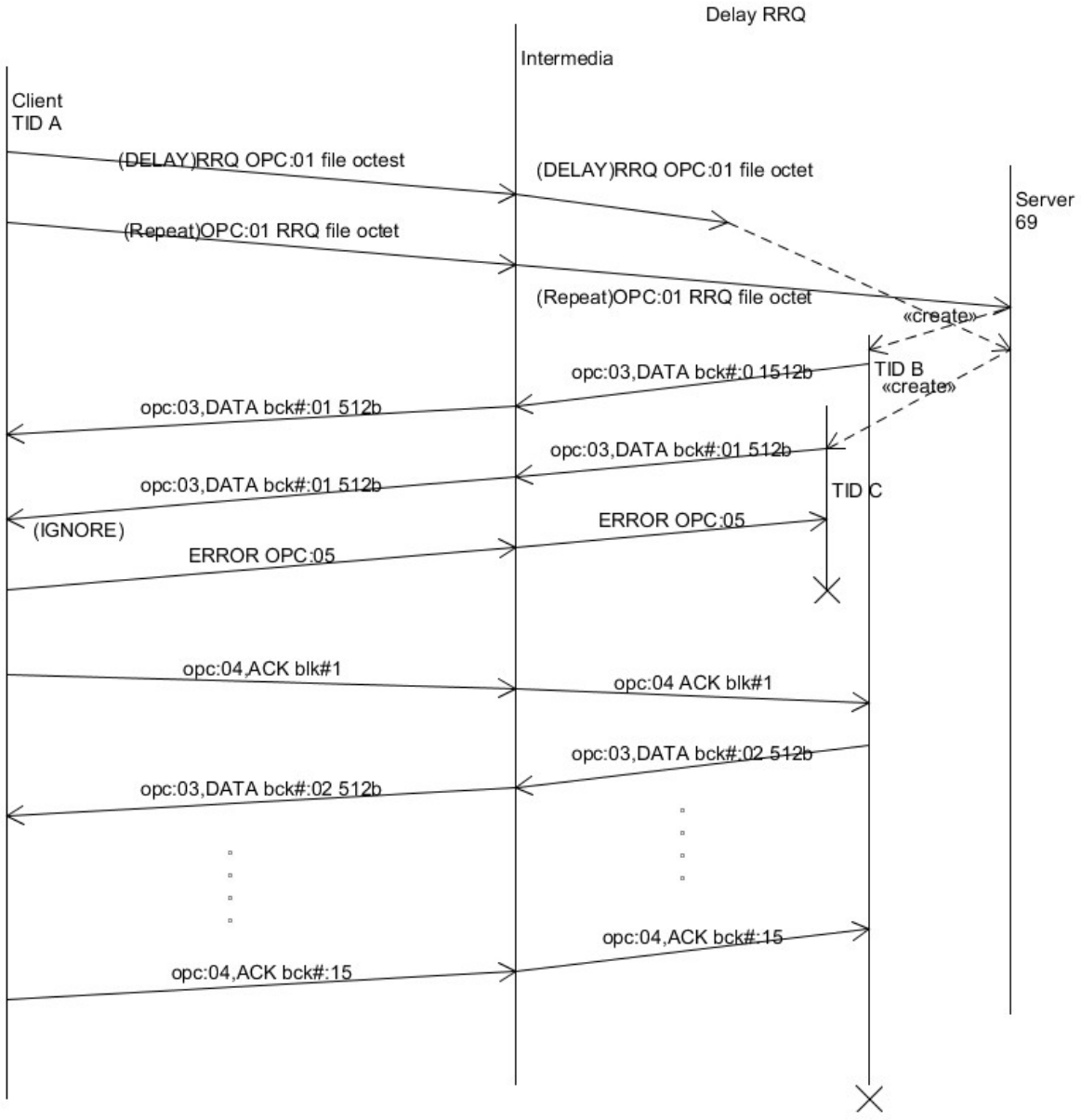


Figure 17: DELAY RRQ

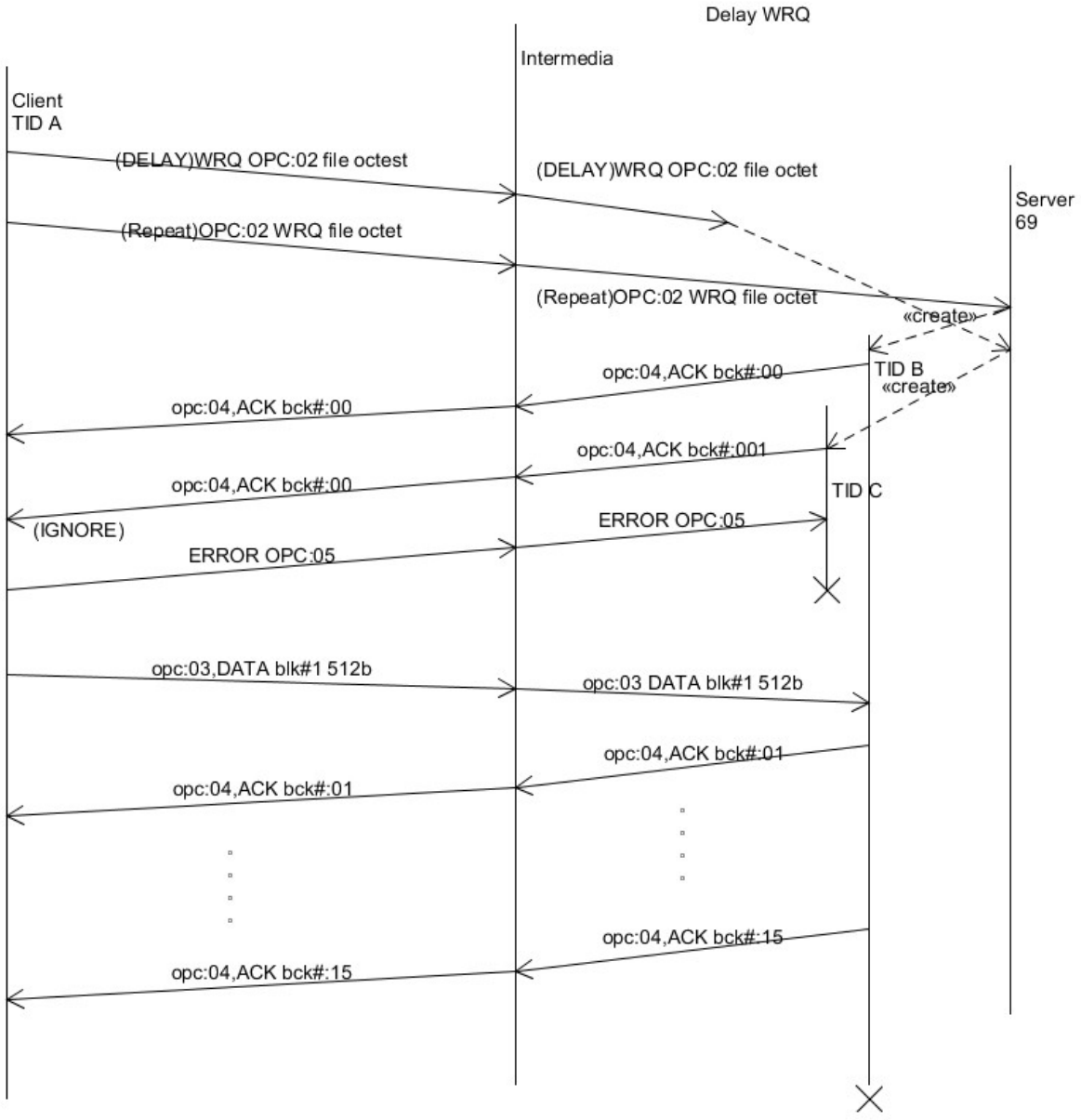
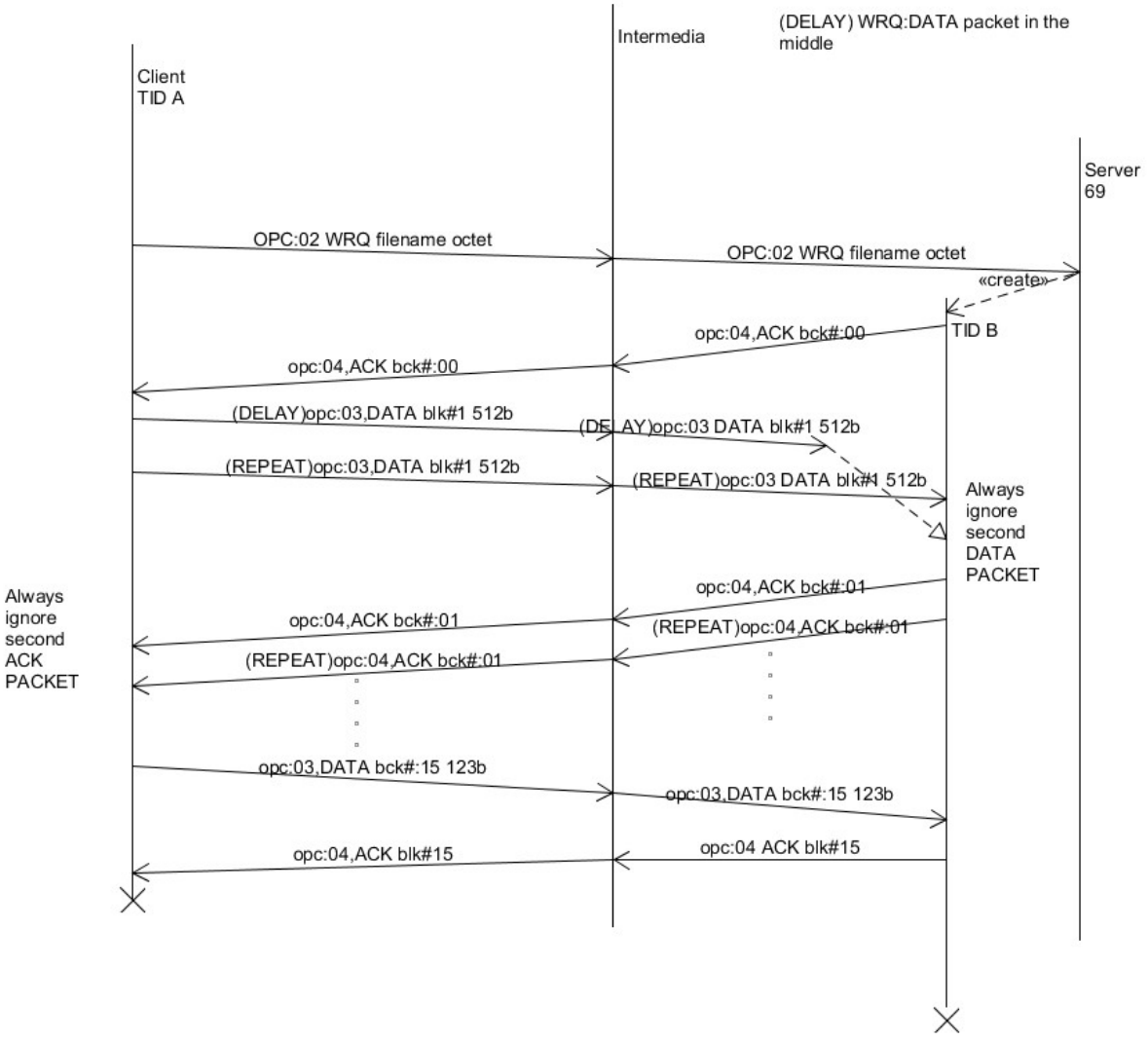


Figure 18: DeELAY WRQ –WRQ



.

