POLITECNICO
MILANO 1863

Progetto di Ingegneria del Software - Documento di Rete

Anno Accademico 2024/2025

Daniele Toniolo

Vittorio Sironi

Lorenzo Trenti

Matteo Zappa

Contents

1 Introduction **2**

1.1 Communication Technology . . . 2

1.2 Netwotk Structure . . . 2

1.2.1 Connection . . . 2

1.2.2 Network Transceiver . . . 6

1.2.3 Events . . . 7

1.2.4 MatchController . . . 7

2 Communication Flow **9**

2.1 Initial Connection and Authentication . . . 9

2.1.1 Connection Establishment (Client) . . . 9

2.1.2 HeartBeat . . . 10

2.2 Synchronous Communication Mechanism: Requester-Responder . . . 10

2.2.1 Description of the model . . . 10

2.2.2 Interaction between Client and Server . . . 11

2.2.3 Functionality and Interactions . . . 11

2.3 Asynchronous Notification from the server . . . 12

2.4 Lobby Managment . . . 12

2.4.1 Creating a Lobby (Client -> Server) . . . 12

2.4.2 Join a Lobby (Client -> Server) . . . 13

2.4.3 Leave a Lobby (Client -> Server) . . . 13

2.4.4 Reporting Player Ready (Client -> Server) . . . 14

2.5 Game actions . . . 15

2.5.1 General schema . . . 15

3 Disconnection **16**

Chapter 1

Introduction

The system uses a client-server architecture where the clients communicate with a central server for participating in a game session.

1.1 Communication Technology

The communication can be done in two different ways, at the user's choice:

- **RMI (Java remote Method Invocation):** The client interacts with the server by obtaining references to remote objects exported by the server to send requests and receive data. This approach is useful for the development of distributed applications by abstracting the complexity of low-level communication between server-client.
- **TCP (Transmission Control Protocol):** The client establishes a direct socket connection with the server. Messages are exchanged as serialized Java objects through input/output streams associated with the socket. This approach provides greater control over the communication process, though it requires more handling about data transmission.

1.2 Netwotk Structure

1.2.1 Connection

The interface connection implements the class TCPConnection and RMIConnection. So we can save a `List<Connection>` and manage the two type of Connection in the same way.
Its main functions:

- **Reader Thread**: The class launches a dedicated thread that waits for new data from the network. When a data is received, it deserializes it into an *Event* and puts it into an input queue shared with the other players (*NetworkTransceiver*)

- **Writer Thread**: This thread waits for data to send. When there is data, it serializes it and sends it over the network. If the queue is empty, the thread is blocked waiting.
- **Competition Management**: It use thread-safe structures (*BlockingQueue*) for thread communication and to avoid busy-waiting. Ensures that reading and writing are independent and don't block each other.

So, in the server we can see that every client that is connected has its own Connection that manages the communication with that client. Meanwhile, the client sees only its own instance of Connection. The most important thing is that the class Connection supplies the queue of inputs/outputs on which the NetworkTransceiver works for changing the data of low level into some high-level data and vice versa.

TCPConnection

The `TCPConnection` class manages a bidirectional TCP connection between client and server, handling object serialization, heartbeat, thread-safe message queuing, and connection closure.

- **Constructors**
  - `TCPConnection(String address, int port)`: Creates a new TCP connection to the specified host and port (client side). Initializes input/output streams and starts heartbeat and reader threads.
  - `TCPConnection(Socket socket)`: Wraps an existing TCP socket (server side). Initializes streams and starts heartbeat and reader threads.
- **void send(Event message)**
  Serializes and sends an `Event` object through the output stream. Throws `DisconnectedConnectio`
  if the connection is closed.
- **Event receive()**
  Returns the next received event from the message queue. Waits if the queue is empty. Throws `DisconnectedConnection` if the connection is closed.
- **void disconnect()**
  Marks the connection as disconnected and notifies all waiting threads.
- **private void read()**
  Starts a thread that continuously reads objects from the input stream. `Event` objects (except `HeartBeat`) are queued for reception. On error, closes the connection.
- **private void hearBeat()**
  Starts a timer that periodically (every 2.5 seconds) sends a `HeartBeat` event to keep the connection alive. Cancels the timer if the connection is interrupted.

