Communication protocol

*Group **AM46** – Beghetto Pietro, de Donato Simone, Dimaglie Gregorio*

GENERAL NETWORK STRUCTURE:

The network mainly revolves around a main LobbyServer and many MatchServers to handle parallel games. Both servers are Echo Servers.
Both servers implement the Runnable interface and their run method continuously accepts client connections.
When a client connects to the ServerSocket, the server will use the newly created socket to create a SocketConnection, an object that also implements Runnable, and run it on a different thread.
The SocketConnection run method continuously listens on the socket for any UserAction the client might send. Whenever a message is received, the SocketConnection starts a new thread to handle it, thus immediately going back to listening on the socket.
The message is initially parsed by the SocketConnection and then forwarded to the server that created this SocketConnection for further parsing.
A LobbyServer will create and run a MatchServer on a different thread and port whenever a client that logged in can't join an existing game.
A MatchServer contains a Controller to which UserActions pertaining to the game itself are forwarded to modify the game state.
To send messages to the client, the controller will forward the message via the MatchServer bound to it, while both servers use the SocketConnection objects to actually send the messages through the network to specific clients.
The sockets, both client- and server-side, have timeouts set so that any networking error (that isn't immediately caught through an exception) will cause this timeout to expire and the connection to be closed from both sides.

COMPLETE MESSAGE LIST:

The idea is to have “UserAction” be the action taken by the client, “Error”, “Info” and “Update” the responses sent by the Server.

A generic abstract serializable class **Message** is the base upon which messages are built.
Each Message contains a String containing the nickname of the sender (or “Server” if it’s a message sent by the server).

From the base class derive four different types of messages:

1. **UserAction**: Sent by a client to the server. It represents the action taken by a user.
2. **Info**: Sent by the server to a client. It contains information about non game-specific information (i.e. login, logout..) and may also serve as confirmation of a successful action.
3. **Update**: Sent by the server to a client. It contains the game state that will be used by the client to update the UI.
4. **Error**: Sent by the server to a client. It notifies the client that something was wrong in their action. Some examples of reasons to throw an Error message are: an action taken in the wrong turn, receiving an unexpected action or receiving incorrect method parameters.

A **UserAction** (which is an abstract class) also contains:

- a UserActionType
- each specific user action will contain the requested information as an attribute (int, Color, etc.) along with getters to retrieve such info.

An **Info** (which is an abstract class) also contains:

- A String containing a description

Infos notify clients of something that has happened that does not pertain to the game itself (login, logout, ping, …), and can double as a confirmation of success regarding a given user action.
Each deriving class contains specific information.

An **Update** (which is a concrete class) also contains:

- the UserAction taken that caused this Update to be sent
- the player who took the action that caused this Update to be sent
- the UserAction that must be taken next in order to progress the game
- the player that must take the next action in order to progress the game
- the state of the game (updated after the last action received) represented by a ObservableByClient object.

Updates are created and sent to the client by the controller through the server after the game state is modified as a result of an action taken by a user. This means that if the model signals an error, the state of the game doesn’t change and an update is not sent. Updates are used by the client to update the playing board and the available user actions that a user can take in the UI.

An **Error** (which is an abstract class) also contains:

- a String containing an error description.

Below is a complete list of all (concrete) UserAction, Info, Update and Error defined thus far:

UserAction

- LoginUserAction
- LobbyDisconnectUserAction
- GameSettingsUserAction
- TowerColorUserAction

- WizardUserAction
- PlayAssistantUserAction
- MoveStudentUserAction
- MoveMotherNatureUserAction
- TakeFromCloudUserAction
- UseCharacterUserAction
- UseAbilityUserAction
- EndTurnUserAction
- LogoutUserAction
- PingUserAction

Info

- ServerLoginInfo
- LogoutSuccessfulInfo
- PingInfo

Update

- Update

Error

- LoginError
- IllegalSelectionError
- IllegalActionError
- DisconnectionError

GAME FLOW WITH MESSAGES:

Labels:

- *“s-”: happens on server/controller*
- *“c-”: happens on client*
- *“>>”: sent to client*
- *“<<”: sent to server*

–Setup–

During setup, messages are sent to specific clients, since not all clients connect simultaneously. But each client will follow this same flow

s- LobbyServer waits for any client to connect
c- Client displays a connection screen, where the user writes the IP and Port
Assuming no error arises, the sockets are then created server and client side. Otherwise, the player is prompted to retry.
c- Client starts a Ping with the LobbyServer. Every second a **PingUserAction** is sent. The server, upon receiving said action, responds by sending a **PingInfo**. This is done in order to keep the connection alive and confirm both ends are still connected
c- Client displays a login screen
<< When the user has entered their nickname, the client sends a **LoginUserAction**
s- LobbyServer runs a check on login credentials, if some error arises a **LoginError** is sent and the user will have to retry with a new nickname
>> LobbyServer sends **LoginInfo** containing the IP and Port of the match server to which the client must connect in order to take part in the game. It also performs other operations knowing the client will now disconnect from the lobby
c- The ping with the lobby is stopped
<< Client sends a **LobbyDisconnectUserAction** then closes the socket
s- Removes the client from the Lobby

c- Using the information contained in the previous LoginInfo, the client automatically connects to the MatchServer
Assuming no error arises, the sockets are then created server and client side.
c- Client starts a Ping with the MatchServer. Every second a **PingUserAction** is sent. The server, upon receiving said action, responds by sending a **PingInfo**. This is done in order to keep the connection alive and confirm both ends are still connected
<< The client automatically sends a **LoginUserAction** to log in the match server
s- MatchServer checks that the client was supposed to connect to this server, otherwise a **LoginError** is sent.
s- Then the LoginUserAction is passed to the controller to create the player in the game state
>> Controller sends (through the server) an **Update** (which, as we stated, contains the game state to update the UI and the next action to take in order to progress)
c- Client parses the Update
if (next action == game settings)

