Relazione network gruppo GC19

Aryan Sood, Federico Villa, Matteo Vitali, Marco Zanzottera

1 Introduzione

In questo documento viene presentata l'architettura dalla parte di rete del nostro progetto.

Le funzionalità avanzate che vogliamo supportare sono:

- Resilienza alle disconnessioni
- Multi-piattaforma
- Chat

2 Server

Per garantire maggiore scalabilità abbiamo scelto di dividere nettamente le classi che rappresentano i *main server* da quelle dei *game server*. Le prime estendono la classe astratta `Server`, si occupano di gestire i *client* (prima connessione, riconnessone e disconnessione) e permettono di dialogare con il `MainController` per registrare il proprio nome, creare o accedere ai giochi. Le altre, invece, vengono istanziate solo quando un giocatore accede (o viene ricollegato) ad un gioco e gestiscono tutte le richieste che un giocatore può fare nel corso di una partita (pescare una carta, mandare un messaggio in chat...).
Notare che due giocatori iscritti alla stessa partita si interfacceranno (mandando messaggi TCP o invocando metodi remoti) con *game server* diversi: il collegamento tra i due avviene a più basso livello mediante il riferimento ad una `GameController` comune.

In `Server` abbiamo implementato il metodo `computeHashOfClientHandler (client : ClientHandler)` che permette di generare, a partire dal nickname del giocatore e dal suo `ClientHandler`, un token segreto. Per essere riconnesso, il *client* dovrà, semplicemente fornire al *main server* a cui era precedentemente registrato il suo nickname ed il suo token.

Client e *server*, ad intervalli regolari, si scambiano messaggi di *heartbeat* (e.g. *ping* o *keep alive*) per notificare alla classe all'altro capo del canale di comunicazione di essere ancora attivi e non avere problemi di rete. Ogni *main server*

ha una `HashMap<_, Long>` dove "_" è l'oggetto che permette la comunicazione (`VirtualClient` per RMI e `Socket` per TCP) e `Long` è il timestamp dell'ultimo *heartbeat* ricevuto (`new Date().getTime()`).
Uno `ScheduledExecutorService`, periodicamente, controlla che non vi siano client per cui `current_time - last_timestamp > MAX_DELTA_TIME_BETWEEN_HEARTBEAT`. In caso ce ne fossero, viene avvisato il `MainController` che aggiornerà lo stato del giocatore e notificherà il `GameController` del gioco a cui è iscritto, se presente; l'informazione sul token, memorizzata in `connectedClients`, non viene eliminata e il *client* viene solo rimosso dalla `lastHeartBeatOfClients`. Alla riconnessone, il *main server* competente, dovrà sostituire l'oggetto "_" precedente con il nuovo "_" del *client*.

La disconnessione, a differenza della mancata ricezione di *heartbeat*, comporta l'eliminazione esplicita dal *server* (*main server* e *game server*) di tutte le informazioni o oggetti che fanno riferimento a quel *client*.
Abbiamo previsto due meccanismi diversi per la disconnessione:

- Disconnessione esplicita: il *client* sceglie volontariamente di disconnettersi dal server, mandando un messaggio `DisconnectMessage` oppure invocando il metodo remoto `disconnectFromServer(VirtualClient, String)`.
- Disconnessione silente: il *client* di cui il server non riceve più *heartbeat* da più di un'ora viene silentemente disconnesso. Questo è utile per pulire le `HashMap` di *main server* e `MainController` da tutti quei *client* non interagenti con il *server* che non sono riusciti a disconnettersi esplicitamente e, quindi, sono ancora proprietari del loro *nickname*.

Ogni `ClientHandler` (e.g. *game server*) ha una coda di messaggi (`MessageToClient`). In maniera asincrona al server, un *thread* li rimuove uno alla volta e notifica il *client* nei modi che saranno di seguito descritti. Quando un giocatore non è iscritto a nessun gioco oppure il *server* non ne riceve più gli *heartbeat*, l'attributo `gameController : GameController` viene settato a `null`. Prima di eseguire ogni azione, si controlla che `gameController` $\neq$ `null`: un utente che viene scollegato dal server (ad esempio per problemi di rete) prima di poter eseguire nuove azioni di gioco deve riconnettersi.

