# UDP Server

For the UDP server I first started by making two versions of the constructor, one where you can name the new server thread and one that defaults to UDPSocketServer. Then for the running of the thread I first defined a bytearray to be written into and the directory for the server to write and read files. I used System.getProperty(“user.dir”) so that I could get the working directory of the project. This way it can be moved and still work. I chose for the server to have one directory to read and write to which was the folder called resources. I did this because normally where the file is stored on a server doesn’t affect the client so it was easier just to keep all of the files in one place. I then have a while true loop inside of a try catch so that the only time the server would stop running is if an error is thrown. I then construct a packet to receive any request sent and wait for a request to be received. One a new request is received I extract all of the data and add the first and second byte together for the opcode. I add them because the first one should always be zero, therefore shouldn’t affect the opcode, so I used this as a form of error checking. I then have an if statement for the opcode which determines the nature of the request and deal with it accordingly. If it is one then it is a read request, in which case I extract the filename by looking at the length of the file minus nine to fine the file length (It is minus nine because of the opcode is two, “octet” is five and the two separating zeros are two.). I then go through the bytes, starting from two to miss the opcode, and read the length of the string into a string builder. I can now construct a file path to the file being requested by combining the directory and the filename. I then check that the filename isn’t empty for error checking. If it is not I save the packet address and port number for later, because otherwise I wouldn’t know where to send the packet back to. I then create a file input stream to use for reading the bytes of the file into packets, then try to make an object of the file using the file path and try hooking the file input stream to this file. This is in a try catch because they can both throw errors like file not found. I then do another error check to check that the file is not empty after creating some variables that will be used later. Now I make an array of datagram packets which is effectively a two dimensional array since a datagram packet just packages an array with some other information. Then I check if the file length is divisible by 512 using file.length() % 512 == 0. If it is I use file.length() / 512 to get the amount of packets I should make and then use a for loop with this number to make the correct amount of formatted packets and add them to the datagram packet array. I then format and add one last packet with only an opcode and block number to signal to the server that there is no more data to read. This is because the server stops waiting for packets when a packet comes with < 512 bytes but a file of size that is a multiple of 512 is an edge case of this method so I deal with it by sending an empty packet. If this is not the case then I use Math.ceil to always round up the division so that there is always enough packets. For this method I also have to size the array and copy it into a second array every time so that, for the last packet no zeros are sent after the data. Then I create a packet for acks to be read into and set the socket to timeout after a second. After this I use a for loop to send a packet, then receive and validate the ack for that packet recursively. If any acknowledgement is wrong I stop sending and print an error message. After this finishes I turn off the timeout. After this is the end of the try catch which catches any file not found exception and sends an error packet to the client. There is also a catch for the input stream not being close which then tries to close it and if this does not work throws an exception. After this is the write option which also starts with getting the filename and making a file object using the same methods. I then create a packet for acks to be sent in and a file output stream with the filepath to reconstruct the file in the resources folder. I then get data packets, write them to the new file and send acks for them until I receive a packet which is < 512 bytes in length. I then send the ack for this last packet. There is also a small piece of code after this to catch invalid opcodes being sent and then a catch for any IO exceptions thrown by the file input and output streams. Finally the socket is closed.

# UDP Client

For the client side I mostly just used the read request code from the server for the write request, and the write request code for the read request. This is because they are almost an inverse of each other on server and client side. One big difference was that I had to always initiate a request from the client side, so I made a make packet method for read and write requests which took the opcode and filename as parameters and then correctly formatted a read or write request to be sent off. This and then waiting for the server response rather than just waiting for a request is the main difference between the server and client. Another big difference is that, because it is client side, it needs some sort of interface. I decided to at first have a text based menu offering to store a file, retrieve a file, or quit the program. Then if they choose to retrieve a file they type the filename and extension (this also must be in the server resources folder). I also did not want the user to have to type out full file paths to where they wanted to retrieve files from or save files to so I implemented a file chooser class with methods to get a file or save a file. These bring up a file explorer so the user can select a file or for saving make a new one. Getting a file returns a string array where the first entry is the file path and the second is the filename. Saving a file returns the file object for data to be saved into. Because they are setup this way they integrate very nicely into the rest of my code, only returning the necessary data.

# TCP Server

The TCP server is setup in the same way as the UDP server however I use data input streams and data output streams to continuously transmit and there are no acks or timeouts since TCP is reliable.

# TCP Client

The TCP client is setup in the same way as the UDP client however I use data input streams and data output streams to continuously transmit and there are no acks or timeouts since TCP is reliable.

# Testing methods

I tested using testdoc.txt which was exactly 1024 bytes to make sure that my program could handle files of sizes that are a multiple of 512. I also used two different photos for different amounts of packets. As well as this I used an empty text file and a large video file to check that there were no scaling issues.

# Evaluation

Overall I am very happy with how my TFTP systems worked out, especially since I think I designed them so that the client side is very user friendly and simple, and the server needs nothing other than to start running it.