Gruppo AM26

Simone Chen, Federico Di Cesare, Marco Donadoni

Premessa

La comunicazione tramite rete, nel nostro progetto, avviene sia tramite protocollo RMI sia tramite comunicazione socket. Per supportare entrambi i protocolli in modo trasparente al resto del progetto, la comunicazione socket è stata progettata in modo da comportarsi allo stesso modo di RMI. In particolare la comunicazione socket viene gestita da due oggetti “endpoint”, che si occupano di leggere e scrivere opportuni messaggi Json sul socket: un oggetto “RemoteServer” che implementa l’interfaccia “Server” e sta dal lato client e un oggetto “RemoteView” che implementa l’interfaccia “View” e sta dal lato del server. Le interfacce “View” e “Server” sono le stesse interfacce usate da RMI, in modo che dall’esterno non ci sia differenza di utilizzo nel caso di connessione tramite RMI o tramite socket. Ad ogni invocazione di un metodo remoto da parte della view, il RemoteServer genera un oggetto di richiesta che serializza e manda sul socket; all’altro capo l’oggetto RemoteView legge questa richiesta, esegue il metodo giusto sul server e genera una risposta che scrive a sua volta sul socket; tale richiesta viene letta dal RemoteServer che ritorna la risposta al chiamante. Analogamente avviene la comunicazione server-view. In questa maniera si sta di fatto imitando tramite socket il funzionamento di una chiamata a metodo remoto tramite RMI.

Prima fase: Login

Dopo aver stabilito una connessione con il server, la view procede ad effettuare il login chiamando l’omonimo metodo. Se la procedura di login va a buon fine, il server ritorna true; altrimenti, nel caso in cui il nickname è già in utilizzo, ritorna false.

Esempio login tramite socket

Client manda richiesta di login:

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

Nel caso in cui il nickname sia già usato il server manda risposta negativa:

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

Client riprova ad effettuare il login con nickname diverso:

```
{
  "_type":"LoginRequest",
```

```
  "nickname":"marco",
  "uuid":"096c4361-fc49-4764-9039-340abaf77309",
  "view":null
}
```

Server manda risposta positiva:

```
{
  "_type":"LoginResponse",
  "uuid":"096c4361-fc49-4764-9039-340abaf77309",
  "res":true
}
```

Il login è ora completato e il server può iniziare a invocare metodi sul client.

Nota: L'UUID della risposta combacia con quello della richiesta, questo perché quando il controller invia una richiesta al remote client, la chiamata rimane in attesa di una risposta con lo stesso UUID della richiesta.

Seconda fase: Model ridotto

Dopo aver completato il login, il server manda una copia ridotta del model per inizializzare la view. Al model ridotto, ovviamente, vengono tolte tutte le informazioni "private" a cui il giocatore non ha accesso (ad esempio i punteggi e le carte powerup dei giocatori avversari).

Terza Fase: controller esegue metodi e fa proseguire il flusso della partita

Una volta inizializzata la view, il server può iniziare a chiamare metodi remoti.

- Selezione di oggetti di gioco

Il primo metodo è quello che si occupa della selezione di oggetti di gioco, utile quando deve il giocatore deve esprimere una scelta (ad esempio su quale cella spostarsi, quale giocatore colpire ecc.). In particolare ad ogni elemento di gioco è assegnato un codice identificativo univoco (UUID) e la selezione avviene scambiandosi semplicemente gli identificatori degli oggetti coinvolti. Oltre alla lista degli identificatori degli oggetti fra cui scegliere viene inviato anche il numero minimo e il numero massimo di oggetti da scegliere.

