Network Architecture Documentation (GC20)

Network Package Structure

```
client.network
├── NetworkManager.java           # Main entry point for network initialization
├── common
│   ├── Client.java               # Abstract client interface
│   └── ClientFactory.java        # Factory for creating clients (Socket or RMI)
├── socket
│   └── SocketClient.java         # Socket-based implementation for Client
└── RMI
    └── RMIClient.java            # RMI-based implementation for Client

server.network
├── common
│   ├── Server.java               # Abstract interface for server
│   ├── HeartbeatService.java     # Detects disconnections through a ping-pong mechanism
│   ├── QueueHandler.java         # Manages a queue to prevent concurrency issues
│   └── ClientHandler.java        # Abstract interface for client handlers
├── socket
│   ├── SocketServer.java         # Socket-based server for handling incoming connections
│   ├── SocketAuthService.java    # Manages authentication for new socket connections
│   └── SocketClientHandler.java  # Handles client logic over socket
├── RMI
│   ├── RMIServer.java                  # Initializes and manages the RMI registry
│   ├── RMIAuthService.java             # Manages authentication for new RMI connections
│   ├── RMIGameControllerService.java   # Manages game logic for a specific RMI client
│   ├── RMIMatchControllerService.java  # Manages lobbies and multiple matches
│   ├── RMIServerHandler.java           # Handles RMI object exporting and registry setup
│   └── RMIClientHandler.java           # Handles logic for clients connected via RMI
├── NetworkFactory.java           # Factory for initializing server-side networking
└── NetworkService.java           # Entry point for controllers to send messages to clients

common
├── interfaces
│   ├── GameControllerInterface.java     # Contains all game-related methods
│   ├── MatchControllerInterface.java    # Contains all lobby/multiple matches related methods
│   ├── AuthInterface.java               # Contains all authentication-related methods
│   └── ViewInterface.java               # Represent the view interface exposed for remote
callbacks
└── message_protocol
│   ├── to_client
│   │   └── *.java                # Contains all server-to-client possible messages
│   └── to_server
│   │   ├── game
│   │   │   └── *.java            # Contains all client-to-server game-related messages
│   │   └── lobby
│   │   │   └── *.java            # Contains all client-to-server lobby-related messages
└── Message.java                  # Interface for all message implementations
```

1. `NetworkManager.java` (Client)

Purpose:

- Initializes the client network connection.
- Uses `ClientFactory` to instantiate either an `RMIClient` or `SocketClient` depending on the user's choice.

Description:

When the player selects a preferred connection type, this manager configures and returns the appropriate client implementation. This implementation is then stored and used when each action is performed.

The Factory Design Pattern has been used to implement Client creation.

---

2. `Client.java` / `Server.java` / `ClientHandler.java`

These are the **core abstractions** for communication:

- `Client` (client-side abstraction for network logic)
- `Server` (server-side abstraction for communication)
- `ClientHandler` (server-side abstraction for managing an individual client)

These interfaces define a unified API for both RMI and Socket backends. `ClientHandler` is crucial for encapsulating per-client logic and message dispatching on the server side.

This implements the Strategy Design Pattern to allow the different network types to provide the same methods.

---

3. `SocketClient.java` & `SocketServer.java`

Client Side:

- Opens a TCP socket to the server.
- Starts a dedicated listener thread.
- Sends and receives serialized `Message` objects using object streams.

Server Side:

- Listens for incoming socket connections using `ServerSocket.accept()`.
- Delegates authentication to `SocketAuthService`, which verifies if the username is already in use.
- Upon successful login, a `SocketClientHandler` is created and associated with the client.
- Communication is handled via `ObjectInputStream` and `ObjectOutputStream`.

---

4. `RMIClient.java` & `RMIServer.java`

Client Side:

- Connects to the RMI registry.
- Retrieves remote service stubs: `GameService`, `MatchService`, and `AuthService`.
- Invokes remote methods via RMI as if they were local.

Server Side:

- Exports implementations of the above services.
- Binds them to the RMI registry under fixed names.
- Accepts client callbacks (e.g., local models exposed as remote interfaces).

---

5. De-synchronization of RMI Calls

Because RMI is a synchronous protocol, concurrent invocations could block the server or result in race conditions. To mitigate this, a `QueueHandler` object is used:

- All actions received via RMI or Socket are enqueued.
- A single thread processes this queue, executing actions one at a time.

The adapter classes `RMIGameControllerService` and `RMIMatchControllerService` enqueue messages on behalf of RMI clients.

---

6. Heartbeat Mechanism

To detect client disconnections, the server includes a `HeartbeatService`:

- Sends a `Ping` every 15 seconds.
- Waits for a corresponding `Pong`.
- If no response is received within 45 seconds, the client is considered disconnected and is removed from the session.

This mechanism applies to both RMI and Socket clients.

---

7. Server-to-Client Communication

Every connected client is associated with a `ClientHandler`, which provides a `sendToClient(Message m)` method implemented appropriately for the connection type:

- `SocketClientHandler` sends messages via the socket output stream.
- `RMIClientHandler` invokes a method on the remote client object (exposed local model through the `updateView(Message m)` method).

The `NetworkService` is responsible for locating the correct handler by username and forwarding the message. This makes server-to-client messaging fully modular and controller-agnostic.