# **ASSIGNMENT 2: TRIVIAL FILE TRANSFER PROTOCOL (TFTP)**

# **REPORT**

# **22359** [**afd22@sussex.ac.uk**](mailto:afd22@sussex.ac.uk) **G5115**

1. **Introduction**

*“TFTP is a very simple protocol used to transfer files. It is from this that its name comes, Trivial File Transfer Protocol or TFTP. Each nonterminal packet is acknowledged separately.”[[1]](#footnote-1)*

The objective of this assignment is to implement a simplified version of the TFTP Protocol based on RFC 1350[[2]](#footnote-2), one that supports only octet mode, handles only a FILE\_NOT\_FOUND error, and without support for error handling in data duplicates. The TFTP server implemented shall be able to handle read and write requests with multiple clients simultaneously with the protocol built on top of UDP i.e. connectionless. As per 1350 requirements, the master port of the Server side is port 69 which shall receive all read and write requests from clients, which will itself generate one slave socket with a port number between 1024 and 49151 inclusive (ports below 1024 are the Well Known Ports and above 49151 are Dynamic ports, both unsuitable for stable usage). The default data size for each DATA packet contents is 512 bytes as per 1350, not including space taken by the opcode and block number (4 bytes). The maximum size of the transmitted file is about 32 MB since each DATA packet is 512 bytes and there are only (255 \* 256 + 255 + 1 = 65536) block numbers available. This document shall be used in conjunction with the comments written in the source codes to explain the logic behind the code design.

1. **Client-Side**

**Front End**

Each client can only communicate with one server at a time. The Client shall start by prompting the user for (1) the main request whether a write or read request, (2) the Internet address of the remote Server which can be the DEFAULT loopback address 127.0.0.1, (3) the intended port number of this Client, accepting any number between 1024 and 49151 inclusive, and (4) the name of the file, including name extension, in request e.g. bob.txt.

**Back End – Read Request**

What follows is a series of messages in the command prompt about the status of the transfer, as seen in *Figure 2.1* if the request is a read request (RRQ) and the file requested is bob and bob.txt. With the first DATA packet, received by the Client, containing the first DEFAULT\_DATA\_SIZE (512 bytes) of the file successfully sent, the first acknowledgment packet (ACK 1) should be sent to the Server with block number matching the received DATA block number. This signals the Client is ready to accept subsequent DATA blocks. Being aware of the possibility of packet losses (in this program, losses are simulated using a random number generator that randomly skips DatagramSocket.send(), the default value LOSS\_PROBABILITY being 0.0 for no losses), an ACK will be sent again if a duplicate block is received until the Server sends the next block.

