Prova finale (Progetto di Ingegneria del Software)

Andrea Altomare
Marco Colombi
Giorgio Corbetta

11 Maggio 2020

Politecnico di Milano

Contents

1 Introduction **3**
1.1 Foundamental concepts . . . 3
1.2 Software components description . . . 3

2 Model-View-Controller pattern **3**
2.1 MVC communication . . . 4

3 Client-Server architecture solution **4**

4 Communication protocol **5**
4.1 Login phase . . . 5
4.2 New Game phase . . . 7
4.3 Game Preparation phase . . . 7
4.4 Playing phase . . . 9
4.5 Game Over phase . . . 10
4.6 Quit scenario . . . 11

5 Network problems detection protocol **11**
5.1 Server-side Ping system . . . 11
5.2 Client-side Ping system . . . 12

Abstract

This document shows the communication protocol for the distributed application system built for the final project of Software Engineering class.

1 Introduction

The built application works as a distributed software. It's a multitier application based upon Client-Server architecture on which, on a higher level concern, relies a Distributed MVC (*Model-View-Controller*) pattern.

1.1 Foundamental concepts

- Distributed MVC-designed application.
- Client-Server architecture.
- Thin Client solution.
- Business logic run entirely on the Server.

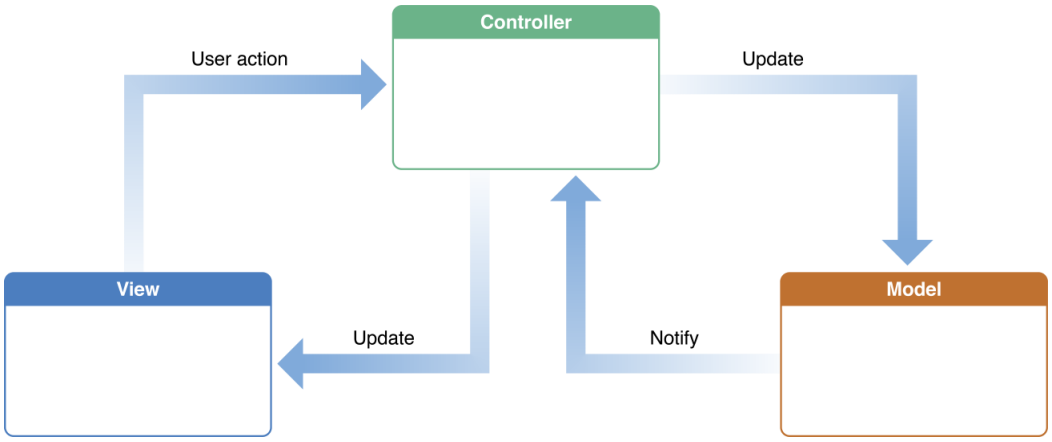
1.2 Software components description

- *Model:* software model with stored game data and operation to modify data.
- *Controller:* queries and update Model. Control the game flow among different players.
- *View:* render the game situation in a suitable way to the end-user.
- *Client:* Users' end-point used to connect with the Server.
- *Server:* Holds Model's data and the business logic to let the application run properly.
- *Observers:* Software components used to let proper View-Controller interaction through an Observer pattern.

2 Model-View-Controller pattern

As said above, the application relies on a (Distributed) MVC pattern which enables a better engineered software regard to development and maintenance aspects of the life-cycle.

Specifically speaking, an Apple-style MVC pattern was applied, this to simplify the overall interaction between users and the Server since the unique entry-point to query data and to receive updates is fully situated into the Controller.



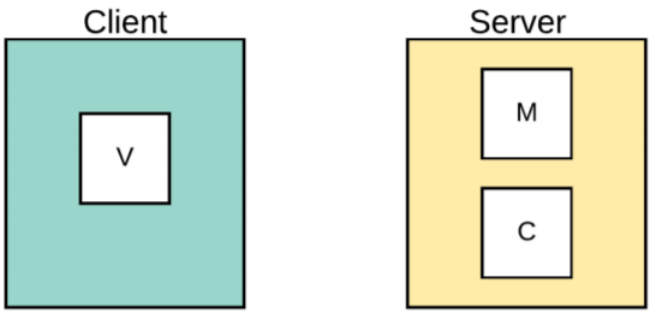
2.1 MVC communication

The scheme above quit-well represents the MVC communication implemented:

- *View-Controller:* View notifies the Controller with actions made by the user.
- *Controller-Model:* Controller processes the request and updates the Model (by methods call).
- *Model-Controller:* Model executes the operation invoked by the Controller and returns the result to the Controller.
- *Controller-View:* Controller processes the result received from the Model and updates the View accordingly.

