TFTP -
The TFTP works by first establishing a socket and sending packets through it. To send the packet you have to assign them port numbers and addresses, rather than just sending them through. Because of this and other factors, it is a UDP method with acknowledgements also being handled directly.

Firstly, the TFTP server creates a socket in which it will receive all its data. An initial port number and address are also established. From there the method to start the server is called and the actual socket under this port number and address is created in the constructor.

Inside the run method, the number of sockets available and the offset for the port number are established, by doing this it means that unless it's the port the servers running on, all threads to be made will run on ports higher than 9000. A thread array is then made, which will store all threads that are and have been established, to ensure that it doesn't try to join threads that are already active. The server enters a while loop so that it will always remain active unless closed directly to allow for new requests to constantly be made even when there are threads going on.

Firstly, the new random port number is made and a check occurs to make sure that the thread is either dead or it is a free port before going ahead. The initial receive packet is made and the server starts to listen for requests.

Once a request is made, the data is taken out and it checks on the type of request being made. If it is not 1 or 2 then declare its an invalid request and state the type before going back to listening. If 1 is chosen, it is a writing request and a new thread of the writing class is made passing in the unique port number and the packet it got.

Inside the writing class, the constructor first reassigns the packet to be a universal version and opens a socket to receive information on using its unique port with a timeout of half a second. The thread then starts properly and sets up its initial variables for the data handling. In these variables is the starting block number and a byte array the length of an acknowledgement packet, which is also created now with the port number and address from the request packet. The request packet data is reassigned with an offset to a new array to detect the length of the filename, which is then assigned to a variable.

A folder is then made to take the file transfer if it doesn't exist already, and the filename is then reassigned to be inside this folder. The file input stream is made with this filename. This is inside a try-catch block which will send a packet to the client with an error code if the file doesn't exist. This is followed by taking the file data and assigning it to a byte array to be transferred over. The code enters a while loop which goes until the entire file has been transmitted, starting by taking the length of the previous packet away from the remaining to send data. Depending on which is smallest between the remaining data and max file length, a byte array of the data to transfer is made with the first four bytes getting the OP code and block number between them. After this, the rest of the file is filled with the data either fully or partially, changing the length as needed which is used to detect when the server is done by the client. Then the packet is assigned the port and address and is sent off, before entering another while loop that waits for an acknowledgement. If this times out it resends the packet, and if this happens more than 5 times it breaks entirely and assumes the connection is lost.

Once the client replies, the block number is checked and incremented up, followed by the loop continuing until all the data has been sent. For the edge cases of there being an empty file, or a file being exactly the length of the packets, a final packet send is done to ensure confirmation of termination followed by the socket being closed.

With the reading class, it is much the same. The request pack is taken in, and a socket is made the same as the writing class. It then creates the first acknowledgement packet and adds the request packets' port and addresses ready to be sent. Following this before sending, the file name is taken from the first received packet. This is copied into an array that is initially 255 long but trimmed to the first 0 byte and finally, the filename is chosen depending on the zero byte. After this, the first ack packet is sent.

Once the first ack packet is sent, the output file is created under the serverside folder and the file output stream is also established. A new 516-byte array is made, which is then used to make the packet that will receive data. A few more variables for the while loop and its breakage are established, and the code enters the while loop. Once inside, the socket receives its first data packet, and the acknowledgement is created using the block number from this first packet. Once the ack has been made, the data packet is taken apart and the data is written to the output file stream with the offset of data written, to ensure that if multiple packets of the same data are received it's not duplicated. Once a packet of less than 516 is received, the code realises its done and quits out, closing the socket and all streams.

In the TFTP client, firstly the random port number is created. A scanner is then made to read the type of request being made, and depending on the answer either the method readfrom or writeto server is executed, passing in the port number.

In writeto server, the file name is input by another scanner and a new file input stream is made if the file exists. Once opened, the socket is established and makes a reply packet ready for the initial port server response. The request is then made according to the specification and sent on port 69 to the server.