    c- Shows game settings selection screen
    << When the user has chosen, the client sends **GameSettingsUserAction**
    s- Controller sets game settings attributes
    s- Controller instantiates Game
    >> Controller sends an **Update**

c- Shows TowerColor selection screen
<< Client sends **TowerColorUserAction**
s- Controller creates a player in the game

```
>> Controller sends an Update
c- Client shows Wizard selection screen
<< Client sends WizardUserAction
s- Controller gives wizard to player
>> Controller sends an Update
s- When all players have connected and decided, Controller starts the game, changing the phase and
selecting first player in turn controller
```

–Planning Phase–

*During the game, each Update is **sent to all clients**, as all players are now connected. This is because everybody should know what the others are doing in their turns and be updated immediately whenever an action is taken*

```
s- Controller refills clouds
if (LastRoundException is caught)
    s- Controller changes its state to not skip the cloud selection
>> Controller sends an Update
c- Client updates UI to accurately display the planning phase
if (it's this Client's turn)
    c- Client shows assistant cards to play for the user to select
    << Client sends PlayAssistantUserAction
    s- Controller plays the card
    if (LastRoundException is caught)
        s- Controller changes its state to not skip the cloud selection
    s- Controller progresses to the next turn with TurnController.
    if (there is a next player)
        >> Controller sends an Update
        -the flow goes back to "if(it's this Client's turn)" above
    else
        s- Controller changes the phase
        -the flow continues at "–Action Phase–" below
else
    c- It's not the client's turn: the client keeps all buttons disabled and waits for another Update
    that says it's their turn
```

–Action Phase–

```
>> Controller sends an Update
c- Client updates UI to accurately display the action phase
if(it's this Client's turn)
    c- Client shows MoveStudent selection screen, and user selects accordingly
    << Client sends MoveStudentUserAction
    s- Controller moves the student
    if (there is a next player)
        >> Controller sends an Update
        -the flow goes back to "Client shows MoveStudent" above
    else
        s- Controller changes the phase
        -the flow continues at ">> Controller sends an Update" below
    >> Controller sends an Update
    c- Client shows MoveMotherNature selection screen
    << Client sends MoveMotherNatureUserAction
    s- Controller moves mother nature (Possible GameOver)
    >> Controller sends an Update
```

```
        c- Client shows TakeFromCloud screen
        << Client sends TakeFromCloudUserAction
        s- Controller takes from cloud
        c- Client shows end turn screen
        << Client sends EndTurnUserAction
        s- Controller progresses to the next turn, with turnController.
        if (there is a next player)
                >> Controller sends an Update
                -the flow goes back to "if(it's this Client's turn)" above
        else
                s- Controller changes the phase
                -the flow continues at "--End of round–" below
else
        c- It's not the client's turn: the client keeps all buttons disabled and waits for another Update
        that says it's their turn
```

**if (EXPERT GAME):*

```
c- When screen is updated to show start of action phase, UI displays the characters and when it's a Client
turn, enables their selection
<< At any time during their action phase, a client can use a character and send UseCharacterUserAction
s- Controller uses the character, and receives the requestParameters
if(no request parameter is needed and the character has only one use)
        s- Controller activates the character's ability
>> Controller sends an Update
c- Client disables selection of characters
if(character has uses left)
        c- Client now lets the player activate the character ability; the player will have to select all of the
        request parameters
        << Client sends UseAbilityUserAction
        s- Controller uses ability
        >> Controller sends an Update
        -the flow continues at "if(character has uses left)" above
else
        c- Client won't allow player to activate the character's ability
```

–End of round–

s- Controller performs end of round operations (Possible GameOver)
s- If the game hasn't ended, switches the phase back to Planning
-Flow goes back to –PlanningPhase–

–End of Game–

s- If at any point a GameOverException is caught by the controller (it can be caught in the flow above where the words (Possible GameOver) are marked), the winner is determined
>> Controller sends an **Update** which specifies that the next UserAction is the "EndGameUserAction", which isn't an actual action but informs the client to show the winner and proceed to end the game
c- Client displays the win/lose screen and awaits for user confirmation
<< Client sends **LogoutUserAction**
s- Logs out the client and closes the match server
>> Server sends a **LogoutSuccesful**
c- Stops the Ping with the match server and closes the connection

–Error Handling–
The controller, before trying to use the given UserAction to change the state, always checks whether the action taken is legal (e.g. if a student has to be moved, it won't accept an action to move mother nature) and if the player who took it matches with the current player. If any of those criteria isn't satisfied, an **IllegalActionError** is sent.

If the action given is legal, the controller changes the state of the game using the data contained in the UserAction. Whenever the controller modifies the game state, it is able to detect any error (i.e. a move mother nature user action that contains a student ID instead of an island ID) by catching any exception thrown by the model. If any exception is caught, an **IllegalSelectionError** is sent.

Both of these situations do not cause the game state to change in any way, meaning no Update is sent.

Whenever the client receives any of the two errors above it asks the user to retake the last action taken. It is equivalent to resuming the flow from the last Update received.

A network error, such as socket exceptions or the expiring of the socket timeout, causes the connection to be closed with the client; the server will send a **DisconnectionError** to any client that hasn't disconnected so that they can know that the connection was closed due to an error. The client will then immediately end the game.