3 Client-Server architecture solution

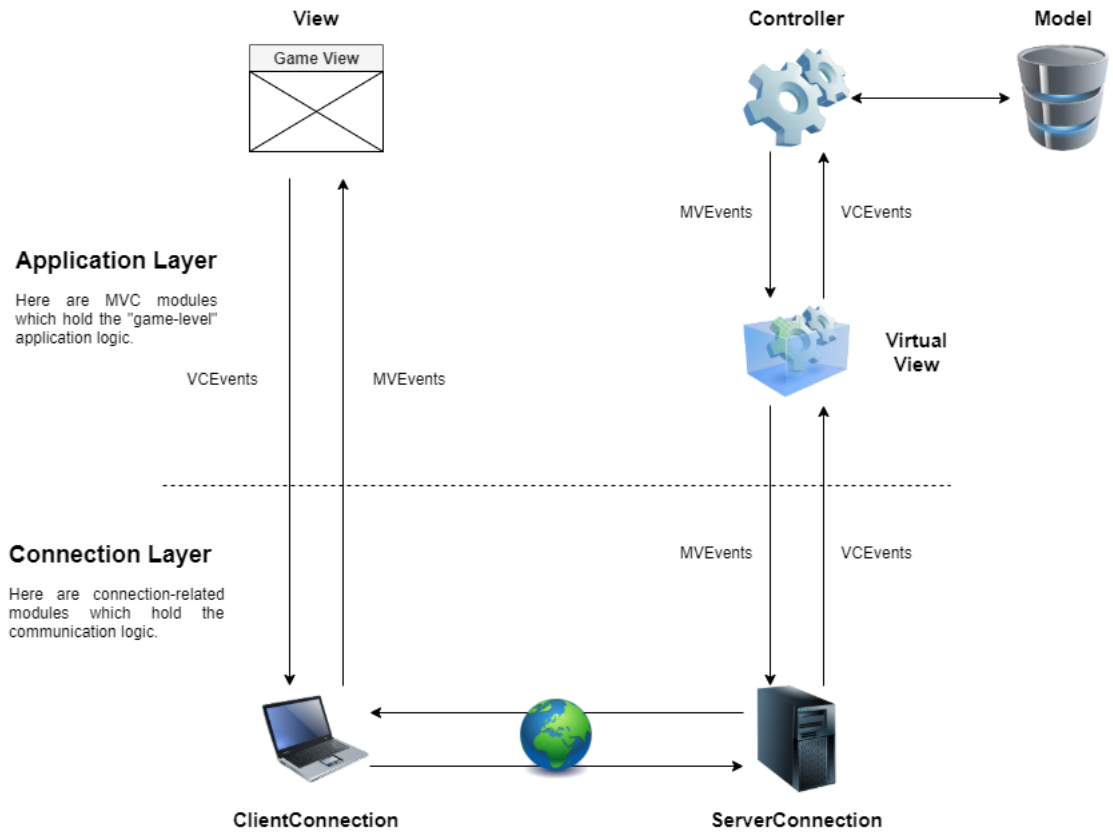
The distributed application is based upon a Client-Server architecture, which means data processing is centralized on the Server, which has the entire business logic and is responsible for the correct flow of the game among different participants. Clients have their own View updated by a series of requests sent to the Server.



The scheme above represents what is implemented: on the Client-side, a *thin-Client* logic which has the only responsibility of game-state scenario rendering (View), by querying the Server and receiving responses. On the Serve-side lies all the logic concerned to the request/response processing (Controller) and data (manipulation/persistence) management (Model).

Being a multitier architecture, MVC modules do not know about connection-related operations, and work as if they were on a single computer (by using the Observer pattern). So the application is structured onto two main layers:

- *Application layer:* implemented as a Model-View-Controller pattern, it's responsible for the game logic.
- *Network layer:* implemented using sockets, it's responsible for Client-Server communication over the network.