So we can describe the protocol details as:

- **Connection Initiation:**
  - Client: Connects to the server using a TCP socket (`new Socket(address, port)`).
  - Server: Accepts incoming sockets and wraps them in a `TCPConnection` instance.
- **Message Exchange:**
  - Messages are Java-serialized `Event` objects.
  - Sending: `send(Event)` writes the object to the output stream.
  - Receiving: A background thread reads objects from the input stream and enqueues them (except for `HeartBeat`).
- **Heartbeat:**
  - A timer sends a `HeartBeat` event every 2.5 seconds to keep the connection alive.
  - If a heartbeat or message cannot be sent/received, the connection is closed.
- **Thread Safety:**
  - All send/receive operations are synchronized on a lock object.
  - The `pendingMessages` queue is used to buffer incoming events for processing.
- **Disconnection:**
  - On error or explicit call, the connection is marked as disconnected, the socket is closed, and waiting threads are notified.

RMIConnection

The `RMIConnection` class manages a bidirectional communication channel between client and server using Java RMI. It handles remote queue lookup, message serialization, heartbeat, thread-safe message queuing, and connection closure.

- **Constructors**
  - `RMIConnection(String address, int port)`: Initializes the connection on the client side. Looks up the remote server and binds to the appropriate sender and receiver queues. Starts heartbeat and reader threads.

  - `RMIConnection(String address, int port, String boundName)`: Initializes the connection on the server side, binding to the correct remote queues. Starts heartbeat and reader threads.

- **void send(Event message)**
  Sends an `Event` to the remote queue. If the connection is closed or a timeout occurs, it throws `DisconnectedConnection`.

- **Event receive()**
  Returns the next received event from the message queue. Waits if the queue is empty. Throws `DisconnectedConnection` if the connection is closed.

- **void disconnect()**
  Marks the connection as disconnected and notifies all waiting threads.

- **private void heartbeat()**
  Starts a timer that periodically (every 2.5 seconds) sends a `HeartBeat` event to keep the connection alive. Cancels the timer if the connection is interrupted.

- **private void read()**
  Starts a thread that continuously reads events from the remote queue. If an event is read, then the `receive()` wakes up. Non-heartbeat events are queued for reception. On error or timeout, closes the connection.

- **private boolean sendWithTimeout(Event message)**
  Tries to send a message to the remote queue within a timeout. Returns true if successful, false otherwise.

- **private Optional<Event> readWithTimeout()**
  Tries to read a message from the remote queue within a timeout. If the message is not read in this time, the connection is considered closed. Otherwise, the event is returned in `reads()`.

So we can describe the protocol details as:

- **Connection Initiation:**
  - Client: Looks up the remote server and binds to sender/receiver queues via RMI registry.
  - Server: Binds to the correct remote queues using the provided bound name.
- **Message Exchange:**
  - Messages are Java-serialized `Event` objects sent via remote queues.

  - Sending: `send(Event)` adds the event to the remote sender queue.
  - Receiving: A background thread polls the remote receiver queue and enqueues events (except `HeartBeat`).

- **Heartbeat:**
  - A timer sends a `HeartBeat` event every 2.5 seconds to keep the connection alive.
  - If a heartbeat or message cannot be sent/received, the connection is closed.

- **Thread Safety:**
  - All send/receive operations are synchronized on a lock object.
  - The `pendingMessages` queue is used to buffer incoming events for processing.

- **Disconnection:**
  - On error or explicit call, the connection is marked as disconnected and waiting threads are notified.

1.2.2 Network Transceiver

This component manages the communication of the event between the logic of the network (*Connection*) and the application level (*EventListener/EventTransceiver*). Its main functions:

- **Event Reception**: It has a thread that every cicle it calls *receive()* on the Connection in order to retrieve the message that arrives from the Client or the Server. The events received are inserted in a BlockingQueue of input. Then the *EventListener*, that are registered, is notified and will handle the event. The handle is done in a different Thread, which is initialized in the constructor.

- **Send Events**: There is also another thread that monitors a BlockingQueue for the output. When a component tries to send an event, it insert the event in the queue and then, there is another thread initialize in the constructor (different from the receieve Threads) which takes the event, it sends it with the method *send()* of the class Connection.

- **BlockingQueue**: There are BlockinQueue because they help the thread to stay blocked when there aren't enough data without consuming CPU in a useless way. Also, the management thread-safe guarantees that more components can send/receive events simultaneously and without conflict.

1.2.3 Events

- **Message Format**: Regardless of the protocol chosen, messages exchanged between client and server are serializable Java objects that implement the Event interface. Each type of interaction (request or notification) is represented by a specific class that extends this interface.
- **EventListener**: It is an interface that receive object Event from the Network-Transceiver and then it process them by making specific action.
- **EventTransceiver**: It is an interface that takes the object Event generated from the application and send them to the NetworkTransceiver

1.2.4 MatchController

The `MatchController` class is the main controller of the game server. It manages lobbies, users, game controllers, and the network communication between server and clients. It is implemented as a singleton and coordinates all network transceivers and event listeners.

