Comparison of TCP and UDP Client-Server APIs

Jason Gardner, *Student Member, IEEE,*

***Abstract*—This project aims to compare the performance and characteristics of Java-based TCP and UDP client-server APIs through the implementation of a custom FTP client and server. This project aims to compare the performance and characteristics of Java-based TCP and UDP client-server APIs through the implementation of a custom FTP client and server using both protocols. Files of sizes 1 MB, 25 MB, 50 MB, and 100 MB were transferred. UDP was made reliable using a cyclic redundancy check (CRC) on each packet. Transfer times were recorded, throughput was calculated, and these results were compared. Ultimately, due to the overhead of checking the full data payload with CRC, UDP transfers had a slower response time than TCP transfers. TCP, being highly optimized, had a smaller computational overhead than the UDP implementation. The UDP implementation had higher throughput. With further optimization, it could potentially outperform the TCP implementation on both metrics. Because reliability is so important to file transfers, TCP is the correct protocol.**

***Index Terms*—TCP, UDP, Client-Server APIs, Java Sockets, FTP Implementation, Network Protocol Comparison, Data Transfer Performance**

I. Introduction

A. Project Overview

THIS project is for the graduate class at University of North Florida, CNT6707 Network Client/Server Architecture with Dr. Sanjay Ahuja during the Fall 2024 term. An FTP server was created in two parts during the term, following the provided instructions. This required the creation of an FTP Server to be written in Java utilizing both TCP and UDP for file transfers. The test files were 1 MB, 25 MB, 50 MB, and 100 MB in size. Response time and throughput were measured and conclusions were drawn about TCP and UDP protocol usage in networking. Understanding the differences between these protocols and knowing when it is appropriate to use each is important in networking.

TCP is a reliable standard that introduces overhead that reduces throughput as the cost of that reliability. UDP is an unreliable standard that increases throughput by discarding reliability as a feature. These were expected results. It was also expected that UDP transfers would have reduced response times due to less overhead. This will be discussed further in the results. The goals of the work were to implement TCP and UDP protocols in a practical project, add reliability to the UDP protocol, and evaluate the performance of TCP and UDP protocols.

While virtual machine servers were provided for the project, it was not possible to get the server and client machines to connect reliably, and the version of Java (OpenJDK 11.0.19) on the machines was past End-of-Life [1][2]. Testing was ultimately performed locally on the development machine and across two machines located on a home Local Area Network (LAN). Those results were consistent, and the home LAN data is the presented results in this work. These machines are described in additional detail in latter sections.

1) Project Description [3]:

Your code must be robust and handle any incorrect user input. Since there can be multiple clients querying the server at the same time with each client being serviced by a different thread on the server, the server must ensure concurrency control on the data file while it is being updated. Thus, only one thread must gain access to the data file during writes (hint: use synchronized keyword for the write method in your code). Be sure to terminate each thread cleanly after each client request has been serviced.

2) Project 1 [3]:

Implement the project as described using Java Sockets (TCP). This will be a connection- oriented client-server system with reliability built-in since TCP provides guaranteed delivery.

3) Project 2 [3]:

Implement the project as described using Java Datagrams (UDP). This will be a connectionless client-server system since UDP is connectionless. However, you will need to provide reliability in your client-side application code since UDP does not guarantee delivery. You may use the CRC32 checksum class available in the java.util.zip package in the JDK for this purpose.

4) Project Description - Continued [3]:

In each project, be sure to measure the mean response time for the server to service the client request. To graph this, have your client make requests in the order of the following data/file sizes: 1 MB, 25 MB, 50 MB, 100 MB. First do this for the PUT command and then do this for the GET command. Determine the response time in each case, then plot the response time vs. offered load (file size) graph. Next, plot the throughout versus offered load graph using the data from the GET graph/command. Throughput in this case is bytes delivered per second (bytes/second). Plot the bytes/second on the y-axis and offered load on the x-axis. So, at the end of each project, you will have 3 graphs: one graph for the response time for the GET command, one graph for the response time for the PUT command, and the throughput graph for the GET command. Your code must be well documented with adequate comments, suitable use of variable names, and follow good programming practices. You will provide a demo at the end of each project to the instructor. Have your server print out diagnostic messages about what it

is doing (e.g., “accepting a new connection”, etc.) Your code will be expected to deal with invalid user commands.

B. Goals

The aim of these projects was to understand TCP and UDP protocols; their practical benefits and drawbacks through implementation and performance analysis.

C. Client-Server Setup

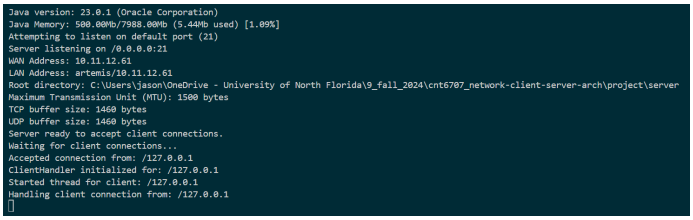


Fig. 1. Server Startup and Client Connection

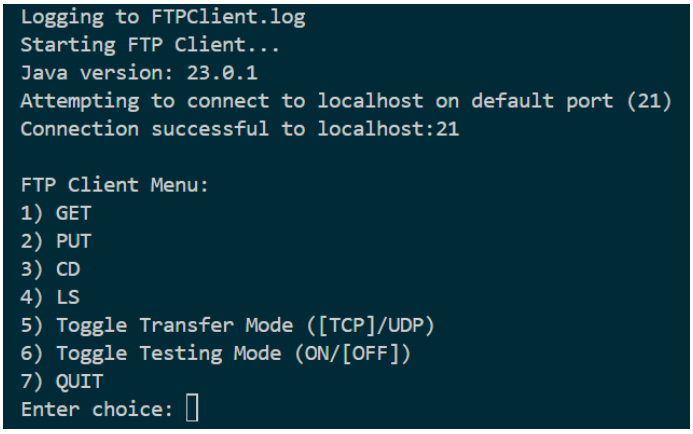
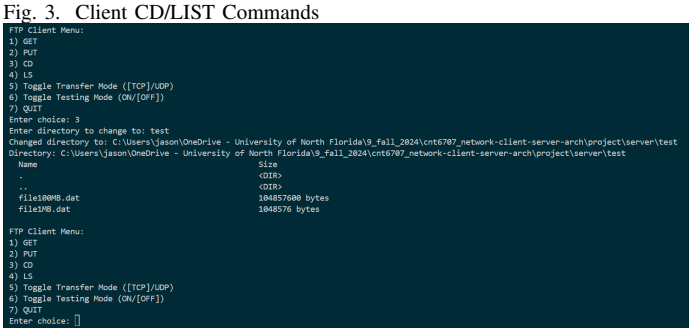


Fig. 2. Client Startup and Server Connection

Fig. 3. Client CD/LIST Commands



The client and server were implemented for both projects as a single server that initially established a TCP connection via *java.net.Socket*. Server and Client startup are shown in Figure 1 and Figure 2, respectively. On the client side, a menu is presented with selections for navigating the server filesystem (CD/LIST) as shown in Figure 3 (the client sends a LIST command after CD to show the client the contents of the directory they have moved to in this example), terminate the connection to the server (QUIT) as shown in Figure 4, a mode select that sends a MODE command to the server to toggle file transfers between TCP (default) and UDP via *java.net.DatagramSocket*, and PUT and GET commands to transfer files to/from the server. Finally, there is an option on the client to toggle a testing mode on the client that transfers 10 files during TCP/UDP PUT/GET operations and averages the results.

Fig. 4. Client QUIT Command

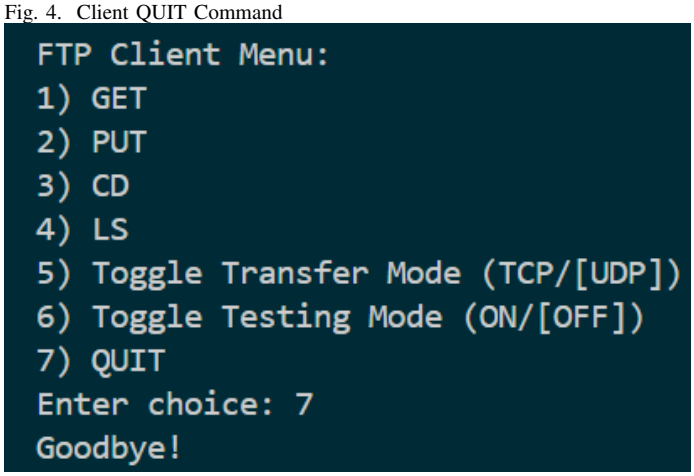
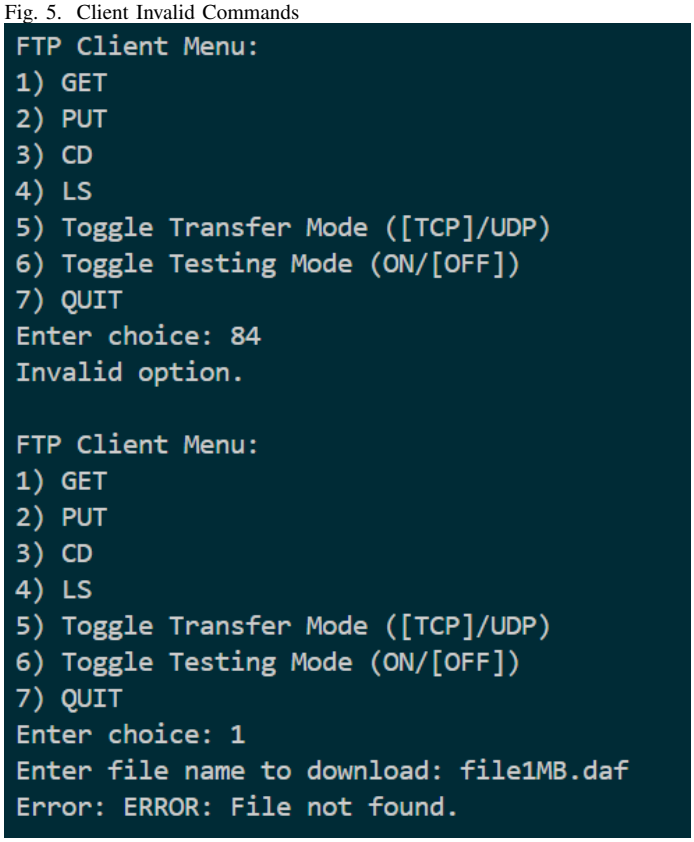


Fig. 5. Client Invalid Commands



The server uses a **ClientHandler** thread to manage multiple clients, and locks the file for writing (returning an error to a client that attempts to write that file at the same time as another) via *java.nio.channels.FileLock*. Invalid menu options and filenames are handled gracefully, as seen in Figure 5.