In the image above we can notice *View*, *Controller* and *Model* classes (implementing a Façade pattern to make the unique entry-point for the related subsystem) representing the high-level MVC pattern, and *ClientConnection*, *ServerConnection* and *VirtualView* classes acting as proxy/mediator entities to enable network-communication between Client and Server.

4 Communication protocol

The communication protocol is based on *Java serialization*.

Ad-hoc-made event objects (which extend from Java EventObject) are used to encapsulate specific requests/responses to/from the Controller (on a "connection-level" point of view, the Server). These objects are serialized by Java and sent to the network through sockets.

Sockets rely on TCP protocol: a Transport-Layer protocol (ISO/OSI stack and TCP/IP stack) which ensures the application does not need to deal with packet corruption and consistency.

4.1 Login phase

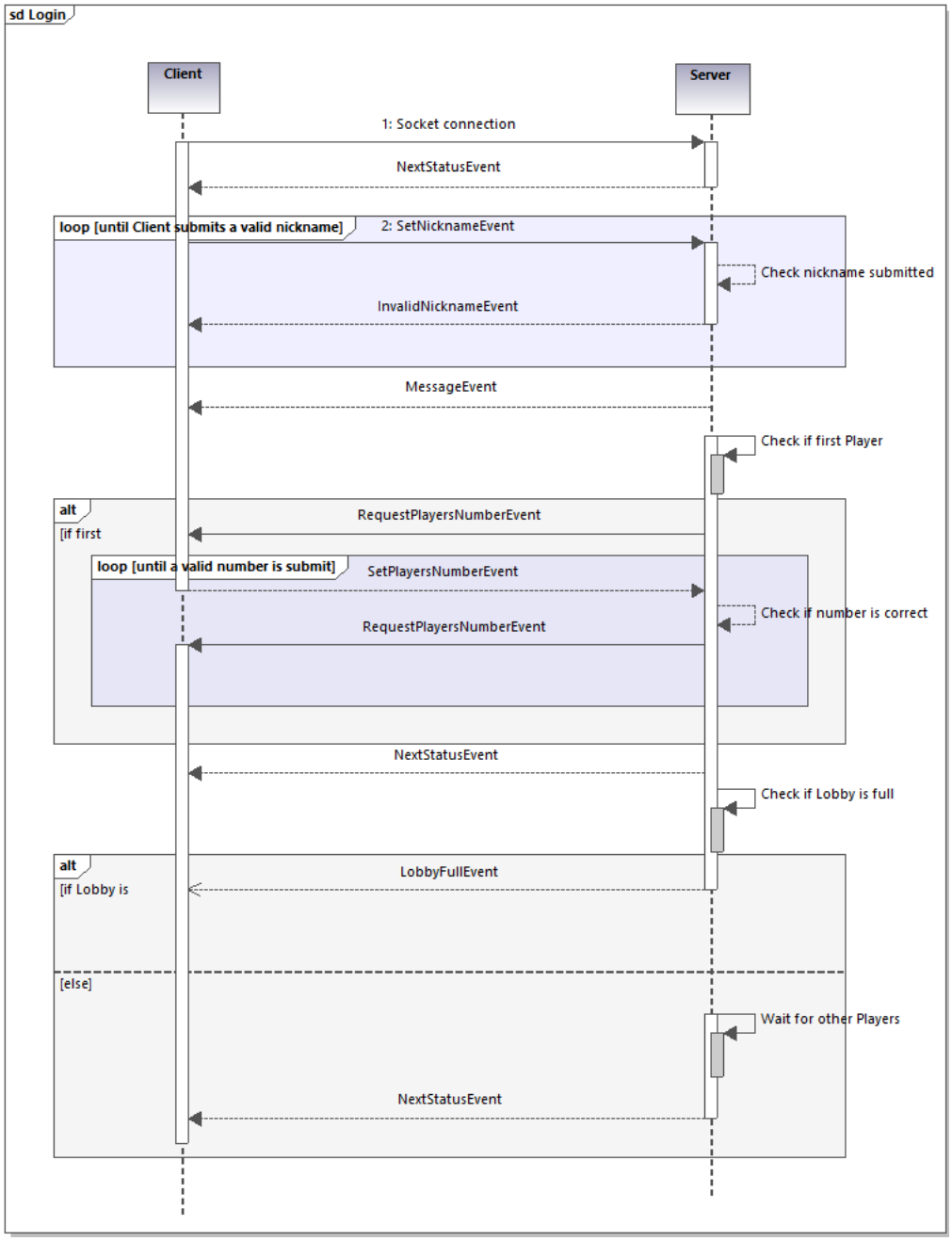
First of all, Client try to connect to the Server. When the connection is established, a *"NextStatusEvent"* is send by the Server, notifying Client that all preliminary connection-related operation has been done.