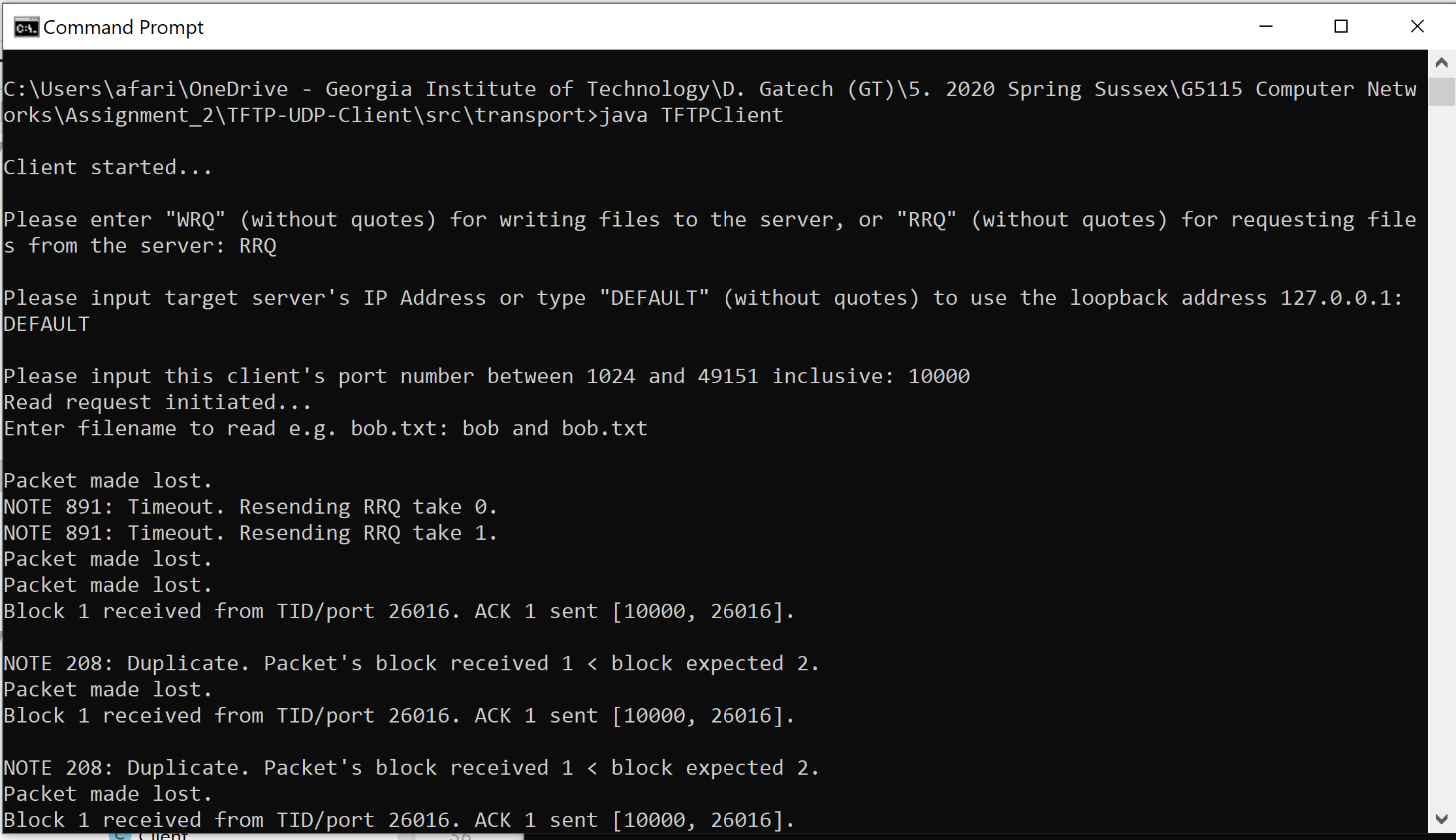


Figure 2.1 Command Prompt of Client Status

This process repeats until the last DATA block is received (content size of 0 to 511 bytes). The last ACK will be sent with the arrival of the final DATA block (the first and its duplicates) until timeout of the Client’s socket occurs at 10 \* TIMEOUT (default value of TIMEOUT is 100 ms), the code shown in *Figure 2.2*. This practice of waiting after sending the final ACK is called “dallying”.



Figure 2.2 Client undergoing timeout at the end of a read request.

**Back End – Write Request**

In an opposite scenario, the Client may ask for a write request (WRQ) to write files onto the Server. The 4 inputs are requested and verified, and like an RRQ, the WRQ is sent repeatedly (up to CLIENT\_BRAKE or 100 times when it is presumed Server is permanently unresponsive) until the Server acknowledges the request with the first ACK (block number 0) to write the file. This initial process is shown in *Figure 2.3* with Queen Liz.txt as the file to be written. After each DATA block sent, the respective ACK will be expected; the same DATA block will be sent again after each TIMEOUT indefinitely until the ACK is received.