NetworkTransceiver Instances

- **Server NetworkTransceiver (`serverNetworkTransceiver`):**
  - There is a single instance of `NetworkTransceiver` used by the server to communicate with all users who are not currently in a lobby.
  - It handles global events such as lobby creation, joining, nickname setting, and user disconnection.
- **Lobby NetworkTransceivers (`networkTransceivers`):**
  - For each lobby created, a dedicated `NetworkTransceiver` instance is created and associated with that lobby. When a user joins a lobby, their connection is moved from the server transceiver to the corresponding lobby transceiver.
  - This transceiver manages all communication between the server and the users within that specific lobby, including all game-related events.
  - The mapping is maintained as `Map<LobbyInfo, NetworkTransceiver>`.

Listener Registration

- **Lobby Listeners:**
  - Registered on the `serverNetworkTransceiver` via the `registerAllLobbyListeners()` method.

- Listeners handle events such as lobby creation (`CreateLobby`), joining (`JoinLobby`), nickname setting (`SetNickname`), and disconnection (`ConnectionLost`).
- Example:

```
CreateLobby.responder(serverNetworkTransceiver, this::createLobby);
JoinLobby.responder(serverNetworkTransceiver, this::joinLobby);
SetNickname.responder(serverNetworkTransceiver, this::setNickname);
ConnectionLost.registerHandler(serverNetworkTransceiver, this::disconnectUse
```

- **Game Listeners:**
  - Registered on each lobby's `NetworkTransceiver` via the `registerAllGameListeners(Netwo` method.
  - Listeners handle all game-related events, such as tile picking, deck usage, player actions, and in-game disconnections.
  - Example:

```
PickTileFromBoard.responder(networkTransceiver, this::pickTileFromBoard);
PlaceTileToBoard.responder(networkTransceiver, this::placeTileToBoard);
PlayerReady.responder(networkTransceiver, this::playerReady);
EndGame.registerHandler(networkTransceiver, this::endGame);
```

Chapter 2

Communication Flow

2.1 Initial Connection and Authentication

2.1.1 Connection Establishment (Client)

- **RMI**:

  1. The client connects to the **RMI Registry** on the server's specified IP address and port. It performs a lookup for the **RemoteServer** object published under the name *"SERVER"*.
  2. On the Server side, this object, corresponds to the **ConnectionAcceptor** class that implements the **RemoteServer** interface. It invokes the *getBoundName()* method on the **RemoteServer** object to obtain a unique session/connection identifier. This name will be used to identify the communication queues dedicated to this client.
  3. Performs a lookup for two **RemoteQueue** objects:
     - *"SENDER_" + boundName*: The queue on which the client will send messages to the server.
     - *"RECEIVER_" + boundName*: The queue from which the client will receive messages from the server.
  4. An instance of **RMIConnection** is created using these remote queues. The server, on its side, will create the UUID representing both the connection and the user to the newly created RMI Connection.

- **TCP**:

  1. The client creates a **Socket** to connect to the specific IP address and TCP port of the server on which the **ServerSocket** is listening.

2. Once the connection is established, the client (and the server on its side) obtains *ObjectOutputStream* and *ObjectInputStream* from the **Socket** to send and receive serialized **Event** objects.
3. An instance of **TCPConnection** (or a similar implementation of the **Connection** interface) is created using these streams. The server, on its side, will create the UUID representing both the connection and the user to the newly created TCP Connection.

2.1.2 HeartBeat

1. Once the **Connection** (either RMIConnection or TCPConnection) is established, a heartbeat mechanism is initiated.
2. Periodically (every TIMEOUT/2 milliseconds, 2.5 seconds), an **HeartBeat** event is sent through the Connection to verify that it is still active.
3. The Connection also has dedicated threads for continuous reading of **Event** and Heartbeat messages, with timeouts to detect disconnections.

2.2 Synchronous Communication Mechanism: Requester-Responder

The system uses a *Requester-Responder* model to handle synchronous requests that need an immediate response during communication between client and server. This mechanism is designed to ensure a controlled flow of events, improving the consistency and traceability of interactions.

2.2.1 Description of the model

1. **Requester**
   - Client-side component that sends a specific request to the server and waits for a response.
   - It ensures that client execution remains blocked until feedback is received from the server. The responses obtained from the server may be success(Tac) or error($Pota$), accompanied by any descriptive data or messages.
2. **Responder**
   - Server-side counterpart of the *Requester*. Receives requests from clients, processes them, and sends an appropriate response.