Fig. 6. Server Logging

```
Java version: 23.0.1 (Oracle Corporation)
Java Memory: 500.00MB/7988.00MB (5.44Mb used) [1.09%]
Attempting to listen on default port (21)
Server listening on /0.0.0.0:21
WAN Address: 10.11.12.61
LAN Address: artemis/10.11.12.61
Root directory: C:\Users\jason\OneDrive - University of North Florida\9_fall_2024\cnt6707_network-client-server-arch\project\server
Maximum Transmission Unit (MTU): 1500 bytes
TCP buffer size: 1460 bytes
UDP buffer size: 1460 bytes
Server ready to accept client connections.
Waiting for client connections...
Accepted connection from: /127.0.0.1
ClientHandler initialized for: /127.0.0.1
Started thread for client: /127.0.0.1
Handling client connection from: /127.0.0.1
Received command from /127.0.0.1: CD test
Changed directory to: C:\Users\jason\OneDrive - University of North Florida\9_fall_2024\cnt6707_network-client-server-arch\project\server\test for client: /127.0.0.1
Received command from /127.0.0.1: LS
LS command executed by /127.0.0.1
Received command from /127.0.0.1: GET file1MB.dat
Received command from /127.0.0.1: PUT file1MB.dat 1048576
Received command from /127.0.0.1: MODE
Received command from /127.0.0.1: GET file1MB.dat
File transfer completed successfully to: /127.0.0.1
Received command from /127.0.0.1: PUT file1MB.dat 1048576
File upload completed in 88 ms. Total bytes transferred: 1077336
File upload completed successfully from: /127.0.0.1
Received command from /127.0.0.1: QUIT
Client issued QUIT. Closing connection for: /127.0.0.1
Client connection closed for: /127.0.0.1
q
Shutting down the server...
```

Both client and server write to the console and a log file, with the client being more verbose and the server mostly echoing commands to the console and log file that are sent by clients as seen in Figure 6. The server has a **shutdownListener** thread that listens for the character 'q' followed by enter to shut down the server, while the client presents an option via the menu that quits the client and notifies the server to shut down the connection and **ClientHandler**.

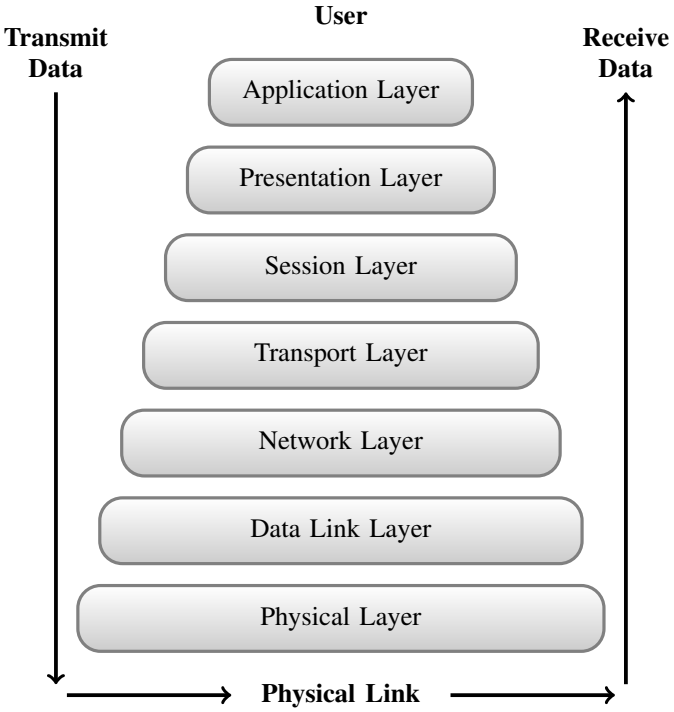
The server maintains a per-client current directory and handles folder navigation within the root folder, preventing traversal to parent directories for security reasons.

Fig. 7. Client TCP GET/PUT Commands

```
FTP Client Menu:
1) GET
2) PUT
3) CD
4) LS
5) Toggle Transfer Mode ([TCP]/UDP)
6) Toggle Testing Mode (ON/[OFF])
7) QUIT
Enter choice: 1
Enter file name to download: file1MB.dat
|================================================| 100% (1077336/1077336 bytes)
GET transfer of file1MB.dat complete.

GET of file1MB.dat completed in 53 ms
File size: 1048576 bytes
Total bytes transferred: 1077336 bytes
Throughput: 20327094 b/s

FTP Client Menu:
1) GET
2) PUT
3) CD
4) LS
5) Toggle Transfer Mode ([TCP]/UDP)
6) Toggle Testing Mode (ON/[OFF])
7) QUIT
Enter choice: 2
Enter file name to upload: file1MB.dat
|================================================| 100% (1077336/1077336 bytes)
PUT transfer of file1MB.dat complete.

PUT of file1MB.dat completed in 47 ms
File size: 1048576 bytes
Total bytes transferred: 1077336 bytes
Throughput: 22922042 b/s
```

For the TCP implementation of file transfers (Figure 7), the server/client exchange a ready message: "READY [PORT]

Fig. 8. OSI Model [4]



[FILESIZE]" and establish a new TCP socket to send the data. Data is sent in data segments of 1,460 bytes to stay within the typical Maximum Transmission Unit (MTU) of 1,500 bytes. This size accounts for the TCP header (20 bytes) and IP header (20 bytes), totaling 1,500 bytes (20 + 20 + 1,460). Since the MTU represents the maximum size of the IP packet payload that the Ethernet frame can carry, we do not need to account for the Ethernet header (14 bytes) in our application.

For the UDP implementation of file transfers (Figure 9), the server/client exchange the same ready message before establishing a **DatagramSocket**. The packets are assembled with a sequence number (8 bytes), data (1,460 bytes), and CRC value (4 bytes) (*java.util.zip.CRC32*) and this datagram is sent over the socket. Our application thus consists of these 12 bytes of application-specific data. Our application data combined with the IP header (20 bytes) and the UDP header (8 bytes), once again comes in at the MTU value of 1,500 bytes. When the packet is received, it is received by a **PacketHandler** thread that was created as to not block the socket connection with packet processing. This thread stores the incoming packets in a map (*java.util.map*) using the sequence number as a key and CRC checks on the receiving end to deal with out-of-order packets.

During development sequence numbers were used in conjunction with Acknowledgments (ACKs). Unfortunately, this leads to two-way communication that decreases throughput, as the sender waits to send the next packet. ACKs were removed in the final version of the code. Negative Acknowledgment (NACKs) to request packet be resent were also considered as a reliability measure, but were outside the scope of the project description. In order to deal with out-of-order packets, some

Fig. 9. Client UDP GET/PUT Commands

```
FTP Client Menu:
1) GET
2) PUT
3) CD
4) LS
5) Toggle Transfer Mode ([TCP]/UDP)
6) Toggle Testing Mode (ON/[OFF])
7) QUIT
Enter choice: 5
Transfer mode switched to UDP

FTP Client Menu:
1) GET
2) PUT
3) CD
4) LS
5) Toggle Transfer Mode (TCP/[UDP])
6) Toggle Testing Mode (ON/[OFF])
7) QUIT
Enter choice: 1
Enter file name to download: file1MB.dat
|==================================================| 100% (1077336/1077336 bytes) CRC32: 0xf4f0ee61 Segment: 718
GET transfer of file1MB.dat complete.

GET of file1MB.dat completed in 104 ms
File size: 1048576 bytes
Total bytes transferred: 1077336 bytes
Throughput: 10359000 b/s

FTP Client Menu:
1) GET
2) PUT
3) CD
4) LS
5) Toggle Transfer Mode (TCP/[UDP])
6) Toggle Testing Mode (ON/[OFF])
7) QUIT
Enter choice: 2
Enter file name to upload: file1MB.dat
|==================================================| 100% (1077336/1077336 bytes) CRC32: 0xf4f0ee61 Segment: 719
PUT transfer of file1MB.dat complete.

PUT of file1MB.dat completed in 94 ms
File size: 1048576 bytes
Total bytes transferred: 1077336 bytes
Throughput: 11461021 b/s
```

form of buffering the packets for processing was necessary. The UDP buffer itself, even set to a high value via *setReceiveBufferSize*, was insufficient for reliable transfers. Both server and client for TCP and UDP send an additional one packet with the file containing the value -1 to signal the end-of-file.

A progress bar updates live during the transfers using the **transferDisplay** method on the client. During UDP transfers, the sequence number and CRC value for the current packet are displayed.

II. Client-Server Computing APIs

The two protocols for this project, TCP and UDP operate at the Transport Layer. They were implemented in Java 23 according to their respective APIs[5][6].

A. TCP Socket API [5]

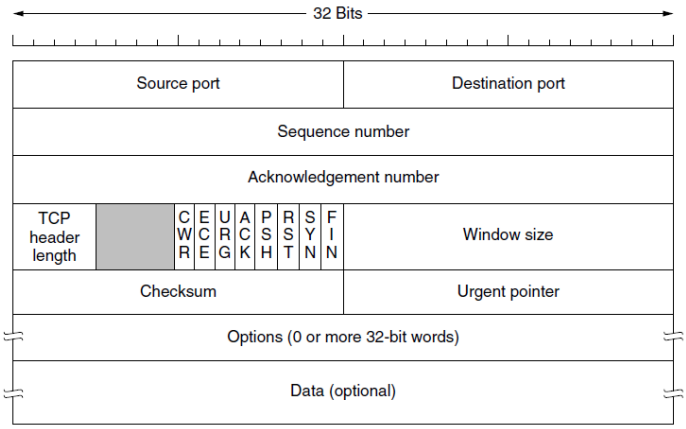
TABLE I
TCP Packet Composition in Bytes [8]

Component	Description	Size (B)
TCP Header	Transmission Control Protocol Header	20
IP Header	Internet Protocol Header	20
Data	Payload (Application Data)	1,460
Total	Total Packet Size (MTU)	1,500

In Java, TCP sockets are used to establish and manage communication between a client and a server using the Transmission Control Protocol (TCP). TCP is a connection-oriented protocol that provides reliable, ordered, and error-checked delivery of data [7].

1) Creating a Server:

Fig. 10. TCP Header [7]



- The server creates a ServerSocket object, which listens for incoming client connections on a specific port.
- When a client connects, the ServerSocket accepts the connection and returns a Socket object for communication with the client.

2) Creating a Client:
- The client creates a Socket object to connect to a server by specifying the server's hostname (or IP address) and port.
- Once connected, the client uses input and output streams to exchange data with the server.

3) Communication:
- Data is exchanged using input and output streams (InputStream and OutputStream).
- Streams can handle raw bytes, characters, or structured data, depending on how they're wrapped (e.g., with BufferedReader or DataOutputStream).

4) Lifecycle:
- Server:
  a) Create ServerSocket.
  b) Accept connections with accept().
  c) Communicate using the Socket object.
  d) Close Socket and ServerSocket when done.
- Client:
  a) Create a Socket and connect to the server.
  b) Communicate using input/output streams.
  c) Close Socket when done.

B. UDP Socket API [6]

UDP does almost nothing besides sending packets between applications, letting applications build their own protocols on top as needed. [7]

1) Creating a Server:
- The server creates a DatagramSocket bound to a specific port.
- It waits for incoming packets using the receive() method.

