Introduction:
The aim of this practical was to gain a better understanding of how networked communication is done and the importance of transport-layer protocols. I decided to develop my client and server using Java and I attempted all the requirement specified in the practical sheet.

Design and implementation:
We will split this section into 2, a server section then a client section. We will then explain how they work together to ensure data consistency.
My goal when designing the server was to work though the requirements. Thus we have 4 bullets points :

- Read an audio file (or any file in this case).

To read the file I want, I simply use a FileInputStream with the file path. I then create a while loop to do the main processing. As soon as I had the correct bytes read by the input stream, I could consider this task done.

- Divide the file in packets for UDP transmission.

Here, we have a tricky requirement where a lot of ways can be correct. Indeed, we know that most protocols limit their packet size to either 512 or occasionally 8192. The reason for this is that when broadcasting across the internet, the larger we go, the more likely we are to run into packet transmission problems and loss. However, since we are on a reliable network and we are not broadcasting to a large number of clients, we can use the maximum UDP message size (around 65500 bytes). In this case, we used the arbitrary number 62800 for packet size in bytes.

- Use UDP to send the packets to the client.

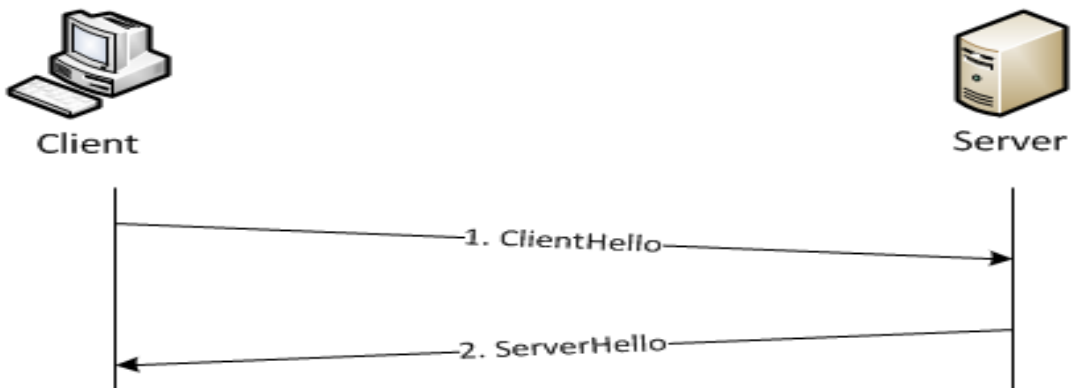
To send data to the client, I used a Datagram socket that would connect on the specific port. Then a Datagram packet is used to get or send data using that socket. The server takes care of sending the packets in the file reading loop. But since I am using a while loop and not a do{}while() loop, I need to send the last packet to complete the transmission then close everything. I use try catch blocks to prevent Exceptions and count the packets sent for debugging purposes.

The last idea I to design a suitable application layer protocol for sending this data. We will get into details for this when explaining how the client and server work together.

The client development used the same bullet point method inspired by the practical specifications.

- Connect to the server and request the file

This is obviously the first step of any network communication, We need a handshake.



Thus, The client here send a message using the same UDP logic explained above and initiates the process. This message requests the file from the server and upon reception of it, the server starts the file reading and packet sending process.

- Receive the file

Receiving the file uses and listening until end logic. For this, I used a while loop with the true condition making it an infinite loop. The client breaks out of the loop upon reception of the last packet. Here, we use a Datagram packet to listen and receive data from the server, we output the length and the packet number for debugging purposes.

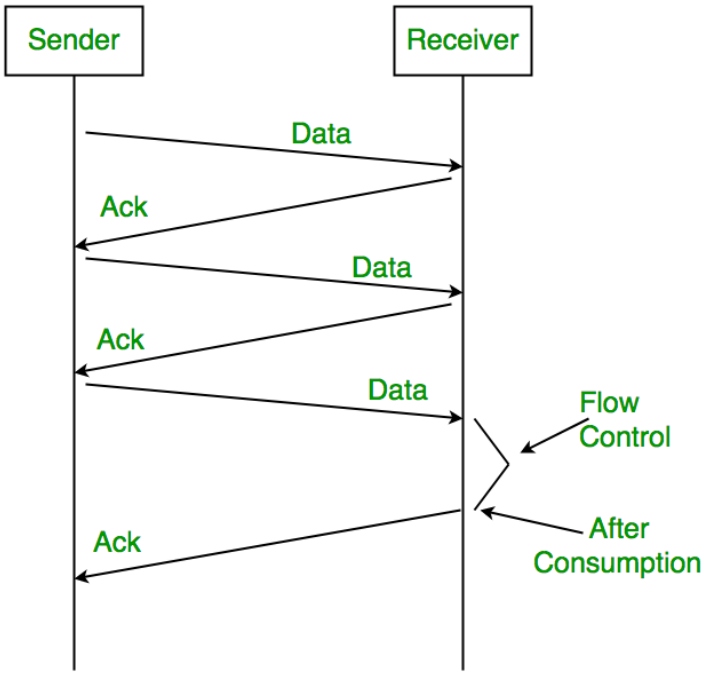
- Write it to a file system

The data received in the process explained above has to be written to a File. To do so I used a simple FileOutputStream and wrote the bytes received into it. When writing this data, I made sure to write the correct amount of bytes using the packets getLength method to make sure the last packet would not get corrupted.