If this doesn't timeout, a reply is received from which the port and address are stripped and applied to all outgoing packets. The code enters a while loop that assembles new packets the same as the server adding their length to the bytesSent int. The server sends off these packets and waits for an ack back. As long as it gets one this continues until all the data is sent, and a final packet is sent to make sure otherwise everything is closed.

In readFrom server, the same scanner takes the file name and this is assembled in the same way as the write request before being sent off. A reply and data packet are then assembled, and ready to be sent. The method then enters a while loop which keeps taking data until done is changed, in the case where the packet length is less than max.
The main difference here is that the data packet is then received, and once received it checks it isn't an error packet. If it is, it states why in the error message otherwise it will create a new output stream and then write to it until the length is less than 516. Once this occurs, a final check for new packets is done and then it closes everything.

TCP -

The TCP Client and Server setup works very similarly to TFTP, with the stark difference in that TCP is TCP, and TFTP is UDP. Because of this TCP does the port, acknowledgements and general security handling itself.

The TCP server starts by opening a socket, under the same port 69 as the TFTP version. It then enters a loop that opens new threads for each of the connections it makes. This means that the server itself is kept open, and new threads can be opened while others are being handled.

Entering the run method, the server opens the data streams in order to communicate back and forth later, which are passed into method parameters. The TCPOPCODE and filename are then received, which determines the type of handling the server is doing and the filename that is being read from or written to. The code is then checked to ensure it is not null and then goes to either method of sendFile or receiveFile.

Receive file only has the parameters of the subfolder and filename in one string, as well as the input stream from the client to receive the data. This then jumps into the method which first creates the file that it will be writing to. A byte array of 516 is chosen to break the data into smaller packets, though with TCP this could be larger as it is more secure but slower. The method then enters a while loop that writes to the file output in chunks. These chunks are the packets and can be of any length other than -1, which results in the while loop terminating. The client sends over the data and then sends the -1 to break it out and say that it is done. Once broken out, it closes the file output stream and the input stream. Finally, out of the method it shuts down the input and output stream of the socket and closes the socket freeing it up for another thread on the same port it generates.

Going into send file, it has three parameters, two are the same as before and the last is the output stream to the client. It then enters the method, which tries to create a new file input stream from the filename provided, if it can then it sends the client “3” which confirms the file exists, otherwise it tells the client it doesn't and both break the connection. Once this is made it starts to send packets in a similar while loop as before however reads from the input stream and changes where it reads from to send the next packet. Once this is done, it sends its last byte of length -1 which indicates an empty packet and breaks out. The file output steam is closed and the socket is terminated on both ends the same as receive.

With the client, the story is vastly the same. The socket is established on port 69 and it tries to establish a connection. If it is successful the server is open and it continues, otherwise, it fails and closes. Once connected it opens the input and output streams and establishes some variables that are used to ensure the user inputs are correct. It enters a while loop that keeps asking for a valid opcode, which decides whether the client will ask to read or write. It doesn't do this for the filename as this has to be checked by the server side on a read request. Once it breaks out of the loop, it gets the file name and checks the opcode.

If the opcode is 1, the send method is called. This method passes in three parameters, which are the output stream, the filename and the opcode to pass to the server. Inside the full filename is established, and the file input stream is made. It sends the opcode to the server to get it ready to receive data followed by the filename it wishes to write to. The same