Fig. 11. UDP Header[7][9]

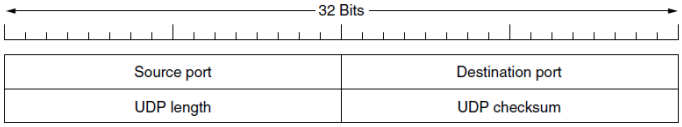


TABLE II
UDP PACKET COMPOSITION [9]

Component	Description	Size (B)
IP Header	Internet Protocol Header	20
UDP Header	User Datagram Protocol Header	8
Sequence Number	Unique Identifier for the Packet	8
Data	Payload (Application Data)	1,460
CRC	Cyclic Redundancy Check	4
Total	Total Packet Size (MTU)	1,500

- Processes the received data and optionally sends a response.

2) Creating a Client:
   - The client creates a DatagramSocket and sends data to the server using the send() method.
   - It may receive a response using the receive() method.
3) Communication:
   - Data is sent and received as discrete packets using DatagramPacket.
   - Each packet contains:
     - Sequence number
     - Data payload
     - Data payload CRC
     - UDP Header (Figure 11)
4) Lifecycle:
   - Server:
     a) Create DatagramSocket.
     b) Receive packets using receive().
     c) Process and respond.
     d) Close DatagramSocket when done.
   - Client:
     a) Create DatagramSocket.
     b) Send data using send().
     c) Receive response (optional) using receive().
     d) Close DatagramSocket.

Reliability was added to UDP by utilizing CRC checks on the data of each packet. This is checked on the receiving end, and the transfer fails and has to be restarted if a CRC check fails. This could be improved using ACK/NACK packets.

C. Comparison of APIs

As described previously, the data contained in Table IV were used to resize the data segment of the packets to ensure the packet size did not exceed the MTU, and to calculate throughput. Because TCP is connection-oriented [7], we added an additional 7 40 byte packets for connection establishment (Figure 12[7]) and shutdown. This overhead calculation is shown in Equation 1. For UDP, the *Protocol Establishment* value is 0.

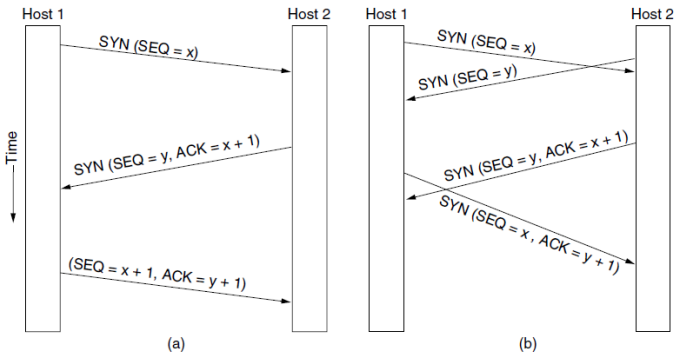
TABLE III
SUMMARY OF TCP AND UDP PROTOCOLS [7]

Aspect	TCP	UDP
Connection Type	Connection-oriented	Connectionless
Reliability	High (built-in error checking and ACKs)	Low (requires manual implementation)
Transfer Speed	Moderate (due to overhead)	High (less overhead)
Implementation Effort	Moderate	Higher (need to handle reliability)
Best Used For	File transfers requiring reliability	Streaming where speed is critical

TABLE IV
OVERHEAD SIZES FOR TCP/UDP PROTOCOLS [7][8][9]

Data	Size (B)
Ethernet MTU	1,500
TCP Overhead	20
IP Overhead	20
UDP Overhead	8
Application Overhead (Segment + CRC)	12
TCP/IP Overhead	40
UDP/IP/Application Overhead	40
TCP Data Size	1,460
UDP Data Size	1,460
TCP EOF Packet	48
UDP EOF Packet	36
TCP Negotiation [7 packets]	280

Fig. 12. Establishing an TCP Connection[7]



$$\text{Size} = (\text{Data Size} + \text{Protocol Overhead}) \times \left\lceil \frac{\text{File Size}}{\text{Data Size}} \right\rceil + \text{EOF} + \text{Protocol Establishment} \quad (1)$$

III. RESULTS AND COMPARISONS

A. Test Bed Specification

Both client/server systems used Microsoft Windows 11 (10.0) x86, with 7988.00 MB of memory allocated to Oracle Java 23.0.1.

B. Experiments Conducted

File sizes tested were 1 MB, 25 MB, 50 MB, and 100 MB. They were generated using the scripts in A. Testing mode was enabled on the client, running a total of 10 transfers in each direction (GET/PUT) on across a LAN.

C. Metrics Measured

- Transfer time (seconds).
- Throughput (bytes per second).
- Total bytes transferred (bytes).
- Number of packets

D. Results

TABLE V
FTP SERVER/CLIENT TCP GET RESULTS

File Size (B)	Transferred (B)	Transfer Time (s)	Throughput (B/s)	Number of Packets
131,072	135,328	0.33	411,331	90
3,276,800	3,367,828	6.9	488,374	2,245
6,553,600	6,733,828	16.5	408,111	4,489
13,107,200	13,467,328	28.03	480,461	8,978

TABLE VI
FTP SERVER/CLIENT UDP GET RESULTS

File Size (B)	Transferred (B)	Transfer Time (s)	Throughput (B/s)	Number of Packets
131,072	135,328	0.2	669,941	90
3,276,800	3,367,828	2.59	1,299,316	2,245
6,553,600	6,733,828	6.17	1,091,736	4,489
13,107,200	13,467,328	10.8	1,247,552	8,978

TABLE VII
FTP SERVER/CLIENT TCP PUT RESULTS

File Size (B)	Transferred (B)	Transfer Time (s)	Throughput (B/s)	Number of Packets
131,072	135,036	0.23	594,872	90
3,276,800	3,367,536	5.46	617,330	2,245
6,553,600	6,733,536	11.25	598,377	4,489
13,107,200	13,467,036	24.24	555,594	8,978

TABLE VIII
FTP SERVER/CLIENT UDP PUT RESULTS

File Size (B)	Transferred (B)	Transfer Time (s)	Throughput (B/s)	Number of Packets
131,072	135,036	0.15	888,395	90
3,276,800	3,367,536	1.92	1,757,587	2,245
6,553,600	6,733,536	3.73	1,805,722	4,489
13,107,200	13,467,036	7.08	1,902,124	8,978

Fig. 13. TCP/UDP PUT Offered Load (B) vs. Response Time (B/s)

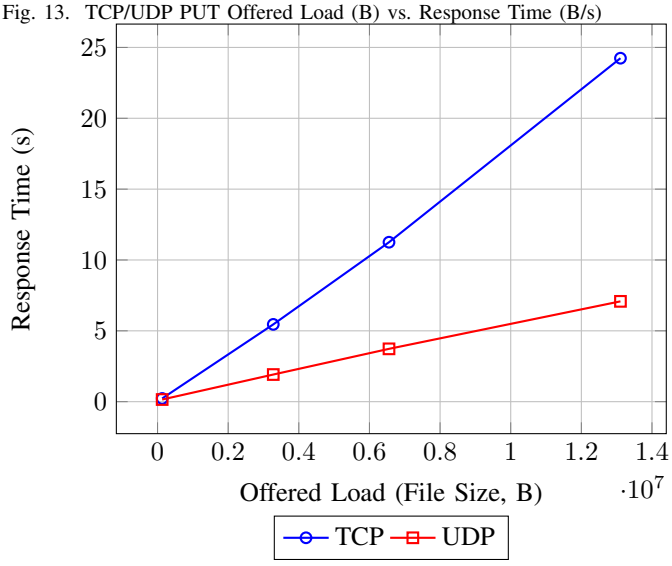


Fig. 14. TCP/UDP GET Offered Load (B) vs. Response Time (B/s)

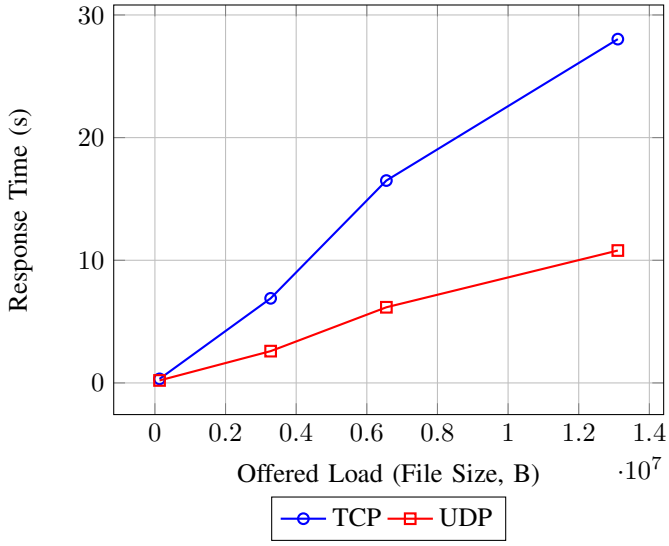
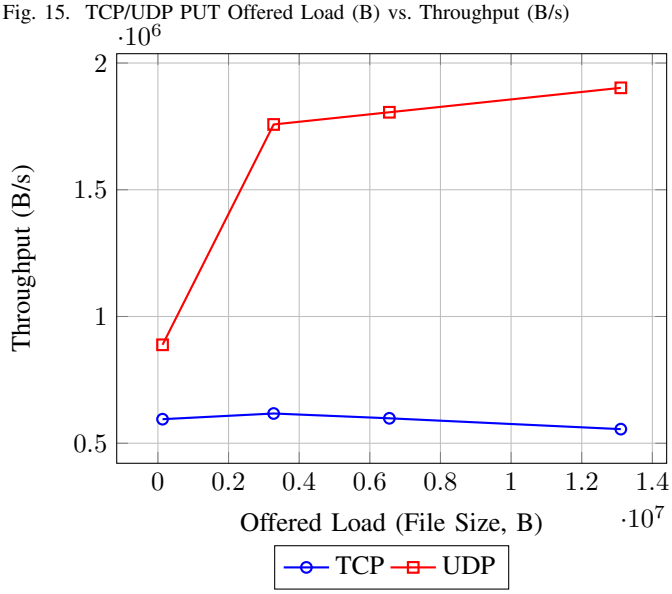


Fig. 15. TCP/UDP PUT Offered Load (B) vs. Throughput (B/s)



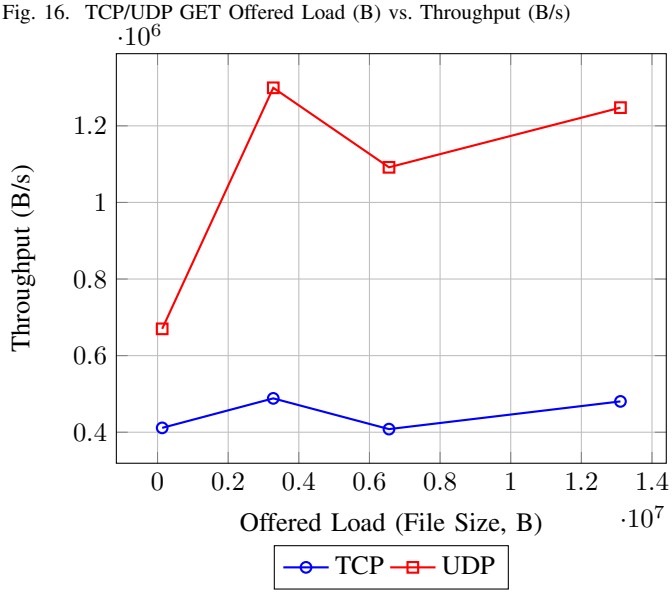
Results are shown in Table V, Table VI, Table VII, Table VIII, Figure 13, Figure 14, Figure 15, and Figure 16.