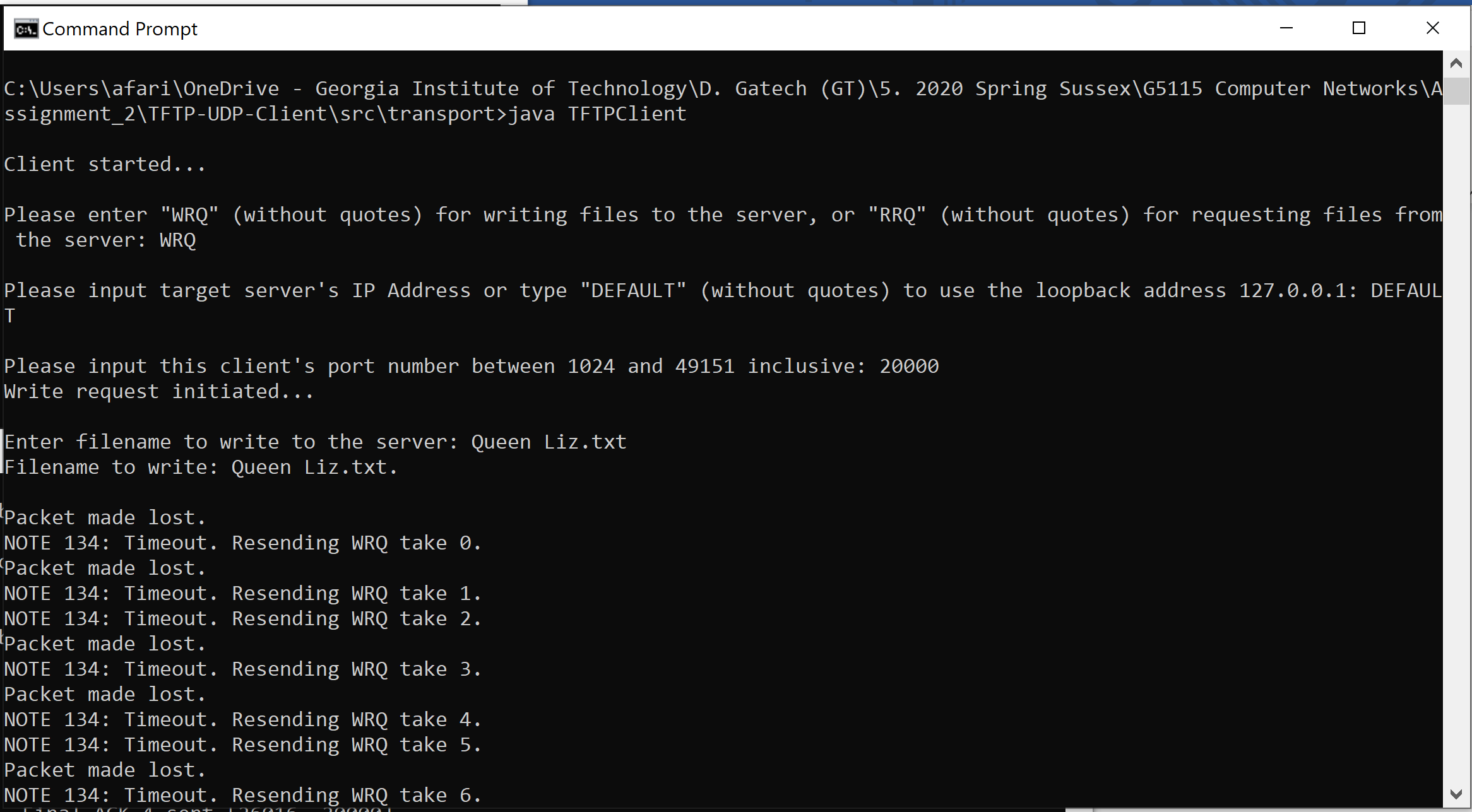
****

Figure 2.3 Initial process of a WRQ in Client-side

To send the file, the file is first broken into pieces that fit into the DATA packets. More formally, a BufferedReader is used to read buffer the contents of the file and is read 512 bytes (or less if the end of file is reached) at a time. Those 512 bytes are packed into a DATA packet, constructed with the DATA opcode and the current block number (blockNumber) to go through an unreliable-data-transfer sending process i.e. through udtSend(DatagramPacket, int, InetAddress), as shown in *Figure 2.4*. The same DATA packet will be sent after each timeout of the client socket indefinitely until the correct acknowledgment is received (ACK’s block number blockReceived equal to expected block number expectedAck). In *Figure 2.5*, receiveAck(int, DatagramPacket) acts to receive any packets that arrive in the Client socket, which also (a) verifies the source internet socket address, (b) verifies the packet is an ACK, (c) checks if the received ACK number is expected, and either (i) resend the previous DATA packet if a previous ACK is received, or (ii) returns the received ACK if the correct is received. To account the possibility the Server terminates connection before the last ACK could be received, the Client will automatically terminate after attempting to send the final DATA FINAL\_LOOP\_LIMIT or 20 times without receiving the last ACK.