At this point the user needs to log into the game, so it sends a *SetNicknameEvent* (containing the nickname submitted by the user) to the Server. Server checks if the nickname is valid (e.g. it's not taken yet): if it isn't, an *InvalidNicknameEvent* is sent to the Client, asking for a new nickname to be submitted.

Server checks if the new Client is the first to connect: if it is, Server asks the user for the number of Players wanted for the upcoming game by sending a *RequestPlayersNumberEvent*. Client is now supposed to respond with a *SetPlayersNumberEvent* containing a valid number (2 or 3), Server checks for its correctness and, if it's not, keeps asking the Client for a valid number of Players by sending new *RequestPlayersNumberEvent* messages.

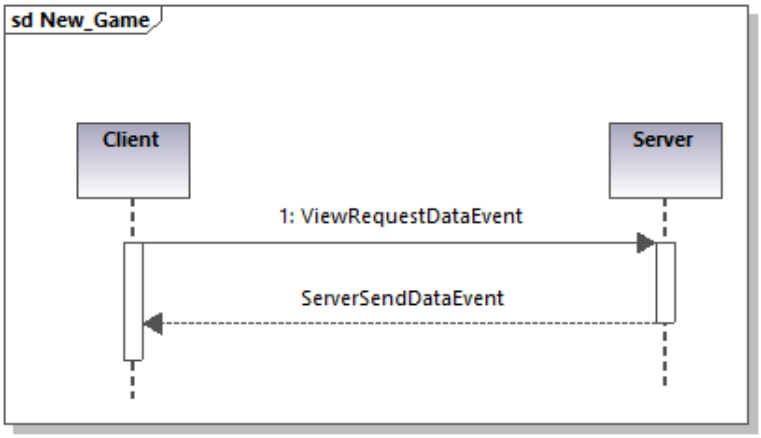
Now the game lobby can be set-up. Server proceeds to create a new lobby. There are now three possibilities:

1. There are not enough players yet, so connected Clients are kept in a waiting list;
2. There are enough players to start the game, so a *NextStatusEvent* is sent to all the Clients, notifying them that a new game match is beginning;
3. The game has already started, so the lobby is full: a *LobbyFullEvent* is sent to the new connected player to inform him/she about the situation, then the Client is unregistered from the Server and the connection is closed.



4.2 New Game phase

When a new game starts, Clients require to the Server some data to show their users some general game info (such as players' name).