These results show that although UDP transfers achieved higher throughput due to lower protocol overhead, the additional processing time required for reliability checks resulted in TCP transfers completing faster overall. It was expected that UDP would be faster by both metrics. We suspect that because TCP is highly optimized and performs CRC checks on the header data rather than the full data packets, even a simple reliability check like the one implemented for UDP in this project caused UDP to have greater overhead, resulting in TCP completing transfers faster in our test environment.

IV. CONCLUSIONS

Based on the data, it is a reasonable conclusion that TCP is the ideal protocol for file transfers. The higher throughput

Fig. 16. TCP/UDP GET Offered Load (B) vs. Throughput (B/s)



observed in UDP transfers is attributed to the minimal protocol overhead inherent in UDP. However, the necessity to implement reliability checks at the application layer introduced additional processing delays. These delays offset the benefits of UDP's lower overhead, resulting in TCP completing transfers faster in our test environment.

A. Performance Comparison

While UDP has the potential to send data faster due to its minimal protocol overhead, the additional processing time required for implementing CRC checks at the application layer offset this advantage. Consequently, TCP ultimately performed transfers faster in our tests. This balance between speed and reliability is the primary consideration between TCP and UDP protocols. TCP is easier to work with, handles all of the reliability, and even flow control, transparent to the user or programmer. If you need to implement any of the features manually with UDP, it can end up being slower.

B. Best Use Cases

UDP excels in applications like video streaming, where high throughput and low latency are crucial, and some data loss is acceptable. While UDP achieved higher throughput due to its lower protocol overhead, the additional overhead of implementing reliability at the application layer negated these advantages, leading to longer overall transfer times compared to TCP. When used properly, like in video streaming, UDP shines. Dropped frames when modern high definition video is anywhere between 30 and 120 frames per second is unlikely to be noticed by the end user. For files, however, every bit matters.

C. Insights Gained

During this project, we learned that UDP is an extremely fast protocol, but that simple things like making sure data is processed in the correct order and ensuring reliability are made more difficult when it is the chosen protocol. UDP is something we will consider using in the future if reliability isn't a priority.

We learned how to implement both TCP and UDP protocols. This work ultimately left us curious, along with some work we was exposed to about lower-level protocols during the term, how one might implement a protocol at the Network layer without relying on TCP or UDP. This is a direction we would like to pursue in the future.

References

[1] Oracle Corporation. (2024) *Oracle Java SE Support Roadmap* [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html. [Accessed: 29 November 2024].

[2] Red Hat. (2024) *OpenJDK Life Cycle and Support Policy - Red Hat Customer Portal* [Online]. Available: https://access.redhat.com/articles/1299013. [Accessed: 29 November 2024].

[3] D. S. Ahuja, "Comparison of tcp and udp client-server apis," 2024, [Handout CNT6707 Fall 2024].

[4] dev-gb. (2024) *TikZ-osi-model* [Online]. Available: https://github.com/dev-gb/TikZ-osi-model. [Accessed: 29 November 2024].

[5] Oracle Corporation. (2024) *Socket (Java SE 23 & JDK 23)* [Online]. Available: https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/net/Socket.html. [Accessed: 29 November 2024].

[6] ——. (2024) *DatagramSocket (Java SE 23 & JDK 23)* [Online]. Available: https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/net/DatagramSocket.html. [Accessed: 29 November 2024].

[7] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. USA: Prentice Hall Press, 2010.

[8] Internet Engineering Taskforce (IETF). (2024) *RFC 793* [Online]. Available: https://datatracker.ietf.org/doc/html/rfc793. [Accessed: 29 November 2024].

[9] ——. (2024) *RFC 768* [Online]. Available: https://datatracker.ietf.org/doc/html/rfc768. [Accessed: 29 November 2024].

