Homework 1

In Homework1, I implemented two-tiered architecture, a client-server system for measuring the time to transfer various amounts of data under different conditions. The client tier is responsible for sending the messages of the data transfer. In this implementation, the client program is tasked with sending data to the server using both UDP and TCP protocols. It also measures and reports the transmission time, the number of sent messages, and the number of bytes sent. The choice between UDP and TCP is made with command-line arguments.

The server tier resides on the receiving end and is responsible for processing and recording the received data. It differentiates between the UDP and TCP protocols, implements streaming and stop-and-wait mechanisms, and outputs relevant statistics after each server session. This includes the used protocol, the number of messages read, and the number of bytes read. For this the choice between UDP and TCP is made through command-line arguments as well.

For the implementation of streaming/stop-and-wait I used command-line arguments as well. If no command-line arguments are specified, the default values are TCP and stop-and-wait.

Furthermore, for both sending messages and receiving them i chose to do the following: The client sends the message length (and server acknowledges if stop-and-wait is active), and then splits the message into chunks of a specified size (1024). The server reads the message length, and then reads chunks until it has read the full message. Here are the code snippets from the TCP variant of the server/client:

```java
private byte[] readMessage(DataInputStream in, DataOutputStream out, long messageSize) throws IOException {
    byte[] buffer = new byte[ServerApp.CHUNK_SIZE];
    long totalBytesRead = 0;
    int bytesRead;

    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    while (totalBytesRead < messageSize) {
        bytesRead = in.read(buffer, off: 0, (int) Math.min(buffer.length, messageSize - totalBytesRead));
        if (bytesRead == -1) {
            break;
        }
        byteArrayOutputStream.write(buffer, off: 0, bytesRead);
        totalBytesRead += bytesRead;

        sendStatus(out, status: "[OK] Chunk Received", force: false);
    }

    return byteArrayOutputStream.toByteArray();
}
```

```java
int totalBytesSent = 0;
int chunkSize = ClientApp.CHUNK_SIZE;

while (totalBytesSent < messageBytes.length) {
    int bytesToSend = Math.min(chunkSize, messageBytes.length - totalBytesSent);
    out.write(messageBytes, totalBytesSent, bytesToSend);
    out.flush();

    totalBytesSent += bytesToSend;

    readStatus(in, force: false);
}

readStatus(in, force: false);
```

The readStatus method is used in the stop-and-wait variant for acknowledgement.

The server:
In the TCP case, the server continuously listens, and then delegates a clientHandler for the communication with the client.
In the UDP case, since there's no connections, the server awaits for a specific message from the clients, and delegates a clientHandler after receiving the message.
In both cases, the clientHandler is done with the client when the "END" message is sent.

Some statistics:

(The avg time includes all the time taken for logs).

- 5 clients, each sending ~100 bytes of data (chunk size of 1024)

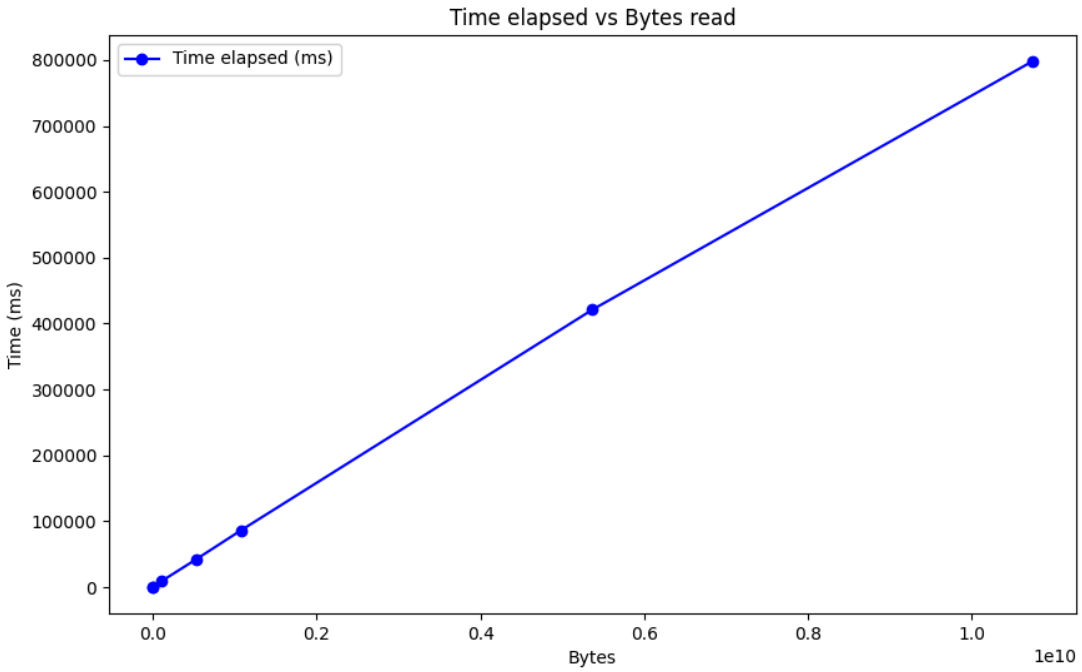
	Avg time (client) (seconds)	Server protocol	Server msg received	Server total bytes
TCP + stopAndWait	0.012	TCP	5	500 bytes
TCP + streaming	0.01	TCP	5	500 bytes
UDP + stopAndWait	0.003	UDP	5	500 bytes
UDP + streaming	0.002	UDP	5	500 bytes