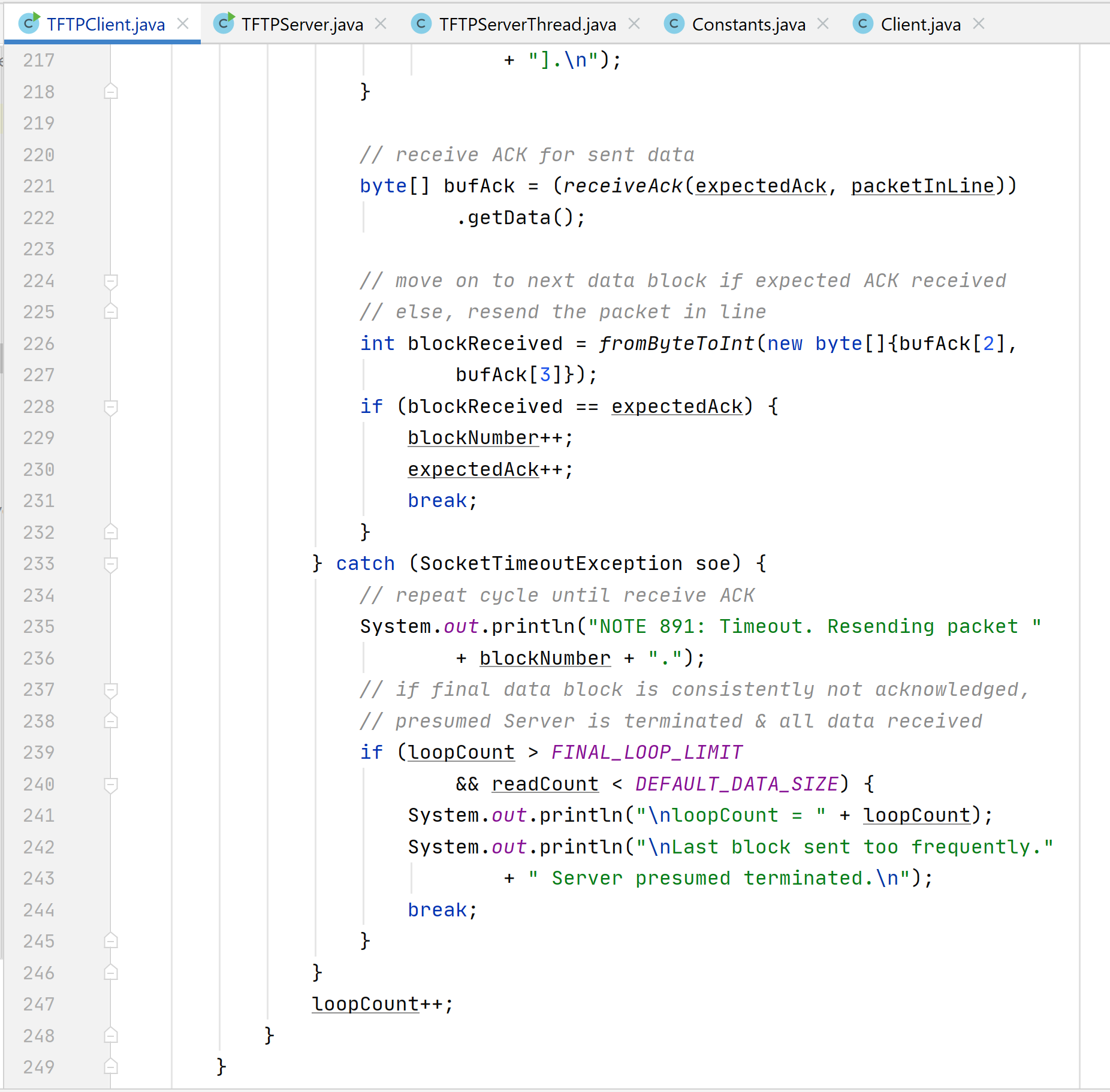


Figure 2.4 WRQ: Sending DATA

Figure 2.5 WRQ: Receiving ACK

1. **Server-Side**

**Introduction**

The Server is more complex than the Client since it is designed to be able to communicate with multiple Clients at one time. No input is requested from the user, unlike the Client side. Constant values are kept separate in Constants.java. The Server keeps track of all current Client objects, each a separate instance of the Client.java class, and their status in a HashMap<InetSocket Address, Client> named mainStatus where each Client is identified by its Internet Socket Address. Threading is used here to make two threads: the main Thread in TFTPServer.java which will forever listen and accept new requests and the secondary thread in TFTPServerThread.java which will forever iterate and loop the Client objects in mainStatus, triggering one state change in each Client before moving on to the next Client (*Figure 3.1*). If another request arrives from an address already in mainStatus*,* the request is assumed as a duplicate of the old request and ignored.