Figure 19: DELAY DATA in the middle –WRQ

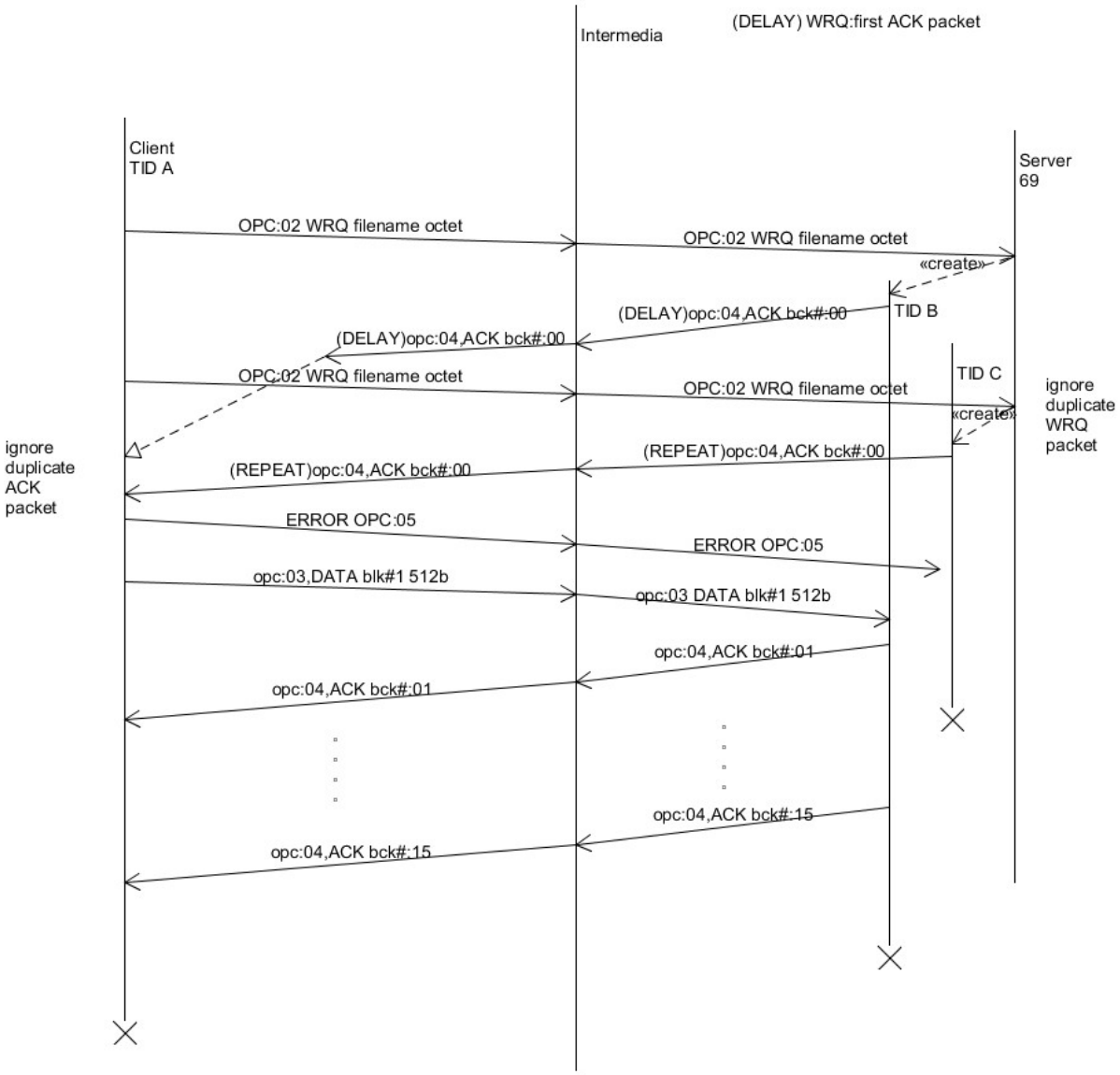


Figure 20: DELAY first ACK packet –WRQ

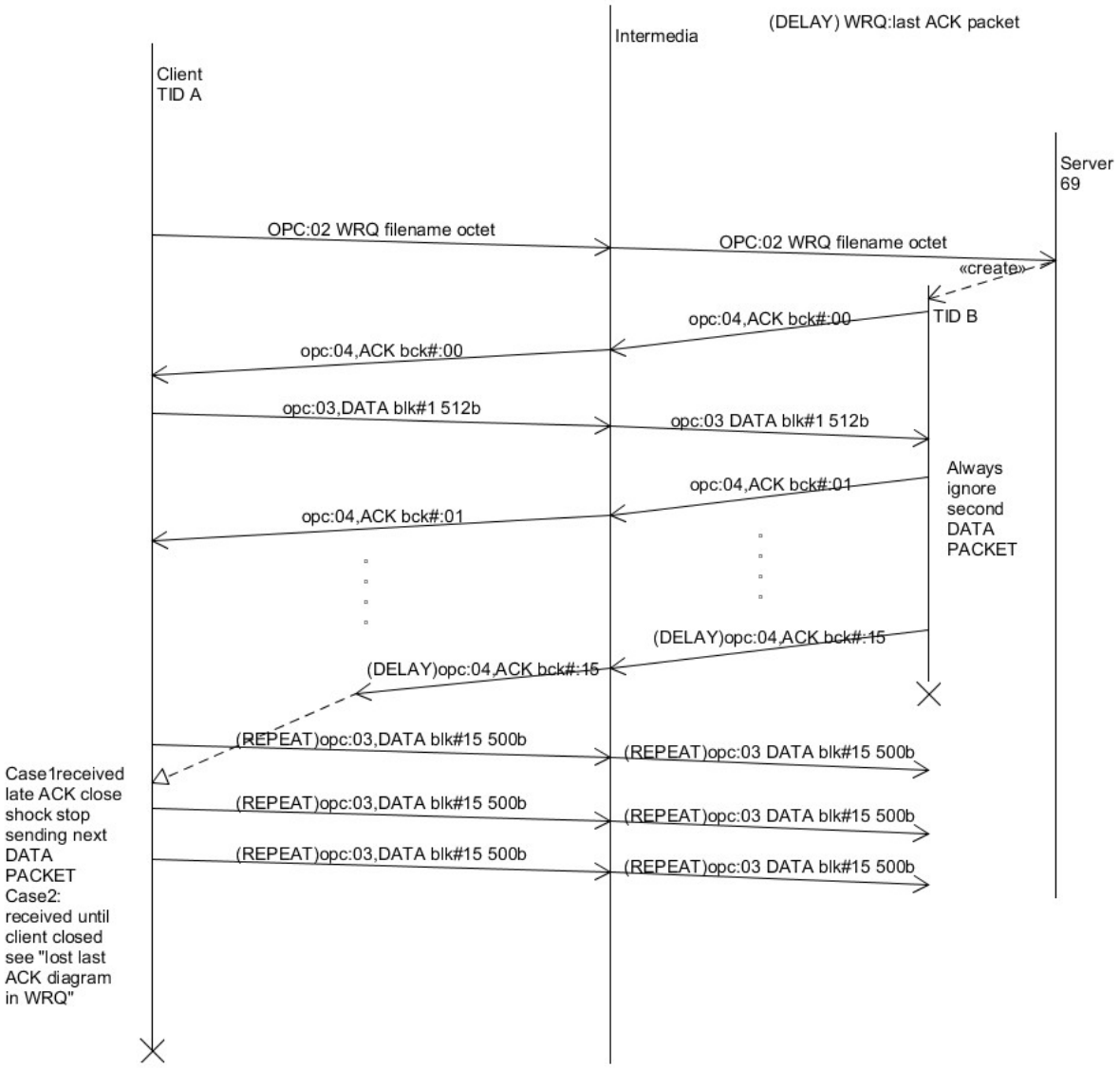


Figure 21: DELAY first ACK—WRQ

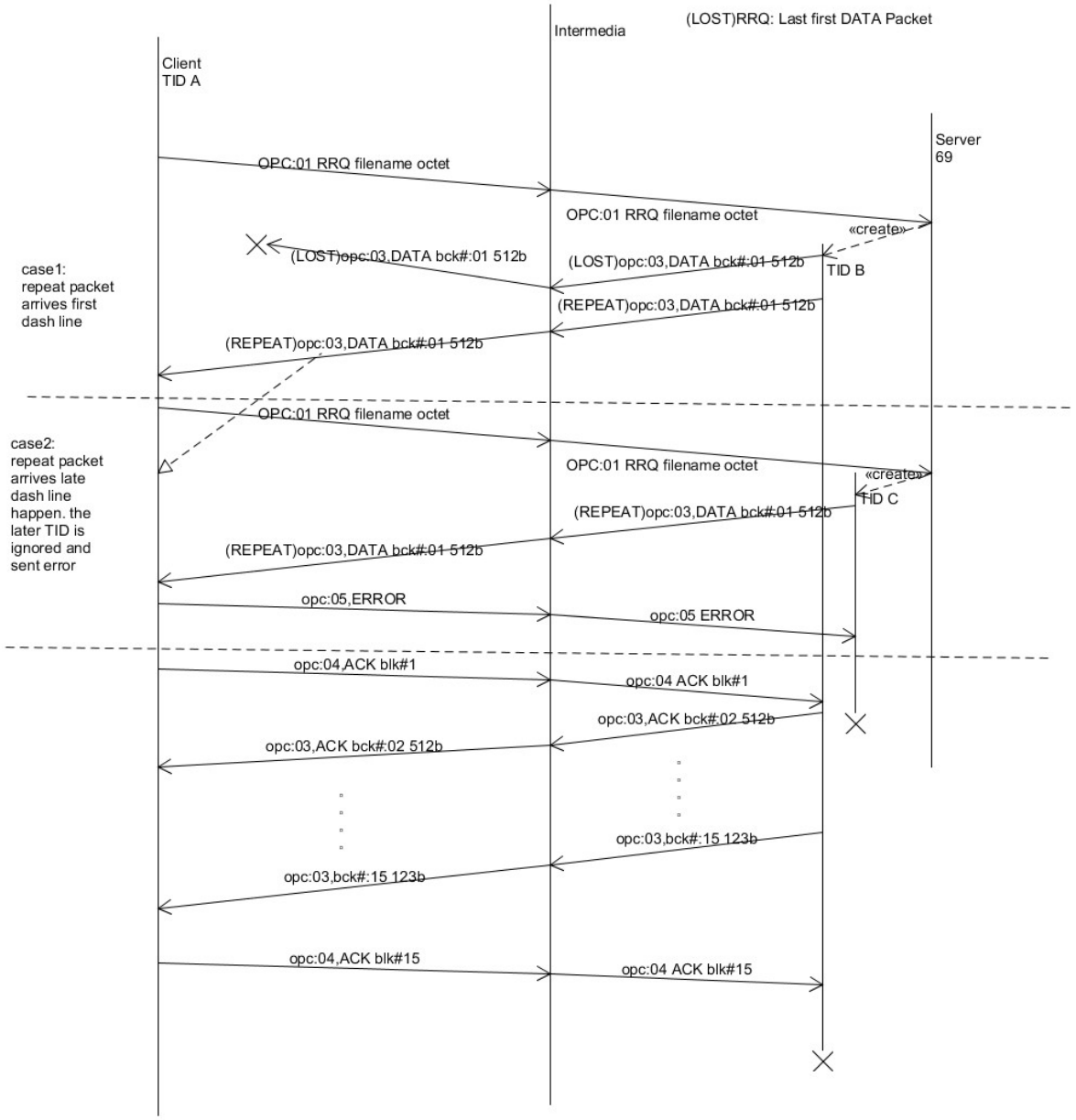


Figure 22: LOST first DATA packet –RRQ

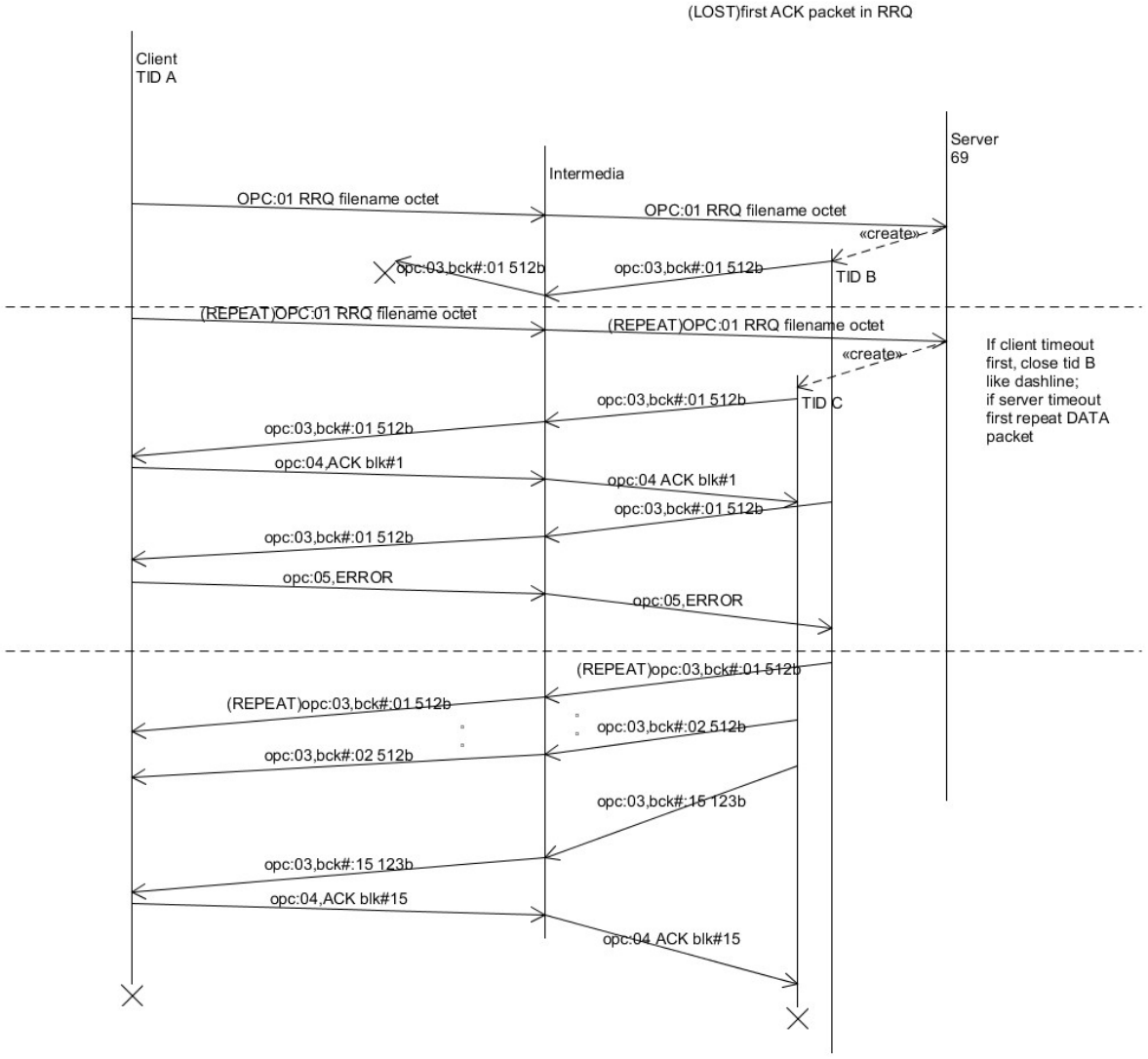


Figure 23: LOST fist ACK—RRQ

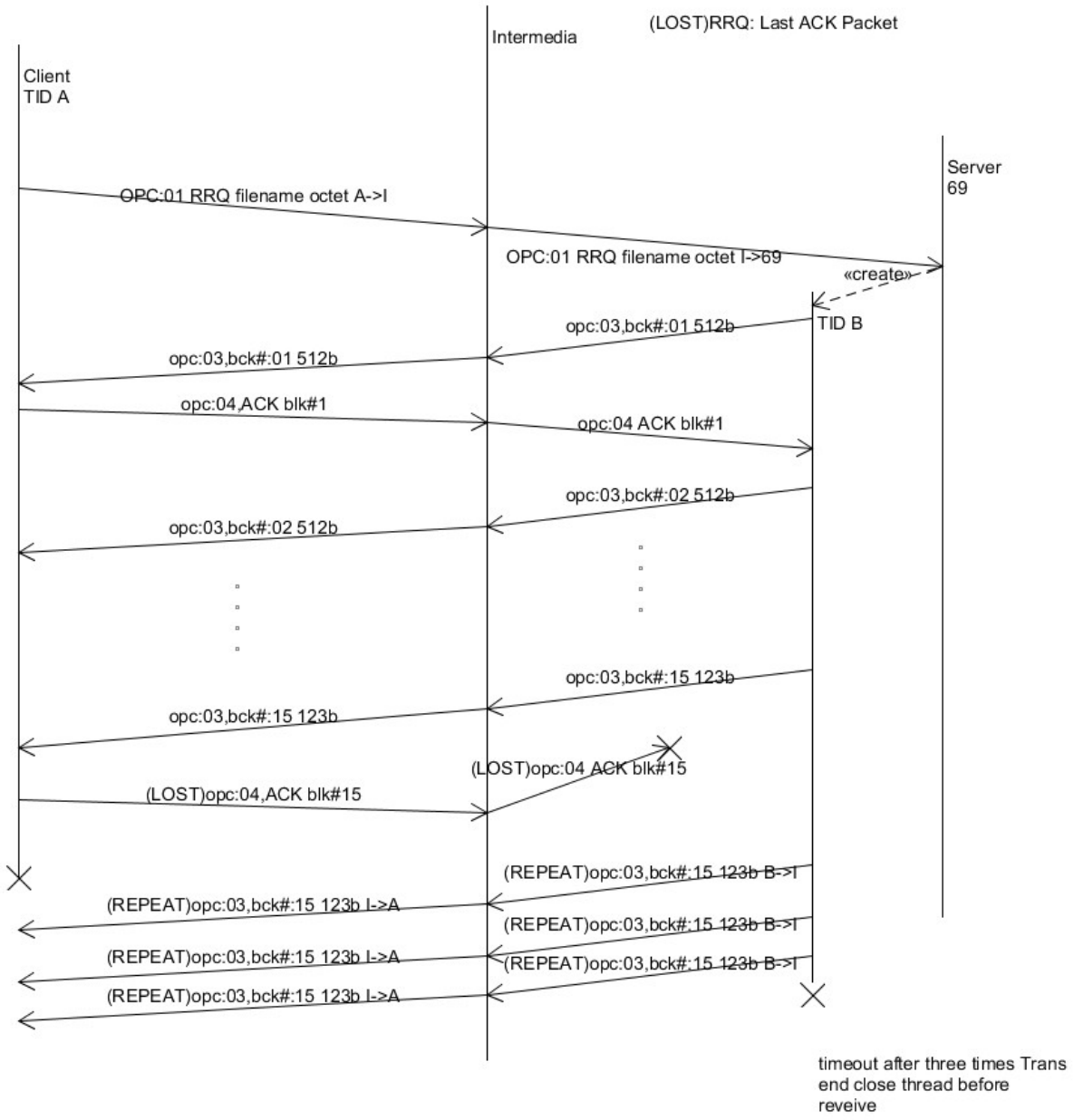


Figure 24: LOST last ACK packet—RRQ

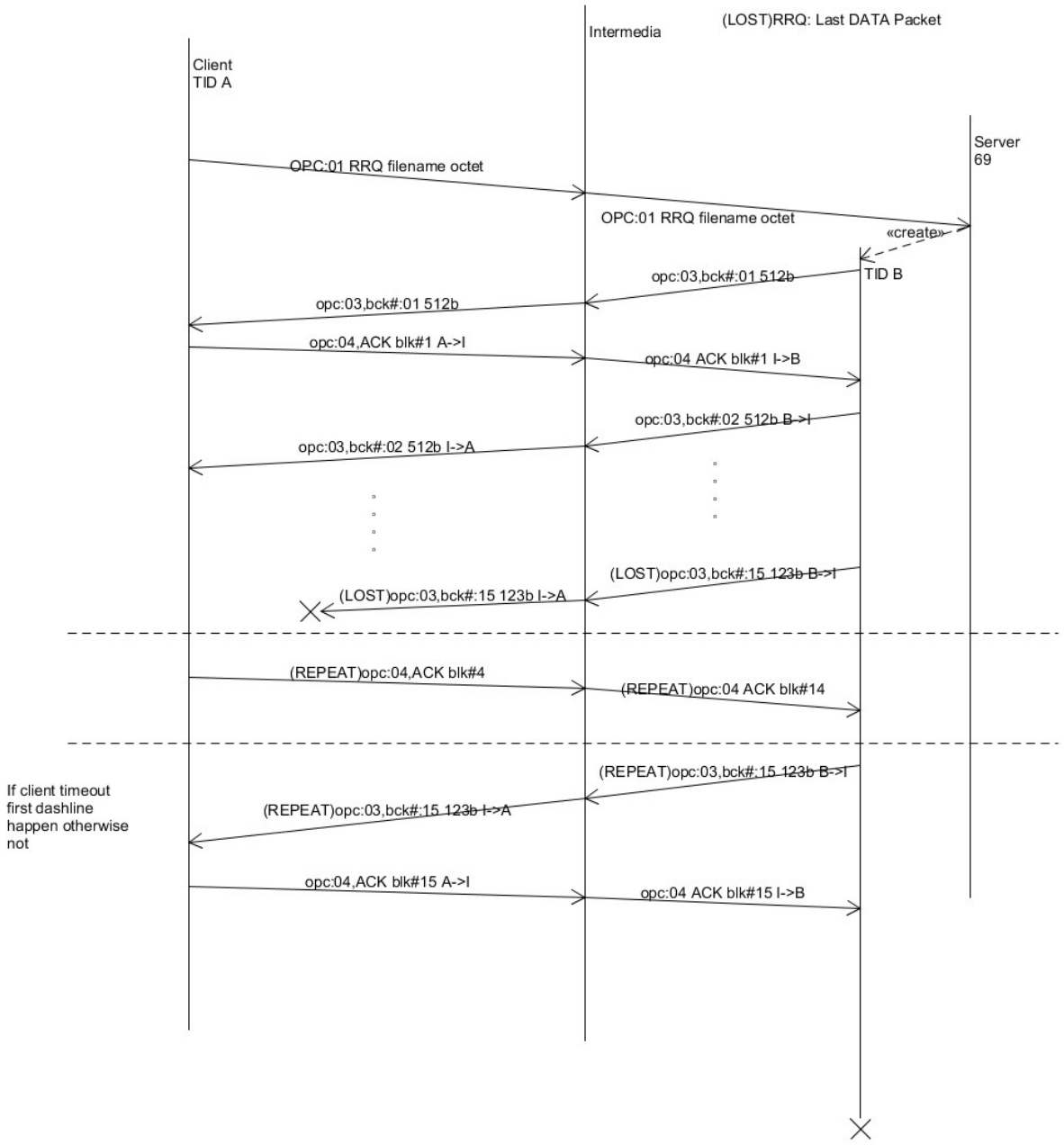


Figure 25: LOST last DATA packet –RRQ

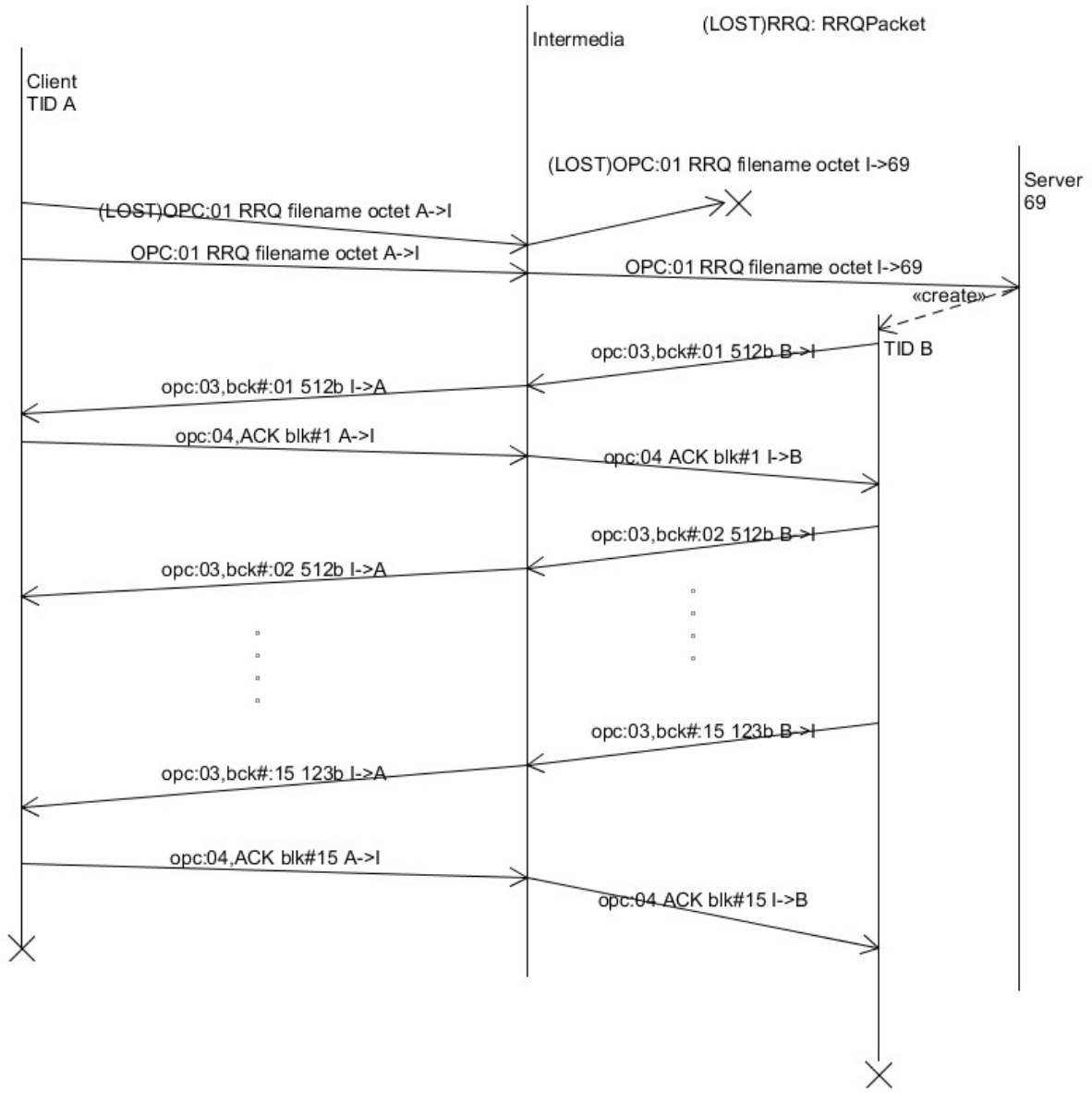


Figure 26:LOST RRQ

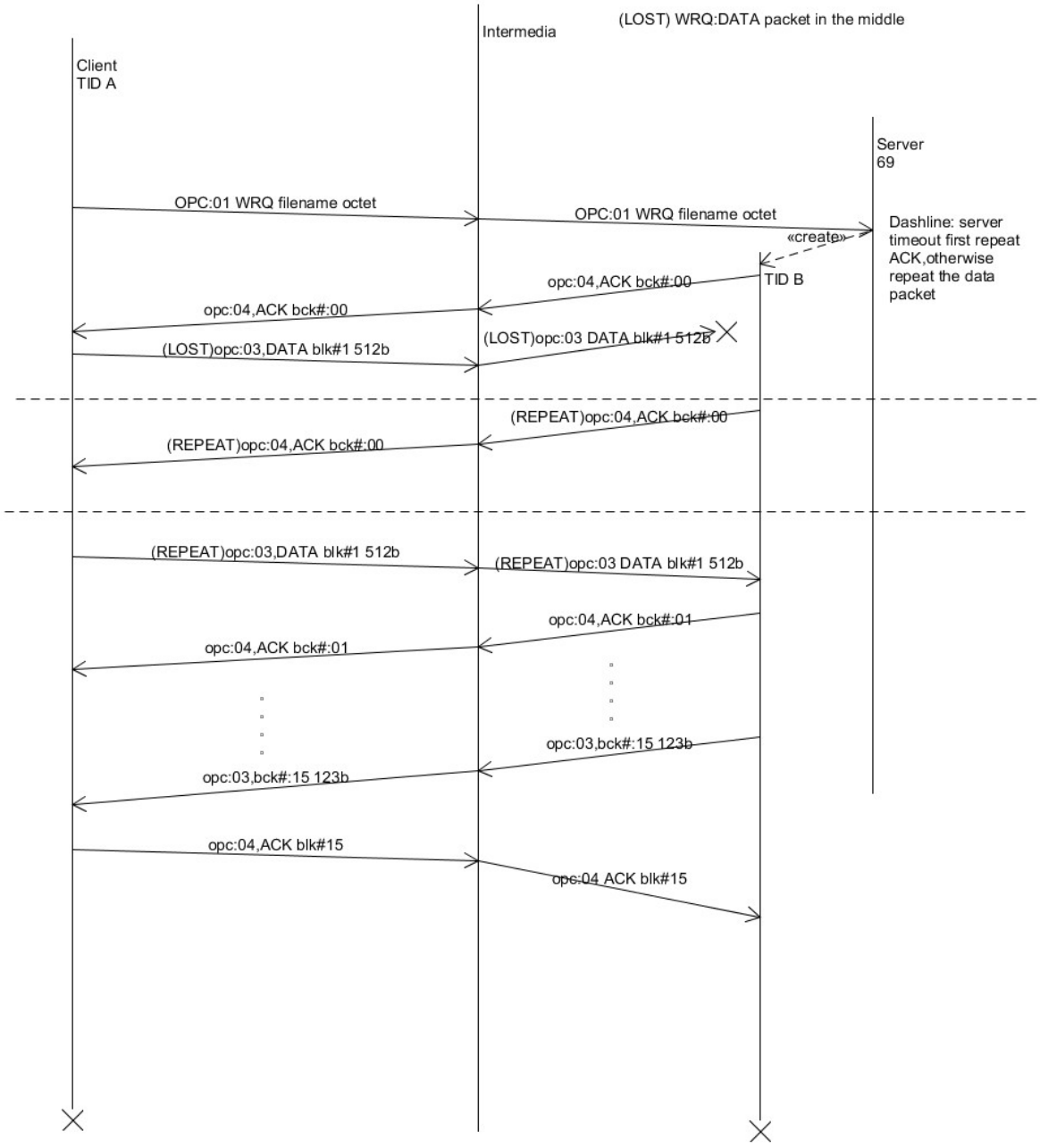


Figure 27: LOST DATA packet in the middle –WRQ

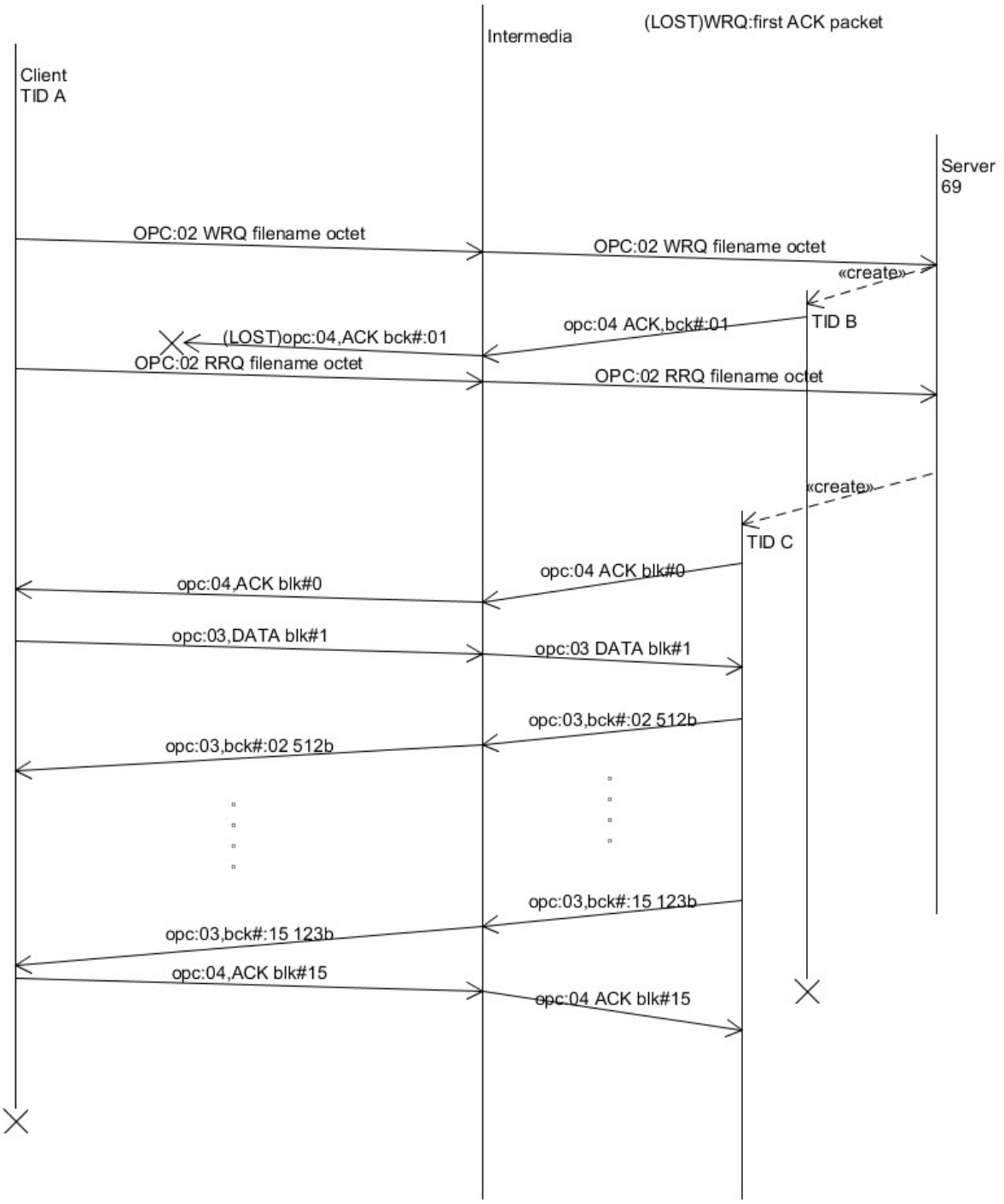


Figure 28: LOST first ACK—WRQ

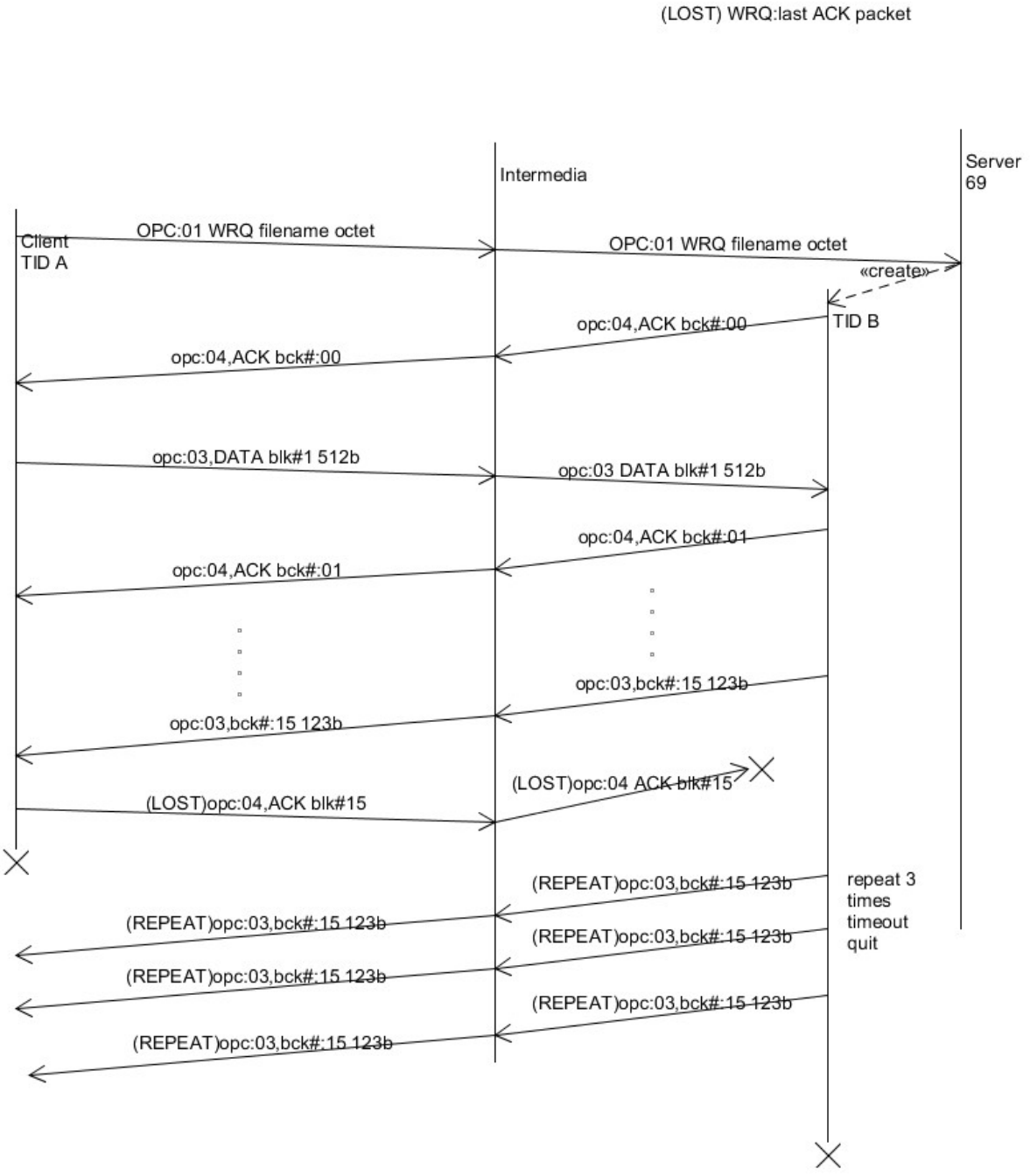


Figure 29: LOST last ACK—WRQ

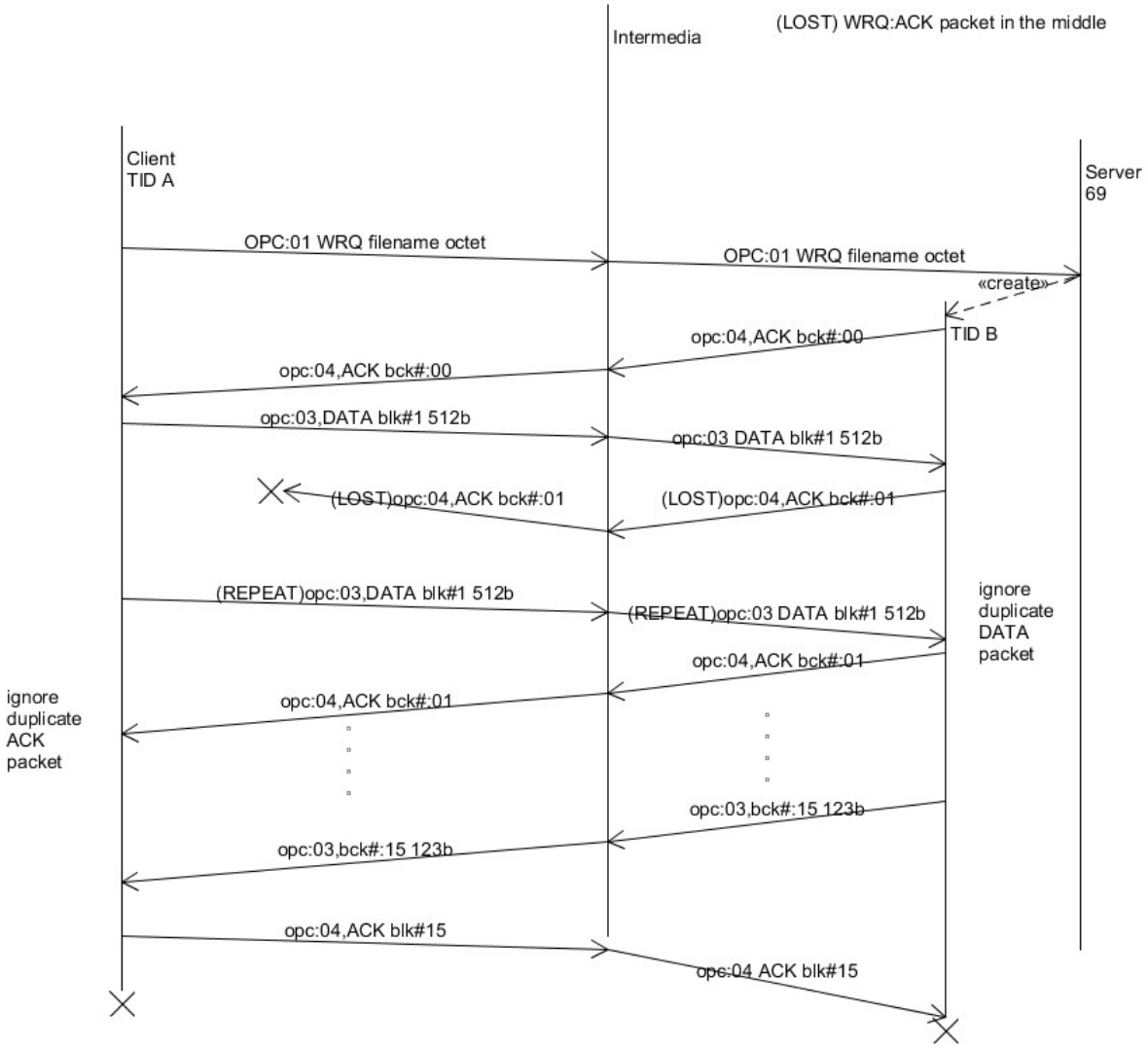


Figure 30: LOST last DATA packet –RRQ

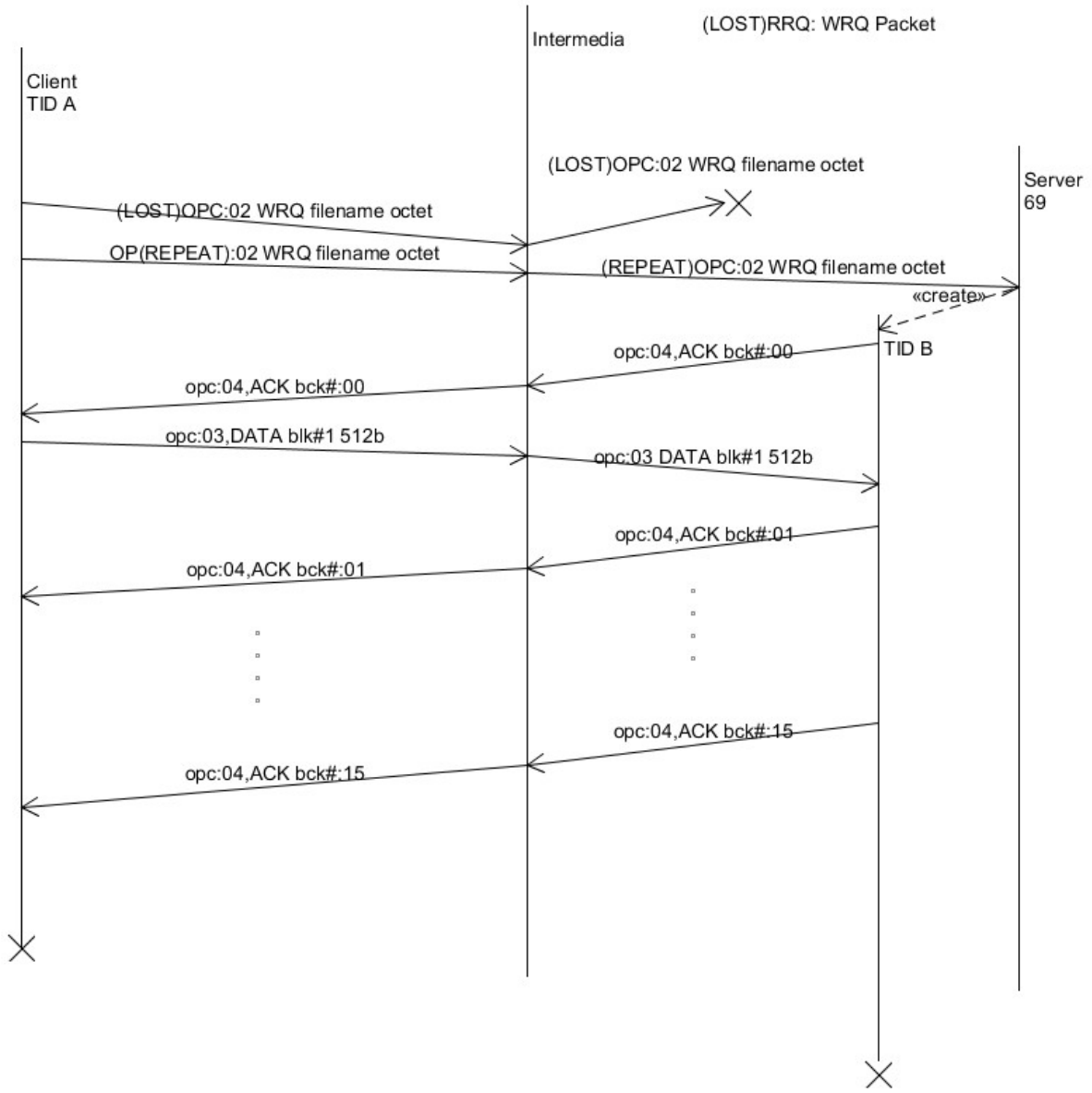


Figure 31: LOST WRQ –WRQ

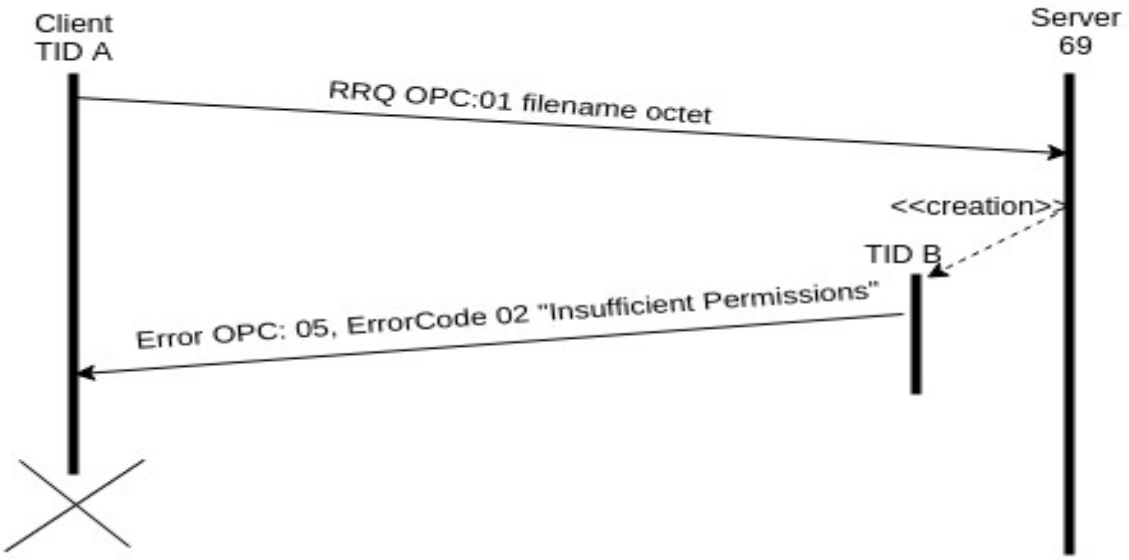


Figure 32: access Violation _ RRQ

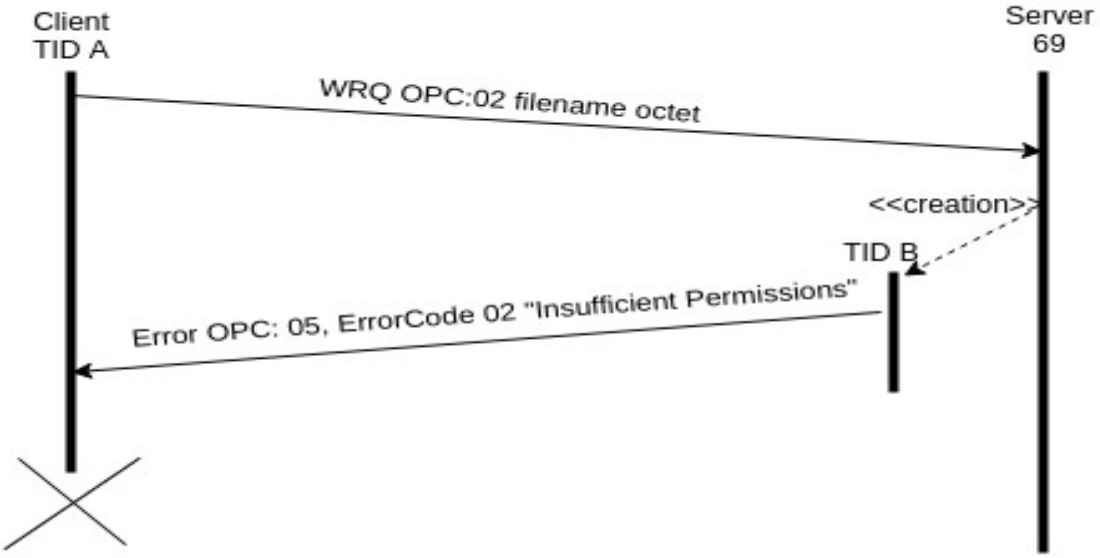


Figure 33: access Violation _ WRQ

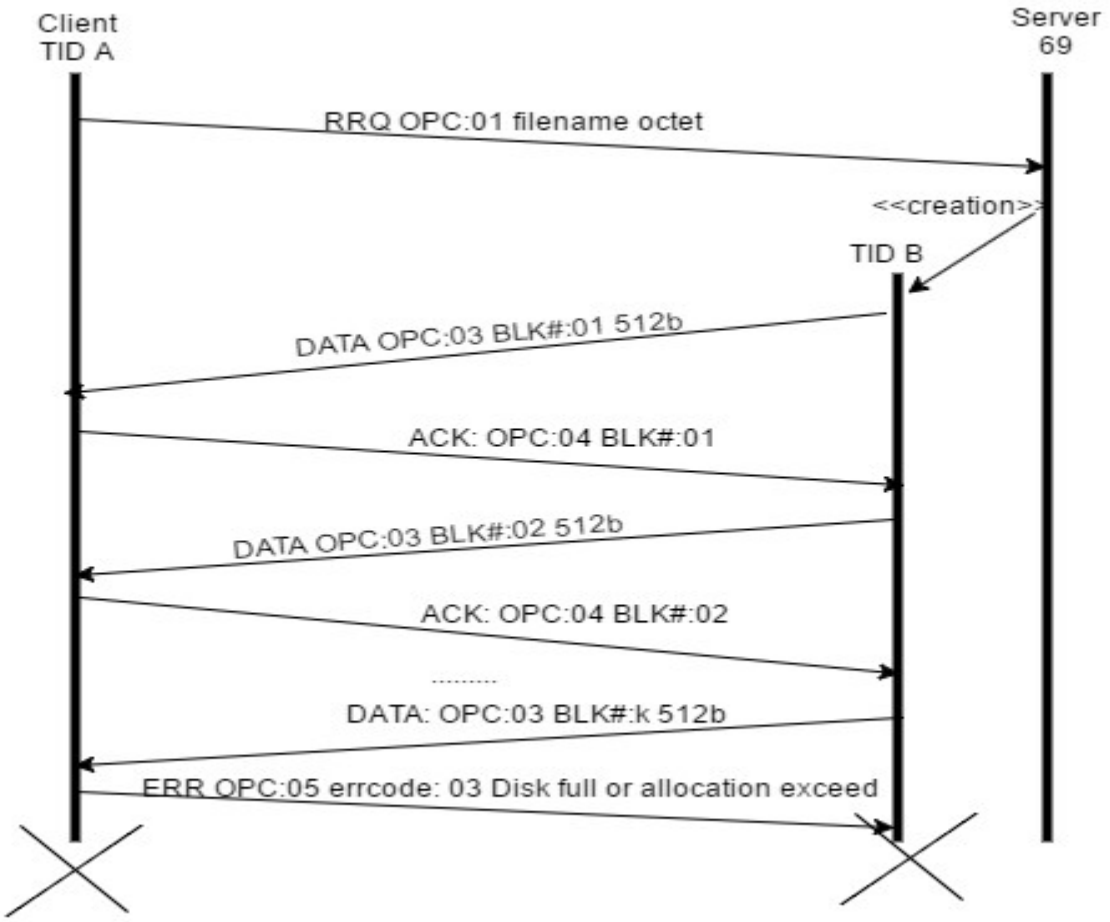


Figure 34: disk Full Or Allocation Exceed _ RRQ

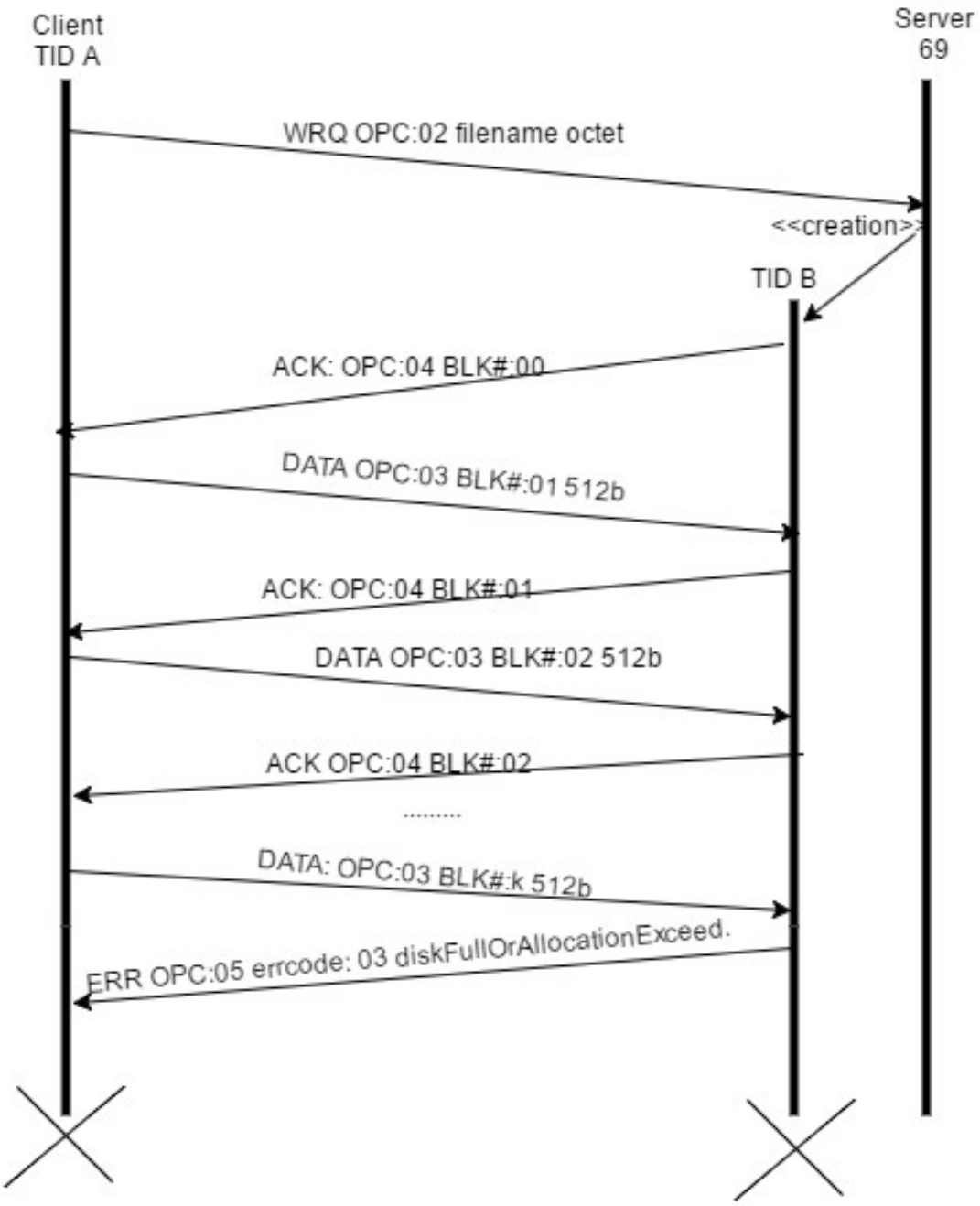