2.1 RMI

L'interfaccia `VirtualClient` rappresenta il *proxy* del *client* visto dal *server* ed ha un unico metodo `pushUpdate(MessageToClient)`. Questa scelta è molto vantaggiosa perché rende simili, almeno nella direzione *server to client*, RMI e TCP.

RMI è un protocollo sincrono, quindi il *client* quando invoca un metodo remoto del `VirtualMainServer` o `VirtualGameServer`, anche se questo è *void*, viene bloccato. Lato *server*, quindi, RMI è stato de sincronizzato utilizzando quanto spiegato precedentemente per il `ClientHandler`.

2.2 TCP

Per TCP occorre gestire esplicitamente la creazione della connessione e si comunica mediante messaggi.

Lato server, abbiamo definito una classe `TCPConnectionAcceptor` che estende `Thread` e, mediante un oggetto `ServerSocket`, attende nuove connessioni. Appena un *client* si collega, viene costruito un nuovo `MessageToServerDispatcher`, viene notificato il `MainServerTCP` e questo diventa osservatore del primo. Il `MainServerTCP` registra il `socket` (notare che il *server* non conosce ancora il nickname, quindi non genera nemmeno il token) mediante `registerSocket (socket)`.

Poiché abbiamo separato `MainServerTCP` e `ClientHandlerSocket`, è stato necessario introdurre il `MessageToServerDispatcher` per inoltrare al componente giusto il messaggio letto dalla rete: ad esempio, `GameHandlingMessage` e `HeartBeatMessage` devono arrivare al `MainServerTCP`, mentre gli altri al `ClientHandlerSocket`.

I messaggi che il *client* manda al *server* mediante TCP sono staticamente tutti `MessageToServer`. Il tipo dinamico potrebbe essere inferito mediante `instance of`: tale soluzione, però, non è ingegneristicamente accettabile oltre che poco mantenibile, quindi si è fatto ricorso ad un pattern `Visitor` (o di `double dispatch`). Ogni classe ha il metodo `accept(visitor : MessageToServerVisitor)` che controlla che `visitor` possa effettivamente visitarla e, in caso affermativo, chiama `visit(_)` passando sé stessa.

3 Comunicazione *server to client*

La logica per l'invio delle informazioni al *client* è differenziale: ad esempio, quando un giocatore piazza una carta, il *server* aggiorna gli altri *client* passando solo la variazione locale del modello e non una sua intera fotografia. Ciò, migliora la manutenibilità del codice e riduce il carico sulla rete.

3.1 Generazione e *routing* dei messaggi

Ogni messaggio è caratterizzato da un *header*, cioè la lista di nickname dei *client* a cui è destinato.
Per gestire i messaggi è stato utilizzato il pattern `Publisher - Subscriber`. I `Publisher` sono tutte le classi che possono generare messaggi (parte del modello e il `GameController`), mentre i `Subscriber` sono i `ClientHandler`. L'oggetto che li collega e fa effettivamente il *dispatch* dei messaggi è la `MessageFactory`. Quando un *client* accede a un gioco, il `GameController` sottoscrive alla `MessageFactory` il suo `ClientHandler`.

Se è necessario generare un messaggio, viene chiamato uno tra i seguenti metodi della `MessageFactory`:

- `sendMessageToPlayer(nick, message)`
- `sendMessageToPlayer(nick message)`
- `sendMessageToAllGamePlayers(message)`
- `sendMessageToAllGamePlayersExcept(nick, message)`

e la classe notificherà gli opportuni `Subscriber`.

Quando un *client* si disconnette, il `GameController` lo rimuove dai `Subscriber` della `MessageFactory`.

