Server ->

**✅ What it does:**

* Starts a **TCP server** on port 12346.
* Listens for incoming connections (serverSocket.accept()).
* Each client is handled by a separate thread (pool.execute(new ClientHandler(...))).

**💡 Concepts:**

* **Thread Pool (ExecutorService)**: Reuses a fixed number of threads (10 here) instead of creating a new one for each connection (which is inefficient).
* **Scalability**: Prevents system overload by limiting simultaneous threads.
* **ClientHandler**: (Assumed) handles communication with a single client (login, messaging, etc).

SignUpController ->

String resp = sendSignup(username, password);

* Uses a **short-lived socket** to send SIGNUP|username|password to the server.
* Server responds with either SUCCESS or ERROR|reason.

**🔑 On success:**

1. Creates a User object and saves it locally (Database.saveUser).
2. Sets the current user in a session (Session.setUser).
3. Opens a **persistent socket** using ClientConnection.getInstance().connect(...) for ongoing messaging.
4. Loads the dashboard scene.

ClientConnection is a **singleton class** that manages the client’s connection to the server **over a socket**. It acts as the **central hub** for:

* Sending messages
* Receiving messages
* Dispatching events (like friend requests or chat messages)
* Managing chat history and group communication
* Uploading profile photos

Private static Instance;

private static means:

* private: can't be accessed directly from outside.
* static: it belongs to the class itself, not an object (shared).
* Since it’s static, all the controller can share this

💡 **Why use singleton?**  
Only one client socket should be active at any time for a user — it prevents duplicate connections or inconsistent state.

**Why don’t we just talk to the server directly from the GUI (e.g., Controller or ClientHandler)?**

Because:

1. **Centralized Control:** You want only **one connection** (Socket) to the server per user. Creating new sockets everywhere (e.g., from each controller) would be inefficient and error-prone.
2. **Threading:** This class handles **background reading** (e.g., new messages) in a separate thread (listen() method), preventing the UI from freezin

 **Message Dispatching:** It acts as a **router**: when a line like PRIVATE|a|b|timestamp|msg comes from the server, it routes that to listeners (your controllers).

 **Queued Messages:** If no controller is listening yet (like before loading the chat UI), messages are **queued temporarily** and dispatched later.

 **Socket Handling:** Handles all low-level details like:

* Opening/closing the socket
* Reusing socket if already connected
* Sending/receiving formatted protocol messages

|  |
| --- |
| 1. **Socket connection** |

|  |
| --- |
| connect() |

|  |  |  |
| --- | --- | --- |
| Connects to the server socket and sets up PrintWriter/thread | | |
| **Receive & dispatch** | listen() + dispatch() | Listens on socket and routes to registered listeners | |

|  |  |  |
| --- | --- | --- |
| **Message listeners** | registerListener() | Lets controllers listen for messages |

|  |  |  |
| --- | --- | --- |
| **Friend listeners** | registerFriendListener() | Triggers friend-related events like onFriendRequestReceived() |

|  |
| --- |
| **Singleton management** |

|  |
| --- |
| getInstance() |

|  |
| --- |
| Ensures only 1 instance is used throughout the app |

When connect() is called:

* It creates a new Socket, PrintWriter, and starts the listen() thread.
* Sends an ONLINE|username line to notify server.

The listen() thread ->

while ((line = br.readLine()) != null) {

dispatchRequests(line); // for friend events

dispatch(line); // for chat/group/system msgs

}

Dispatch handles the activities.

**✅ Extra: Why it’s a Singleton**

The use of private static ClientConnection INSTANCE ensures that:

* Only **one connection** is maintained per client
* All controllers and views share that single communication line

private final ExecutorService dispatchPool = Executors.newSingleThreadExecutor();

This executor handles **incoming events**, such as messages received from the server. It dispatches them to all registered listeners:

* **this creates:**
* A **thread pool with only 1 thread** (a **single-threaded executor**).
* Used to **queue and run listener tasks one at a time**, preserving **order of messages**.

💡 **Why?**  
If 3 listeners are registered and a message arrives, they should be notified **in order**, without race conditions.

**Why single-threaded?**

Because:

* UI updates in JavaFX **must run on the JavaFX Application Thread**.
* Dispatching on multiple threads could lead to **race conditions** or **concurrent UI modifications**, which are illegal and crash-prone.
* A single-threaded executor **ensures ordering**: if messages come in as A → B → C, they are processed in that exact order.

⚠️ If you used a thread pool with multiple threads, your UI might try to display C before A finishes processing, which leads to inconsistencies.

private final List<FriendListener> friendListeners = new CopyOnWriteArrayList<>();

public interface FriendListener {

void onFriendRequestReceived(String fromUser);

}

The List holds all interested listeners (like controllers) that want to get notified when a friend request arrives. (listeners are my controllers like friendRequestController, viewProfileController

for (MessageListener l : listeners) {

dispatchPool.submit(() -> l.onMessageReceived(groupName, body));

}

->

* For each listener registered for chat messages (MessageListener interface), it **submits a task** to the dispatchPool.
* The dispatchPool (single thread) ensures **one task runs at a time**, keeping order intact.

|  |
| --- |
| * Dispatch pool ensures ordered, serialized event delivery to UI |

This makes sure your UI isn’t flooded with out-of-order or conflicting message updates.

