CSCI 345 Computer Networks Final Project

By: Emily Prieto, Delren Divinagracia, Leysha Charles, Arib Chowdhury

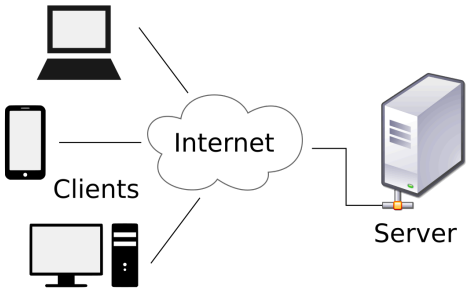
I. Introduction

For this project we developed a functional Instant Messaging (IM) service that has the capabilities of servers connecting to multiple clients and holding communication by using TCP/IP protocol. The core objectives were to implement secure client authentication and facilitate communication between multiple clients through a centralized server. Clients can hold peer-to-peer connections between clients to facilitate messages between users, and conduct file sharing using socket programming. The system is built upon Socket Programming, which are defined as networks that communicate to each other over a bridge of APIs known as sockets. Sockets work in the way that they establish a connection from the user to the server for quick interaction and quickly disconnect after information is exchanged between the two. These sockets will allow us to create communication from both the client and server side to then find other users that are also connected to the server and then directly communicate to other clients. Our objective is to share files and share messages using a peer to peer connection while authenticating the status

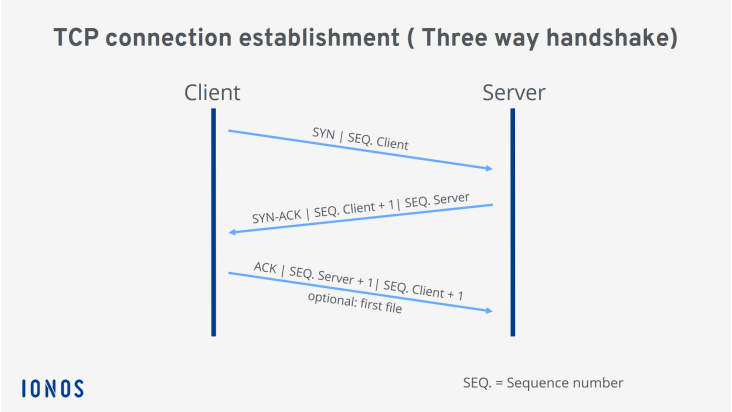
II. System Architecture and Protocol Design

A. Network Topology and Transport Layer

This system operates on a Centralized Relay Architecture, with the server functioning like a middleman for all communication. Sockets are necessary for client-server interactions and are responsible for a flow of data between the communication models of the transport layer and application layer. While sockets is not technically a layer in the communication layer, it allows applications to interact with networking and transport layers. The standard for TCP/IP network configurations requires Socket APIs to establish the network endpoints and relies on TCP/IP protocols for maintaining communication.



For the Transport protocol, we used TCP (Transmission Control Protocol) for all communication. TCP is a connection-oriented protocol intended to transfer packets across the internet and to ensure the successful transmission of data and messages between networks. TCP is essential because it provides reliability, guaranteeing that every byte of data, especially important for file integrity, is delivered. It also ensures ordered delivery, wherein chat messages and file segments are reassembled in the correct order. Finally, TCP connections are established through a three-way handshake before any data transfer occurs, ensuring a reliable and stable connection.



B. Application Layer Protocol

We designed a simple text-based protocol inspired by HTTP and suitable for reading by Java's BufferedReader class. Each packet has three components:

- a *method*, such as `LOGIN, WHOIS, GOODBYE` or `SUCCESS`, which describes what the packet is intended to do;
- a collection of *headers*, which, like HTTP headers contain additional information for the packet structured as key-value pairs:
    - `contentLength` – the length, in characters, of the message body
    - `username` – the username for a login request
    - `address` – the IP address of a peer; and
- a (possibly empty) *content*, which represents the message body or, in the case of a file transfer, the file contents.

Here is an example of a `FILE` transfer packet, as it is sent over the TCP socket. Newline characters are shown explicitly with `\n`.

```
FILE\n
contentLength: 12\n
filename: hello.txt\n
\n
Hello world!
```

The first line indicates the *method*. All following lines up to the blank line are headers, including the `contentLength` header, which indicates the length of the message body. The last line contains the 12 character *content*, which in this case is the contents of the *hello.txt* file. The content may be empty, such as in this `LOGIN` packet, and thus the `contentLength` header may be omitted.

```
LOGIN\n
username: admin\n
password: hunter2\n
\n
```

Note that the blank line is still mandatory, as otherwise the reader would not know where to stop. If the credentials are incorrect, the server may respond with this `FAILURE` packet.

```
FAILURE\n
error: WRONG_CREDENTIALS\n
\n
```

The command set contains fundamental IM functions such as `LOGIN` to initiate login, `SUCCESS` to indicate success, `STATUS` for presence updates, `WHOIS` for looking up a peer's status and IP, `MESSAGE` for text messaging, and `FILE` to initiate a file transfer request.

C. Client Concurrency and State Management

The client, implemented in `src/messenger/Client.java`, makes use of concurrency to handle incoming data without blocking the user interface. During a chat session, the client must be able to accept and display incoming messages and file transfer requests while simultaneously allowing the user to use the UI and send messages back. To accomplish this, a separate `IncomingPacketHandler` is dispatched whenever a connection is established, which handles and displays incoming packets as they appear. Any incoming packets are handled by the thread, leaving the main thread free to control the user interface and send packets to the peer. To avert concurrent programming errors such as race conditions we make use of Java's `synchronized` keyword for certain operations, such as for closing the socket on receiving a `GOODBYE` from the peer.