- 5 clients, each sending ~1GB of data (16_385 messages of 65535 bytes each) (chunk size of 1024)

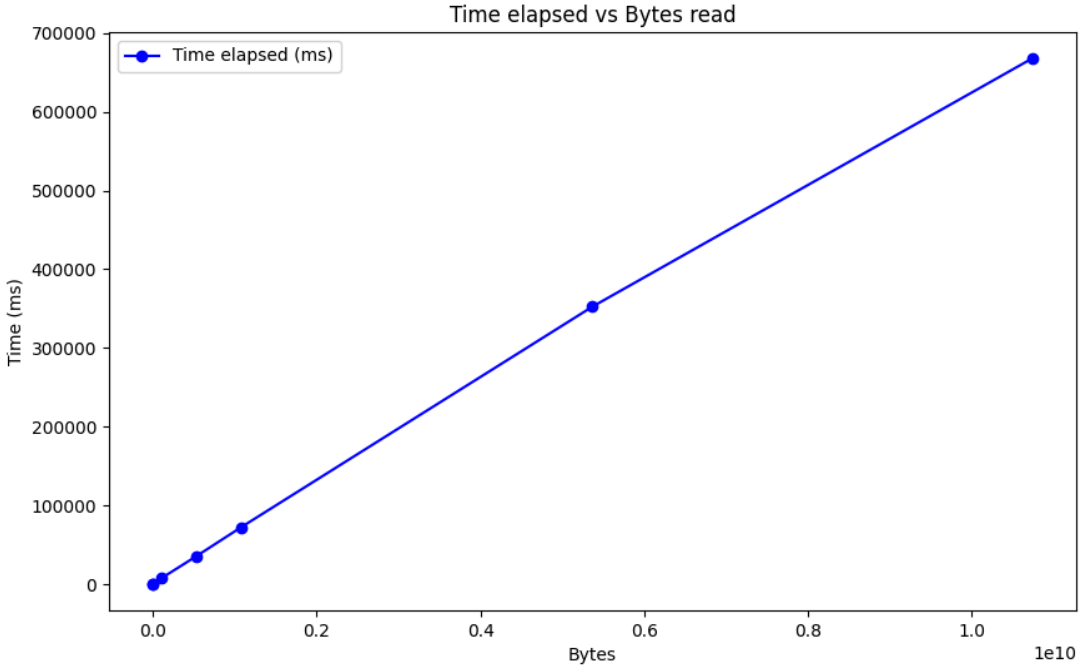
	Avg time (client) (seconds)	Server protocol	Server msg received	Server total GB
TCP + stopAndWait	109.71	TCP	81925	5.0001 GB
TCP + streaming	91.84	TCP	81925	5.0001 GB
UDP + stopAndWait	13.45	UDP	81925	5.0001 GB
UDP + streaming	3.43	UDP	46613	2.84 GB

And graphs (1 client sending X messages of 65535 bytes, equating to 64KB, 1MB, 100MB, 500MB, 1GB, 5GB, 10GB).

