Group AM26

Simone Chen, Federico Di Cesare, Marco Donadoni

Introduction

In our project, communication between client and server is handled using both RMI and sockets. In order to support both protocols in a transparent way, the socket communication mimics RMI's behaviour. Socket communication is managed by two "endpoint" objects, whose job is to read from the socket and write to the socket Json messages: a "RemoteServer" on the client-side which implements the "Server" interface and a "RemoteView" on the server-side which implements the "View" interface. "View" and "Server" are the same interfaces used by RMI so that from the outside there is no difference between using RMI or sockets. At every remote method invocation made by the view , the RemoteServer generates a request object which gets serialized and sent on the socket. On the other end of the socket (server-side) the RemoteView reads this request, invokes the right method on the server and generates a response object, which is sent back on the socket. At this point the RemoteServer reads the response, which is then returned to the caller. The communication from the server to the view is made at the same way. By managing the method invocation this way we are de facto imitating a call to a remote method using RMI.

First phase: Login

After establishing a successful connection with the server, the view calls the login method, passing the nickname chosen by the user as a parameter. If the nickname is already used by some other player, the server returns false and the view proceeds to make another login request with a different nickname chosen by the user. If the login procedure is successful, the server returns true. The view then waits until the match starts, at which point the server sends the reduced model. During this waiting no messages are exchanged between server and client.

Example of socket login

Client sends login request:

```
{
  "_type":"LoginRequest",
  "nickname":"luca",
  "uuid":"342c7ab4-df11-44c0-91a7-e795a9cb8bac",
  "view":null
}
```

If the nickname is already used, the server returns a negative response:

```
{
  "_type":"LoginResponse",
  "uuid":"342c7ab4-df11-44c0-91a7-e795a9cb8bac",
  "res":false
}
```

Then the client tries to login with another nickname:
```
{
  "_type":"LoginRequest",
  "nickname":"marco",
  "uuid":"096c4361-fc49-4764-9039-340abaf77309",
  "view":null
}
```

If the nickname is note used, the server returns a positive response:
```
{
  "_type":"LoginResponse",
  "uuid":"096c4361-fc49-4764-9039-340abaf77309",
  "res":true
}
```

Login is now complete and the view waits for the start of the match. When the match is started, a reduced copy of the model is sent by the server to the view.

Note: the UUID of the response is the same UUID of the request because when a remote method is called, a request is sent and then the method waits for a response with the same UUID, which is then returned to the caller.

Second phase: Reduced model

After the login phase, when the match starts the server sends a reduced copy of the model to every view. Every confidential information is obviously removed from the reduced copy of the model, such as points or powerups of other players.

Third phase: controller executes methods and makes the gameflow carry on

After the view is initialized, the server starts to call remote methods on the views.

- ### Select game objects

The first remote method is used to make a selection of game objects, useful when the player needs to make a choice (for example which square to move on, which player to be shot etc.). Every game object has an unique identifier (UUID) and the selection process is made only by sending and receiving this identifiers. To ask the user to make a selection, a list of identifiers is sent along with the minimum and maximum number of objects to be chosen.

Example of socket selection

Server sends a request to select at least 1 and at most 2 objects among those given:
```
{
  "_type":"SelectObjectRequest",
  "objUuid":[
    "106051b8-13b0-41e3-a593-95a643851449",
    "46779cda-31fb-4131-a25a-26566bf72f88",
    "138bd9ba-8dbf-46fd-b651-4d4807092157"
  ],
  "min":1,
  "max":2,
  "uuid":"a969d8bc-1e20-45c3-80e8-2f46c2c2d07e"
}
```

Client selects only one object and sends the response back with the identifier of the object chosen:

```
{
  "_type":"SelectObjectResponse",
  "uuid":"a969d8bc-1e20-45c3-80e8-2f46c2c2d07e",
  "res":[
    "106051b8-13b0-41e3-a593-95a643851449"
  ]
}
```

Note: every UUID in the response must be one of the UUIDs present in the request.

- Show message

Another remote method is the one used to send a message to a player.