III. Server Module and Client Module

The system is composed of two major components: the server module and the client module. The server module contains all logic for authentication, command parsing, and state management. It handles the organization of connected users, routing of server-mediated messages, and the introduction of clients to each other before a peer-to-peer connection can be made. In contrast, the client module is responsible for establishing the initial TCP connection to the server or to a peer, sending authentication commands, handling user-issued operations such as messaging or file transfer, responding to server notifications, and maintaining local status information. The client interface is console-based and designed to present messages, presence updates, and file-transfer prompts clearly while supporting stable interaction with the server's protocol.

IV. Implementation Details

The implementation uses Java as the primary development language, leveraging its built-in networking libraries such as ServerSocket and Socket for TCP communication. Multithreading is achieved using Java's Thread class, and thread safety is enforced using the `synchronized` keyword to lock some instance variables for the duration of an operation. File operations rely on FileReader and FileWriter to read and write byte sequences as strings during file transfers. The

final implementation includes secure authentication through a structured command protocol, real-time TCP messaging, status tracking for all connected users, and support for direct client-to-client file sharing using dedicated data sockets. Significant technical challenges arose during development, including managing concurrency without race conditions, ensuring synchronization of authentication and presence updates, safely transferring large files without corruption, and designing a message protocol that is simple but robust enough to handle multiple types of communication.

User Interface

Our client presents a console interface inspired by IRC. Actions are performed by inputting various commands beginning with slashes. Any input line that does not begin with a slash is interpreted as a message to be sent to the currently connected peer. There are a number of commands, including:

- `/server <ip> <port>` – Use a different server than the default
- `/register <username> <password>` – Register with the server as a new user
- `/getstatus <username>` – Get the current status of a peer (`BUSY`, `READY`, `CHATTING`)
- `/connect <username>` – Begin a chat session with a peer
- `/sendfile <file path> <filename>` – Register with the server as a new user

Here is an example session where a user `alice` registers a new account and begins a chat session with `bob`, sending a file along the way.

```
/register alice password123
/getstatus bob
User bob: READY
/connect bob
hi bob!
>hi alice!
/getstatus bob
User bob: CHATTING
/sendfile file.txt cool_file.txt
File sent successfully!
/quit
Bye.
```

Client-to-client operations

A client-to-client operation refers to a direct communication between two client systems. It does not require a central server to handle the exchange of data. The exchange is done directly and doesn't need to route everything through a server. As long as the clients know each other's information, they don't need to rely on the central server, reducing server load and network congestion at the expense of requiring a reachable port on the other client, which can be complicated by NAT. It can also be substantially faster for users who are on the same network. In our application, messaging and file sharing takes place over peer-to-peer connections, and the server is only relied upon to provide authentication and information about their peer's IP addresses and statuses.

File Transfer over TCP

Our `/sendfile` command sends the entire file in a single application layer packet, relying on TCP to handle chunking, congestion control and streaming. This results in a simpler, more robust implementation at the cost of loading the entire file into memory at once, which can be expensive for very large files. To keep things simple, the sender can specify an alternate filename to send the file under and all file transfer requests are accepted automatically. This way of doing things introduces a vulnerability to directory traversal attacks that may allow remote code execution. Since this is a toy project, we did not worry overmuch about the security implications, but if we were to continue development this would be one of the first things to address.

V. Testing and Achievements

We used JUnit to write unit tests and integration tests for much of the new functionality, enabling us to catch errors early. To test the interaction between the server and multiple client processes on one machine, we assigned each process a unique port. In production, both clients and servers would all use port `6789`. Testing was conducted using multiple simultaneous client instances to evaluate authentication accuracy, message reliability, presence tracking, and file-transfer integrity. Authentication tests verified that the server consistently accepted valid credentials and rejected invalid ones. Messaging tests confirmed that TCP ensured ordered and complete delivery across several concurrent sessions. Status tracking tests demonstrated that changes to status were properly distributed in response to future `WHOIS` requests. File-transfer tests verified that files were transmitted and reconstructed accurately using the TCP data stream. Overall, the system achieved secure authentication, reliable messaging, consistent status synchronization, and dependable file-transfer performance while maintaining performance under realistic usage conditions.

VI. Project Management: Timeline and Workflow

The project followed a structured planning and workflow model combining Waterfall planning and Agile sprints. Early work focused on requirements, architecture, and task division; later weeks were dedicated to implementing core subsystems, debugging, and testing. Weekly team meetings allowed progress tracking and redistribution of tasks as needed. Each member contributed to essential components of the system. Emily developed the client logic, packet format and console interface while contributing to the presentation. Delren produced much of the written documentation and assisted with the presentation. Leysha implemented the concurrency model and file-transfer subsystem while helping organize the final demonstration. Arib developed server-side authentication, session management, and command parsing and contributed to refining the technical report. Together, these contributions ensured that all major features were implemented effectively.

VII. Individual Contributions

- Emily Prieto – Coded client-side logic and console UI and worked on the project presentation. Designed packet format.
- Delren Divinagracia – Coded protocol integration logic and worked on the technical report and project presentation
- Leysha Charles – Coded Server concurrency and File Transfer logic. Created the project presentation.
- Arib Chowdhury – Coded Server Side Authentication and Worked on the Technical Report

VIII. Conclusion

In conclusion, the completed Instant Messaging system integrates secure authentication, multithreaded client management, real-time messaging, presence tracking, and TCP-based file transfer in a cohesive Java implementation. Through careful protocol design, reliable transport-layer integration, and controlled use of concurrency, the system demonstrates practical application of core networking concepts. Testing confirmed stable and consistent communication performance, and the project as a whole illustrates how socket programming and TCP/IP principles can be combined to build a functional multi-client communication platform.