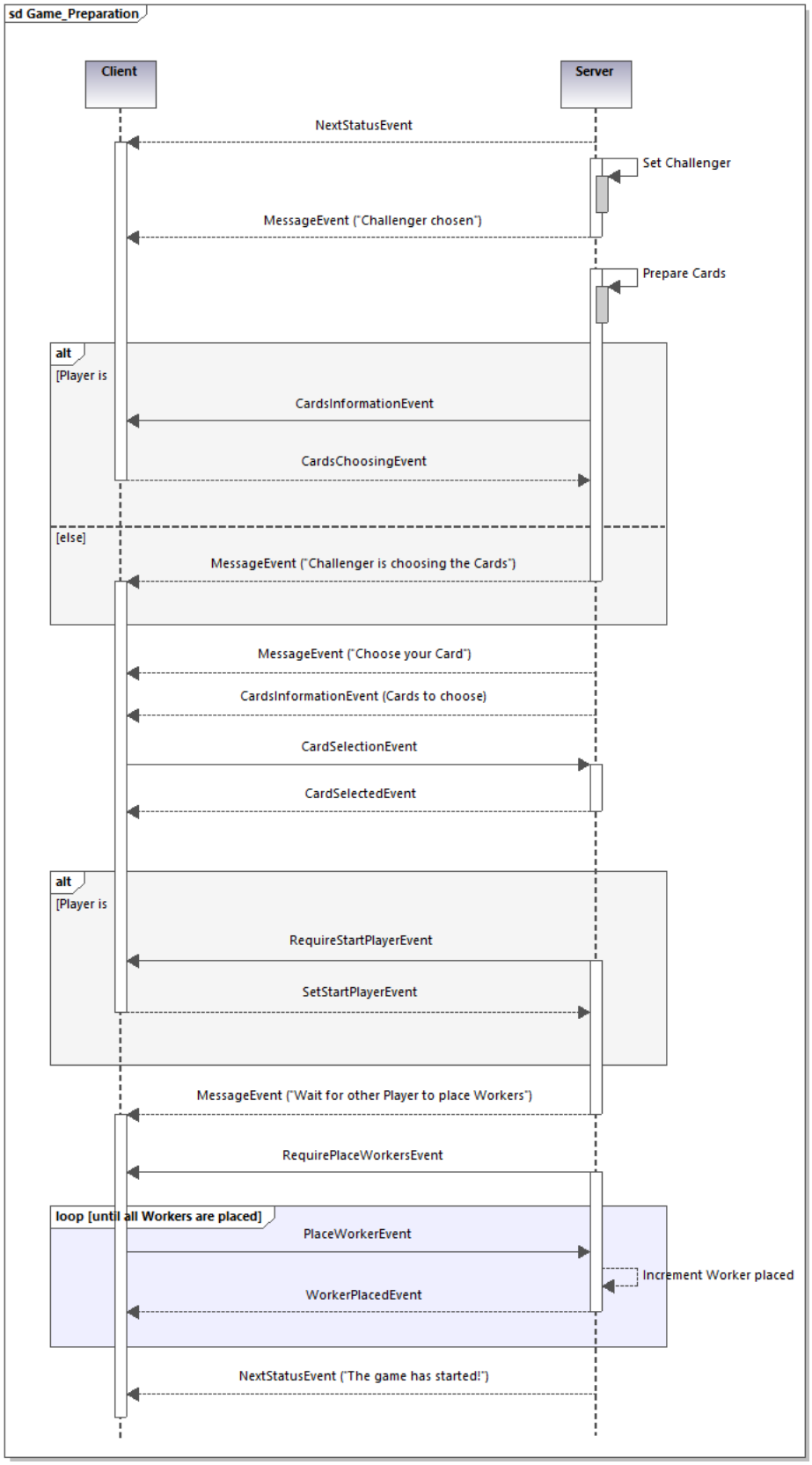
4.3 Game Preparation phase

In this game the game as started. The first step is to set the ground for the game scenario.

Server choose the Challenger player, and asks him/her to choose Cards for this game (*CardChoosingEvent*). Then, every player is asked to select a Card among those chosen previously. *CardsInformationEvent* convey Cards information to the players whereas *CardSelectionEvent* holds the Card selected by the player; a *CardSelectedEvent* is broadcasted by the Server to inform every user about a choice made by a player.

Challenger is also asked to choose the Start Player (*SetStartPlayerEvent*) by a *RequireStartPlayerEvent*.

In the end, every player places his/her own Workers through a *PlaceWorkerEvent* and the Server broadcast the updated Board scenario by a *WorkerPlacedEvent*.


sd Game_Preparation
Client
Server
NextStatusEvent
Set Challenger
MessageEvent ("Challenger chosen")
Prepare Cards
alt
[Player is
CardsInformationEvent
CardsChoosingEvent
[else]
MessageEvent ("Challenger is choosing the Cards")
MessageEvent ("Choose your Card")
CardsInformationEvent (Cards to choose)
CardSelectionEvent
CardSelectedEvent
alt
[Player is
RequireStartPlayerEvent
SetStartPlayerEvent
MessageEvent ("Wait for other Player to place Workers")
RequirePlaceWorkersEvent
loop [until all Workers are placed]
PlaceWorkerEvent
Increment Worker placed
WorkerPlacedEvent
NextStatusEvent ("The game has started!")

4.4 Playing phase

This is the main phase of the game.