APPENDIX

FTPServer.java

```
/* Author:  Jason Gardner (n01480000),
 * Date: 23 October 2024
 * Project: Project 2
 * File: FTPServer.java
 * CNT6707 - Network Architecture and Client/Server Computing
 * Description: Mutlithreaded FTP server program that uses threads to handle multiple clients
 *              Commands: GET, PUT, CD, LS, QUIT
 *              Transfer modes: TCP, UDP
 *              Testing mode: GET/PUT performed NUM_TESTS times and average time/throughput is calculated
 */

import java.io.*;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.FileChannel;
import java.util.Scanner;
import java.util.TreeMap;
import java.util.Arrays;
import java.util.zip.CRC32;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;
import java.util.logging.FileHandler;
import java.util.logging.Logger;

/**
 * FTPServer handles incoming FTP client connections, executing commands such as LS, CD, GET, and PUT.
* It uses a single persistent connection and ensures clean file transfers with proper stream management.
*/
public class FTPServer {
    private static int listenPort = 21;
    private static boolean running = true; // Server running flag
    private static ServerSocket serverSocket; // Class-level ServerSocket for handling shutdown
    private static final Logger LOGGER = Logger.getLogger("FTPServer");
    private static final int MTU = 1500;  // Maximum Transmission Unit (MTU) for Ethernet
    private static final int IP_OVERHEAD = 20; // 20 bytes for IP header
    private static final int TCP_OVERHEAD = 20; // 20 bytes for TCP header
    private static final int TCP_IP_OVERHEAD = IP_OVERHEAD + TCP_OVERHEAD; // Total TCP/IP overhead
    private static final int TCP_BUFFER_SIZE = MTU - TCP_IP_OVERHEAD; // Final payload size
    private static final int UDP_OVERHEAD = 8; // 8 bytes for UDP header
    private static final int APPLICATION_OVERHEAD = Long.BYTES + Integer.BYTES; // 8 bytes for sequence + 4 bytes for ↩
        ↪ CRC
    private static final int UDP_IP_OVERHEAD = IP_OVERHEAD + UDP_OVERHEAD; // Total UDP/IP overhead
    private static final int UDP_IP_APPLICATION_OVERHEAD = UDP_IP_OVERHEAD + APPLICATION_OVERHEAD; // 8 bytes for ↩
        ↪ sequence + 4 bytes for CRC
    private static final int UDP_BUFFER_SIZE = MTU - UDP_IP_APPLICATION_OVERHEAD; // Final payload size
    private static final int TIMEOUT = 2000; // Timeout in milliseconds
    private static final int UDP_RECV_BUFFER = 100000000; // 100MB buffer size for UDP
    private static final int UDP_DELAY = 0; // Delay in milliseconds for UDP mode

    private static class PacketHandler extends Thread {
        private final DatagramSocket socket;
        private final FileOutputStream fos;
        //private final long expectedFileSize;
        //private final int timeout;
        private final long startTime;
        private long totalBytesTransferred = 0;
        private volatile boolean transferActive = true;
        private long bytesPerFile = 0;

        private PacketHandler(DatagramSocket socket, FileOutputStream fos, long expectedFileSize, int timeout) {
            this.socket = socket;
            this.fos = fos;
            //this.expectedFileSize = expectedFileSize;
            //this.timeout = timeout;
            this.startTime = System.currentTimeMillis();
            this.bytesPerFile = expectedFileSize + UDP_IP_APPLICATION_OVERHEAD * (int)Math.ceil((double) ↩
                ↪ expectedFileSize/UDP_BUFFER_SIZE);
        }

        @Override
        public void run() {
```

```
            try {
                long expectedSequence = 0;
                Map<Long, byte[]> packetBuffer = new TreeMap<>();

                byte[] buffer = new byte[Long.BYTES + UDP_BUFFER_SIZE + Integer.BYTES];

                while (transferActive && totalBytesTransferred < bytesPerFile) {
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    try {
                        socket.receive(packet);
                    } catch (SocketTimeoutException e) {
                        if (totalBytesTransferred >= bytesPerFile) {
                            break;
                        }
                        printAndLog("Timeout waiting for next packet from client. Aborting transfer.");
                        transferActive = false;
                        return;
                    }

                    ByteBuffer byteBuffer = ByteBuffer.wrap(packet.getData(), 0, packet.getLength());
                    byteBuffer.order(ByteOrder.BIG_ENDIAN);
                    long sequenceNumber = byteBuffer.getLong();

                    if (sequenceNumber == -1L) {
                        transferActive = false;
                        break;
                    }

                    int dataLength = packet.getLength() - Long.BYTES - Integer.BYTES;
                    if (dataLength <= 0) {
                        printAndLog("Invalid packet received from client. Skipping.");
                        continue;
                    }

                    byte[] data = new byte[dataLength];
                    byteBuffer.get(data);
                    int receivedChecksum = byteBuffer.getInt();

                    CRC32 crc = new CRC32();
                    crc.update(data, 0, dataLength);
                    long calculatedChecksum = crc.getValue() & 0xFFFFFFFFL;
                    if (calculatedChecksum != (receivedChecksum & 0xFFFFFFFFL)) {
                        printAndLog("CRC mismatch for sequence " + sequenceNumber + " from client. Ignoring packet.");
                        continue;
                    }

                    // Add packet to buffer for reassembly
                    packetBuffer.put(sequenceNumber, data);

                    while (packetBuffer.containsKey(expectedSequence)) {
                        byte[] nextData = packetBuffer.remove(expectedSequence);
                        fos.write(nextData);
                        totalBytesTransferred += (nextData.length + UDP_IP_APPLICATION_OVERHEAD); // data size + TCP ↩
                            ↪ Header + IP Header
                        expectedSequence++;
                    }
                }

                fos.flush();
            } catch (IOException e) {
                printAndLog("Error in packet handler: " + e.getMessage());
            } finally {
                try {
                    fos.close();
                } catch (IOException e) {
                    printAndLog("Error closing file: " + e.getMessage());
                }
                socket.close();
                long duration = System.currentTimeMillis() - startTime;
                printAndLog("File upload completed in " + duration + " ms. Total bytes transferred: " + ↩
                    ↪ totalBytesTransferred);
            }
        }

        //public long getTotalBytesTransferred() {
        //    return totalBytesTransferred;
        //}
    }

    public static void main(String[] args) throws IOException {
        LogToFile.logToFile(LOGGER, "FTPServer.log"); // Log to file
        printAndLog("Logging to FTPServer.log");
        printAndLog("Starting FTP server...");
        final String javaVersion = System.getProperty("java.version");
        final String javaVendor = System.getProperty("java.vendor");
        final String osVersion = System.getProperty("os.version");
```

```
        final String osName = System.getProperty("os.name");
        final String osArch = System.getProperty("os.arch");
        final Runtime runtime = Runtime.getRuntime();
        final long totalMemory = runtime.totalMemory();
        final long maxMemory = runtime.maxMemory();
        final long freeMemory = runtime.freeMemory();
        final long usedMemory = totalMemory - freeMemory;
        final String MEMORY_FORMAT = "Java Memory: %.2fMb/%.2fMb (%.2fMb used) [%.2f%%]";
        printAndLog("OS: " + osName + " " + osVersion + " (" + osArch + ")");
        printAndLog("Java version: " + javaVersion + " (" + javaVendor + ")");
        printAndLog(String.format(MEMORY_FORMAT, totalMemory / 1048576.0, maxMemory / 1048576.0, usedMemory / 1048576.0,↩
            ↪  (usedMemory * 100.0) / totalMemory));

        if (args.length > 0) {
            listenPort = Integer.parseInt(args[0]);
        } else {
            printAndLog("Attempting to listen on default port (" + listenPort + ")");
        }

        // Start the server shutdown listener (listens for "q" to quit)
        new Thread(FTPServer::shutdownListener).start();

        try {
            serverSocket = new ServerSocket(listenPort, 50, InetAddress.getByName("0.0.0.0")); // Bind to all interfaces
            printAndLog("Server listening on " + serverSocket.getInetAddress() + ":" + serverSocket.getLocalPort());
            try (final DatagramSocket datagramSocket = new DatagramSocket()) {
                datagramSocket.connect(InetAddress.getByName("8.8.8.8"), 12345);
                printAndLog("WAN Address: " + datagramSocket.getLocalAddress().getHostAddress());
                // Display server interface IP address
                printAndLog("LAN Address: " + InetAddress.getLocalHost());
            } catch (Exception e) {
                printAndLog("Server WAN IP: Unable to Reach Google DNS: " + e.getMessage());
            }

            printAndLog("Root directory: " + System.getProperty("user.dir"));
            printAndLog("Maximum Transmission Unit (MTU): " + MTU + " bytes");
            printAndLog("TCP buffer size: " + TCP_BUFFER_SIZE + " bytes");
            printAndLog("UDP buffer size: " + UDP_BUFFER_SIZE + " bytes");
            printAndLog("Server ready to accept client connections.");
            printAndLog("Waiting for client connections...");

            // Main loop to accept client connections
            while (running) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    printAndLog("Accepted connection from: " + clientSocket.getInetAddress());

                    // Handle client connection in a new thread
                    Thread clientThread = new Thread(new ClientHandler(clientSocket));
                    clientThread.start();
                    printAndLog("Started thread for client: " + clientSocket.getInetAddress());

                } catch (IOException e) {
                    if (running) { // Only log if still running
                        printAndLog("Error accepting connection: " + e.getMessage());
                    }
                }
            }

        } catch (IOException e) {
            if (running) {
                printAndLog("Could not listen on port " + listenPort + ": " + e.getMessage());
            }
        } finally {
            if (serverSocket != null && !serverSocket.isClosed()) {
                serverSocket.close();
            }
        }
    }

    /**
     * Listens for "q" input to shut down the server.
    */
    private static void shutdownListener() {
        Scanner scanner = new Scanner(System.in);
        while (running) {
            if (scanner.nextLine().equalsIgnoreCase("q")) {
                running = false;
                printAndLog("Shutting down the server...");
                try {
                    if (serverSocket != null && !serverSocket.isClosed()) {
                        serverSocket.close();  // Close the server socket to unblock accept()
                    }
                } catch (IOException e) {
                    printAndLog("Error closing the server socket: " + e.getMessage());
                }
```

```
                break;
            }
        }
        scanner.close();
    }

    /**
     * ClientHandler class to handle individual client connections in separate threads.
    * Each client connection is handled by a separate instance of this class.
    * The class implements the Runnable interface to run in a separate thread.
    * The class handles the LS, CD, GET, PUT, and QUIT commands.
    * The class maintains the current directory for each client.
    * The class uses a static ROOT_DIR for the server root directory.
    */
    private static class ClientHandler implements Runnable {
        private final Socket clientSocket;
        private final String clientAddress; // Store client address for logging
        private static final String ROOT_DIR = System.getProperty("user.dir");
        private String currentDir;
        private boolean udpMode = false; // UDP mode flag

        ClientHandler(Socket clientSocket) {
            this.clientSocket = clientSocket;
            this.clientAddress = clientSocket.getInetAddress().toString(); // Capture client address
            this.currentDir = ROOT_DIR; // Start in the root directory
            printAndLog("ClientHandler initialized for: " + clientAddress);
        }

        /**
         * Run method to handle client connections and execute commands.
        */
        @Override
        public void run() {
            printAndLog("Handling client connection from: " + clientAddress);
            try (
                BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)
            ) {
                String inputLine;
                while ((inputLine = in.readLine()) != null) {
                    printAndLog("Received command from " + clientAddress + ": " + inputLine); // Log with client info
                    String[] command = inputLine.split(" ");
                    switch (command[0].toUpperCase()) {
                        case "LS":
                            handleLS(out);
                            break;
                        case "CD":
                            handleCD(command, out);
                            break;
                        case "GET":
                            handleGET(command, out);
                            break;
                        case "PUT":
                            handlePUT(command, out, in);  // Pass 'in' to handlePUT
                            break;
                        case "MODE":
                            udpMode = !udpMode; // Toggle UDP mode
                            break;
                        case "QUIT":
                            handleQUIT(out);
                            return;  // Close this client handler after QUIT
                        default:
                            out.println("Unknown command");
                            break;
                    }
                }
            // Handle exceptions and close the client connection
            } catch (IOException e) {
                printAndLog("Exception in client handling for " + clientAddress + ": " + e.getMessage());
            }
        }

        /**
         * Handles the LS command to list files in the current directory in the desired format.
        * @param out The output writer to communicate with the client.
        */
        /**
         * Handles the LS command to list files in the current directory in the desired format.
        * @param out The output writer to communicate with the client.
        */
        private void handleLS(PrintWriter out) {
            File dir = new File(currentDir);
            File[] files = dir.listFiles();

            if (files != null) {
                Arrays.sort(files, (f1, f2) -> f1.getName().compareToIgnoreCase(f2.getName()));
```

```
            String OUTPUT_FORMAT = "  %-50s %-30s";

            out.println("Directory: " + currentDir);
            out.println(OUTPUT_FORMAT.formatted("Name", "Size"));
            out.println(OUTPUT_FORMAT.formatted(".", "<DIR>"));
            out.println(OUTPUT_FORMAT.formatted("..", "<DIR>"));

            for (File file : files) {
                if (file.isDirectory()) {
                    out.println(OUTPUT_FORMAT.formatted("/" + file.getName() + "/", "<DIR>"));
                }
            }

            for (File file : files) {
                if (!file.isDirectory()) {
                    out.println(OUTPUT_FORMAT.formatted(file.getName(), file.length() + " bytes"));
                }
            }
        }
        out.println("EOF"); // Mark the end of listing
        out.flush();
        printAndLog("LS command executed by " + clientAddress);
    }

    /**
     * Handles the CD command to change the current directory.
    * @param command The command array containing the directory to change to.
    * @param out The output writer to communicate with the client.
    */
    private void handleCD(String[] command, PrintWriter out) {
        if (command.length > 1) {
            File newDir = new File(currentDir + File.separator + command[1]);
            try {
                if (newDir.isDirectory() && newDir.getCanonicalPath().startsWith(ROOT_DIR)) {
                    currentDir = newDir.getCanonicalPath(); // Update current directory
                    out.println("Changed directory to: " + currentDir);
                    printAndLog("Changed directory to: " + currentDir + " for client: " + clientAddress);
                } else {
                    out.println("Directory not found or permission denied.");
                }
            } catch (IOException e) {
                printAndLog("Error changing directory for " + clientAddress + ": " + e.getMessage());
            }
        } else {
            out.println("ERROR: No directory specified.");
        }
        out.flush();
    }

    /**
     * Handles the QUIT command, closing the client connection.
    * @param out The output writer to communicate with the client.
    * @throws IOException If an I/O error occurs.
    */
    private void handleQUIT(PrintWriter out) throws IOException {
        out.println("Goodbye!"); // Inform the client the server is closing the connection
        printAndLog("Client issued QUIT. Closing connection for: " + clientAddress);

        // Close the client socket
        clientSocket.close();
        printAndLog("Client connection closed for: " + clientAddress);
    }

    /**
     * Handles the GET command for file download.
    * @param command The command array containing the file to download.
    * @param out The output writer to communicate with the client.
    * @throws IOException If an I/O error occurs while sending the file.
    */
    private void handleGET(String[] command, PrintWriter out) throws IOException {
        if (command.length > 1) {
            File file = new File(currentDir + File.separator + command[1]);
            if (file.exists() && !file.isDirectory()) {
                long fileSize = file.length();  // Get file size
                if (!udpMode) {
                    try (ServerSocket transferSocket = new ServerSocket(0)) {
                        out.println("READY " + transferSocket.getLocalPort() + " " + fileSize);  // Send file size
                        try (Socket fileTransferSocket = transferSocket.accept();
                            FileInputStream fis = new FileInputStream(file);
                            BufferedOutputStream bos = new BufferedOutputStream(fileTransferSocket.getOutputStream()↩
                                ↪ )) {
                            byte[] buffer = new byte[TCP_BUFFER_SIZE];
                            int bytesRead;
                            while ((bytesRead = fis.read(buffer)) != -1) {
                                bos.write(buffer, 0, bytesRead);
                            }
```

```
                            bos.flush();
                        }
                    }
                } else {
                    // UDP mode
                    DatagramSocket datagramSocket = new DatagramSocket(); // Create DatagramSocket for sending data
                    datagramSocket.setSoTimeout(5000); // Timeout for receiving packets
                    InetAddress clientAddress = clientSocket.getInetAddress(); // Client IP
                    out.println("READY " + datagramSocket.getLocalPort() + " " + fileSize);  // Server tells client ↩
                        ↪ it's ready

                    // Wait for the client to send its local port
                    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                    String clientResponse = in.readLine();
                    if (clientResponse != null && clientResponse.startsWith("CLIENT_READY")) {
                        int clientPort = Integer.parseInt(clientResponse.split(" ")[1]);  // Get client's port

                        // Start sending file data
                        FileInputStream fileInputStream = new FileInputStream(file);
                        CRC32 crc = new CRC32();
                        byte[] buffer = new byte[UDP_BUFFER_SIZE];
                        int bytesRead;

                        // Initialize sequence number
                        long sequenceNumber = 0;

                        // Read and send data packets
                        while ((bytesRead = fileInputStream.read(buffer)) != -1) {
                            // Calculate CRC32 for the data
                            crc.update(buffer, 0, bytesRead);
                            int checksum = (int) crc.getValue(); // Use int for CRC32

                            // Create a packet with sequence number, data, and CRC32
                            // Format: [sequence number (8 bytes)][data (MTU)][CRC32 checksum (4 bytes)]
                            ByteBuffer byteBuffer = ByteBuffer.allocate(Long.BYTES + bytesRead + Integer.BYTES);
                            byteBuffer.order(ByteOrder.BIG_ENDIAN); // Ensure consistent byte order
                            byteBuffer.putLong(sequenceNumber);     // Sequence number
                            byteBuffer.put(buffer, 0, bytesRead);   // Data
                            byteBuffer.putInt(checksum);            // CRC32 checksum

                            byte[] packetData = byteBuffer.array();
                            DatagramPacket packet = new DatagramPacket(packetData, packetData.length, clientAddress,↩
                                ↪ clientPort);
                            datagramSocket.send(packet); // Send the packet

                            sequenceNumber++;
                            crc.reset(); // Reset CRC for next packet

                            // Introduce a small delay
                            try {
                                Thread.sleep(UDP_DELAY);
                            }
                                catch(InterruptedException e) {
                                printAndLog("Thread interrupted: " + e.getMessage());
                            }
                        }

                        // Send end-of-file signal with sequence number -1
                        ByteBuffer endBuffer = ByteBuffer.allocate(Long.BYTES);
                        endBuffer.order(ByteOrder.BIG_ENDIAN);
                        endBuffer.putLong(-1L); // Special sequence number for EOF
                        DatagramPacket endPacket = new DatagramPacket(endBuffer.array(), endBuffer.capacity(), ↩
                            ↪ clientAddress, clientPort);
                        datagramSocket.send(endPacket);

                        fileInputStream.close();
                        datagramSocket.close();
                        printAndLog("File transfer completed successfully to: " + clientAddress);

                    }
                }
            } else {
                out.println("ERROR: File not found.");
            }
        } else {
            out.println("ERROR: No file specified for GET command.");
        }
        out.flush();
    }

    /**
 * Handles the PUT command for file upload.
 * @param command The command array containing the file to upload.
 * @param out The output writer to communicate with the client.
 * @param in The existing BufferedReader to read client messages.
 * @throws IOException If an I/O error occurs while receiving the file.
```

```
    */
    private void handlePUT(String[] command, PrintWriter out, BufferedReader in) throws IOException {
        if (command.length > 2) {
            final long fileSize;
            try {
                fileSize = Long.parseLong(command[2]);  // Get file size from the client
            } catch (NumberFormatException e) {
                out.println("ERROR: Invalid file size.");
                out.flush();
                return;
            }

            File file = new File(currentDir, command[1]);

            // Attempt to lock the file
            try (RandomAccessFile raf = new RandomAccessFile(file, "rw");
                FileChannel channel = raf.getChannel()) {
                java.nio.channels.FileLock fileLock = channel.tryLock();
                if (fileLock == null) {
                    out.println("ERROR: File is currently in use.");
                    out.flush();
                    return;
                }

                if (!udpMode) {
                    // TCP mode
                    try (ServerSocket transferSocket = new ServerSocket(0);
                        FileOutputStream fos = new FileOutputStream(raf.getFD())) {
                        out.println("READY " + transferSocket.getLocalPort() + " " + fileSize);  // Send file size
                        out.flush();

                        try (Socket fileTransferSocket = transferSocket.accept();
                            BufferedInputStream bis = new BufferedInputStream(fileTransferSocket.getInputStream())) {
                            byte[] buffer = new byte[TCP_BUFFER_SIZE];
                            int bytesRead;
                            long currentBytes = 0;
                            while ((bytesRead = bis.read(buffer)) != -1) {
                                fos.write(buffer, 0, bytesRead);
                                currentBytes += bytesRead;
                            }
                            fos.flush();
                        }
                    }
                } else {
                    // UDP mode
                    DatagramSocket datagramSocket = new DatagramSocket();
                    datagramSocket.setSoTimeout(TIMEOUT);
                    datagramSocket.setReceiveBufferSize(UDP_RECV_BUFFER);

                    out.println("READY " + datagramSocket.getLocalPort() + " " + fileSize);
                    out.flush();

                    // Start the PacketHandler thread
                    FileOutputStream fos = new FileOutputStream(raf.getFD());
                    PacketHandler handler = new PacketHandler(datagramSocket, fos, fileSize, TIMEOUT);
                    handler.start();

                    // Wait for the handler to finish
                    try {
                        handler.join();
                    } catch (InterruptedException e) {
                        printAndLog("File transfer was interrupted: " + e.getMessage());
                        Thread.currentThread().interrupt();
                        return;
                    }

                    printAndLog("File upload completed successfully from: " + clientAddress);
                }
            } catch (IOException e) {
                out.println("ERROR: Could not lock file for writing: " + e.getMessage());
                out.flush();
            }
        } else {
            out.println("ERROR: No file specified for PUT command.");
            out.flush();
        }
    }
}

    /**
     * Utility method to print messages to the console and log them.
    * @param message The message to log.
    */
    private static void printAndLog(String message) {
        System.out.println(message);
        LOGGER.info(message);
```

```
    }

    /**
     * Utility class to set up logging to a file.
     */
    private static class LogToFile {
        public static void logToFile(Logger logger, String logFile) {
            try {
                FileHandler fh = new FileHandler(logFile, true);  // Append mode
                logger.addHandler(fh);

                // Use the custom formatter for the log format
                CustomLogFormatter formatter = new CustomLogFormatter();
                fh.setFormatter(formatter);

                // Disable console output for the logger (remove default handlers)
                logger.setUseParentHandlers(false);
            } catch (SecurityException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        /**
         * Custom log formatter to format log messages with a timestamp and log level.
         */
        private static class CustomLogFormatter extends Formatter {
            // Define the date format
            private static final SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy@HH:mm:ss");

                @Override
                public String format(LogRecord record) {
                    // Get the current date and time
                    String timeStamp = dateFormat.format(new Date(record.getMillis()));

                    // Get the log level (severity)
                    String logLevel = record.getLevel().getName();

                    // Format the log message
                    return String.format("%s:%s:\t%s%n",
                            timeStamp,              // Short date and time
                            logLevel,               // Log level (severity)
                            record.getMessage()     // Actual log message
                    );
                }
            }
    }
}
```