Potrebbe sembrare che non sia stato rispettato il normale funzionamento previsto dal pattern `Publisher - Subscriber`, in quanto questo prevede che il `Publisher` notifichi sempre tutti i `Subscriber` allo stesso modo, per garantire modultarità e permettere in futuro l'aggiunta di ulteriori classi che sono `Subscriber`, ma con funzioni differenti. Abbiamo previsto questa possibilità di evoluzione: da un lato, manteniamo la classica implementazione `Publisher - Subscriber` con quelli che chiamiamo `Subscriber` anonimi, dall'altro aggiungiamo dei `Subscriber` con nome (come sono ad esempio i nostri `ClientHandler`). Possiamo selettivamente inviare messaggi tramite il nickname, ma i `Subscriber` anonimi riceveranno sempre tutti i messaggi inviati, permettendo ad esempio l'introduzione di una classe che effettua il log dei messaggi.

3.2 Messaggi *server to client*

I `MessageToClient` sono organizzati gerarchicamente e possono essere suddivisi come segue:

- *AnswerToActionMessage*

  Rappresenta una generica risposta (positiva o negativa) ad un'azione di gioco che il *server* manda al *client*. In particolare:

Messaggio	*Uso e contenuto*
AcceptedChooseGoalCardMessage	La scelta della carta obbiettivo segreta è stata accettata.
AcceptedColorMessage	La scelta del colore è stata accettata.

AcceptedPickCardMessage	*Abstract.* Il pescaggio di una carta per il giocatore con nome *"nickname"* è stato accettato. Vengono fornite informazioni sulla carta pescata e sul nuovo simbolo in cima al mazzo.
AcceptedPickCardFromTableMessage	Il pescaggio di una carta di una carta dal tavolo è stato accettato. Vengono fornite informazioni sulla nuova carta da mettere sul tavolo e sui mazzi.
AcceptedPlaceCardMessage	*Abstract.* Il giocatore *"nick"* ha piazzato una carta e questo è stato accettato. Vengono fornite informazioni sui simboli visibili.
AcceptedPlaceInitialCard	La carta iniziale è stata piazzata correttamente.
AcceptedPlacePlayableCardMessage	Il piazzamento di una carta giocabile è andato a buon fine. Vengono date informazioni sui punti aggiornati e su come la carta è stata piazzata.
OwnAcceptedPlaceCardMessage	È la versione di *AcceptedPlacePlayableCardMessage* che viene mandata al giocatore che ha piazzato la carta.
OwnAcceptedPickCardFromDeckMessage	È la versione di *AcceptedPickCard* che viene mandata al giocatore che ha pescato la carta.
OtherAcceptedPickCardFromDeckMessage	È la versione di *AcceptedPickCard* che viene mandata agli altri giocatori quando uno pesca una carta.
OtherAcceptedPlacePlayableCardMessage	È la versione di *AcceptedPlacePlayableCardMessage* che viene mandata agli altri giocatori.
RefusedActionMessage	Descrive il tipo di errore occorso a seguito di una certa mossa.

- *NotifyChatMessage*

  Questo messaggio viene utilizzato dal *server* per notificare al *client* che è arrivato un nuovo messaggio in chat.

- *ConfigurationMessage*

  I messaggi che estendono questa classe sono utilizzati per supportare la resilienza alle disconnessioni e per passare al *client* informazioni del gioco che non possono essere rappresentate in forma differenziale. In particolare:

Messaggio	*Uso e contenuto*
OwnStationConfigurationMessage	Rappresenta la postazione di gioco del giocatore. È utilizzato sia quando il *player* entra in un gioco, sia quando si ricollega.
OtherStationConfigurationMessage	Rappresenta la postazione degli altri giocatori. È utilizzato quando un nuovo *player* si (ri -) collega al proprio gioco.
TableConfigurationMessage	Rappresenta il tavolo da gioco alla (ri -) connessione.
GameConfigurationMessage	Da informazioni sul gioco: lo stato, il numero di giocatori, il *player* corrente al momento della riconnessone.