Figure 35:disk Full Or Allocation Exceed _ WRQ

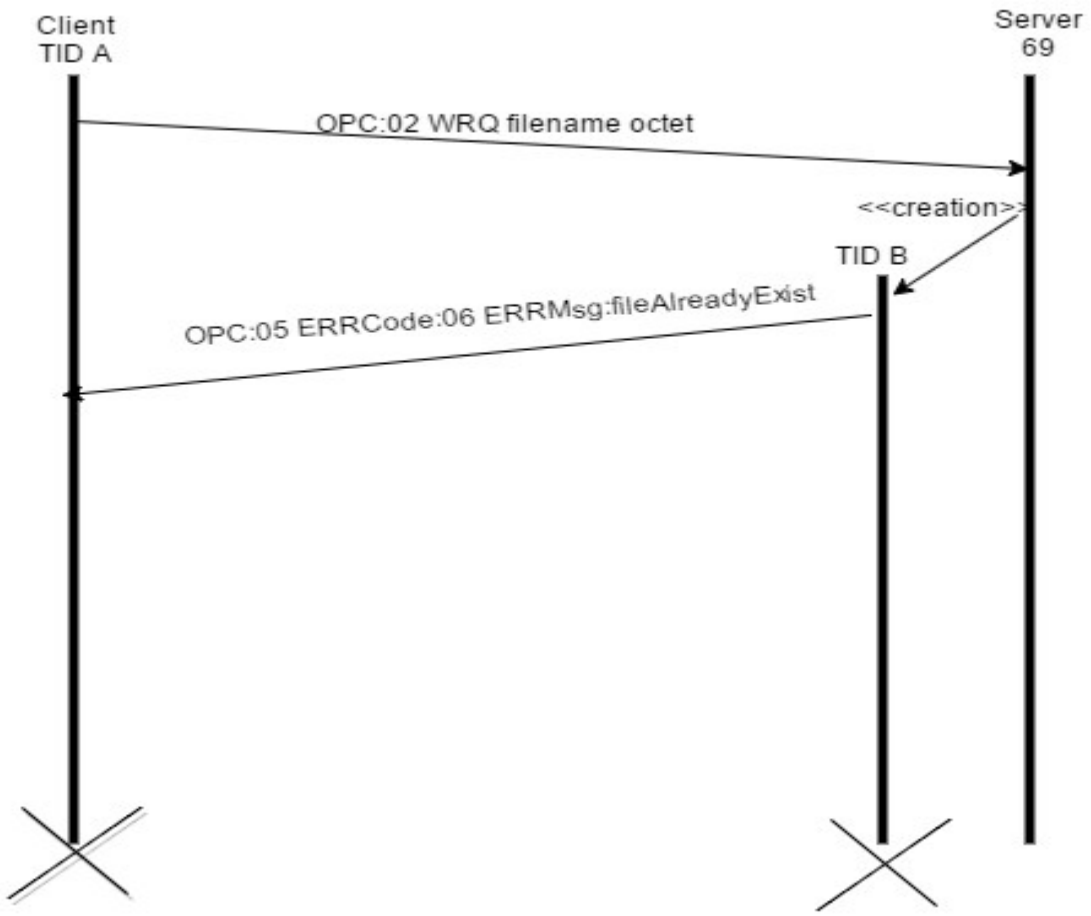


Figure 36: file Already Exis t_ WRQ

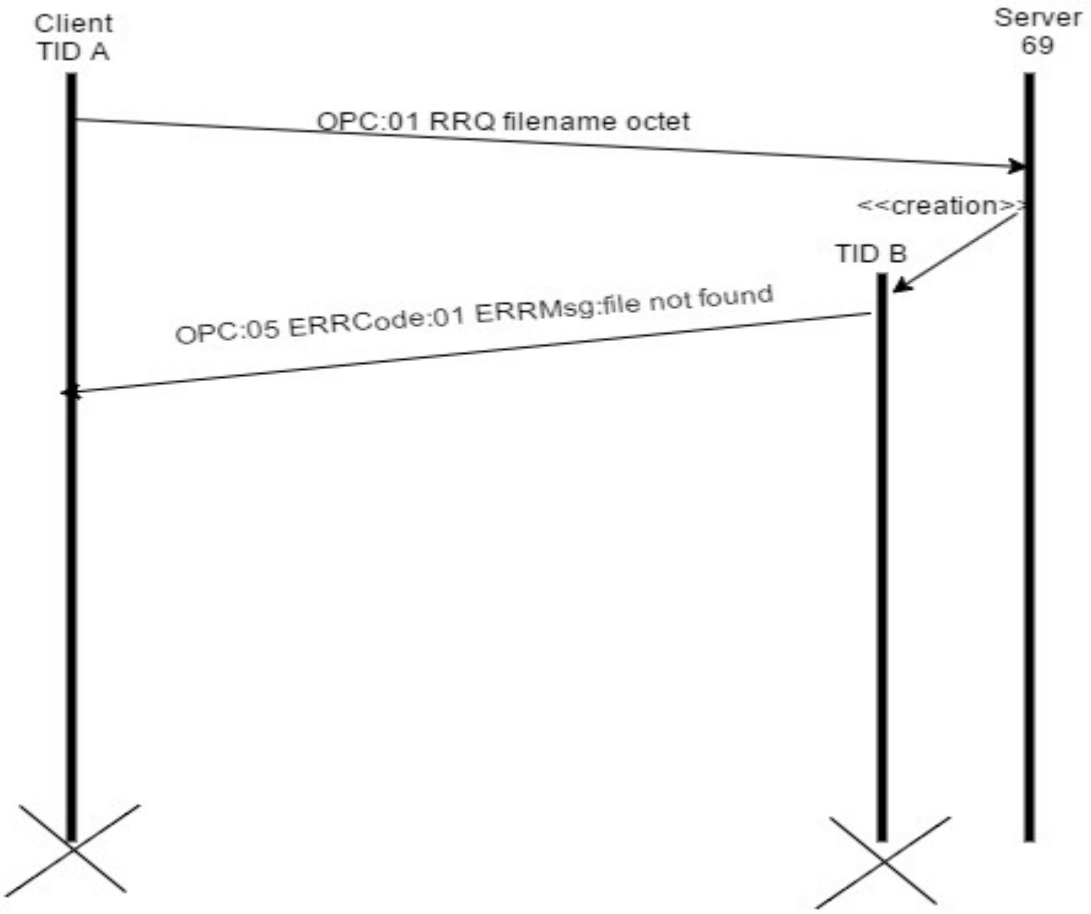


Figure 37: file Not Found _ RRQ

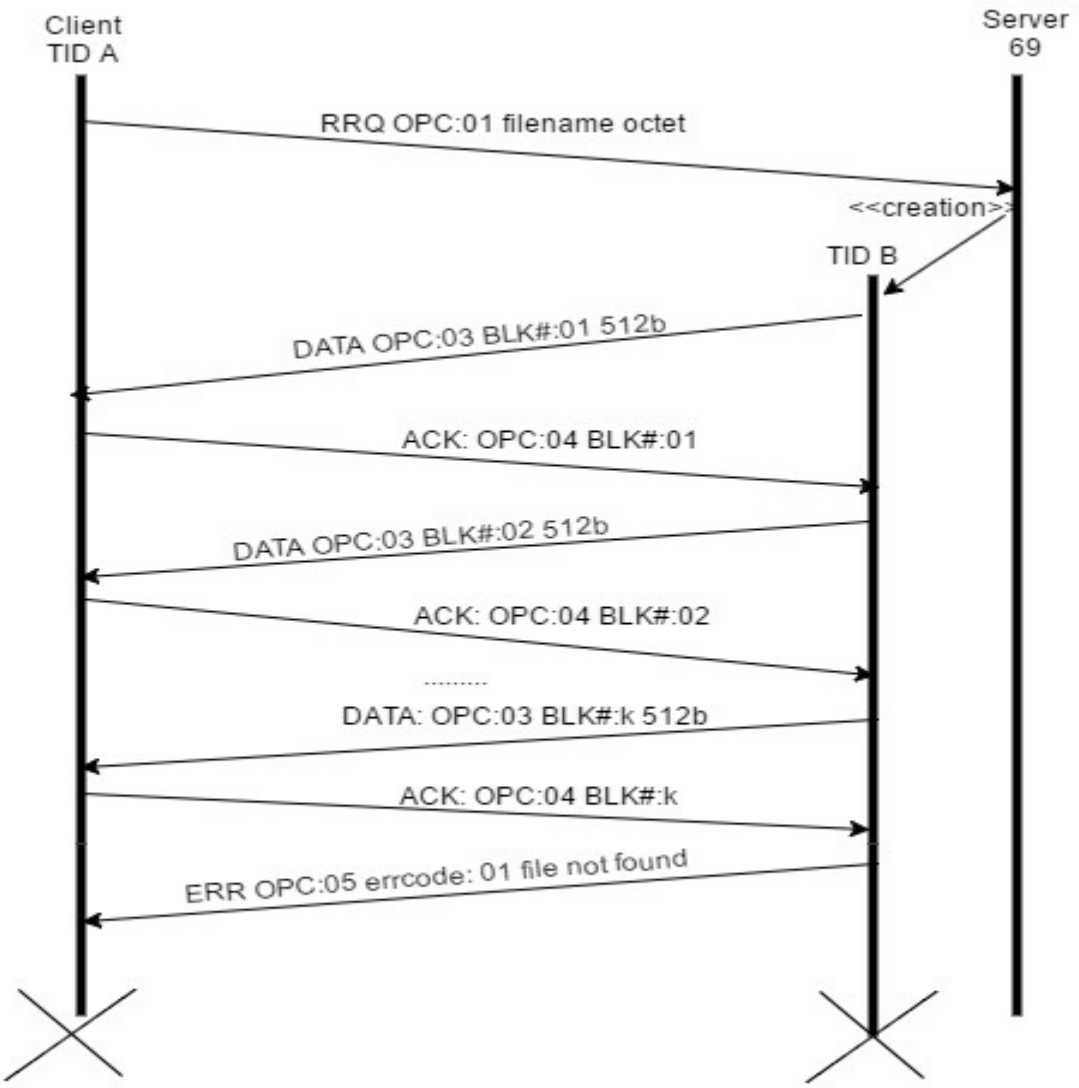


Figure 38: file not found _ RRQ2

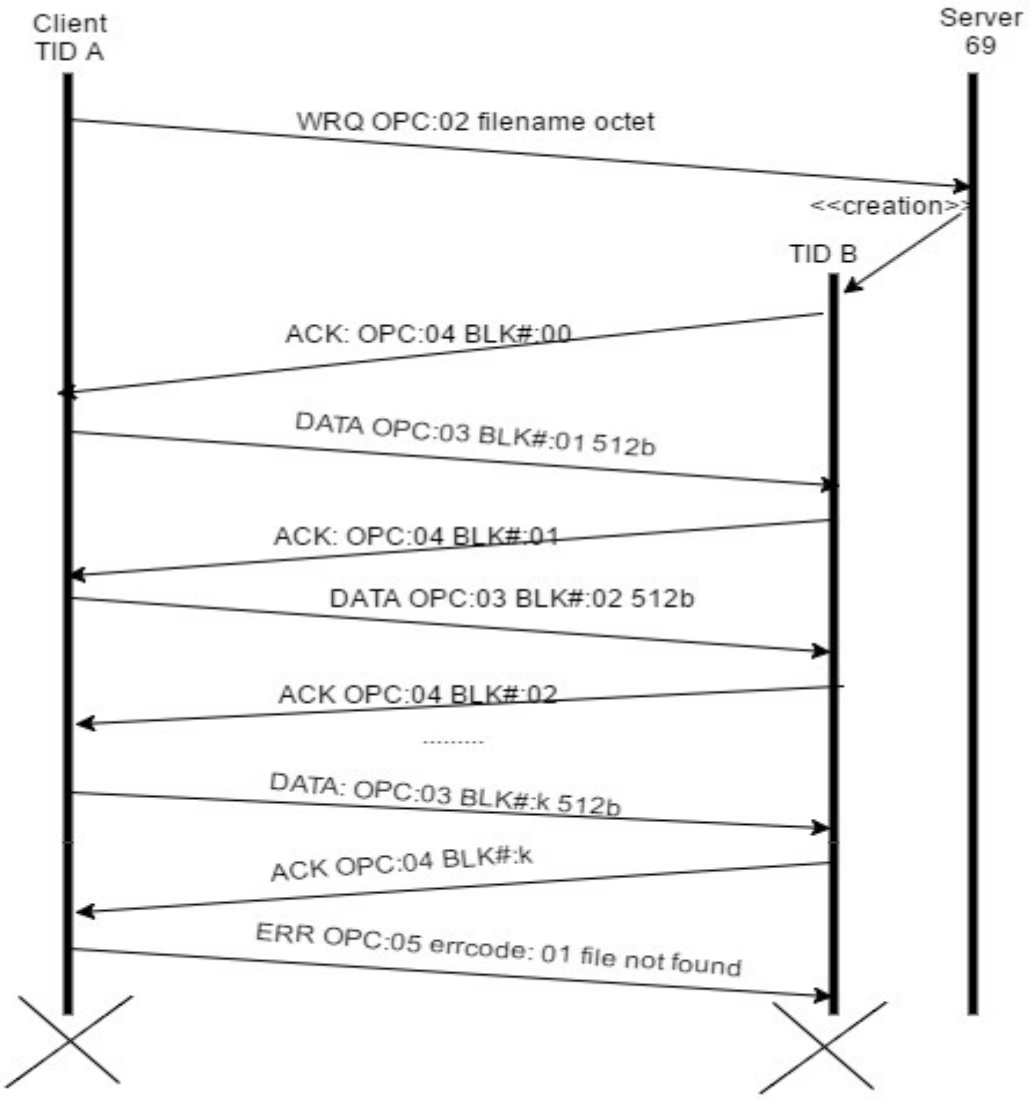


Figure 39: file Not Found _ WRQ

UML Class Diagrams

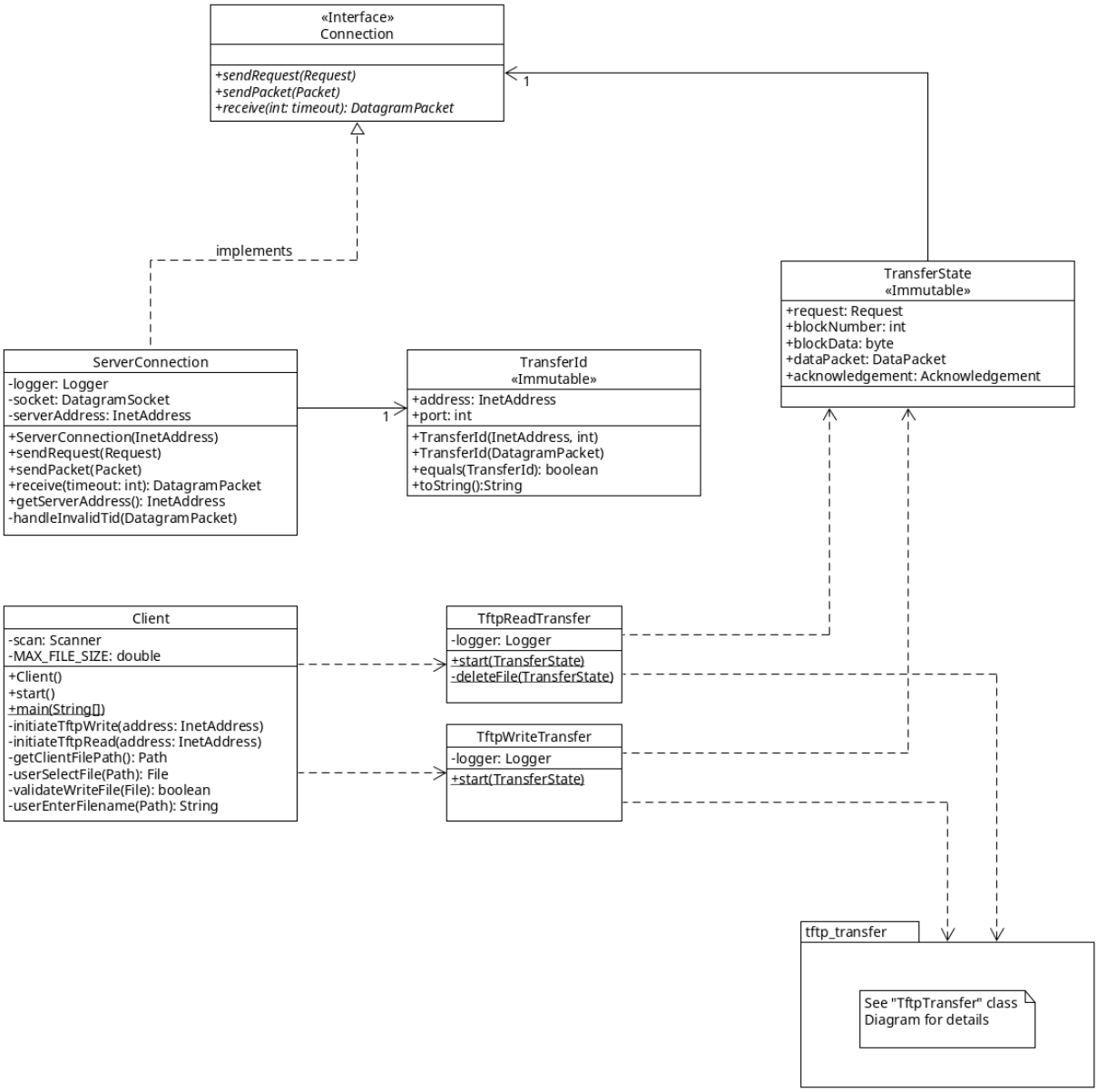


Figure 40: Client

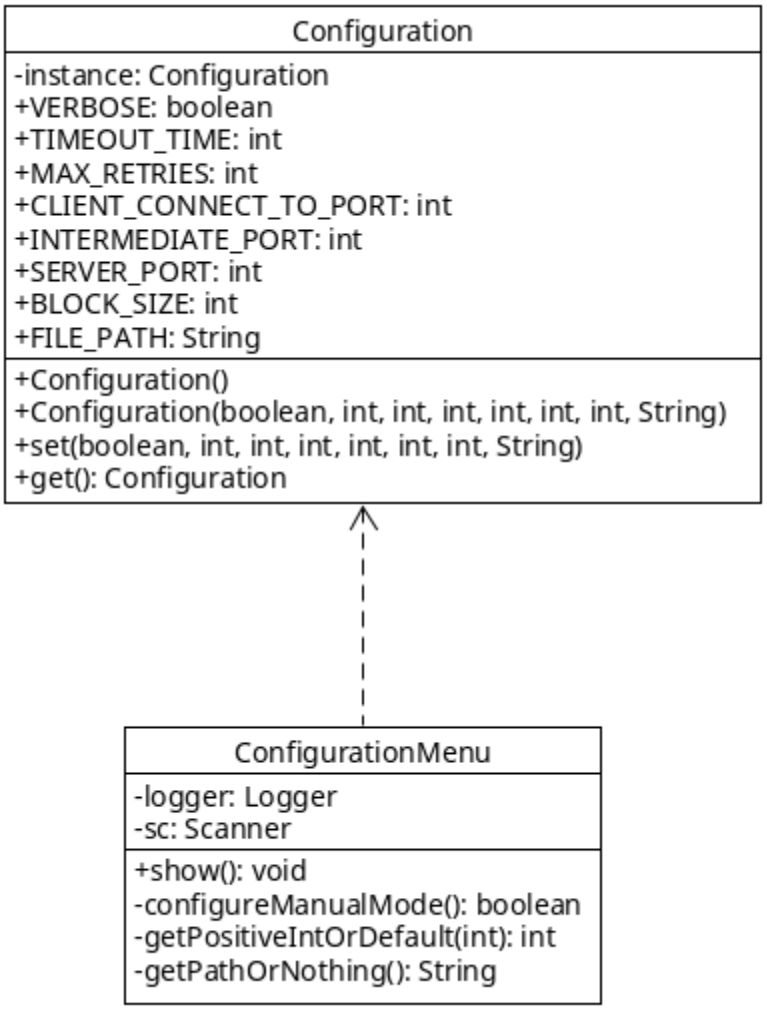


Figure 41: Configuration

Figure 42: FileIO

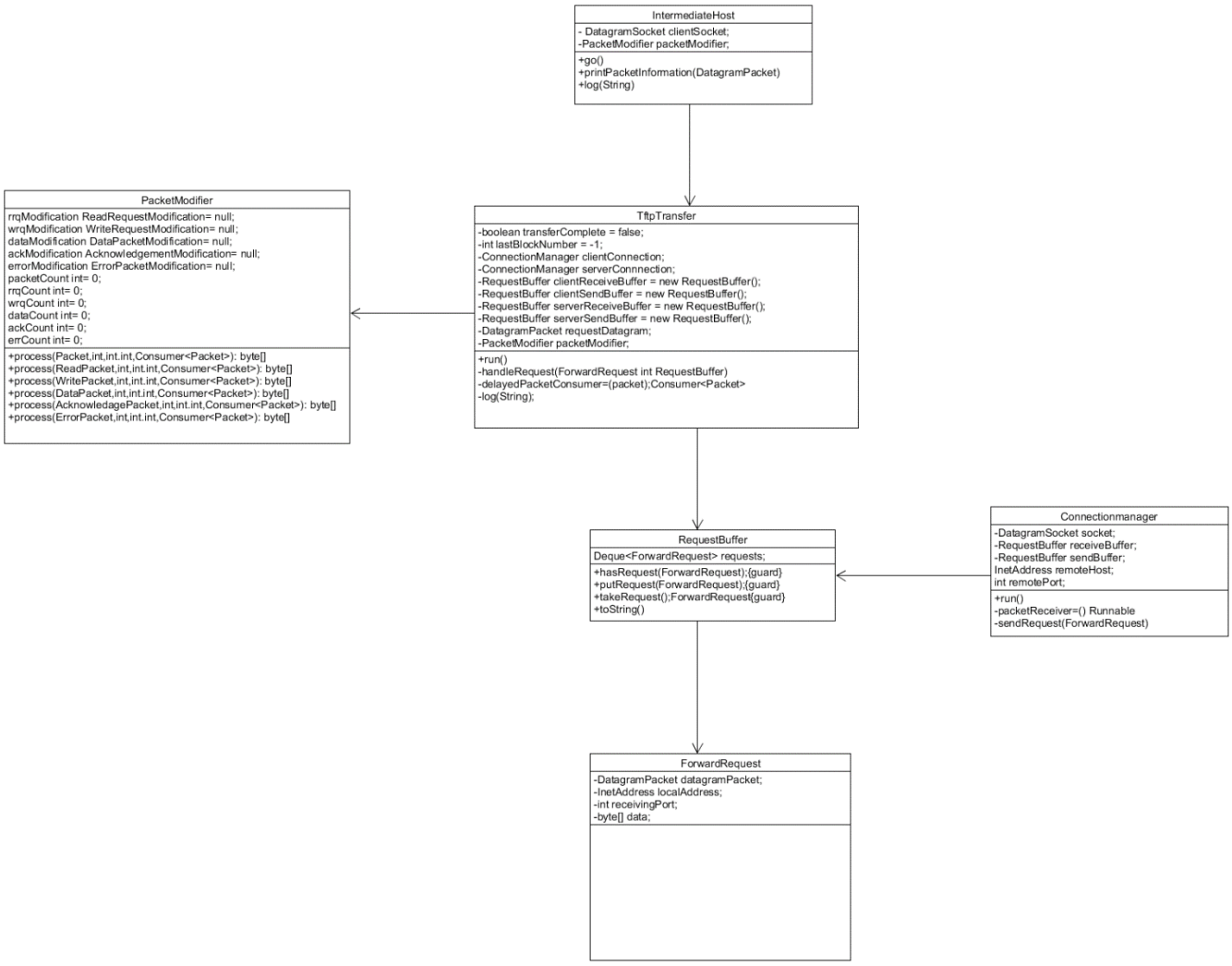


Figure 43: Intermediate Host

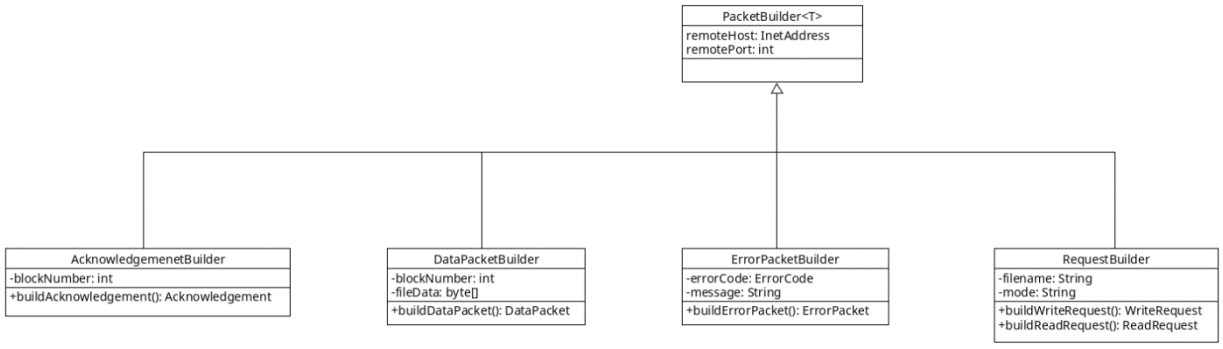


Figure 44: Packet Builder

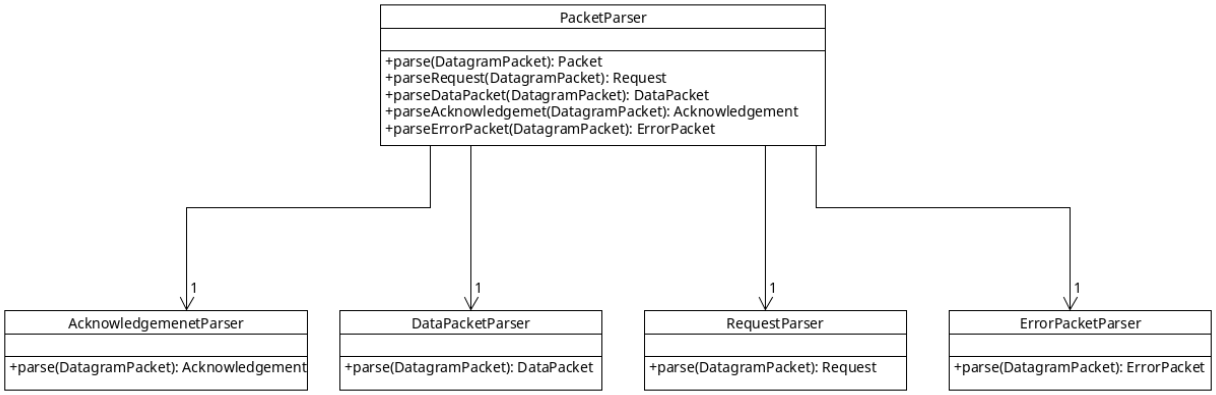


Figure 45: Packet Parser

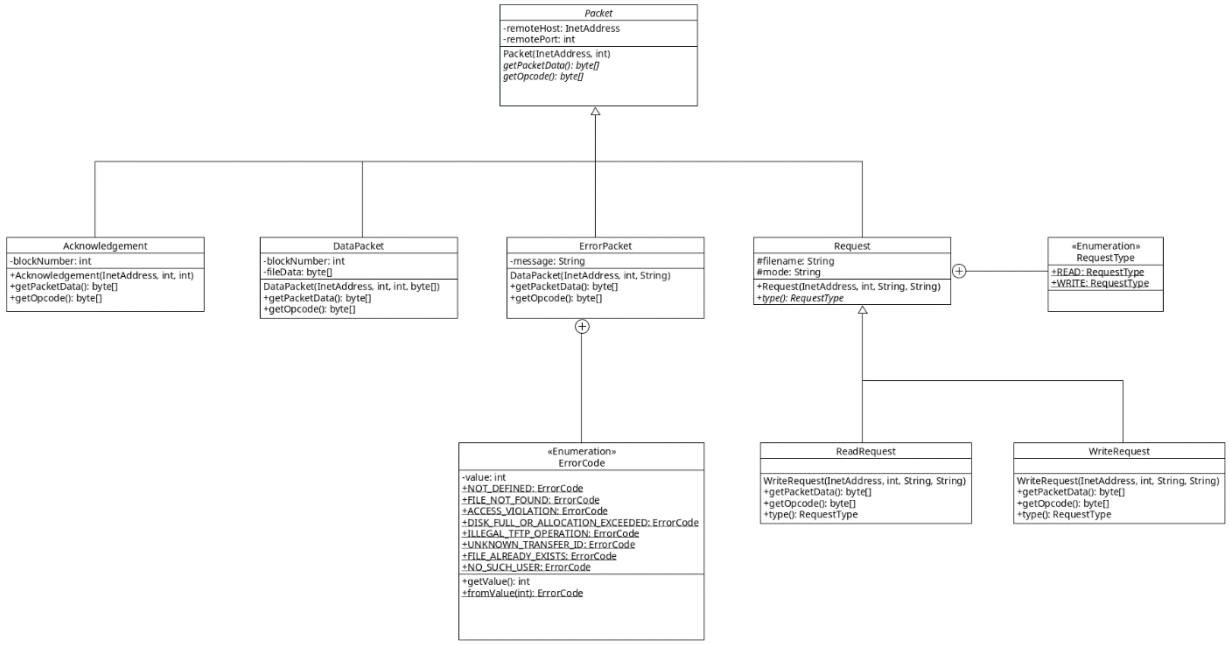


Figure 46: Packets

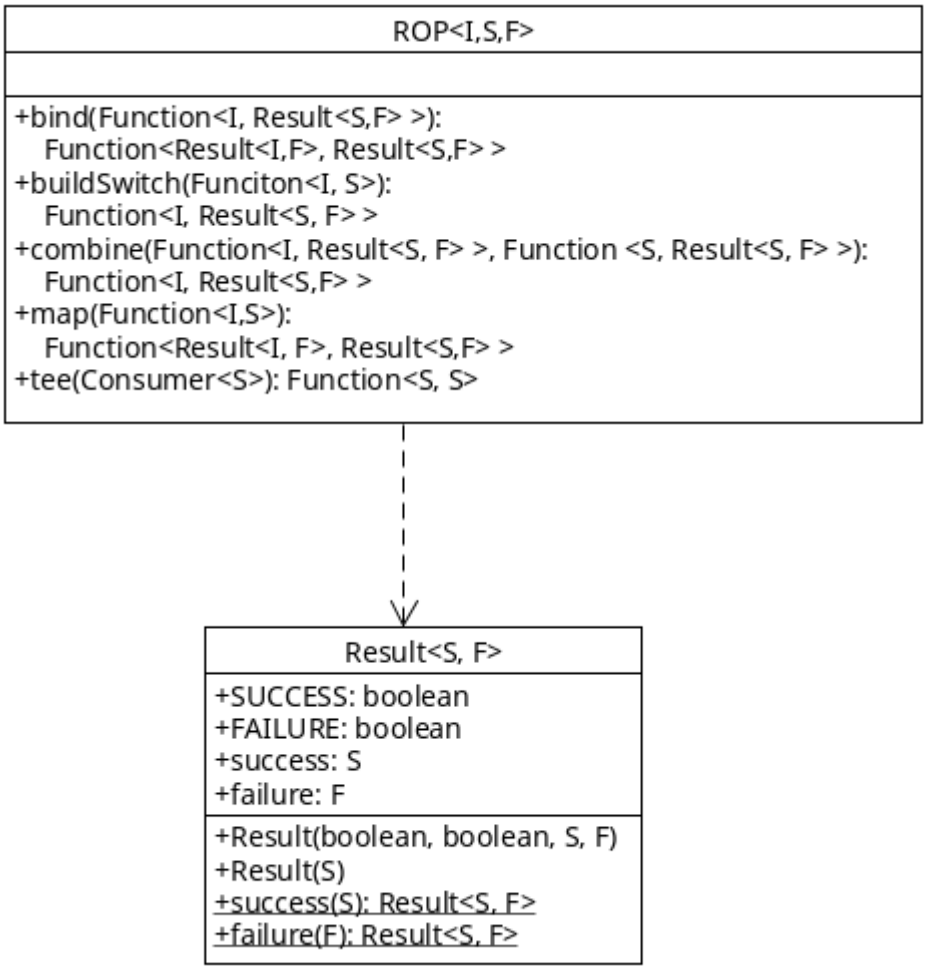


Figure 47: ROP

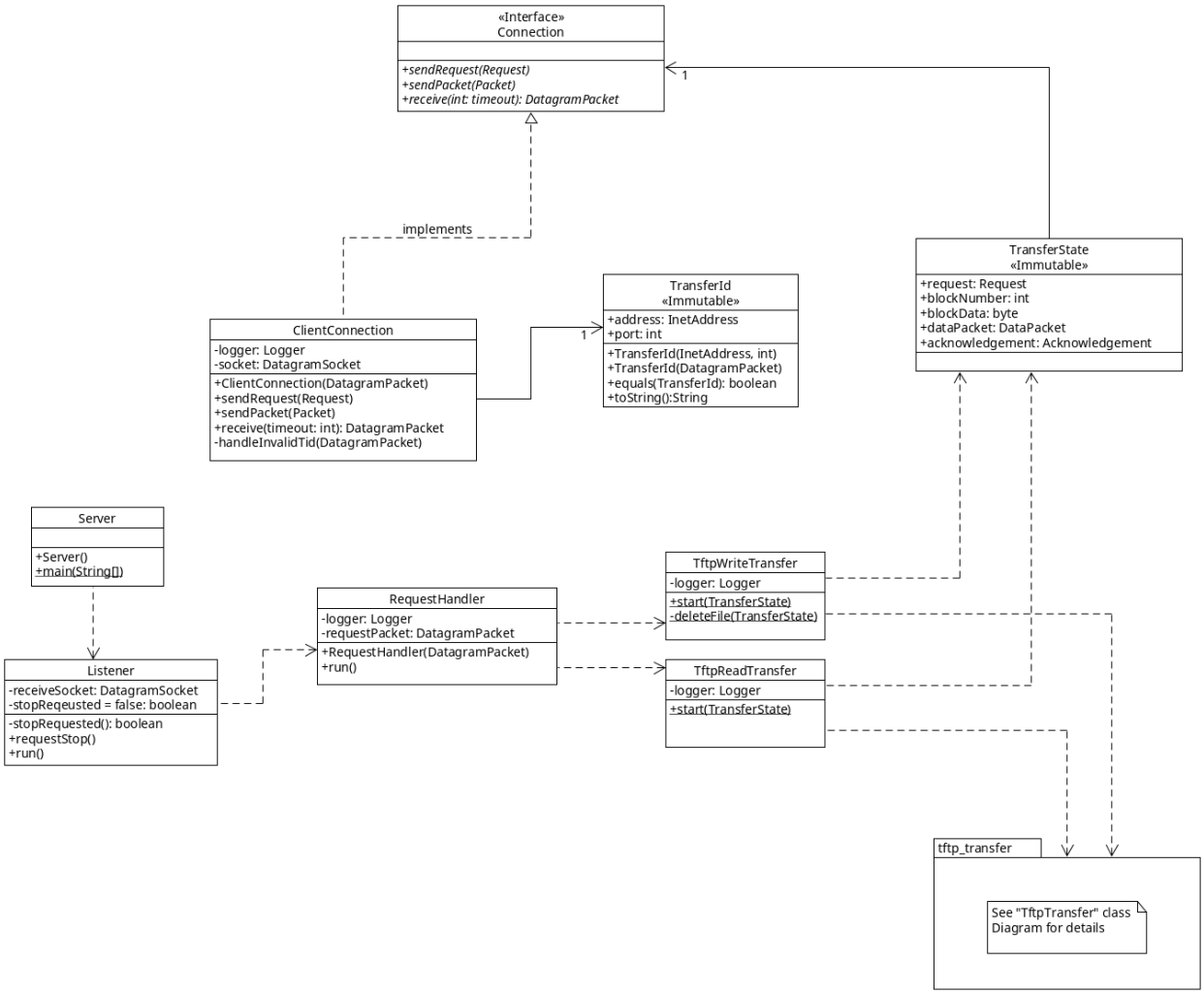


Figure 48: Server

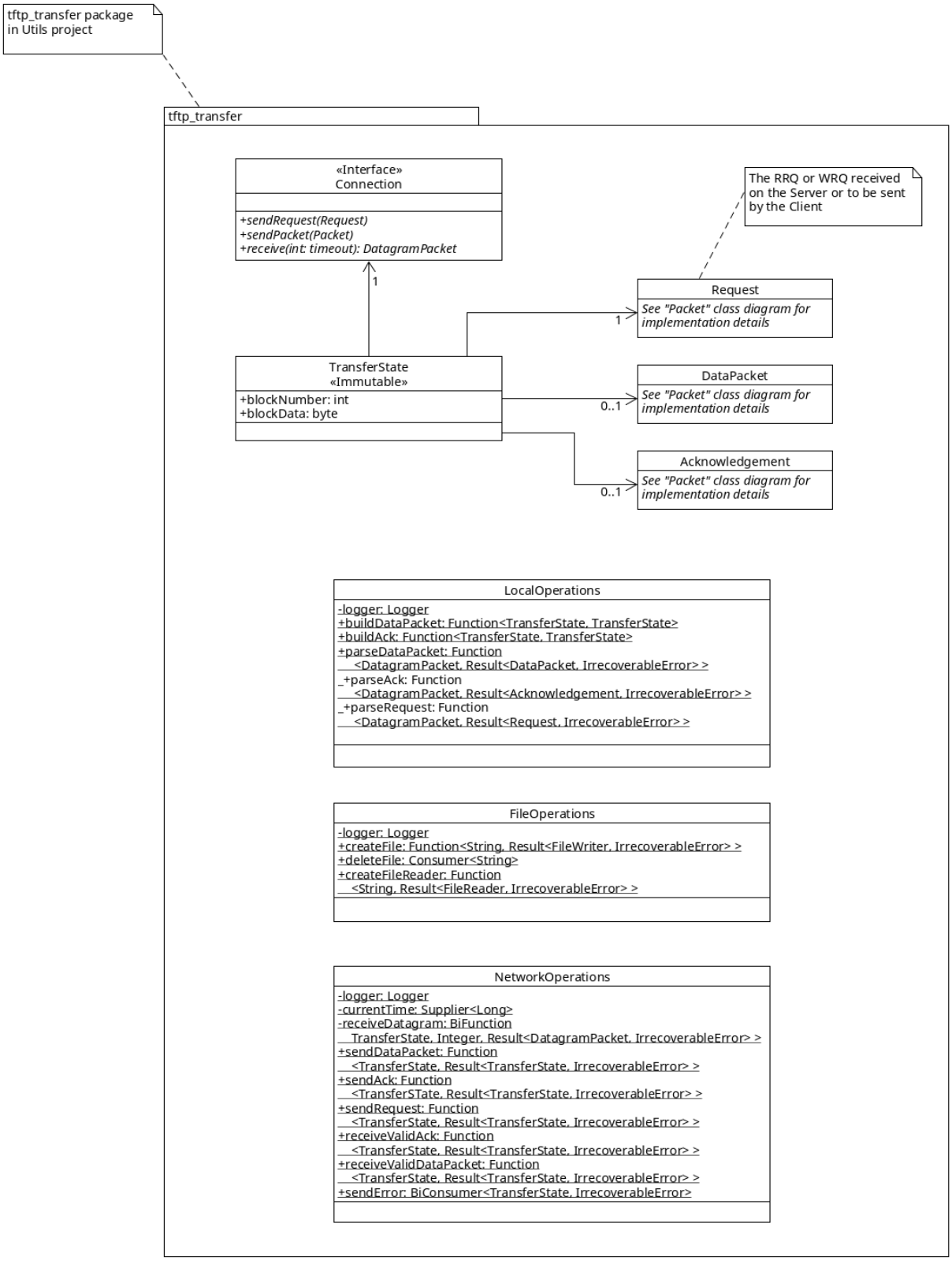

tftp_transfer package
in Utils project
tftp_transfer
«Interface»
Connection
+sendRequest(Request)
+sendPacket(Packet)
+receive(int: timeout): DatagramPacket
The RRQ or WRQ received on the Server or to be sent by the Client
1
Request