FTPClient.java

```
/* Author:  Jason Gardner (n01480000),
 * Date: 23 October 2024
 * Project: Project 2
 * File: FTPClient.java
 * CNT6707 - Network Architecture and Client/Server Computing
 * Description: FTP server program
 *              Commands: GET, PUT, CD, LS, QUIT
 *              Transfer modes: TCP, UDP
 *              Testing mode: GET/PUT performed NUM_TESTS times and average time/throughput is calculated
 */

import java.io.*;
import java.net.*;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.zip.CRC32;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.TreeMap;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;
import java.util.logging.FileHandler;
import java.util.logging.Logger;

/**
 * FTP client program that connects to an FTP server and allows the user to interact with the server using the following↩
    ↪  commands:
 * 1) GET <file> - Download a file from the server
 * 2) PUT <file> - Upload a file to the server
 * 3) CD <directory> - Change the current directory on the server
 * 4) LS - List the contents of the current directory on the server
 * 5) Switch transfer mode (TCP/UDP)
```

```
 * 6) Enable testing mode (GET/PUT performed NUM_TESTS times and average time/throughput is calculated)
 * 7) QUIT - Disconnect from the server and exit the client
 */
public class FTPClient {
    static final Logger LOGGER = Logger.getLogger("FTPClient"); // Logger for logging to file
    static final int NUM_TESTS = 10;  // Number of tests for testing mode
    static boolean testingMode = false;  // Default to testing mode off
    static boolean udpMode = false;  // Default to TCP mode
    private static final int MTU = 1500;  // Maximum Transmission Unit (MTU) for Ethernet
    private static final int IP_OVERHEAD = 20; // 20 bytes for IP header
    private static final int TCP_OVERHEAD = 20; // 20 bytes for TCP header
    private static final int TCP_IP_OVERHEAD = IP_OVERHEAD + TCP_OVERHEAD; // Total TCP/IP overhead
    private static final int TCP_BUFFER_SIZE = MTU - TCP_IP_OVERHEAD; // Final payload size
    private static final int UDP_OVERHEAD = 8; // 8 bytes for UDP header
    private static final int APPLICATION_OVERHEAD = Long.BYTES + Integer.BYTES; // 8 bytes for sequence + 4 bytes for ↩
        ↪ CRC
    private static final int UDP_IP_OVERHEAD = IP_OVERHEAD + UDP_OVERHEAD; // Total UDP/IP overhead
    private static final int UDP_IP_APPLICATION_OVERHEAD = UDP_IP_OVERHEAD + APPLICATION_OVERHEAD; // 8 bytes for ↩
        ↪ sequence + 4 bytes for CRC
    private static final int UDP_BUFFER_SIZE = MTU - UDP_IP_APPLICATION_OVERHEAD; // Maximum UDP payload size
    private static final int TIMEOUT = 2000;  // Timeout in milliseconds
    private static final int PORT = 21;  // Default port number
    private static final int UDP_RECV_BUFFER = 100000000; // 100MB buffer size for UDP
    private static String serverIP;  // Server IP address
    private static int serverPort;  // Server port number
    private static final int UDP_DELAY = 0;  // Delay in milliseconds for UDP transfer

    @FunctionalInterface
    interface QuadConsumer<A, B, C, D> {
        void accept(A a, B b, C c, D d);
    }

    private static class PacketHandler extends Thread {
        private final DatagramSocket socket;
        private final FileOutputStream fos;
        private final long expectedFileSize;
        private final int timeout;
        private final QuadConsumer<Long, Long, Integer, Long> transferDisplay; // Updated functional interface
        private long totalBytesTransferred = 0; // For metrics
        private long duration = 0; // For metrics
        private volatile boolean transferActive = true;
        private long bytesPerFile = 0;
        private int runNumber = 0;

        private PacketHandler(DatagramSocket socket, FileOutputStream fos, long expectedFileSize,
                              QuadConsumer<Long, Long, Integer, Long> transferDisplay, int timeout, int runNumber) {
            this.socket = socket;
            this.fos = fos;
            this.expectedFileSize = expectedFileSize;
            this.transferDisplay = transferDisplay;
            this.timeout = timeout;
            this.bytesPerFile = expectedFileSize + UDP_IP_APPLICATION_OVERHEAD * (int)Math.ceil((double) ↩
                ↪ expectedFileSize/UDP_BUFFER_SIZE);
            this.runNumber = runNumber;
        }

        @Override
        public void run() {
            long startTime = System.currentTimeMillis();
            try {
                long expectedSequence = 0;
                Map<Long, byte[]> packetBuffer = new TreeMap<>(); // Buffer for out-of-order packets

                byte[] buffer = new byte[Long.BYTES + UDP_BUFFER_SIZE + Integer.BYTES];

                while (transferActive && totalBytesTransferred < bytesPerFile) {
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    try {
                        socket.receive(packet);
                    } catch (SocketTimeoutException e) {
                        if (totalBytesTransferred >= bytesPerFile) {
                            break; // All packets received
                        }
                        printAndLog("Timeout waiting for next packet. Aborting transfer.", true);
                        transferActive = false;
                        return;
                    }

                    ByteBuffer byteBuffer = ByteBuffer.wrap(packet.getData(), 0, packet.getLength());
                    byteBuffer.order(ByteOrder.BIG_ENDIAN);
                    long sequenceNumber = byteBuffer.getLong();

                    if (sequenceNumber == -1L) {
                        // End-of-file signal
                        transferActive = false;
                        break;
```

```
                }

                int dataLength = packet.getLength() - Long.BYTES - Integer.BYTES;
                if (dataLength <= 0) {
                    printAndLog("Invalid packet received. Skipping.", true);
                    continue;
                }

                byte[] data = new byte[dataLength];
                byteBuffer.get(data);
                int receivedChecksum = byteBuffer.getInt();

                // Perform CRC validation
                CRC32 crc = new CRC32();
                crc.update(data, 0, dataLength);
                long calculatedChecksum = crc.getValue() & 0xFFFFFFFFL;
                if (calculatedChecksum != (receivedChecksum & 0xFFFFFFFFL)) {
                    printAndLog("CRC mismatch for sequence " + sequenceNumber + ". Ignoring packet.", false);
                    continue;
                }

                // Add packet to buffer for reassembly
                packetBuffer.put(sequenceNumber, data);

                // Process packets in order
                while (packetBuffer.containsKey(expectedSequence)) {
                    byte[] nextData = packetBuffer.remove(expectedSequence);
                    fos.write(nextData);
                    totalBytesTransferred += (nextData.length + UDP_IP_APPLICATION_OVERHEAD); // data size + TCP ↩
                        ↪ Header + IP Header
                    expectedSequence++;

                    // Update the transfer display with sequence and CRC
                    transferDisplay.accept(totalBytesTransferred, bytesPerFile, receivedChecksum, sequenceNumber);
                }
            }

            fos.flush();
        } catch (IOException e) {
            printAndLog("Error in packet handler: " + e.getMessage(), true);
        } finally {
            duration = System.currentTimeMillis() - startTime;
            try {
                fos.close();
            } catch (IOException e) {
                printAndLog("Error closing file: " + e.getMessage(), true);
            }
            socket.close();
        }
    }

    public long getTotalBytesTransferred() {
        return totalBytesTransferred;
    }

    public long getDuration() {
        return duration;
    }
}

public static void main(String[] args) throws IOException {
    LogToFile.logToFile(LOGGER, "FTPClient.log"); // Log to file
    printAndLog("Logging to FTPClient.log", true);

    System.out.println("Starting FTP Client...");

    String javaVersion = System.getProperty("java.version");
    printAndLog("Java version: " + javaVersion, true);

    if (args.length == 2) {
        printAndLog("Connecting to " + args[0] + " on port " + args[1], true);
    } else if (args.length == 1) {
        printAndLog("Attempting to connect to " + args[0] + " on default port (" + PORT + ")", true);
    } else {
        printAndLog("Usage: java FTPClient <hostname> <port number>", true);
        System.exit(1);
    }

    serverIP = args[0];
    serverPort = args.length == 2 ? Integer.parseInt(args[1]) : PORT;

    try (
        Socket ftpSocket = new Socket(serverIP, serverPort);
        PrintWriter out = new PrintWriter(ftpSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(ftpSocket.getInputStream()));
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))
```

```
        ) {
            printAndLog("Connection successful to " + serverIP + ":" + serverPort, true);

            menu(out, in, stdIn);

        } catch (UnknownHostException e) {
            // If the host is not found, log the error and exit
            printAndLog("Unknown host: " + serverIP, false);
            System.exit(1);
        } catch (IOException e) {
            // If an I/O error occurs, log the error and exit
            printAndLog("Couldn't get I/O for the connection to " + serverIP + ":" + serverPort, true);
            LOGGER.severe(e.getMessage());
            e.printStackTrace();
            System.exit(1);
        }
    }

    /**
     * Menu system for user interaction
     * @param out The PrintWriter for sending commands to the server
     * @param in The BufferedReader for reading responses from the server
     * @param stdIn The BufferedReader for reading user input
     * @throws IOException If an I/O error occurs while reading user input
     */
    private static void menu(PrintWriter out, BufferedReader in, BufferedReader stdIn) throws IOException {
        while (true) {
            String transferModeMenu = "Toggle Transfer Mode ("+ (!udpMode ? "[" : "") + "TCP" + (!udpMode ? "]" : "") + "/" + (udpMode ? "[" : "") + "UDP" + (udpMode ? "]" : "") + ")";
            String testingModeMenu = "Toggle Testing Mode (" + (testingMode ? "[" : "") + "ON" + (testingMode ? "]" : "" ) + "/" + (!testingMode ? "[" : "") + "OFF" + (!testingMode ? "]" : "") + ")";
            System.out.printf("\nFTP Client Menu:\n1) GET\n2) PUT\n3) CD\n4) LS\n5) %s\n6) %s\n7) QUIT\n", transferModeMenu, testingModeMenu);
            System.out.print("Enter choice: ");
            String choice = stdIn.readLine();
            switch (choice) {
                case "1":
                    System.out.print("Enter file name to download: ");
                    String getFileName = stdIn.readLine();
                    receiveFile(getFileName, out, in);
                    break;
                case "2":
                    System.out.print("Enter file name to upload: ");
                    String putFileName = stdIn.readLine();
                    sendFile(putFileName, out, in);
                    break;
                case "3":
                    System.out.print("Enter directory to change to: ");
                    String dirName = stdIn.readLine();
                    out.println("CD " + dirName);
                    String cdResponse = in.readLine();
                    printAndLog(cdResponse, true);
                    if (!cdResponse.startsWith("Error")) {
                        // Run LS after CD to list directory contents if directory change is successful
                        out.println("LS");
                        String responseLine;
                        while (!(responseLine = in.readLine()).equals("EOF")) {
                            printAndLog(responseLine, true);
                        }
                    }
                    break;
                case "4":
                    out.println("LS");
                    String responseLine;
                    while (!(responseLine = in.readLine()).equals("EOF")) {
                        printAndLog(responseLine, true);
                    }
                    break;
                case "5":
                    udpMode = !udpMode;
                    out.println("MODE");
                    printAndLog("Transfer mode switched to " + (udpMode ? "UDP" : "TCP"), true);
                    break;
                case "6":
                    testingMode = !testingMode;
                    printAndLog("Testing mode " + (testingMode ? "enabled" : "disabled"), true);
                    break;
                case "7":
                    out.println("QUIT");
                    printAndLog(in.readLine(), false);
                    return;
                default:
                    System.out.println("Invalid option.");
                    break;
            }
        }
```

```
    }

    /**
     * Handles the file receiving for the GET command.
     * @param fileName The name of the file to download.
     * @param out The PrintWriter for sending commands to the server.
     * @param in The BufferedReader for reading responses from the server.
     * @throws IOException If an I/O error occurs while receiving the file.
     */
    private static void receiveFile(String fileName, PrintWriter out, BufferedReader in) throws IOException {
        long totalDuration = 0;  // Accumulate transfer times
        long totalBytesTransferred = 0;  // Accumulate bytes transferred
        int numRuns = testingMode ? NUM_TESTS : 1;
        long fileSize = 0;
        long bytesPerFile = 0;
        boolean transferSuccess = false;  // Flag to indicate if transfer was successful

        for (int i = 0; i < numRuns; i++) {
            if (i > 0) {
                // If we're in testing mode, display a separator between runs
                System.out.println("\n-------------------------------------------------");
            }
            if (testingMode) {
                System.out.println("Starting run " + (i + 1) + " of " + numRuns + " for " + fileName + " transfer.");
            }

            out.println("GET " + fileName);  // Send GET command to the server
            out.flush();
            String serverResponse = in.readLine();

            if (serverResponse != null && serverResponse.startsWith("READY")) {
                transferSuccess = true;  // Transfer is going to happen
                String[] readyResponse = serverResponse.split(" ");
                int port = Integer.parseInt(readyResponse[1]); // Server's transfer port
                fileSize = Long.parseLong(readyResponse[2]);  // File size from server

                if (!udpMode) {
                    // TCP Mode
                    try (Socket transferSocket = new Socket(serverIP, port);
                         BufferedInputStream bis = new BufferedInputStream(transferSocket.getInputStream());
                         FileOutputStream fos = new FileOutputStream(fileName)) {
                        byte[] buffer = new byte[TCP_BUFFER_SIZE];
                        int bytesRead;
                        long currentBytes = 0;
                        long startTime= System.currentTimeMillis();  // Start time for each file