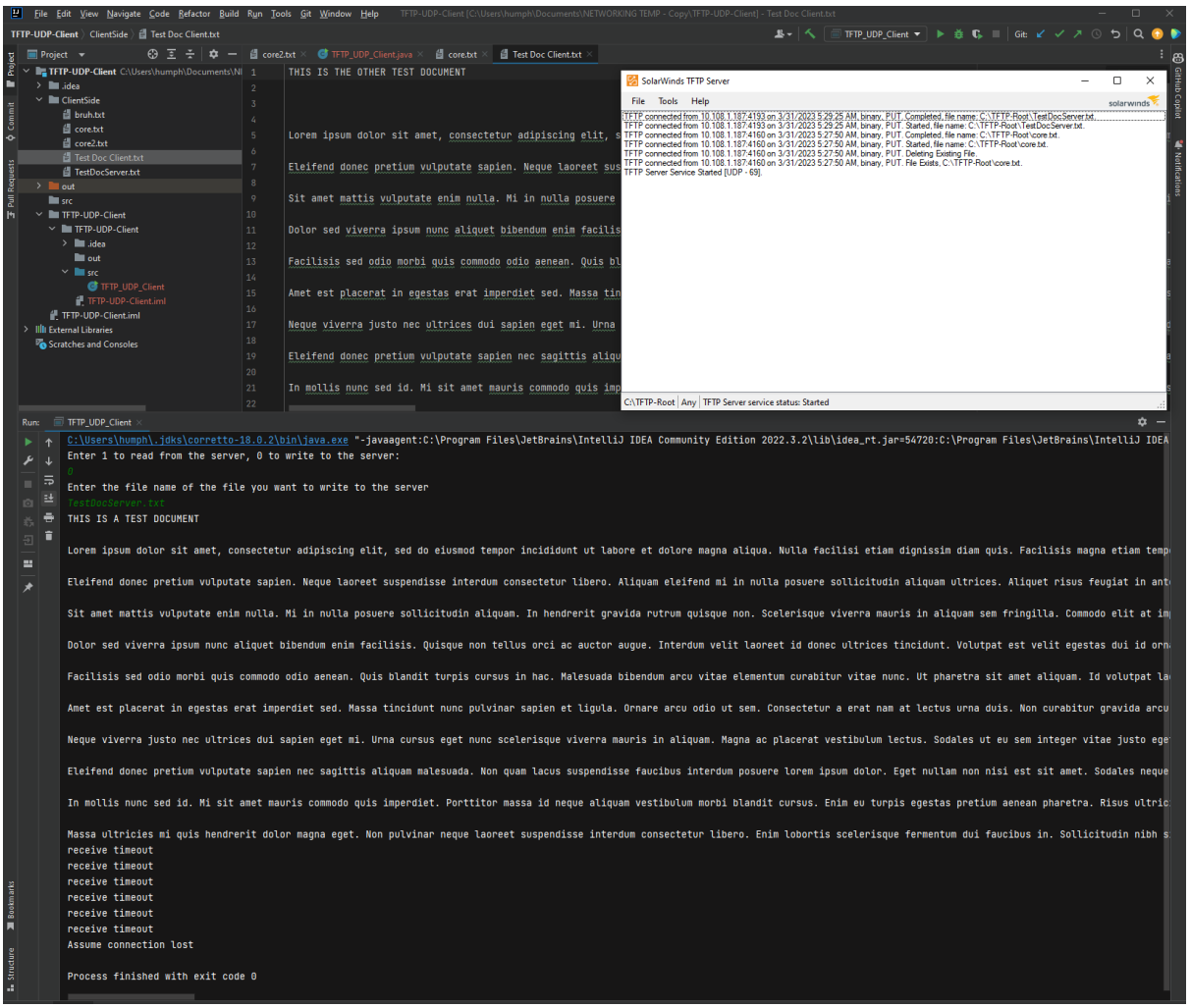
as the server, the code then enters a while loop which breaks the file input stream into packets for the server to receive and once the bytes read is -1 it breaks out having transferred its last packet. The file input stream is closed and the output stream is closed.

If the opcode is 2, the receive method is called. This method passes in four parameters, three are the same as the send but the last is the input stream from the server. Once inside the method, the opcode and the filename are transferred over. It enters a wait for the reply and the server says whether or not the file exists in another confirmation code. If it is not valid, everything is closed and the socket is terminated. If it is valid, then the file output stream is established to now make the file output, rather than making it and then finding out the file doesn't exist on the other end. The data transfer exists inside the while loop, and once it is completed the output stream is closed.