See "Packet" class diagram for implementation details
1
TransferState
«Immutable»
+blockNumber: int
+blockData: byte
DataPacket
See "Packet" class diagram for implementation details
0..1
Acknowledgement
See "Packet" class diagram for implementation details
0..1
LocalOperations
-logger: Logger
+buildDataPacket: Function<TransferState, TransferState>
+buildAck: Function<TransferState, TransferState>
+parseDataPacket: Function
<DatagramPacket, Result<DataPacket, IrrecoverableError> >
_+parseAck: Function
<DatagramPacket, Result<Acknowledgement, IrrecoverableError> >
_+parseRequest: Function
<DatagramPacket, Result<Request, IrrecoverableError> >
FileOperations
-logger: Logger
+createFile: Function<String, Result<FileWriter, IrrecoverableError> >
+deleteFile: Consumer<String>
+createFileReader: Function
<String, Result<FileReader, IrrecoverableError> >
NetworkOperations
-logger: Logger
-currentTime: Supplier<Long>
-receiveDatagram: BiFunction
<TransferState, Integer, Result<DatagramPacket, IrrecoverableError> >
+sendDataPacket: Function
<TransferState, Result<TransferState, IrrecoverableError> >
+sendAck: Function
<TransferSTate, Result<TransferState, IrrecoverableError> >
+sendRequest: Function
<TransferState, Result<TransferState, IrrecoverableError> >