Esempio di selezione tramite socket

Server manda messaggio per selezione fra tre oggetti:

```
{
  "_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 manda risposta con identificatore dell'oggetto scelto:

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

Nota: L'UUID degli oggetti scelti contenuti nel payload della risposta deve ovviamente essere uno di quelli contenuti nel payload della richiesta.

- Visualizzazione messaggio

Un altro metodo remoto è quello che permette di mandare e visualizzare un messaggio al giocatore.

Esempio visualizzazione messaggio tramite socket

Server manda richiesta di visualizzazione di un messaggio:

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

Client risponde confermando la visualizzazione tramite una risposta vuota:

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

- Update della view

Un altro metodo è quello che permette di aggiornare la view in seguito ad eventi (ad esempio spostamenti di giocatori, danni a giocatore, ricarica arma, raccoglimento munizioni ecc.). In particolare la view osserva il model e ogni volta che il controller genera una modifica su esso viene automaticamente mandato un update alle view che devono essere aggiornate (in alcuni casi non tutte le view devono ricevere update, ad esempio nel caso in cui cambi il punteggio di un giocatore solo la view di tale giocatore deve essere aggiornata).

Quarta fase: disconnessione

Una volta terminata la partita, il server procede a disconnettere le view.

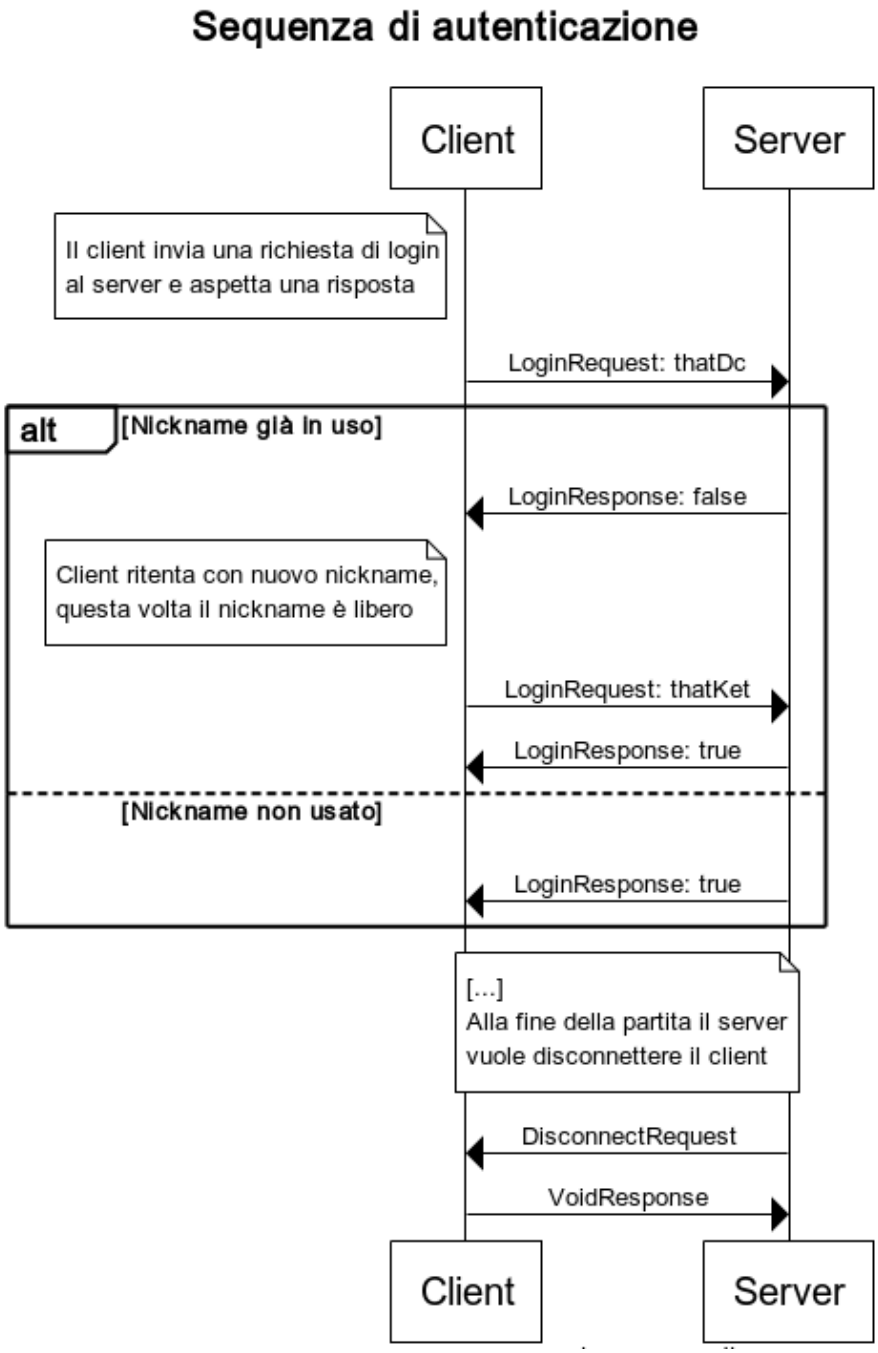
Esempio di disconnessione tramite socket

Server manda:


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


Client conferma che procede alla disconnessione tramite una risposta vuota:


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



Esempi di interazione di gioco:

Movimento

Messaggi scambiati per eseguire un'azione di movimento.
Un giocatore può, durante il suo turno, decidere di eseguire un'azione di movimento, il flusso logico è il seguente:

- Il Controller verifica di quanti movimenti si può spostare effettivamente il giocatore
- Il Controller raccoglie in una lista tutti i riferimenti alle Squares in cui il giocatore si può spostare
- Il Controller, a partire dalla lista appena citata, genera dunque una lista di UUID da mandare alla view remota (client).
- Il Controller chiama il metodo selectObject sulla remote view (client), passandogli la lista di

UUID appena generata.

Messaggio da Server a Client:

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

Essendo richiesta un'azione di movimento, c'è da scegliere un solo Square, perciò min e max sono settati a 1.
Il client riceve il messaggio e mostra un'interfaccia di selezione all'utente, quando l'utente ha selezionato la Square dove desidera muoversi, il client invia al remote server (controller) la risposta:

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

A questo punto il Controller:

- Riceve l'UUID della Square scelta
- Ricava il riferimento della Square a partire dall'UUID
- Sposta nel model il giocatore fino alla Square selezionata

Raccolta

Messaggi scambiati per eseguire un'azione di raccolta.
Un giocatore può, durante il suo turno, decidere di eseguire un'azione di raccolta, il flusso logico è il seguente:

- Il Controller verifica quali sono le Square nelle quali il giocatore si può spostare per eseguire la raccolta.
- Il Controller prepara una lista delle appena citate Square.
- A partire da questa lista, il Controller genera un'ulteriore lista di UUID da mandare al client.
- Il Controller chiama il metodo selectObject sulla remote view (client) passandogli la lista di UUID appena generata

Messaggio da Server a Client:

```
{
  "_type":"SelectObjectRequest",
```

```
  "objUuid":[
    "uuidSquare1",
    "uuidSquare2",
    "uuidSquare3"
  ],
  "min":1,
  "max":1,
  "uuid":"uuidMessaggio"
}
```

Essendo richiesta un'azione di movimento, ai fini della raccolta, c'è da scegliere un solo Square, perciò min e max sono settati a 1.
Il client riceve il messaggio e mostra un'interfaccia di selezione all'utente, quando l'utente ha selezionato la Square dove desidera muoversi per poter raccogliere l'oggetto, il client invia al remote server (controller) la risposta:

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

A questo punto il Controller:

- Riceve l'UUID della Square scelta.
- Ricava il riferimento della Square a partire dall'UUID.
- Sposta nel model il giocatore fino alla Square selezionata.
- Esegue l'azione di raccolta nella Square selezionata.

Ora abbiamo due diversi scenari:

1. Il giocatore si trova in una Square ordinaria
2. Il giocatore si trova in una SpawnSquare

Nel primo caso l'azione di raccolta viene eseguita autonomamente, raccogliendo l'unica tesserina di munizioni presenti sulla Square.
Nel secondo caso il Controller deve chiedere all'utente quali delle Weapon presenti sulla Square intende raccogliere:

Messaggio da Server a Client:

```
{
  "_type":"SelectObjectRequest",
  "objUuid":[
    "uuidWeapon1",
    "uuidWeapon2",
    "uuidWeapon3"
  ],
  "min":1,
  "max":1,
```

```
  "uuid":"uuidMessaggio"
}
```

Similmente a prima, l'utente deve scegliere una ed una sola arma, quindi min e max sono ancora settati a 1. Il client fornisce all'utente la schermata di dialogo per la selezione della Weapon da raccogliere, una volta che l'utente ha scelto, viene inviata la risposta.

Messaggio da Client a Server:

```
{
  "_type":"SelectObjectResponse",
  "uuid":"uuidMessaggio",
  "res":[
    "uuidWeapon3"
  ]
}
```

Ricevuta la risposta, il Controller procede a:

- Controllare la validità dell'UUID.
- Riconoscere la Weapon corretta a partire dall'UUID.
- Aggiungere la Weapon carica all'utente.

Sequenza di raccolta