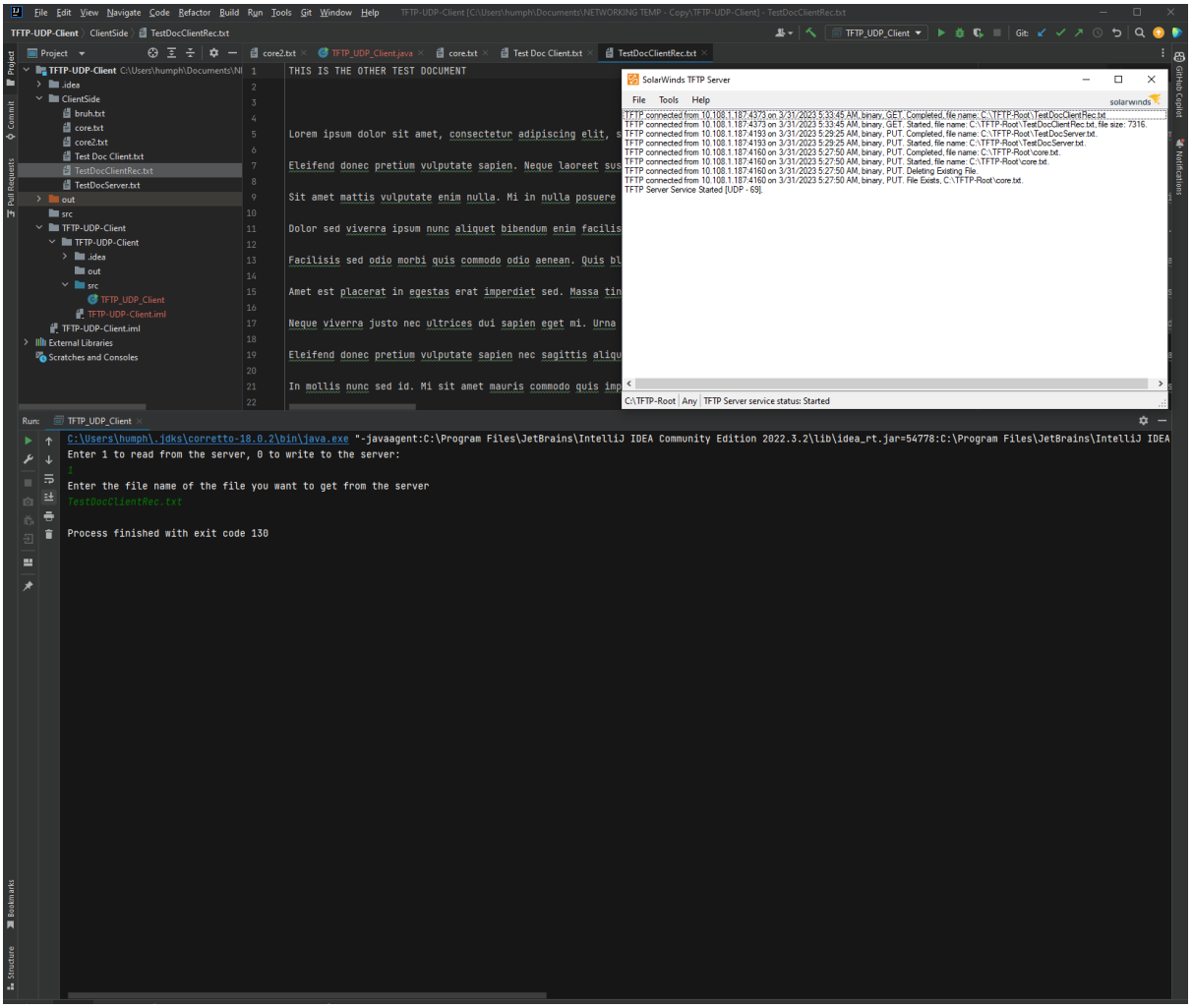
After both of these cases and in the case where the opcode is incorrect and somehow got through the validity check, the socket is closed and the connection is terminated.

Interoperability -

I used the solar winds TFTP third-party software to demonstrate that my code works in both reading and writing. Seen below is the screenshot of the transferred file in solar winds, as well as the exit of my IntelliJ when writing to the server. The file sent is called TestDocServer.txt.



Seen below this is the reading of another document TestDocClientRec.txt with the same information as the one sent, just renamed.



These are examples of the code working with a third party device