- Stream the audio

Now to stream the audio to the user, I decided to use audioInputStream to read into the file written above and output the sound. We make sure that the clip is played in its entirety using an empty while loop where the conditional makes the program wait for the clip to be read fully before closure. This part was tricky since Java is not asynchronous in opposition to JavaScript for example. This made reading the clip inside the main loop impossible without Threading my client tasks. A solution would have been to use the AudioFormat class and stream the byte arrays in the while loop, But that makes the program very fragile to jitter and packet loss.

On that note, We need to take care of the client and server communication to make sure the data is well sent and well received. To do that, I used a Stop and Wait protocol logic.



https://www.geeksforgeeks.org/stop-and-wait-arq/

After transmitting one packet, the sender waits for an acknowledgment (ACK) from the receiver before transmitting the next one. In this way, the sender can recognize that the previous packet is transmitted successfully. Thus, "stop-n-wait" guarantees reliable transfer between nodes.

Normally, we would keep a record of each packet sent and use a unique sequence number to support this feature. But we don't need to do that here because we force the server to wait a specific amount of time for a unique ACK, jitter or packet loss/ packet reordering is made impossible because the server doesn't process the next packet. Indeed in the server side we have the following loop (pseudo-code):

```
socket listening;
time-out set for socket;

 While(ack not received){
        try{
                send packet again;
                listen for ack;
        } catch {
                time-out;
        }
}
```

This piece of pseudo code is that main part of my Stop and Wait implementation. The main shortcoming of this algorithm is that it allows the sender to have only one outstanding frame on the link at a time. The sender should wait till it gets an ACK of previous frame before it sends next frame. As a result, it wastes a substantial amount of network bandwidth. To improve efficiency while providing reliability, We can use "sliding window" protocol "Go back N" protocol. However, since this project is so small we don't really need to go further when it comes to optimisation.

Testing:

I had 5 things to test to make my program full-proof.
The first thing I had to check was If it actually worked. The program ran but I found myself stuck on some problems I could not fix. Indeed, Depending on whether It was running on SSH or on the localhost, the sound streaming would not always work. I then understood that it was mainly an issue of the AudioInputStream library and could not easily be fixed. However, the file was correctly written and played back afterward. The streaming worked fine when using the localhost and not ssh-ing. In addition, the testing script could not be run properly so I had to do testing myself (at the best of my ability).

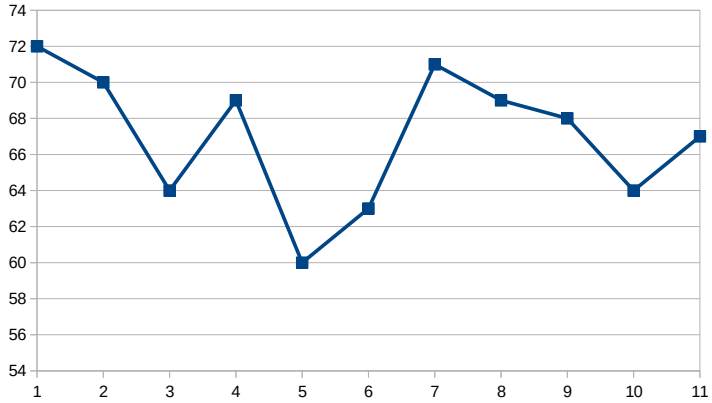
To test packet loss, packet reordering and delay, I coded a loop using a random number generator to randomly mess with my packets. In addition to this, in Java, without using a thread sleep on the server sending process, we already had a lot of jitter.

```
mmd4@klovia:~/Documents/cs3102/Practical1 $ ./server.sh
port : 3002
name : files/Chuck_D_-_No_Meaning_No_(feat._Fine_Arts_Militia).wav
Total Length :33988652
No of packets : 541

Last packet Length : 13852
```

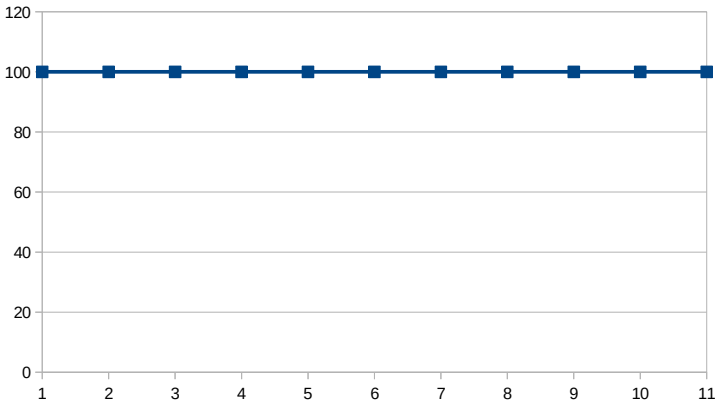
```
 Packet length: 62800

Packet 324 written to file


 Packet length: 13852

Packet 325 written to file

ending...
^Cmmd4@klovia:~/Documents/cs3102/Practical1 $
```

This is one of many examples of what would happen without the protocol implementation. As we can see, Not all packets are received and the streamed file lacks some parts.