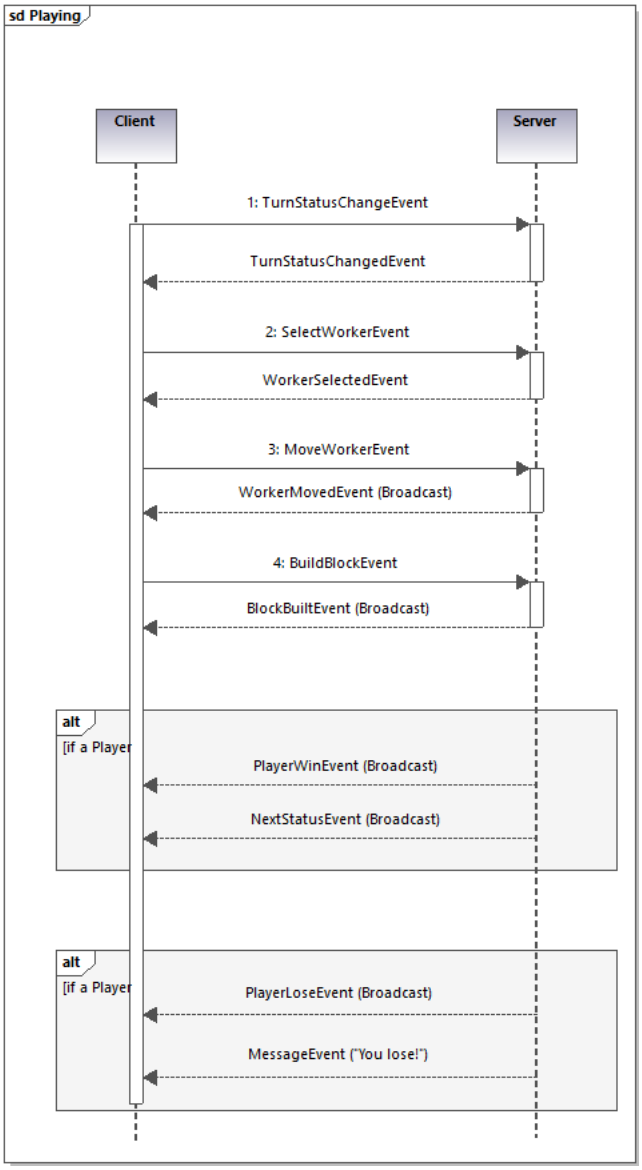
Each player starts his/her own turn by receiving a *TurnStatusChangedEvent* by the Server (which inform a user that his/her turn is started).

Users can switch their turn at any time as they please, since it's the very game to let this possibility (by using some specific Cards), through a *TurnStatusChangeEvent*. The Server of course has the responsibility to validate user's choice by sending him/her a *TurnStatusChangedEvent* as a confirmation.

Users' turn works in a very standard way:

1. Player select a worker by which make a move (*SelectWorkerEvent* – *WorkerSelectedEvent*);
2. A *Movement move* is made by sending a *MoveWorkerEvent* (and receiving a *WorkerMovedEvent* as a confirmation);
3. A *Build move* is made by sending a *BuildBlockEvent* (and receiving a *BlockBuiltEvent* as a confirmation);
4. If a Player wins, a *PlayerWinEvent* is sent in broadcast by the Server;
5. If a Player lose, a *PlayerLoseEvent* is sent in broadcast by the Server to inform all the players.

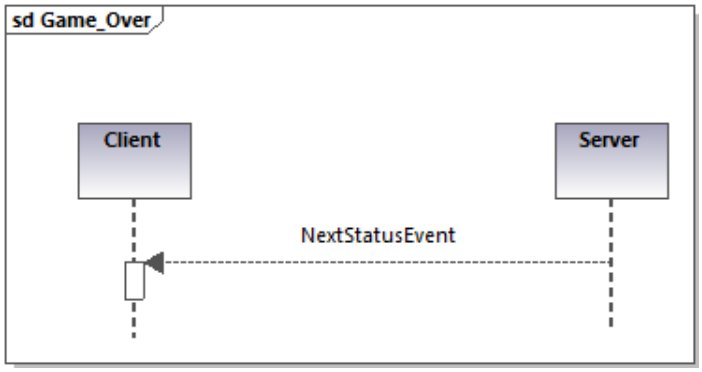
When a game is finished, a *NextStatusEvent* is broadcasted to let Clients go to the next state (of the View).