- It integrates application logic rules to verify and execute the requested operations, thereby returning an outcome that conforms to the conditions set by the system.

2.2.2 Interaction between Client and Server

1. The client sends a request using the *Requester*.
2. The server receives the event on the NetworkTransceiver on which the user is connected
3. The server processes the request with the associate handle
4. The server sends back to the client a `StatusEvent` by usign the *Responder*
5. The client receives the response and can continue its execution based on the result received.

2.2.3 Functionality and Interactions

The *Requester-Responder* model is used in numerous scenarios, including:

- **Nickname Setting**
  - The client sends a request to register a nickname on the server.
  - The server checks the availability of the name and responds with a confirmation message(*Tac*) if the nickname is acceptable or an error message(*Pota*) if the name is already in use.
  - The response received determines the client-side UI update.
- **Lobby Creation and Management**
  - The client sends requests related to lobby management (e.g., creation, access, or exit).
  - The server processes these requests by making sure that the parameters meet the defined rules, providing a clear response to the client on the outcome.
- **Game Actions**
  - During the game, the client uses the synchronous model to invoke gameplay-related actions, such as placing tiles, managing resources, or using special abilities.
  - The server checks the validity of the action in the context of the current game. If valid, it returns a success confirmation(*Tac*) and updates the game status, while If invalid, it sends an error(*Pota*) with a message describing the problem.

2.3 Asynchronous Notification from the server

In addition to direct *Tac/Pota* responses, the GameController sends broadcast events to all clients in the game to inform them of changes in game state. These are crucial for keeping the view of the game synchronized across all clients.
They are important because the client can do other actions while it is waiting for a response about a previous event.
There are some asynchronous notifications that are sent to the client when he is waiting for a synchronous response (in order to notify of some changing in the model) or when another client has done an action which modify the model.

2.4 Lobby Managment

Lobby-related interactions take place via the serverNetworkTransceiver until a user enters a specific lobby. At that point, his connection (regardless of protocol) is transferred to a dedicated NetworkTransceiver for that lobby. The NetworkTransceiver operates on the Connection abstraction, making the lobby and game management logic independent of the underlying transport protocol.

2.4.1 Creating a Lobby (Client -> Server)

- **Client side**: The client sents the event **CreateLobby**. That event rappresent the fact that him or another player has created a lobby.

- **Server side**: The server uses the method **MatchController.createLobby** for initializing the lobby in the server. This is the process:

  1. Creates a *LobbyInfo* object and attempts to create the *Board* for the specified level
  2. Creates a new *'NetworkTransceiver'* for the lobby, registers listeners for game events and the *'LeaveLobby'* event on this new transceiver.
  3. Transfers the creator user's connection from the *'serverNetworkTransceiver'* to the lobby transceiver'(networkTransceiver.connect(...)', 'serverNetworkTransceiver.disconnect
  4. Creates *'PlayerData'* for the founding user. Creates and associates a *'Game-Controller'* with the lobby and then adds the player to the lobby in the 'Game-Controller'.
  5. Then it responds to the founder client about the creation of new lobby:
     - Sends a *Tac* event for reporting that the lobby is correctly initialized.

    - Sends a *PlayerAdded* event for notifying that the founder client is being added to the lobby.

6. To the other client that will want to join or not the lobby, the server will send them the event *LobbyCreated*, which indicates the creation of a lobby (broadcast event).

- In case of errors (for example board creation failed), the server will send broadcast the event *Pota*, which indicates that has appened a failure in the progamm.

2.4.2 Join a Lobby (Client -> Server)

- **Client side**: The client sents the event **JoinLobby**. That event rappresent the fact that he has joined a lobby.
- **Server side**: The server uses the method **MatchController.JoinLobby** for joining the lobby in the server. The process that is used:
    1. Checks if the lobby exists and is not full. If so, assigns an available color to the player and creates PlayerData for the player.
    2. Adds the player to the lobby in the *GameController* and transfers the user's connection from the *serverNetworkTransceiver* to the lobby transceiver.
    3. Response to Client (unicast) with the event *Tac* if all was right and the server notify to all clients (broadcast) the event *LobbyJoined.* For the players present in the lobby, it sends also the event *PlayerAddes*, which means that a player is being added to the lobby.
- In case of error (lobby full/not found), it sends the event *Pota* only to the client that is trying to connect to a lobby (unicast).