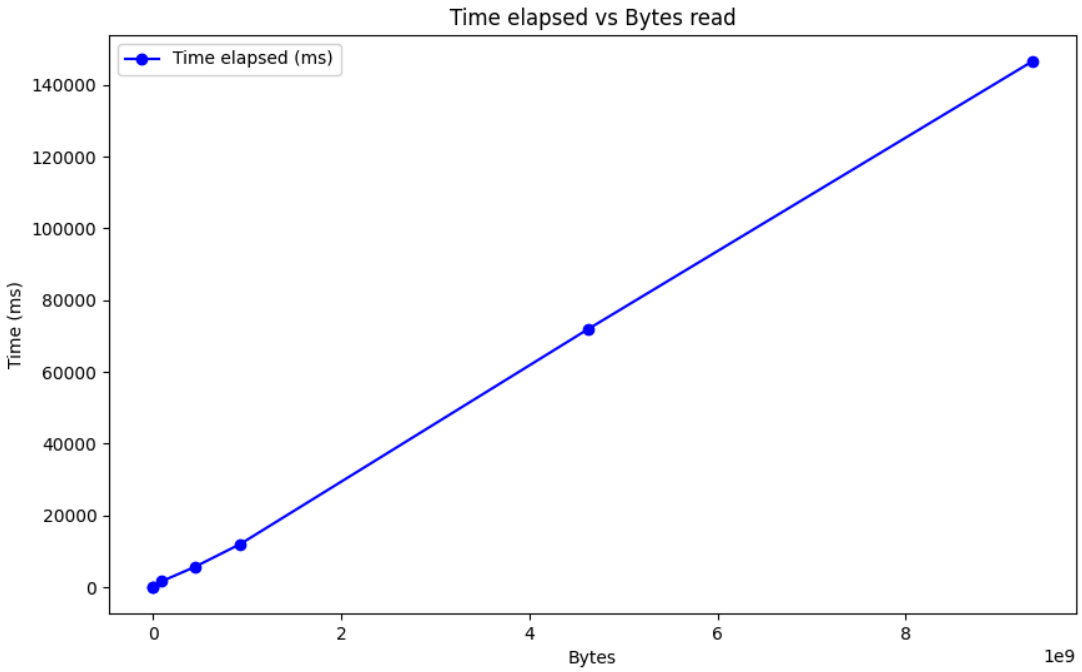
TCP + stop-and-wait



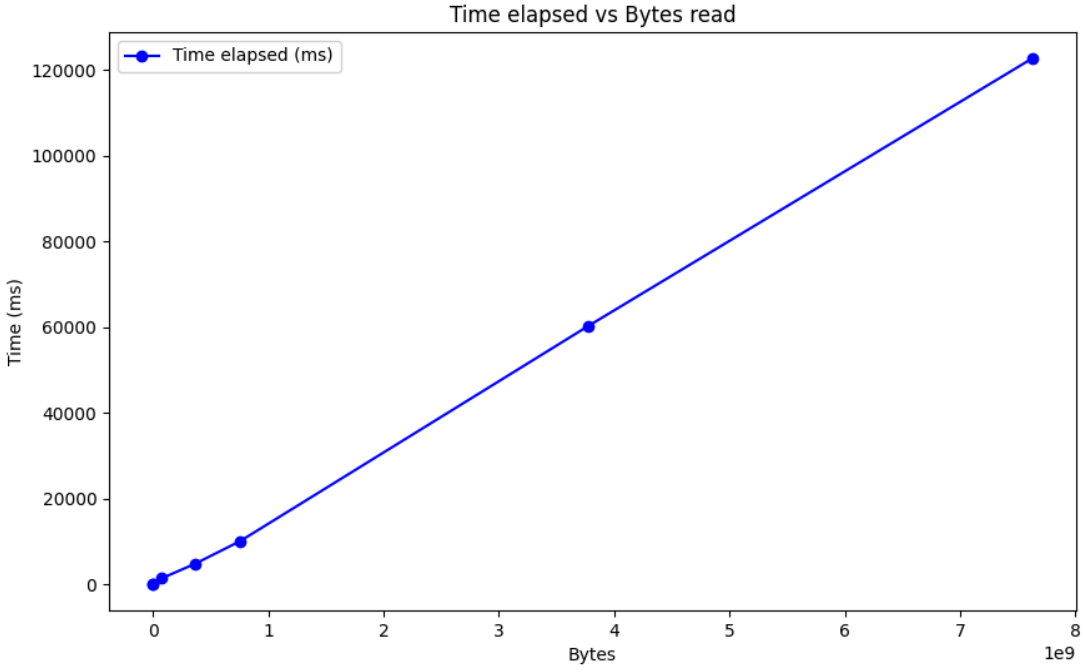
TCP + streaming



UDP + stop-and-wait



UDP + streaming



Extra tests: I tried to transfer a real file, and rebuild it on the server. I also tested the transferring the maximum chunk size (65000 — because for UDP, 65535 throws error “Message too long”). For this experiment i also counted the “num messages” as the number of times we send/receive chunks.

	Avg time (client) (ms)	Server protocol	Server msg received	Server total bytes	Was file fully rebuilt?
TCP + stopAndWait	500	TCP	28	1775258	Yes
TCP + streaming	480	TCP	28	1775258	Yes
UDP + stopAndWait	329	UDP	28	1775258	Yes
UDP + streaming	204	UDP	5	260000	No

One more interesting test would be that by adding a 10 ms between each chunk sent by the client, the UDP + streaming does manage to send the whole file. While not different from stop and wait, since there’s no acknowledgement, there is still a “stop” of 10 ms.

UDP + streaming (+10ms delay)	300	UDP	28	1775258	Yes

Finally, i’ve deployed my server app to AWS EC2 instance in order to test the file transfer from a remote location.

	Avg time (client) (ms)	Server protocol	Server msg received	Server total bytes	Was file fully rebuilt?
TCP + stopAndWait	3240	TCP	28	1775258	Yes
TCP + streaming	2972	TCP	28	1775258	Yes
UDP + stopAndWait	1241	UDP	28	1775258	Yes
UDP + streaming	297	UDP	28	1775258	Yes