Clients are added or removed from a shadow of mainStatus called mainStatusPending. mainStatus updates the list of Clients for processing from mainStatusPending only after each Client has completed one state change to prevent a ConcurrentModificationException.

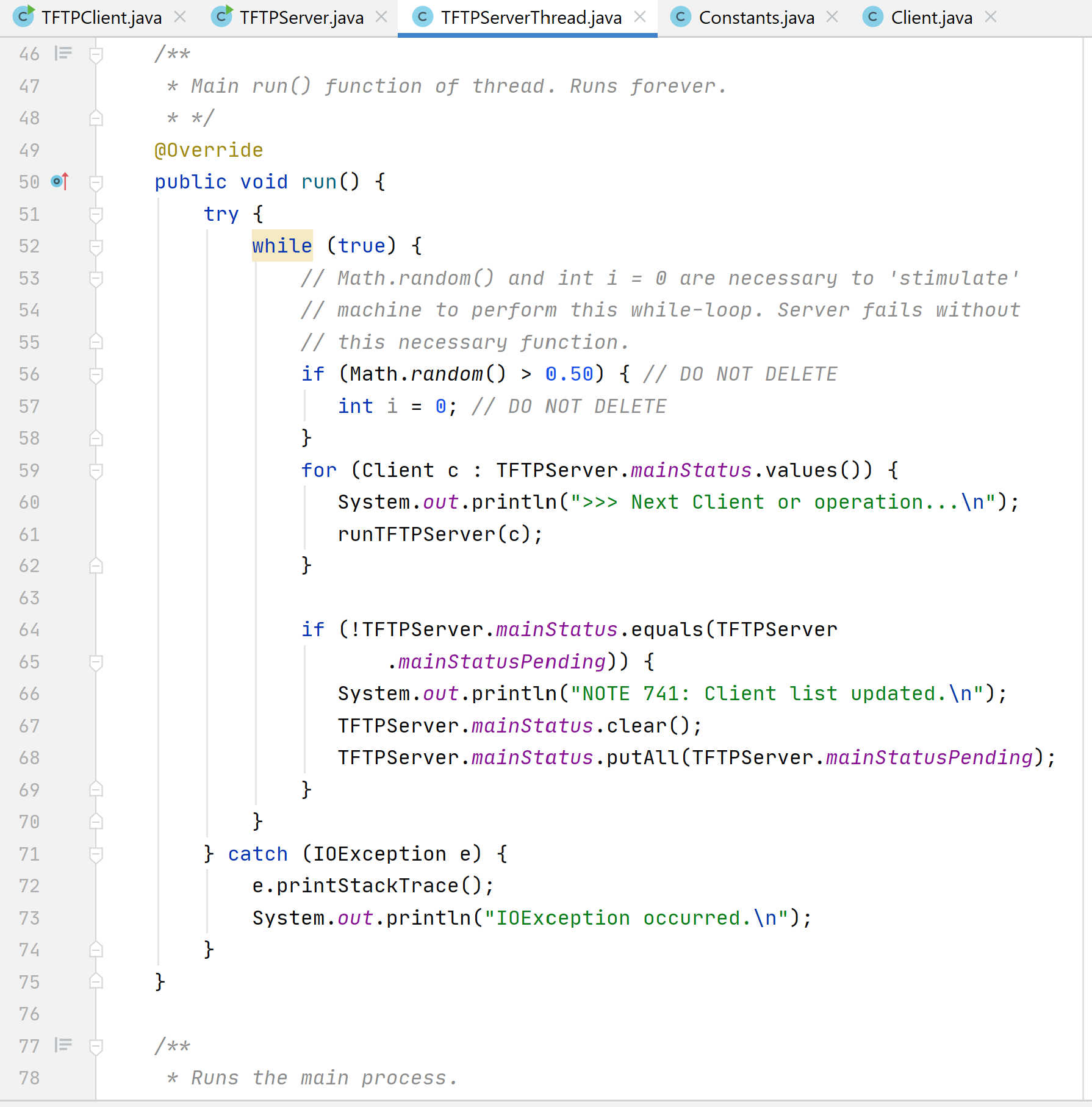


Figure 3.1 Server triggering process for each Client

**Back End – Read Request**

For each Client in mainStatus*,* this thread will first check the request of the Client. If it is an RRQ, the file will be checked if it exists in the database (when blockNumber in process is 1). A FILE\_NOT\_FOUND error is sent if file not found, else a readRequestServer(DatagramPacket) method will be called to trigger the change from the first to the second state involving making a buffer to read the file, and sending DATA 1 repeatedly until ACK 1 is received. This process returns and the next Client will be processed and repeats for subsequent DATA blocks. If the file size is a multiple of 512 bytes, then a final DATA packet with zero content will be sent until either (i) the final ACK is received, or (ii) if the final DATA packet has been sent 20 times (LOOP\_LIMIT), both triggering the removal of this Client from mainStatus and ends the processing for this Client’s request. Like the Client’s write request, a BufferedReader is used to send the file contents in 512 bytes of DATA packets.

Since the Server may be processing multiple Clients, a limit is placed on the number of allowed attempts as shown in *Figure 3.2* at an amount half of LOOP\_LIMIT or 10 attempts. If the same DATA packet has been sent without its acknowledgment received 10 times, the process returns, and TFTPServerThread.java will process the next Client object.

