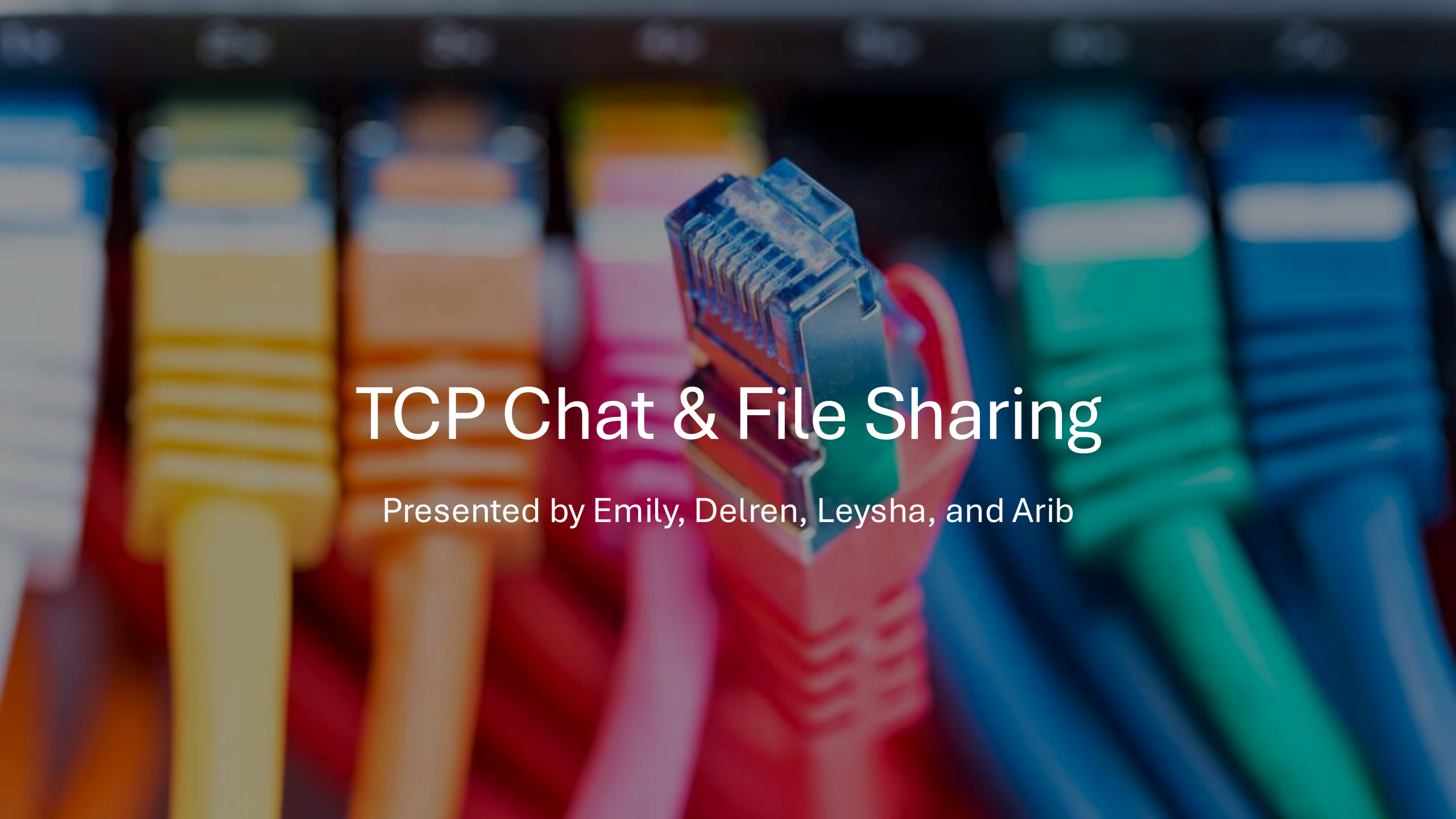

TCP Chat & File Sharing

Presented by Emily, Delren, Leysha, and Arib

Introduction

- **Goal**: To build a chat program where many clients can connect to one server
- What the project does: login, messaging, status updates, and file transfer
- A server provides authentication, and clients are the users. The server makes sure everyone can talk to each other reliably. We used TCP/IP sockets, which guarantee that data arrives safely and in order.

Config & Constants

```java
public class Config {  6 usages
    public static final String SERVER_IP = "127.0.0.1";
    public static final int SERVER_PORT = 6789;  4 usages
}
```

- Config.java holds all the network settings (IP and Port). This centralizes control, making the system easier to manage
- We keep connection settings in one place. There is a default hardcoded server address, set here to the loopback address for testing.
- We chose port **6789** for both client-server and client-client communication.

Peer & Status

- Peer.java records (username, IP, status)
- Status.java enum (READY/BUSY)
- Clients can get information about a particular user they want to contact, including their IP address and status.