                        bytesPerFile = fileSize + TCP_IP_OVERHEAD * (int)Math.ceil((double) fileSize/TCP_BUFFER_SIZE);  ↩
                            ↪ // Total bytes to transfer

                        while ((bytesRead = bis.read(buffer)) != -1) {
                            fos.write(buffer, 0, bytesRead);
                            currentBytes += bytesRead;
                            totalBytesTransferred += (bytesRead + TCP_IP_OVERHEAD); // bytesRead + TCP Header + IP ↩
                                ↪ Header

                            // Display progress for the current run
                            transferDisplay(totalBytesTransferred - i * bytesPerFile, bytesPerFile, 0, 0);

                        }

                        long endTime = System.currentTimeMillis();
                        totalDuration += (endTime - startTime);  // Accumulate total time for all runs
                        fos.flush();
                    }
                } else {
                    // UDP Mode
                    try (DatagramSocket datagramSocket = new DatagramSocket();
                        FileOutputStream fileOutputStream = new FileOutputStream(fileName)) {
                        datagramSocket.setSoTimeout(TIMEOUT); // Set timeout for receiving packets
                        datagramSocket.setReceiveBufferSize(UDP_RECV_BUFFER); // Set buffer size
                        //printAndLog("Actual UDP receive buffer size on server: " + datagramSocket.getReceiveBufferSize↩
                            ↪ () + " bytes", true);
                        InetAddress serverAddress = InetAddress.getByName(serverIP);

                        out.println("CLIENT_READY " + datagramSocket.getLocalPort());
                        out.flush();

                        PacketHandler handler = new PacketHandler(
                                datagramSocket,
                                fileOutputStream,
                                fileSize,
                                FTPClient::transferDisplay, // Pass transferDisplay method reference
                                TIMEOUT,
                                i
                        );
                        handler.start();
```

```
                    try {
                        handler.join();
                    } catch (InterruptedException e) {
                        printAndLog("File transfer was interrupted: " + e.getMessage(), true);
                        Thread.currentThread().interrupt(); // Restore interrupt status
                        return;
                    }

                    // Collect metrics from the PacketHandler
                    totalBytesTransferred += handler.getTotalBytesTransferred();
                    totalDuration += handler.getDuration();
                }
            }
        } else {
            printAndLog("Error: " + serverResponse, true);
            transferSuccess = false;  // No transfer occurred
            break;  // Exit the loop since there's an error
        }
    }

    if (transferSuccess) {
        // Log transfer details
        logTransferDetails(numRuns, fileSize, totalDuration, totalBytesTransferred, fileName, "GET");
    }
}

/**
 * Handles the file sending for the PUT command.
 * @param fileName The name of the file to upload.
 * @param out The PrintWriter for sending commands to the server.
 * @param in The BufferedReader for reading responses from the server.
 * @throws IOException If an I/O error occurs while sending the file.
 */
private static void sendFile(String fileName, PrintWriter out, BufferedReader in) throws IOException {
    long totalDuration = 0;  // Accumulate transfer times
    long totalBytesTransferred = 0;  // Accumulate bytes transferred
    int numRuns = testingMode ? NUM_TESTS : 1;
    long fileSize = 0;
    long bytesPerFile = 0;
    boolean transferSuccess = false;  // Flag to indicate if transfer was successful

    for (int i = 0; i < numRuns; i++) {
        if (i > 0) {
            // If we're in testing mode, display a separator between runs
            System.out.println("\n--------------------------------------------------");
        }
        if (testingMode) {
            System.out.println("Starting run " + (i + 1) + " of " + numRuns + " for " + fileName + " transfer.");
        }
        long startTime = System.currentTimeMillis();  // Start time for each run
        File file = new File(fileName);
        fileSize = file.length();  // Get the actual file size

        out.println("PUT " + fileName + " " + fileSize);  // Send PUT command with file size
        out.flush();

        String serverResponse = in.readLine();
        if (serverResponse != null && serverResponse.startsWith("READY")) {
            String[] readyResponse = serverResponse.split(" ");
            int port = Integer.parseInt(readyResponse[1]); // Server's transfer port
            transferSuccess = true;  // Transfer is going to happen

            if (!udpMode) {
                // TCP mode
                try (Socket transferSocket = new Socket(serverIP, port);
                    BufferedOutputStream bos = new BufferedOutputStream(transferSocket.getOutputStream());
                    FileInputStream fis = new FileInputStream(fileName)) {
                    byte[] buffer = new byte[TCP_BUFFER_SIZE];
                    int bytesRead;
                    long currentBytes = 0;
                    bytesPerFile = fileSize + TCP_IP_OVERHEAD * (int)Math.ceil((double) fileSize/TCP_BUFFER_SIZE);  ↩
                        ↪ // Total bytes to transfer

                    while ((bytesRead = fis.read(buffer)) != -1) {
                        bos.write(buffer, 0, bytesRead);
                        currentBytes += bytesRead;

                        totalBytesTransferred += (bytesRead + 40); // bytesRead + TCP Header + IP Header

                        // Display progress for the current run
                        transferDisplay(totalBytesTransferred - i * bytesPerFile, bytesPerFile, 0, 0);
                    }

                    bos.flush();

                }
```

```
            } else {
                // UDP mode
                try (DatagramSocket datagramSocket = new DatagramSocket();
                    FileInputStream fis = new FileInputStream(fileName)) {
                    datagramSocket.setSoTimeout(TIMEOUT);
                    InetAddress serverAddress = InetAddress.getByName(serverIP);

                    byte[] buffer = new byte[UDP_BUFFER_SIZE];
                    long sequenceNumber = 0;
                    long currentBytes = 0;
                    bytesPerFile = fileSize + UDP_IP_APPLICATION_OVERHEAD * (int)Math.ceil((double) fileSize/↩
                        ↪ UDP_BUFFER_SIZE);

                    while (true) {
                        int bytesRead = fis.read(buffer);
                        if (bytesRead == -1) break;

                        CRC32 crc = new CRC32();
                        crc.update(buffer, 0, bytesRead);
                        int checksum = (int) crc.getValue();

                        ByteBuffer packetBuffer = ByteBuffer.allocate(Long.BYTES + bytesRead + Integer.BYTES);
                        packetBuffer.order(ByteOrder.BIG_ENDIAN);
                        packetBuffer.putLong(sequenceNumber);
                        packetBuffer.put(buffer, 0, bytesRead);
                        packetBuffer.putInt(checksum);

                        DatagramPacket packet = new DatagramPacket(packetBuffer.array(), packetBuffer.position(), ↩
                            ↪ serverAddress, port);
                        datagramSocket.send(packet);

                        sequenceNumber++;
                        currentBytes += bytesRead;
                        totalBytesTransferred += (bytesRead + UDP_IP_APPLICATION_OVERHEAD);

                        // Display progress for the current run
                        transferDisplay(totalBytesTransferred - i * bytesPerFile, bytesPerFile, checksum, ↩
                            ↪ sequenceNumber);

                        // Introduce a small delay
                        try {
                            Thread.sleep(UDP_DELAY);
                        }
                            catch(InterruptedException e) {
                            printAndLog("Thread interrupted: " + e.getMessage(), true);
                        }
                    }

                    // End-of-file signal
                    ByteBuffer endBuffer = ByteBuffer.allocate(Long.BYTES);
                    endBuffer.order(ByteOrder.BIG_ENDIAN);
                    endBuffer.putLong(-1L);
                    DatagramPacket endPacket = new DatagramPacket(endBuffer.array(), endBuffer.capacity(), ↩
                        ↪ serverAddress, port);
                    datagramSocket.send(endPacket);
                }
            }
        } else {
            printAndLog("Server error: " + serverResponse, true);
            transferSuccess = false;  // No transfer occurred
            break;  // Exit the loop since there's an error
        }

        long endTime = System.currentTimeMillis();
        totalDuration += (endTime - startTime);
    }

    if (transferSuccess) {
        // Log details
        logTransferDetails(numRuns, fileSize, totalDuration, totalBytesTransferred, fileName, "PUT");
    }
}

/**
 * Logs and prints the details of a file transfer, handling both single run and test mode.
 * @param numRuns The number of runs (1 for a single run, NUM_TESTS for test mode).
 * @param totalDuration The total duration of all runs in milliseconds.
 * @param totalBytesTransferred The total number of bytes transferred.
 * @param fileName The name of the file being transferred.
 * @param operation The operation type ("GET" or "PUT").
 */
private static void logTransferDetails(int numRuns, long filesize, long totalDuration, long totalBytesTransferred, ↩
    ↪ String fileName, String operation) {
    printAndLog("\n" + operation + " transfer of " + fileName + " complete.", true);
    if (numRuns > 1) {
        // Test mode: display average statistics
```

```
            long averageDuration = totalDuration / numRuns;
            double averageThroughput = totalBytesTransferred / (averageDuration * numRuns/ 1000.0);  // Throughput in b/↩
                ↪ s
            System.out.println("");  // New line for clarity
            printAndLog("Average transfer time for " + numRuns + " runs: " + averageDuration + " ms", true);
            printAndLog("File size: " + filesize + " bytes", true);
            printAndLog("Total bytes transferred: " + totalBytesTransferred / numRuns + " bytes", true);
            printAndLog("Average throughput: " + (long) averageThroughput + " b/s", true);
        } else {
            // Single run: display detailed stats
            long duration = totalDuration;  // Total duration is for the single run
            double throughput = totalBytesTransferred / (duration / 1000.0);  // Throughput in b/s
            System.out.println("");  // New line for clarity
            printAndLog(operation + " of " + fileName + " completed in " + duration + " ms", true);
            printAndLog("File size: " + filesize + " bytes", true);
            printAndLog("Total bytes transferred: " + totalBytesTransferred + " bytes", true);
            printAndLog("Throughput: " + (long) throughput + " b/s", true);
        }
    }

    /**
     * Displays a progress bar for the file transfer.
     * @param currentBytes The number of bytes transferred so far.
     * @param totalBytes The total number of bytes to transfer.
     * @param crc The CRC32 checksum of the transferred data.
     * @param segment The current segment/sequence number being processed.
     */
    private static void transferDisplay(long currentBytes, long totalBytes, int crc, long segment) {
        // Clear the line if we're about to hit 100% to ensure no residual characters are present
        if (currentBytes >= totalBytes) {
            System.out.print("\r" + " ".repeat(150) + "");
        }
        int transferPercentage = (int) ((currentBytes / (double) totalBytes) * 100);

        // Calculate how much of the bar to fill. Minimum is 0, and maximum is 50.
        int arrowPosition = Math.max(0, Math.min(50, transferPercentage / 2));

        // Build the progress bar display
        String progressBar = "|"
                + "=".repeat(arrowPosition)  // Filled portion
                + (arrowPosition < 50 ? ">" : "") // Arrow
                + " ".repeat(50 - arrowPosition)  // Empty portion
                + "| ";

        // Print transfer percentage and byte count
        System.out.print("\r" + progressBar
                + transferPercentage + "% ("
                + currentBytes + "/" + totalBytes + " bytes)");

        // Only print CRC if it is non-zero
        if (crc != 0) {
            System.out.print(" CRC32: 0x" + Integer.toHexString(crc));
        }

        if (segment != 0) {
            System.out.print(" Segment: " + segment);
        }

        // Flush the output to ensure it updates in real-time
        System.out.flush();
    }

    /**
     * Utility method to print messages to the console and log them.
     * @param message The message to log.
     * @param newline Whether to include a newline character at the end of the console output.
     */
    private static void printAndLog(String message, boolean newline) {
        if (newline) {
            System.out.println(message);
        } else {
            System.out.print(message);
        }
        LOGGER.info(message);
    }

    /**
     * Utility class to set up logging to a file.
     */
    private static class LogToFile {
        public static void logToFile(Logger logger, String logFile) {
            try {
                FileHandler fileHandler = new FileHandler(logFile, true);  // Append mode
                logger.addHandler(fileHandler);

                // Use the custom formatter for the log format
```

```
                CustomLogFormatter formatter = new CustomLogFormatter();
                fileHandler.setFormatter(formatter);

                // Disable console output for the logger (remove default handlers)
                logger.setUseParentHandlers(false);
            } catch (SecurityException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        /**
         * Custom log formatter to format log messages with a timestamp and log level.
         */
        private static class CustomLogFormatter extends Formatter {
            // Define the date format
            private static final SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy@HH:mm:ss");

                @Override
                public String format(LogRecord record) {
                    // Get the current date and time
                    String timeStamp = dateFormat.format(new Date(record.getMillis()));

                    // Get the log level (severity)
                    String logLevel = record.getLevel().getName();

                    // Format the log message
                    return String.format("%s:%s:\t%s%n",
                            timeStamp,              // Short date and time
                            logLevel,               // Log level (severity)
                            record.getMessage()     // Actual log message
                    );
                }
            }
    }
}
```