**🔧 General Role of ClientHandler**

This class handles one client connection on the server side. Each time a client connects, the server spins up a new thread running ClientHandler, which:

* Parses messages (commands) from the client,
* Updates server state (like online users, groups, etc.),
* Sends responses or broadcasts to appropriate clients.

**ONLINE\_WRITERS**

java

CopyEdit

private static final Map<String, PrintWriter> ONLINE\_WRITERS = new ConcurrentHashMap<>();

* Maps usernames → PrintWriter used to send messages back to that client.
* Used to broadcast or send messages to specific online users.

**GROUP\_MEMBERS**

java

CopyEdit

private static final Map<String, Set<String>> GROUP\_MEMBERS = new ConcurrentHashMap<>();

* Keeps track of all group names and their members.
* Every time a group is created or a message is sent to a group, this map is used to fetch members.

**CHAT\_FOLDER and GROUP\_FOLDER**

Paths for saving chat logs and group info persistently on disk.

public void run() {

try {

prepareIOStreams(); // setup reader/writer

listenForCommands(); // begin infinite loop to receive and dispatch commands

} ...

}

**📥 prepareIOStreams()**

Creates UTF-8 input/output streams from the raw socket. This ensures that multibyte characters (like Unicode emojis or Bengali script) are transmitted correctly.

<https://chatgpt.com/share/688a4bc8-ab3c-8010-9e01-81aaf2f640c1>

Great — let’s walk through the **message flow** step by step for your LAN messenger, focusing on how private and group messages travel from one user’s input to another’s chat UI.

**🔄 Flow: Sending and Receiving a Message (PRIVATE)**

**👤 1. User Types and Sends a Message**

This happens in the **ChatBoxController**.

java

CopyEdit

clientConnection.sendPrivateMessage(toUser, messageText);

This calls ClientConnection.send(...), which writes a line like this to the socket:

makefile

CopyEdit

PRIVATE|alice|bob|2025-07-30T14:55:01|Hey Bob!

**📡 2. ClientConnection → Socket → Server**

The line goes through the socket to the **Server**, which spawns a ClientHandler thread for each client.

java

CopyEdit

String line = in.readLine(); // inside listenForCommands()

This line is read from the input stream of the socket connected to Alice.

**🧠 3. ClientHandler Parses Message**

java

CopyEdit

case "PRIVATE": handlePrivate(pieces); break;

The server-side ClientHandler calls this:

java

CopyEdit

private void handlePrivate(String[] p) {

String from = p[1];

String to = p[2];

String timestamp = ...;

String body = ...;

Then it builds this:

java

CopyEdit

String forward = "PRIVATE|alice|bob|2025-07-30T14:55:01|Hey Bob!";

**📬 4. Send to Recipient OR Store Offline**

java

CopyEdit

PrintWriter targetWriter = ONLINE\_WRITERS.get(to); // Find Bob's socket writer

if (targetWriter != null) {

targetWriter.println(forward); // Sends directly to Bob

} else {

saveOfflineMessage(...);

}

So yes: ClientHandler uses PrintWriter to **send a message back to the client**.

📘 **What is PrintWriter?**  
It wraps the socket’s output stream and lets you send text (like protocol lines) with println().

**⬇️ 5. Bob's Client Receives It**

In **Bob’s ClientConnection**, a thread is running:

java

CopyEdit

String line;

while ((line = br.readLine()) != null) {

dispatch(line);

}

So it gets:

makefile

CopyEdit

PRIVATE|alice|bob|2025-07-30T14:55:01|Hey Bob!

**🧭 6. dispatch(...) Routes It**

java

CopyEdit

if ("PRIVATE".equals(header)) {

String from = p[1];

String body = p[3] + "|" + p[4]; // timestamp|message

for (MessageListener l : listeners) {

dispatchPool.submit(() -> l.onMessageReceived(from, body));

}

}

This sends it to all MessageListeners.

**💬 7. ChatBoxController Gets the Message**

Your ChatBoxController implements MessageListener. When registered:

java

CopyEdit

ClientConnection.getInstance().registerListener(this);

The method:

java

CopyEdit

public void onMessageReceived(String from, String body) {

// display the message in the chat UI

}

Now the message appears in the chat window!

**🔁 Summary: Message Flow (Private)**

plaintext

CopyEdit

User Input (ChatBoxController)

↓

ClientConnection.send() writes to socket

↓

Server's ClientHandler reads it

↓

ClientHandler finds receiver & forwards via PrintWriter

↓

ClientConnection on receiver runs dispatch(line)

↓

dispatch() calls all onMessageReceived() listeners

↓

ChatBoxController receives and shows the message

**⚠️ Clarifying Your Question**

❓ "Does ClientHandler send msg back to ChatBoxController?"

Not directly. **ClientHandler sends it through the socket to ClientConnection,** which then dispatches it to the UI (e.g. ChatBoxController).

* In client handler, server is always running the listenForCommands() to listen to commands from client connection.
* In clientConnection, listen() loop is only used by clients to listen to servers.
* Controllers send direct messages by clientConnection.getInstance().sendRequest().

**📤 uploadProfileImage(File file) – How Profile Image Upload Works**

**🔄 Flow Summary:**

1. Client sends a **header**: filename, username, and length.
2. Then sends **raw image bytes** over the same socket.
3. Server reads the header first, then reads the bytes.
4. Server writes the image to disk.