Packets

- Custom plain-text packet format inspired by HTTP. Each has a:
  - Method – The type of action to perform
  - Headers – Key-value store of method-specific data (username, status, contentLength, etc.)
  - Content – (Possibly empty) packet body. Used for message text and file transfers.

```
public record Packet(Method method, Map<String, String> headers, String content, InetAddress address) {

public static void sendPacket(Socket socket, Packet packet) throws IOException {  10 usages  Emily Prieto
    var output = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
    output.write( str: packet.method() + "\n");
    for (var header: packet.headers().entrySet())
        output.write("%s: %s\n".formatted(header.getKey(), header.getValue()));
    output.write( str: "\n");
    output.write(packet.content());
    output.flush();
}
```

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

FILE transfer packet, showing headers and body (file contents).

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

LOGIN packet with empty body. Note the trailing newline.

Server Architecture

- Server.java
- The server listens for connections using a server socket. It stores usernames and passwords in a map, and keeps track of active sessions in another map. each session has a username, ip, port, and status. this is how the server knows who is online.

Server Request Handling

- Server handles packets with four kinds of methods:
  - REGISTER – register as a new user
  - LOGIN – perform authentication and announce your IP
  - WHOIS – get information about another user, including their status and peer-to-peer IP
  - STATUS – set your status with the server
- On Success – sends back SUCCESS packet with requested information in the headers
- On Failure – sends back FAILURE packet with 'error' header indicating problem
- Custom syntax for specifying packet handlers:

```
@HandlesMethod(Method.REGISTER)  1 usage  Emily Prieto
public Packet onRequestREGISTER(Packet request) {
```

Client Architecture

- Client.java
- Can send REGISTER, LOGIN, WHOIS, STATUS packets to configured server
- Can initialize messaging session with peer, sending HELLO, MESSAGE, FILE, GOODBYE packets
- Simple terminal command line interface
- IRC inspired syntax:
  - /login <username> <password>
  - /connect <username>
  - /sendfile <path> <filename>
  - Lines without a slash are sent as messages.

```
switch (cmd) {
    case "/server": {...} break;
    case "/register": {...} break;
    case "/login": {...} break;
    case "/getstatus": {...} break;
    case "/connect": {...} break;
    case "/disconnect": {...} break;
    case "/sendfile": {...} break;
    case "/quit": {...} return;
    default: sendMessage(line);
}
```

```
/register alice password1
/getstatus bob
User bob: READY
/connect bob
hi bob!
>hi alice!
/sendfile date.txt cool_file.txt
File sent successfully!
/quit
Bye.
```

Messaging & File Transfer

- Messages are sent with a MESSAGE packet directly to the peer's TCP socket
- The user must first initialize the connection with **/connect** <username>
- File transfer is done with **/sendfile** <path> <filename>

```java
public Error sendMessage(String message) throws IOException {
    if (peerSocket == null)
        return Error.MALFORMED_REQUEST;
    var packet = new Packet(Method.MESSAGE, message);
    Packet.sendPacket(peerSocket, packet);
    var response = Packet.readPacket(peerSocket);
    return response.getError();
}
```

```java
public void sendFile(String path, String filename) {  1 usage  Emily Prieto +1
    String contents;
    try (var reader = new BufferedReader(new FileReader(path))) {
        contents = reader.readAllAsString();
    } catch (FileNotFoundException ex) {
        IO.println("File not found!");
        return;
    } catch (IOException ex) {
        IO.println("An error has occurred!");
        return;
    }
    var request = new Packet(Method.FILE, Map.of(
            k1: "filename", filename
    ), contents);
```

Client Testing

- We used JUnit to write unit tests for our helper functions, Server class and Client class
- To test client-client communication on the same machine, we had to assign the server and each client a different port number for use during testing

```
@Test  Emily Prieto
@Order(3)
void whois() {
    assertAll(
            () -> assertNull(client.whois( username: "nonexistantuser")),
            () -> assertEquals(new Peer(
                    username: "alice",
                    InetAddress.getLoopbackAddress(),
                    CLIENT_PORT,
                    Status.BUSY
            ), client.whois( username: "alice"))
    );
}

@Test  Emily Prieto
@Order(4)
void status() {
    assertAll(
            () -> assertEquals(Status.BUSY, client.whois( username: "alice").status()),
            () -> assertEquals(Error.OK, client.setStatus(Status.READY)),
            () -> assertEquals(Status.READY, client.whois( username: "alice").status()),
            () -> assertEquals(Error.OK, client.setStatus(Status.BUSY)),
            () -> assertEquals(Status.BUSY, client.whois( username: "alice").status())
    );
}
```

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

Conclusion

- Recap: working Java messaging system with authentication, messaging, status, file transfer
- We built a working chat system with authentication, messaging, status updates, and file transfer. We proved it works with automated tests. In the future, we'd like to add a graphical interface, encryption, and saving data permanently
- Future: encryption or persistent storage, concurrent connections, more robust error messages