- *NotifyEventOnGame*

  I messaggi che estendono questa classe sono usati dal *server* per notificare ai *client* che c'è stato qualche evento nel gioco a cui sono iscritti (ad esempio un nuovo giocatore si è connesso). In particolare:

Messaggio	*Uso e contenuto*
AvailableColorsMessage	Un giocatore ha scelto il suo colore: gli altri devono essere notificati dei nuovi colori disponibili.
EndGameMessage	Il gioco è finito: si mostrano i vincitori.
NewPlayerConnectedToGameMessage	Un nuovo giocatore si è connesso al gioco: all'inizio se ne mostra solo il nome.
BeginFinalRoundMessage	Notifica i giocatori che è iniziato l'ultimo round.

GamePausedMessage	Notifica i giocatori attualmente connessi che il gioco è andato in pausa.
PlayerReconnectedToGame	Notifica che un giocatore precedentemente disconnesso dal gioco si è ricollegato.
DisconnectedPlayerMessage	Notifica i giocatori connessi al gioco che un giocatore è stato disconnesso (per *heartbeat* o per propria volontà).
GameResumedMessage	Notifica i giocatori connessi al gioco che il gioco può riprendere.
StartPlayingMessage	Notifica che i giocatori possono effettivamente partire a giocare.

- *GameHandlingMessage*

  I messaggi che estendono questa classe sono usati dal *server* per notificare ai *client* informazioni in merito alla gestione dei giochi. In particolare:

Messaggio	*Uso e contenuto*
AvailableGamesMessage	Contiene l'elenco dei giochi disponibili a cui il giocatore può collegarsi.
CreatedPlayerMessage	La registrazione del nome del *player* è andata a buon fine.
JoinedGameMessage	La registrazione del *player* in un certo gioco è andata a buon fine.
CreatedGameMessage	Notifica che il gioco è stato correttamente creato.
GamePausedMessage	Notifica i giocatori attualmente connessi che il gioco è andato in pausa.
DisconnectGameMessage	Notifica che la disconnessione del giocatore dal gioco è andata a buon fine.
DisconnectServerMessage	Notifica che la disconnessione del giocatore dal *server* è andata a buon fine.

- *HeartBeatMessage*

  Il *server* manda al *client* questo messaggio ogni volta che riceve da questo un *HeartBeatMessage*; il *client* deve essere attivo e registrato sul *server*.

- *NetworkHandlingErrorMessage*

  Il *client* riceve questo messaggio ogni qualvolta fa una richiesta che però non è soddisfacibile per motivi legati alla logica con cui il *networking* è stato disegnato: ad esempio, il giocatore, dopo essersi disconnesso, vuole fare una mossa senza prima riconnettersi.

- *TurnStateMessage*

  Il messaggio *TurnStateMessage* viene mandato ogni volta che c'è da cambiare turno. Viene specificato quale diventa il giocatore in turno e quale azione (`TurnState.DRAW` o `TurnState.PLACE`) può eseguire.

4 Client

`ClientRMI` e `ClientTCP` rappresentano le interfacce di rete con cui il *client* può dialogare con il *server*. Entrambe implementano tre interfacce :

- `GameManagementInterface`: contiene tutti i metodi per creare, iscriversi o abbandonare un gioco e per avere i giochi disponibili.
- `NetworkManagementInterface`: contiene tutti i metodi che il `Client-Controller` potrebbe avere la necessità di invocare per interagire con il *server*. Ad esempio, registrare il *nickname* (`void connect(String nick)`) o disconnettersi.
- `ConfigurableClient`: i `Client` che implementano quest'interfaccia sono configurabili, cioè deve essere possibile registrare in essi, dopo la creazione, il *"cookie"* dell'utente. Quando il `ClientController` riceve un `Created-PlayerMessage`, infatti, registra nell'interfaccia di rete il *nickname* ed il *token*, in modo che questa possa gestire in maniera autonoma e trasparente la riconnessone.

  Le configurazioni sono gestite dalla classe statica `ConfigurationManager`. Il *"cookie"* viene salvato localmente in un file JSON *read-only* e finché l'utente non si disconnette esplicitamente viene mantenuto. Quindi, anche se la macchina dell'utente dovesse spegnersi, facendo ripartire il `Client` ci si può riconnettere.

Notare l'importanza di queste tre interfacce: componenti diverse della logica locale possono vedere solo i metodi di cui hanno effettivamente bisogno senza nemmeno sapere quale tecnologia di comunicazione sia impiegata.