+receiveValidAck: Function
<TransferState, Result<TransferState, IrrecoverableError> >
+receiveValidDataPacket: Function
<TransferState, Result<TransferState, IrrecoverableError> >
+sendError: BiConsumer<TransferState, IrrecoverableError>

Figure 49: TFTP Transfer

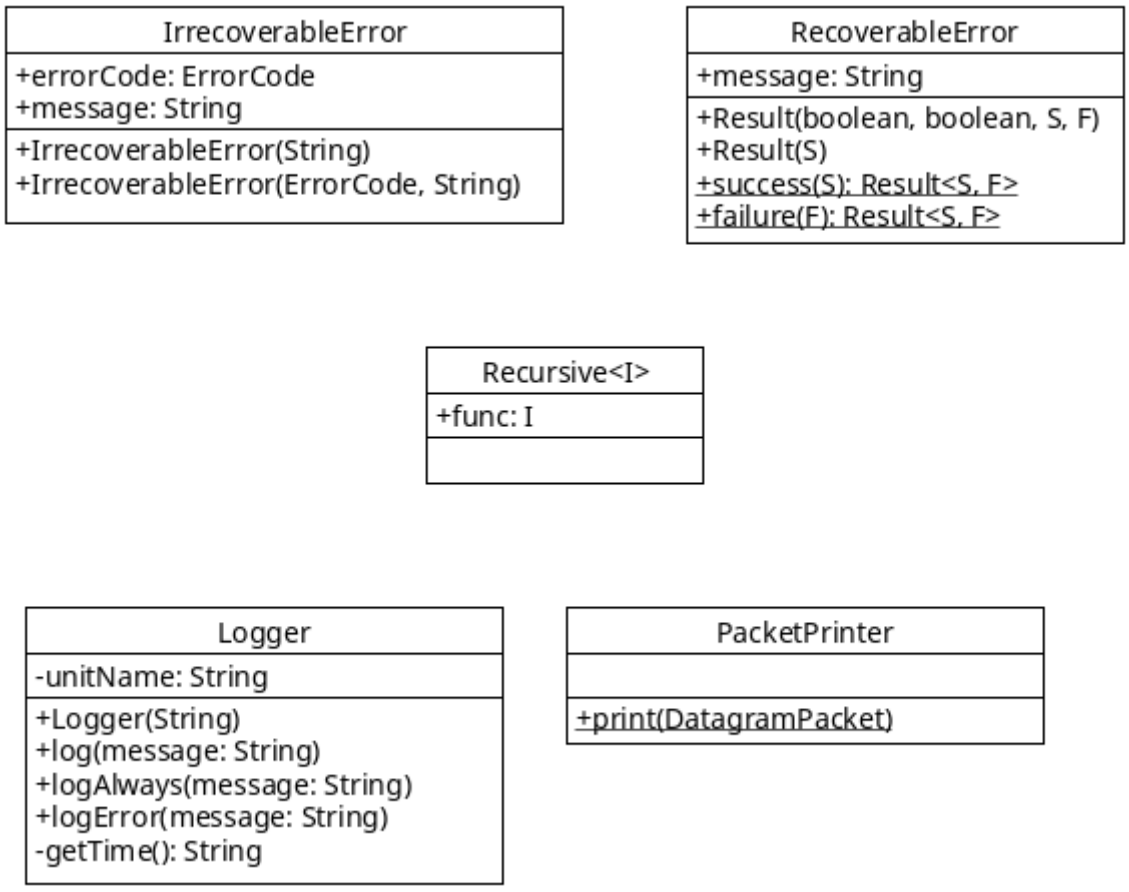


Figure 50: Utils

Readme

Files

- ReadMe.txt - this file
- Code/
  - Client/ - *Code for Client*
  - IntermediateHost/ - *Code for Intermediate Host*
  - Server/ - *Code for Server*
  - Utils/ - *Common code between Server, Client, Intermediate Host*
- 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 timing diagrams (iteration 4)*

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)

Select a Configuration for Client/Server/IntermediateHosr

Upon start the Client/Server/IntermediateHost will ask which configuration mode to use. The mode should always be the same in all three programs, except when using Manual mode, which allows to modify all configuration values for a particular program.

Modes

Debug Mode: Verbose output, use Intermediate Host

- Verbose logging output
- 5000ms socket timeout
- Make 3 attempts to re-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 Intermediate Host

- Verbose logging output