Example of socket show message

Server sends a request to show a message:

```
{
  "_type":"ShowMessageRequest",
  "message":"Ciao marco! Io sono il server!",
  "uuid":"793ebc85-d813-4e2d-b374-1cba79c55ef5"
}
```

Client sends back a void response after showing the message:

```
{
  "_type":"VoidResponse",
  "uuid":"793ebc85-d813-4e2d-b374-1cba79c55ef5"
}
```

- Update of the view

Another method is the one used to update the view after the models gets modified by an user (for example movement of a player, damage to a player, weapon recharged etc.). The view observes the model and every time the controller modifies the model an update is automatically sent to the views that need to be updated. In some cases, not every view needs to be updated (for example when the score of a player changes only the view of that player needs to be updated).

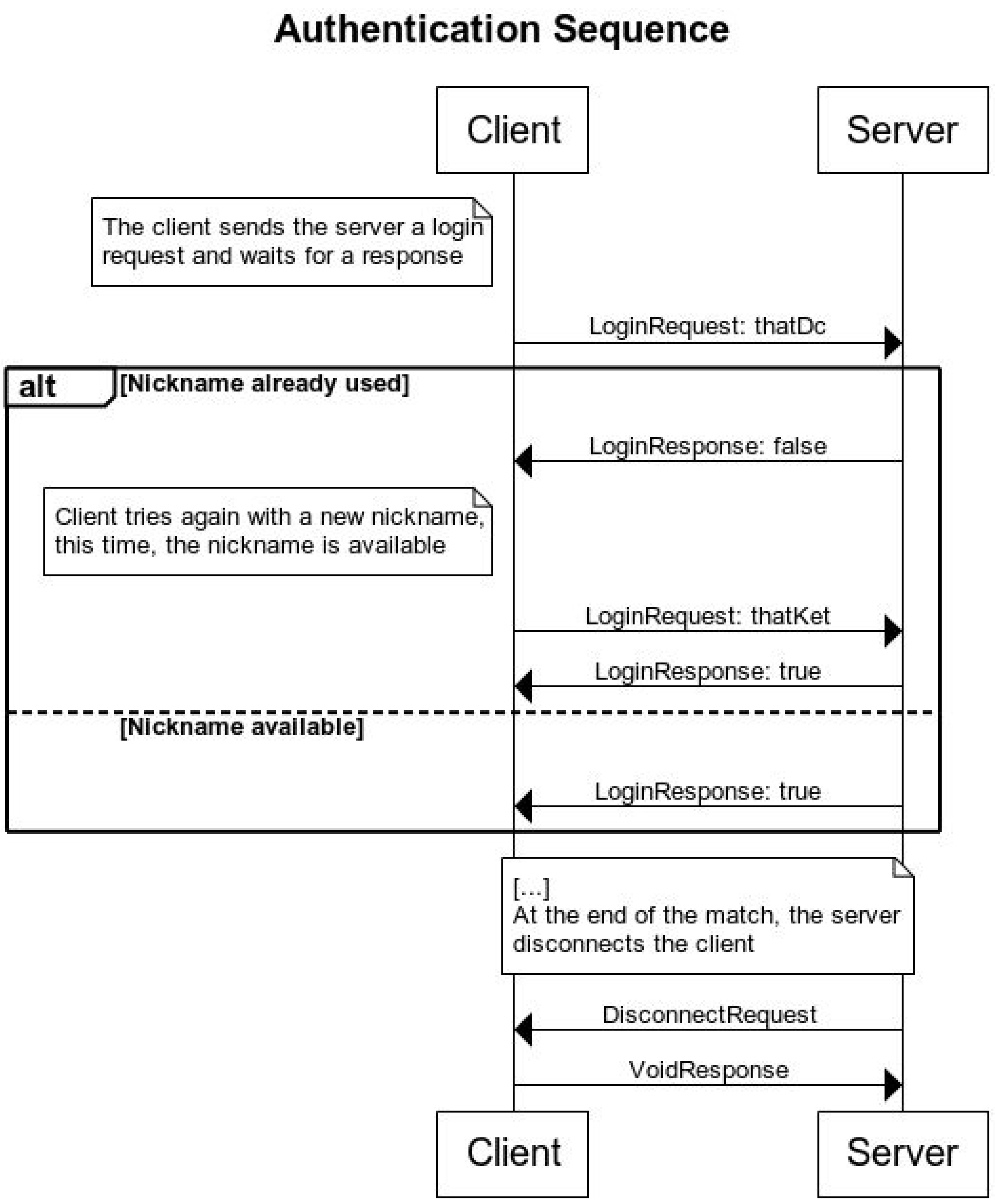
Fourth phase: Disconnection

When the match is finished, the server disconnects all the views.