generate_files.sh

```
#!/bin/bash

# Create files with specified sizes using fallocate
echo "Generating files with random data..."

# 1MB file
dd if=/dev/urandom of=file1MB.dat bs=1M count=1

# 25MB file
dd if=/dev/urandom of=file25MB.dat bs=1M count=25

# 50MB file
dd if=/dev/urandom of=file50MB.dat bs=1M count=50

# 100MB file
dd if=/dev/urandom of=file100MB.dat bs=1M count=100
```

generate_files.ps1

```
# Function to generate a random binary file of a specified size (in bytes)
function Generate-RandomFile {
    param (
        [string]$fileName,
        [int]$fileSizeBytes
    )

    $random = New-Object Random
    $buffer = New-Object byte[] $fileSizeBytes
    $random.NextBytes($buffer)

    # Write the binary data to the file
    [System.IO.File]::WriteAllBytes($fileName, $buffer)
}

# Generate test files of various sizes
Generate-RandomFile -fileName "file1MB.dat" -fileSizeBytes 1048576   # 1MB
Generate-RandomFile -fileName "file25MB.dat" -fileSizeBytes 26214400 # 25MB
Generate-RandomFile -fileName "file50MB.dat" -fileSizeBytes 52428800 # 50MB
Generate-RandomFile -fileName "file100MB.dat" -fileSizeBytes 104857600 # 100MB
```