I ran the client and server 30 times to check if the packet corruption was consistent. It was and here are my results over the last 10 runs:

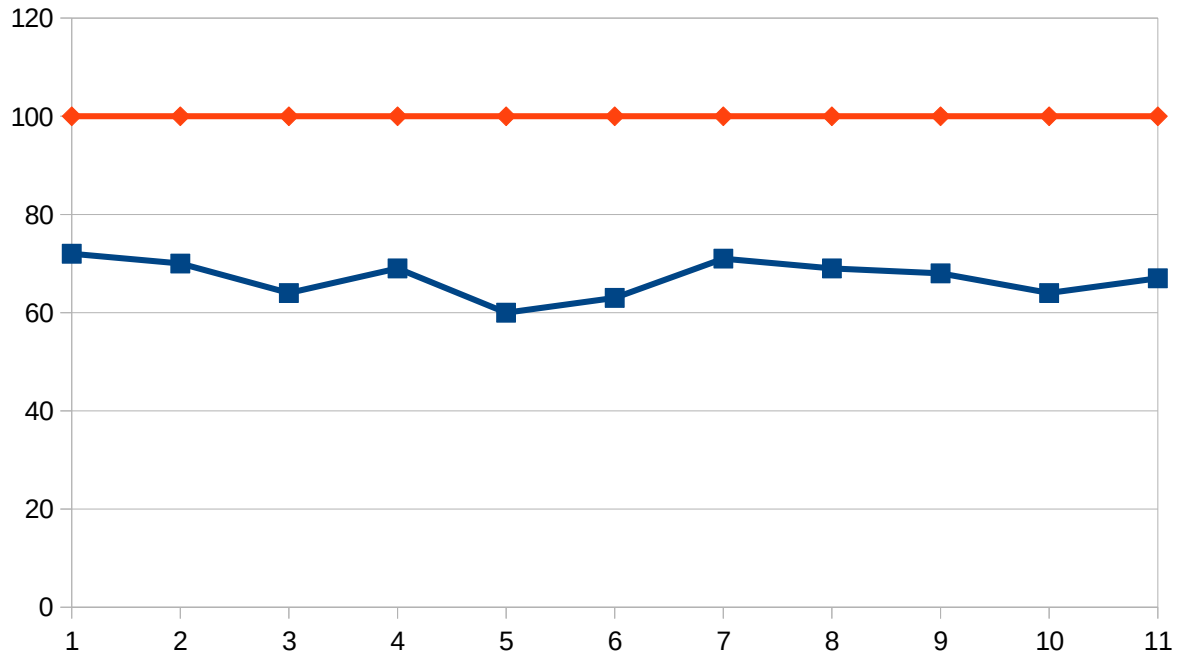


This graph shows the percentage of packets received by the client without the protocol.



This graph shows the percentage of packets received by the client with the protocol implemented.

Now let’s compare those 2 graphs to show the impact of my Stop and wait Implementation.



We can clearly see that when the jitter, the packet loss and the packet reordering are taken care of, we receive 100% of the packets in the right order. I tested the packet ordering by calling the diff command on the source file and the output file and they were the same on the 10 runs.

Now I had to test the delay. For this, I used the time-out command for the Datagram socket so that my program would exit if the client is not responding after 3 seconds. In this example I make my client wait 5 seconds before sending the ACK. We can see that the time-out was a success.

```
ERROR!
java.net.SocketTimeoutException: Receive timed out
        at java.net.PlainDatagramSocketImpl.receive0(Native Method)
        at java.net.AbstractPlainDatagramSocketImpl.receive(AbstractPlainDatagramSocketImpl.java:143)
        at java.net.DatagramSocket.receive(DatagramSocket.java:812)
        at Server.main(Server.java:117)
mmd4@klovia:~/Documents/cs3102/Practical1 $
```

Program running commands and results:

This part is very important as my program did not behave normally on SSH.
When in the Practical1 folder, using the terminal, compile the server using javac server/src/Server.java ; and the client using client/src/Client .java.
Now calling the client and server script will send the packets and create the audio file that can be played back.
To stream the audio file, don't use ssh and replay the host String by localhost. This allows the AudioInputStream library to actually work and stream the sound on the client terminal.

Conclusion:

I struggled a lost with external libraries in this practical and did not have the opportunity to focus on the main goal of. I managed to do all the requirements and testing when the program was running on localhost. However, the streaming functionality doesn’t work on ssh when calling the client script. Thus, I am a bit disappointed by the end result. However, I learned a lot about network communication and now feel confident to develop a reliable networked client and server for a simple purpose.