Example of socket disconnection

Server sends:
```
{
  "_type":"DisconnectRequest",
  "uuid":"3435b20a-5171-4c23-8714-edb08642fc64"
}
```

Client confirms that it is going to disconnect by sending back a void response:
```
{
  "_type":"VoidResponse",
  "uuid":"3435b20a-5171-4c23-8714-edb08642fc64"
}
```



Game interaction examples:

Movement

Here are the messages exchanged to perform a movement action.
A player, during his turn, can decide to take a movement action, the logical steps are the following:

- The Controller checks how far the player can move.
- The Controller builds a list containing all the square eligible for the movement.
- The Controller builds a list containing the UUIDs of the squares included in the just mentioned list.
- The Controller calls the selectObject method on the Remote View (client), passing the list of UUIDs just generated.

Message from Server to Client:

```
{
  "_type":"SelectObjectRequest",
  "objUuid":[
    "uuidSquare1",
    "uuidSquare2",
    "uuidSquare3"
  ],
  "min":1,
  "max":1,
  "uuid":”messageUuid"
}
```

Since the request was sent because of a movement action, only a single Square must be selected, so the min and max value are set to 1 both.
The client receives the message and shows the user an interface that allows him to select the target square, as soon as the user decides where to move, the client builds a response and answers the server:

```
{
  "_type":"SelectObjectResponse",
  "uuid":”messageUuid",
  "res":[
    "uuidSquare2"
  ]
}
```

Now the Controller:

- Receives the selected square's UUID.
- Obtains the Square reference based on the UUID.
- Modifies the model so that the player is now on the selected square.

Grab

Here are the messages exchanged to perform a grab action.
A player, during his turn, can decide to take a grab action, the logical steps are the following:

- The Controller verifies which are the squares where the player can move to perform a grab action.
- The Controller builds a list containing the just mentioned squares.
- Based on this list, the Controller generates another list, containing the UUIDs of the squares.
- The Controller calls the selectObject method on the Remote View (client), passing the list of UUIDs just generated.

Message from Server to Client:
```
{
  "_type":"SelectObjectRequest",
  "objUuid":[
    "uuidSquare1",
    "uuidSquare2",
    "uuidSquare3"
  ],
  "min":1,
  "max":1,
  "uuid":”messageUuid"
}
```

Since the request was sent because of a grab action, only a single Square must be selected, so the min and max value are set to 1 both.
The client receives the message and shows the user a select square interface, when the user has selected the target square where he desires to move to grab the object, the client sends the Remote Server (controller) the following response:

```
{
  "_type":"SelectObjectResponse",
  "uuid":"messageUuid",
  "res":[
    "uuidSquare2"
  ]
}
```

Now the Controller::

- Receives the UUID of the selected square.
- Obtains the square's reference from the just received UUID.
- Changes the model so that the player moves to the selected square.
- Performs the grab action on the selected square.

Now we have two different scenarios:

1. The player stands on a StandardSquare.
2. The player stands on a SpawnSquare

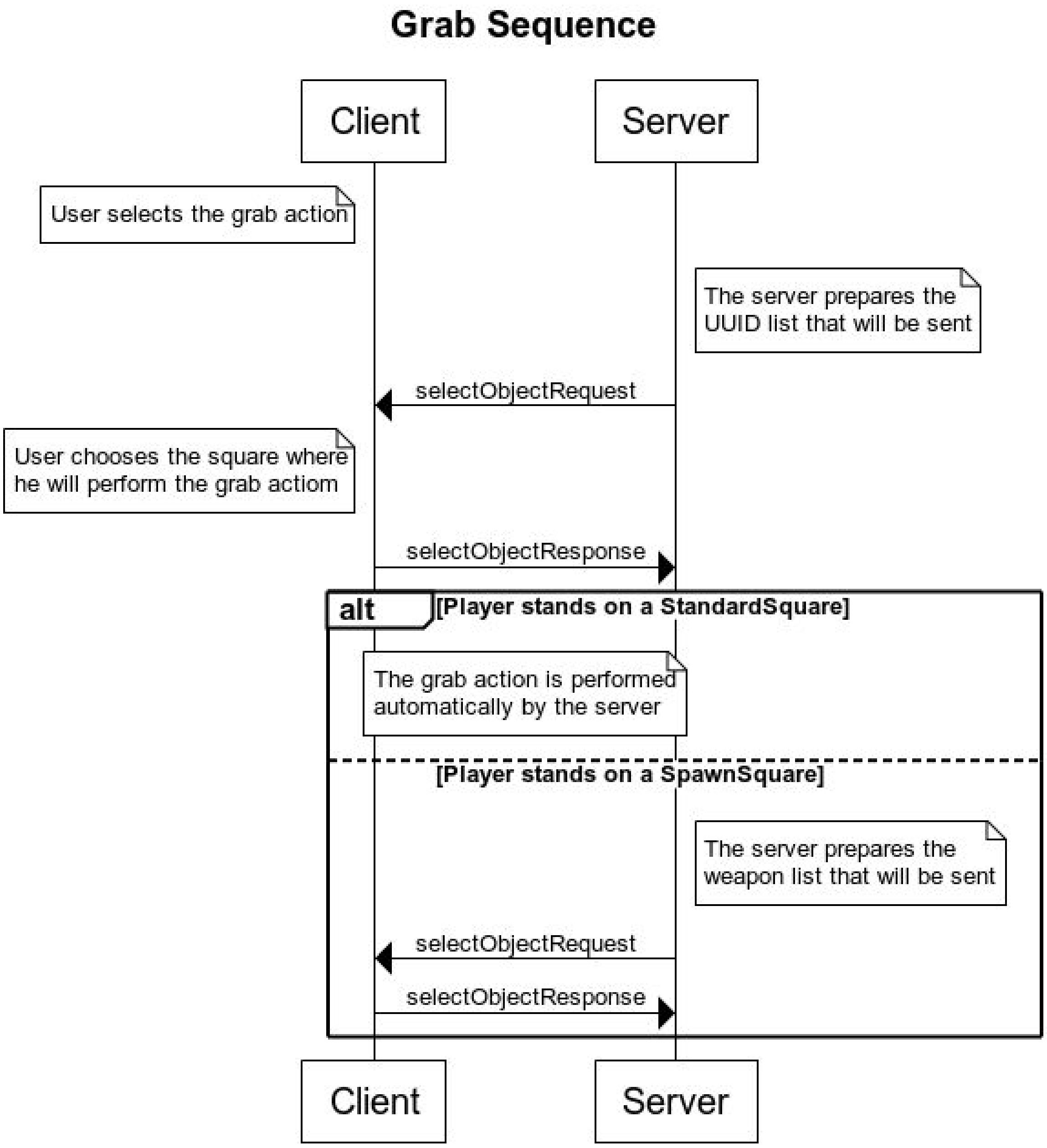
In the first scenario, the grab action is performed automatically, since there is only one AmmoTile on the StandardSquare.
Otherwise the Controller must ask the Client which one of the three available weapons he wants to pick:

Message from Server to Client:
```
{
  "_type":"SelectObjectRequest",
  "objUuid":[
    "uuidWeapon1",
    "uuidWeapon2",
    "uuidWeapon3"
  ],
  "min":1,
  "max":1,
  "uuid":"messageUuid"
}
```

Just like before, a grab action allows the user to grab only a single weapon, so min and max are both set to 1. The client shows the user a select weapon interface, as soon as the user chooses the weapon, the client answers the server:

Message from Client to Server:
```
{
  "_type":"SelectObjectResponse",
  "uuid":"messageUuid",
  "res":[
    "uuidWeapon3"
  ]
}
```

Once the response is received, the Controller:
- Checks the validity of the UUID.
- Obtains the weapon’s reference.
- Adds the player the chosen weapon, charged.


Grab Sequence
Client
Server
User selects the grab action
The server prepares the UUID list that will be sent
selectObjectRequest
User chooses the square where he will perform the grab actiom
selectObjectResponse
alt
[Player stands on a StandardSquare]
The grab action is performed automatically by the server
[Player stands on a SpawnSquare]
The server prepares the weapon list that will be sent
selectObjectRequest
selectObjectResponse
Client
Server