Infine, ogni interfaccia di rete ha un suo `HeartBeatManager` che gestisce l'invio e la ricezione degli `HeartBeatMessage`: se `current_time` − `last_heartbeat_time` è maggiore di un certo valore soglia viene segnalato un possibile problema di rete.

`ClientRMI` e `ClientTCP` ricevono dal server dei messaggi che staticamente sono di tipo `MessageToClient`. Anche qui, quindi, come per il `MainServerTCP` non abbiamo utilizzato la `instance of`, ma abbiamo fatto ricorso al pattern *Visitor* per il *double-dispatch*.

4.1 Ricezione e gestione dei messaggi

I messaggi sono salvati ed elaborati in una classe comune alle due implementazioni di rete: `MessageHandler`. La classe crea un thread che estrae, uno alla volta, i messaggi da una coda e li elabora.

4.2 Interazione con il *server*

Per evitare ritardi nell'interfaccia utente, alla chiamata di funzioni remote si è deciso, in aggiunta a quanto fatto per il *server*, di desincronizzare le chiamate di rete dei *client*.

4.2.1 Client RMI

Se il thread che fa la chiamata remota è il thread principale dell'applicazione, l'intera *view* dell'utente risulterà bloccata. Per non incorrere in questo problema, il `ClientRMI` delega l'invocazione dei metodi remoti del `VirtualMainServer` o del `VirtualGameServer` ad un `ExecutorService` creato ad hoc.

4.2.2 Client TCP

La richiesta di invio di un messaggio ne provoca l'aggiunta in una coda. Sarà poi compito di un apposito thread rimuovere e mandare ordinatamente al server, uno per uno, i messaggi che tale coda contiene.

5 Messaggi *server to client*

I `MessageToServer` sono organizzai gerarchicamente e possono essere suddivisi come segue:

- *ActionMessage*

  Le classi che estendono `ActionMessage` incapsulano una determinata azione che dovrà essere processata da un *game server*. In particolare:

Messaggio	*Uso e contenuto*
ChoosenColorMessage	Contiene il colore che l'utente ha scelto.
DirectionOfInitialCardMessage	È il messaggio con cui il *player* notifica al *server* che vuole piazzare la sua carta iniziale con una certa `CardOrientation`.
PickCardFromTableMessage	Il *player* comunica quale carta sul tavolo vuole pescare.
ChosenGoalCardMessage	Notifica che il *player* ha scelto la sua carta obbiettivo privata.
PickCardFromDeckMessage	Il *player* comunica di voler pescare una carta dal mazzo.
PlaceCardMessage	Il giocatore comunica di aver piazzato una carta: non è necessario serializzare, di nuovo, l'oggetto carta perché è sufficiente il suo codice univoco.

- *GameHandlingMessage*

  Le classi che estendono `GameHandlingMessage` incapsulano una determinata azione che dovrà essere processata da un *main server*. In particolare:

Messaggio	*Uso e contenuto*
CreateNewGameMessage	Richiede la creazione di un nuovo gioco con il nome speificato.
JoinFirstAvailableGame	Il *player* chiede di essere registrato nel primo gioco disponibile.
NewUserMessage	Il *player* comunica di voler registrare il proprio nome presso il server.
RequestAvailableGamesMessage	Il *player* richiede informazioni sui giochi disponibili.
DisconnectMessage	Il *player* comunica di volersi esplicitamente disconnettere dal *server*.

JoinGameMessage	Il giocatore chiede di essere registrato al gioco con il nome specificato.
ReconnectToServerMessage	Il giocatore chiede di essere riconnesso al *server*.
RequestGameExitMessage	Il giocatore notifica di voler abbandonare il gioco cui è iscritto.
JoinGameMessage	Il giocatore chiede di essere registrato al gioco con il nome specificato.

- *PlayerChatMessage*

  La classe rappresenta un messaggio da inviare in chat.

- *HeartBeatMessage*

  La classe rappresenta il messaggio di *heartbeat* che l'interfaccia di rete del *client* manda al *server*.