2.4.3 Leave a Lobby (Client -> Server)

- **Client side**: The client sents the event **LeaveLobby**. That event rappresent the fact that he has leaved a lobby.
- **Server side**: The server uses the method **MatchController.LeaveLobby** for joining the lobby in the server. The process that is used:
    - Identifies the user and the lobby and response to Client (unicast, on lobby transceiver) with the event *Tac* for saying that it found the lobby.
    - If the user is the founder:

        1. Removes the lobby, the associated GameController, and the lobby's NetworkTransceiver and notify the Clients in the Lobby (broadcast) with the event *LobbyRemoved*.
        2. All users in the decommissioned lobby are transferred back to the serverNetworkTransceiver with the notification for saying that the Client is being transferred. This event (unicast) is called *Lobbies*, beacuse it shows the fact that the Client is now where it can see all the available lobbies.
        3. The it sends the notification to all Clients (broadcast) called *LobbyRemoved* that symbolized the elimination of the previous lobby.
    - If the user is not the founder:
        1. Removes the player from the lobby in the GameController and MatchController data structures and then it transfers the user to the serverNetworkTransceiver.
        2. Notifies the User (unicast) with the event *Lobbies* for showing the fact that he left the lobby and now the Client can choose another lobby.
        3. It sends the notification to all Clients (broadcast) with the event *LobbyLeft* for saying that a player has left the lobby.

- In case of error (lobby not found), it sends the event *Pota* only to the client that is trying to connect to a lobby (unicast).

2.4.4 Reporting Player Ready (Client -> Server)

- **Client side**: The client sents the event **PlayerReady**. That event rappresent the fact that he is ready for playing.
- **Server side**: The server uses the method **MatchController.PlayerReady** for saying to all the other players that that particular Client is ready to play. The process that is used:
    1. Updates the "ready" status of the player and response to Client (unicast) with the event *Tac*, for saying that all went right.
    2. If all required players are ready:
        - The server sends a notification to all Clients (broadcast) called *LobbyRemoved* for saying that the lobby is no longer visible/accessible.
        - It notify all Clients in the Lobby (broadcast) with the event *StartingGame*. Then a timer is started; when it expires, *GameController.startGame()* is invoked to start the game.
- In case of error, it sends the event *Pota*.

2.5 Game actions

Once the game has started, all communication occurs through the lobby's Network-Transceiver, which in turn uses the appropriate Connection instance (RMI or TCP) for that client. The MatchController receives the events and delegates them to the correct GameController.

2.5.1 General schema

1. Client sends a specific event to the server with the *Requester*. Then the MatchController receives the event, identifies the user, lobby, and associated GameController.
2. The MatchController invokes the appropriate method on the GameController, which executes the logic of the action:
   - If the action is valid and allowed:
     - Performs the action
     - Sends an event *Tac* to the requesting client for saying that it has been done the action
     - Sends one or more game state update events to all clients in the game via an asynchronous event.
   - If the action is invalid or not allowed:
     - Send an event *Pota* to the requesting client to inform him that the action can't be done.

Chapter 3

Disconnection

Detection and Propagation of Disconnection

Client disconnections caused by network errors are detected at the network layer (e.g., in `TCPConnection` or `RMIConnection`) through heartbeat timeouts or exceptions such as `DisconnectedConnection`. When a disconnection is detected, the `NetworkTransceiver` creates and injects a `ConnectionLost` event into its internal event queue. This is done by the receiver thread associated with the client connection, which, upon catching a disconnection, enqueues a `ConnectionLost` event for that user.

Forcing Disconnection Handling in NetworkTransceiver

The `NetworkTransceiver` is responsible for managing all active client connections and their associated event threads. When a network error occurs, the receiver thread for that client:

- Catches the `DisconnectedConnection` exception.
- Logs the disconnection.
- Creates a `ConnectionLost` event with the user's UUID.
- Adds this event to the transceiver's received event queue, ensuring it will be processed by the main event dispatch thread.

This mechanism guarantees that the server is always notified of a client disconnection, regardless of where the error originated.

Handling Disconnection in MatchController

The `MatchController` registers a listener for `ConnectionLost` events on both the global and lobby-specific `NetworkTransceiver` instances. When a `ConnectionLost` event is received, the `disconnectUser` method is invoked. This method performs the following actions:

1. Identifies the user and their associated lobby (if any).
2. The user is remove from the lobby and from the server. If the game is already started the game finish and all the other players return to the lobby. Otherwise, they simply return to the lobby.
3. The system ensures that no resources or threads remain associated with the disconnected client.
4. The user is removed from all relevant server-side maps and data structures.
5. All other clients in the same lobby or game session receive a broadcast event informing them of the disconnection, so their UI can update accordingly.