- 5000ms socket timeout
- Make 3 attempts to re-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 Intermediate Host

- Very little logging
- 5000ms socket timeout

- Make 3 attempts to re-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 re-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

Configure the Intermediate Host

After selecting a configuration 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.

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.

Included test files

- one-block 510 bytes
- one-block-exactly 512 bytes
- two-blocks 1020 bytes
- two-blocks-exactly 1024 bytes
- three-blocks 1530 bytes
- three-blocks-exactly 1536 bytes
- four-blocks 2040 bytes
- four-blocks-exactly 2048 bytes
- large-file 51000 bytes
- huge-file 33551360 bytes
- too-big 33556480 bytes

- access-restricted-file 2040 bytes

Error Scenarios & Testing

Note:
The test files are identical on both the client and server, so you may need to delete some of them on either side to test successful transmissions.

Testing Setup on separate machines

1) On one machine start the Server and select a configuration

2) On a second machine start the Client and Intermediate Host

- in the Client: enter 127.0.0.1 as the Server IP (or just hit enter)

- in the Intermediate Host: enter the IP of the first machine as the server IP

Concurrent File Access

The server and client allow only one Thread at a time to read or write from a file. This is a design decision we made to prevent any problems with concurrent file access. Thus, if one client is writing a file "x" to the server, and then another client tries to read the file "x" while it is still being written, the server will send an error code 2 (Access Violation) to the second client. This is implemented using File locks.

The same happens if two clients try to read from the same file on the server. Only one client can read from a file, the second one will receive an error code 2.

This has some implications:
When duplicating a RRQ or WRQ there is a subtle race condition that cannot be avoided. The server will simultaneously open two threads that both try to read or write to the same file. One of the threads will succeed, while the other will fail to acquire a lock on the file. The first thread will send a DataPacket or ACK to the client, the second thread will send an error code 2

(Access Violation) to the client. If the client receives the DataPacket/ACK first, it will continue the transfer and ignore the ErrorPacket from the second thread. However, if the client receives the ErrorPacket first, it will abort the transfer completely.

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 WRQ 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 that some file data is buffered by the JVM and the OS, so for small files this will not occur, since the required data may 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.

Other 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 on Iteration four.

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 behaviour 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 ,...) 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 behaviour, and moving the behaviour required for a transfer into small re-usable functions, we were able to break out all of that behaviour 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. 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 behaviour 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 concept used in Functional Programming that can aid in error handling. Essentially, all functions return a Result object 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 the functions 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/