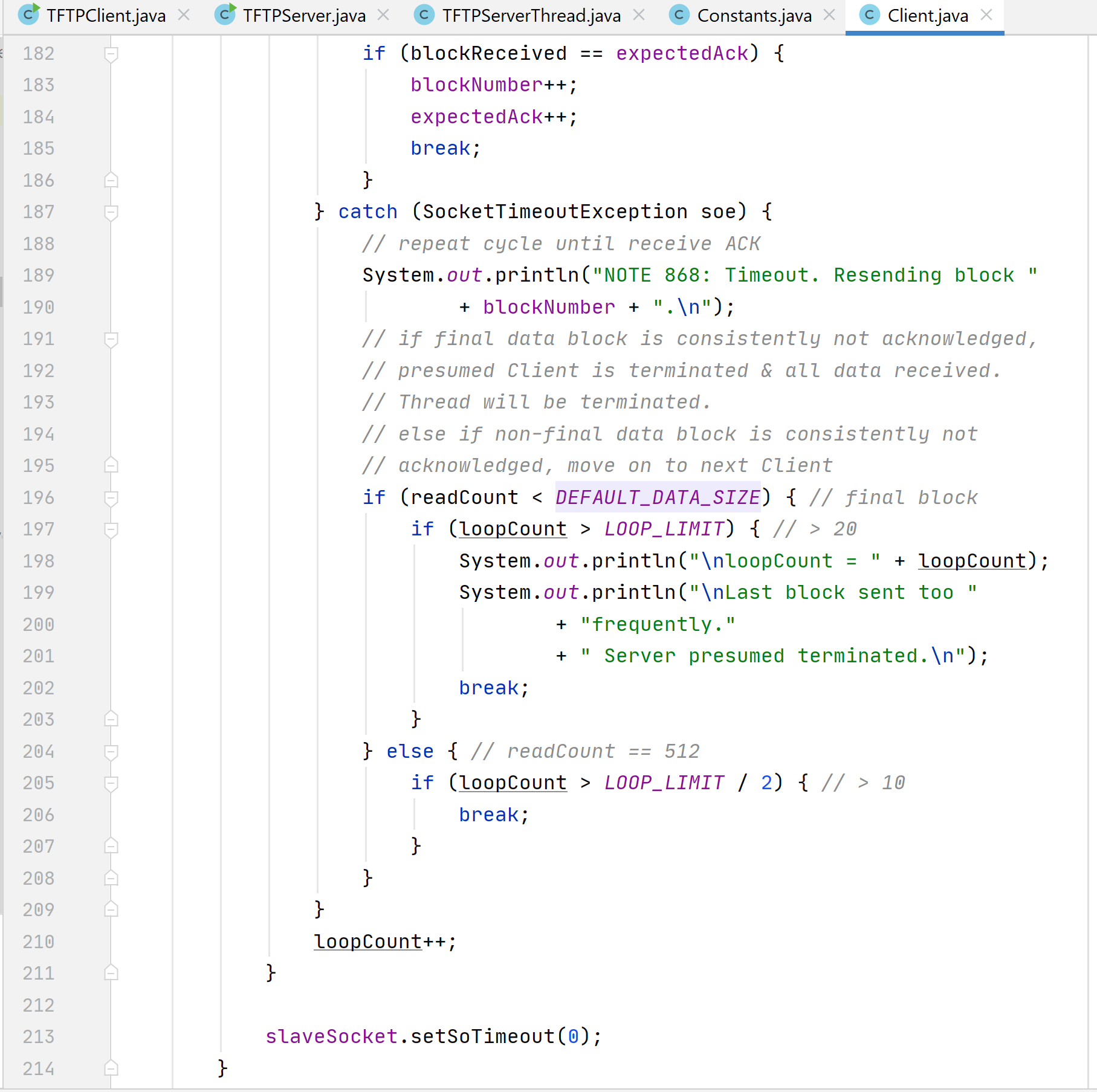


Figure 3.1 RRQ: Limit placed on the number of allowed attempts for each block

Hence, the definition of a ‘change of state’ in RRQ, i.e. the completion of which will end the processing for this Client, can be either:

1. If the blockNumber is 1 when the request is new, (1) checking the file exists in the database, (2) initializing the BufferedReader for reading the file in 512-byte blocks, (3) sending the first DATA packet up to LOOP\_LIMIT / 2 times or until the respective ACK is received, in that order assuming file exists; or
2. If blockNumber = *x* > 1, sending DATA *x* packet up to LOOP\_LIMIT / 2 times or until ACK *x* is received; or
3. If the final DATA block (content < 512 bytes) is being sent and packet content size is not a multiple of 512 bytes, either (i) sending the last DATA packet up to LOOP\_LIMIT, or (ii) receiving the last ACK before the 21st DATA packet is sent\*; or
4. If the final DATA block is being sent and file size is a multiple of 512 bytes, either (i) a final DATA block with zero content is sent up to LOOP\_LIMIT times, or (ii) receiving the last ACK before the 21st zero DATA packet is sent\*.

\* the completion of causes the Client to be removed from further processing.

**Back End – Write Request**

The WRQ process in the Server-side is roughly like the Client’s Read Request in that the file will be received in 512-byte DATA blocks with an ACK sent to acknowledge each block before the next DATA block is sent.

If the request is new and no ACK was sent yet i.e. expectedBlock = 1, the first ACK (ACK 0) is sent once to the Client as an acknowledgment of the request and to begin sending DATA packets. The Server calls the main write process to handle each incoming DATA packets as shown in *Figure 3.2*.