4.5 Game Over phase

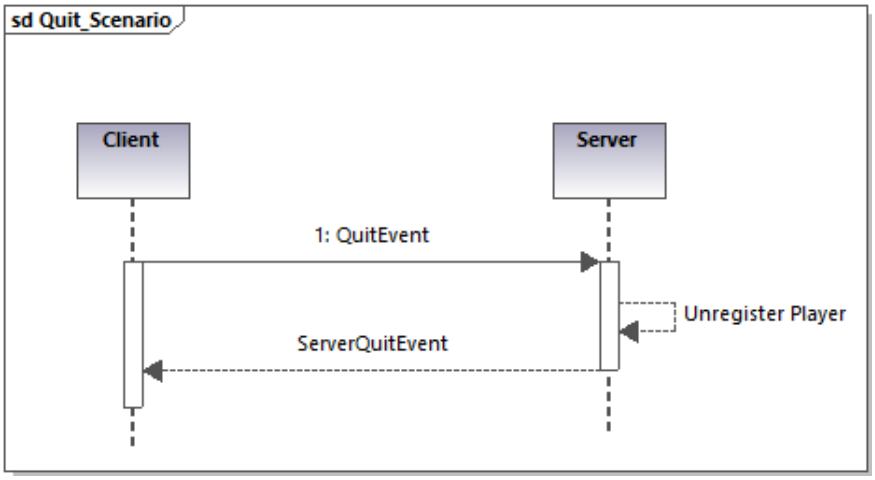
This phase is just for informing players that the game is over.

A new game can start (Server would broadcast a *NextStatusEvent* message).



4.6 Quit scenario

At any time, if a Player wants to quit the game (even if it has not started yet) he/she can send through his/her Client a *QuitEvent* to the Server; the Server would unregister the Player form the Clients in playing list, then a *ServerQuitEvent* is sent to the Client as response and the connection is eventually closed.



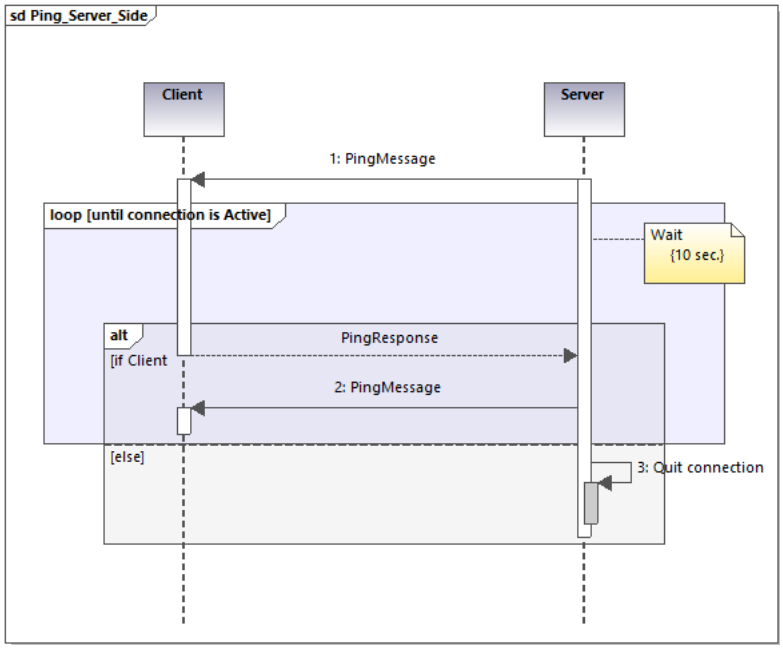
5 Network problems detection protocol

With Java's sockets, Server can detect whether a Client disconnects in an anomalous way, but it cannot detect if a Client is unreachable for network-related problems.

This part of the protocol is designed to resolve this kind of situation. A Ping-like system has been implemented.

5.1 Server-side Ping system

Every ten seconds, Server sends to the Client a *PingMessage*, then wait for a *PingResponse* from it. If no response arrives from the Client within 10 seconds, Server declares that Client as unreachable, closes the connection and unregister it from the list of Clients connected. Otherwise, every 10 seconds, the protocol is repeated with a new *PingMessage* from the Server.



5.2 Client-side Ping system

On the Client side, Client waits for the first *PingMessage* from the Server then responds with a *PingResponse*. At this point the protocol is this: Client waits for 20 seconds, then

- If a new PingMessage arrives from the Server, just respond with a PingResponse message; the protocol is now repeated every 20 seconds;
- Otherwise, Server is unreachable. Client closes the connection.