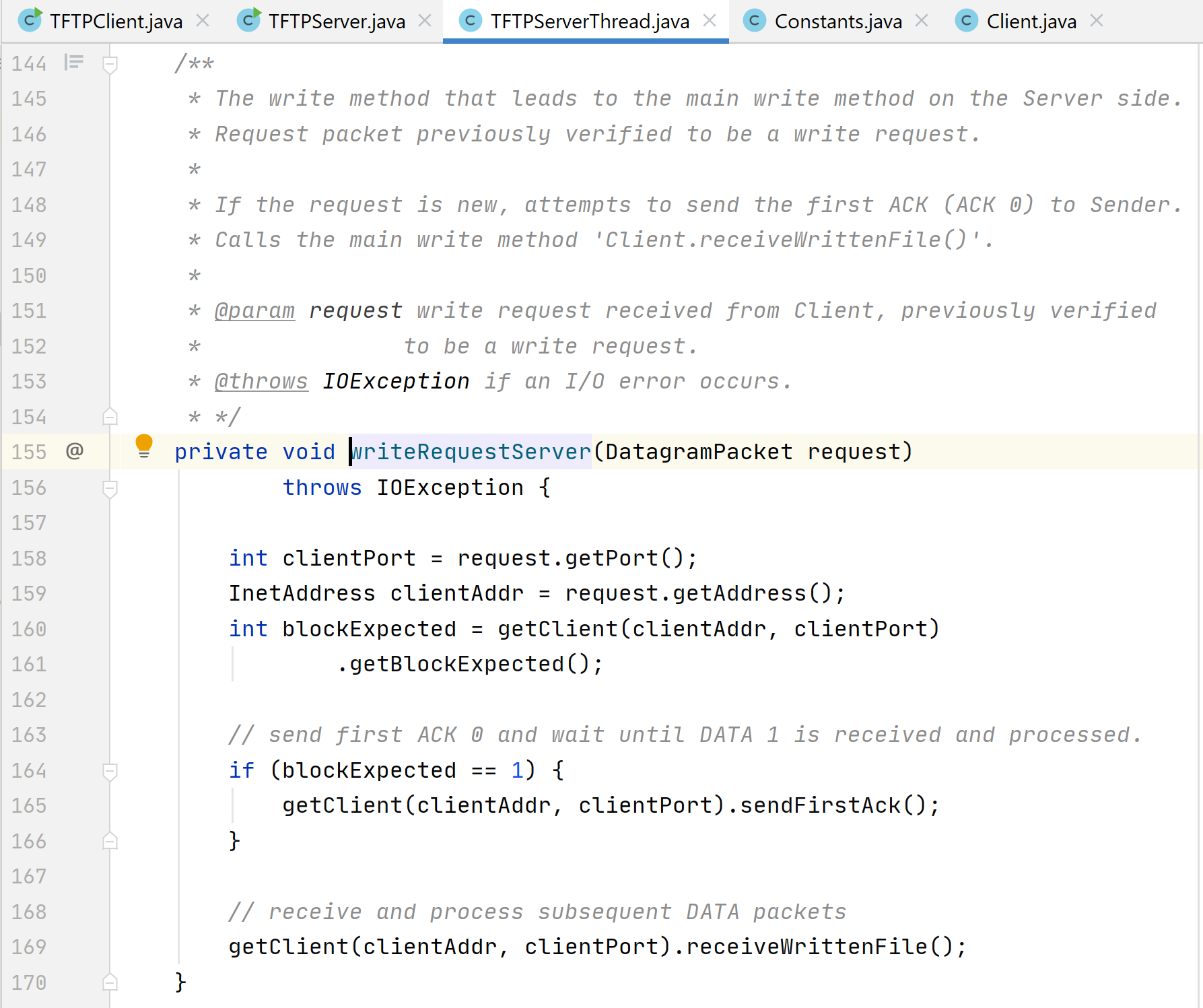


Figure 3.2 Leading to main write request for the Client

In the main write process Client.receiveWrittenFile(), ACK 0 will be sent after each timeout until DATA 1 is received. As each DATA packet is received, the contents are appended to a StringBuilder instance and the respective ACK sent. Any duplicate DATAs are responded with their respective ACKs and contents are ignored. As shown in *Figure 3.3*, after the final DATA is received, the final ACK is sent and a FileWriter initialized to write the file into Server.



Figure 3.3 WRQ: Server receiving DATA and responding with ACKs

But to account the possibility the final ACK did not reach the Client, the process sets a Boolean value writeRequestCompleted to true. This Boolean value keeps the slave port open for 10 \* TIMEOUT to receive final DATA duplicates and respond appropriately until timeout when the Client is removed from further processing. Hence, a ‘change of state’ in a write request is either:

1. If the first DATA is to be received (blockExpected = 1), send the first ACK 0 repeatedly until DATA 1 is received, appending contents to StringBuilder, send ACK 1; or
2. If blockExpected = *x* > 1 and the correct block number *x* is received, sending ACK *x*, appending its contents, and to receive the next DATA *x* + 1 packet (waiting indefinitely); or
3. If blockExpected = *x* > 1 and a duplicate block *x* – 1 is received, sending ACK *x –* 1; or
4. The final DATA packet is received, and the file is successfully written; or
5. Keeping slave socket open for 10 \* TIMEOUT to receive and acknowledge duplicates of the last DATA, timeout triggers removal of Client from further processing.

**Acknowledgments**

The writer would like to thank Dr. Parisis, Wikipedia, and Google for providing advice throughout the creation of this protocol.

1. Sollins, K. “The TFTP Protocol (Revision 2)”, *Request For Comments: 1350,* MIT,published July 1992. [www.ietf.org/rfc/rfc1350.txt](http://www.ietf.org/rfc/rfc1350.txt). Accessed 2020-07-04. pp. 1 [↑](#footnote-ref-1)
2. Ibid. [↑](#footnote-ref-2)