Documento Protocollo di Rete

Riccardo Piantoni, Matteo Rossi, Federico Pinto, Jacopo Sacramone
Gruppo 12

May 2024

1 Descrizione della comunicazione

Il protocollo di comunicazione è stato progettato tenendo conto della necessità di ottenere trasparenza rispetto alle tecnologie Socket ed RMI. Ad alto livello, i singoli messaggi scambiati sono delle vere e proprie "richieste di esecuzione di azioni", tradotte nel protocollo di rete sottostante desiderato, sia da Client a Server sia viceversa.
Una tipica interazione consiste nell'invio da parte del Client di una richiesta, opportunamente incapsulata nel protocollo di rete scelto, di eseguire una determinata azione: dopo averla ricevuta e deincapsulata, se la richiesta riguarda un'azione consentita in base allo stato del giocatore o della partita, il Server eseguirà tale azione, altrimenti lancerà una eccezione. In entrambi i casi, il Server risponderà a sua volta con una richiesta in base all'esito dell'operazione: aggiornare il Model ridotto presente sul Client, oppure gestire l'eventuale eccezione a lui propagata. E' importante sottolineare che anche il Server può avviare da sé una comunicazione verso uno o più Client, al fine di inviare aggiornamenti relativi allo stato delle lobby, della partita o di una chat.
Di seguito sono descritti i package e i componenti principali dell'architettura di rete distribuita.

2 Descrizione del package Controller

2.1 Classi e interfacce comuni a Client e Server

2.1.1 ControllerInterface

Definisce la struttura generica di un controller di rete. Da essa ereditano le due interfacce ClientControllerInterface e ServerControllerInterface, che specificano i metodi che i rispettivi controller devono implementare, in modo che i comandi creati sul Client possano chiamare i comandi del ServerController (e viceversa) pur mantenendo nascosta e privata l'implementazione effettiva a fini di sicurezza.

2.1.2 SocketHandler

Classe astratta che contiene attributi e metodi per l'inizializzazione e la gestione di una connessione via Socket. A livello tecnologico il meccanismo di comunicazione utilizzato è quello degli AsynchronousSocketChannel di Java, che gestiscono automaticamente le operazioni di *accept()*, *connect()*, *read()* e *write()* in modo asincrono tramite una thread pool interna. Quando una connessione viene stabilita, viene creata un'istanza di un sottotipo opportuno di SocketHandler, che inizializza gli ObjectInputStream e ObjectOutputStream, incapsulandoli attraverso varie operazioni nei ByteBuffer necessari ad un AsynchronousSocketChannel per funzionare. Inoltre, la classe SocketHandler definisce anche le routine da eseguire al completamento o al fallimento di un'operazione di read o write: quando un thread termina una lettura, esso invoca la routine completed, la quale ne avvia asincronamente una nuova tramite un executor appartenente alla thread pool interna e poi passa ad eseguire il Command ricevuto.

2.1.3 Interfacce VirtualClient e VirtualServer

Contengono rispettivamente i metodi *requestToClient()* e *requestToServer()*, invocati per inviare le richieste. Sono estese dalle interfacce RMIVirtualClient e RMIVirtualServer, ovvero le loro declinazioni Remote proprie di RMI.

2.2 Package ClientController

2.2.1 SocketClient

Inizializza la connessione Socket, creando e memorizzando un riferimento ad un SocketServerHandler. Implementa l'interfaccia VirtualServer, il cui metodo tuttavia si limita a delegare la richiesta al ServerSocketHandler interno.

2.2.2 SocketServerHandler

Estende SocketHandler e implementa l'interfaccia VirtualServer. Riceve le richieste dal canale di connessione del client con il server ed esegue i comandi contenuti all'interno.

2.2.3 RMIClientSkeleton

Inizializza la connessione RMI, ottenendo il riferimento remoto al server e fornendogli il proprio in modo che il server possa comunicare con esso. Inoltre implementa l'interfaccia RMIVirtualClient, consentendo di al server di chiamare remotamente il metodo requestToClient() e quindi eseguire sul client i comandi desiderati.

2.2.4 ClientController

Gestisce l'esecuzione vera e propria dei comandi ricevuti dalla rete: implementando l'interfaccia ClientControllerInterface, deve ridefinire le funzioni che possono essere chiamate dai rispettivi ClientCommands all'interno del loro metodo *execute()*, di fatto aggiornando il model ridotto presente sul client oppure gestendo un'eccezione quando richiesto dal server.

2.3 Package ServerController

2.3.1 Server

Inizializza le connessioni Socket e RMI, mettendosi in ascolto su entrambi i protocolli e creando la thread pool che eseguirà i comandi ricevuti dal client.

2.3.2 SocketClientHandler

Estende SocketHandler e implementa l'interfaccia VirtualServer. Riceve le richieste dal canale di connessione del client con il server ed affida alla thread pool l'esecuzione dei comandi ricevuti.

2.3.3 RMIServerStub

Rappresenta l'oggetto remoto RMI, implementando l'interfaccia RMIVirtualServer e quindi il metodo requestToServer() che sarà chiamato dal client.

2.3.4 ServerController

Implementa l'interfaccia ServerControllerInterface, ridefinendo quindi i metodi che possono essere chiamati all'interno dei ServerCommands ricevuti ed effettuando controlli di integrità sui parametri presenti all'interno di ogni comando, così da poter segnalare un'eccezione in caso di richiesta errata o impossibile da eseguire.

2.4 Package Commands

Le azioni da eseguire sono incapsulate all'interno di comandi. Ogni specifico comando sovrascrive il metodo *execute()* e successivamente alla ricezione viene eseguito, come previsto dal Command Pattern.
Sono divisi in due categorie principali: **ClientCommands**, che il server chiede al client di eseguire, e **ServerCommands**, nella direzione opposta della comunicazione.

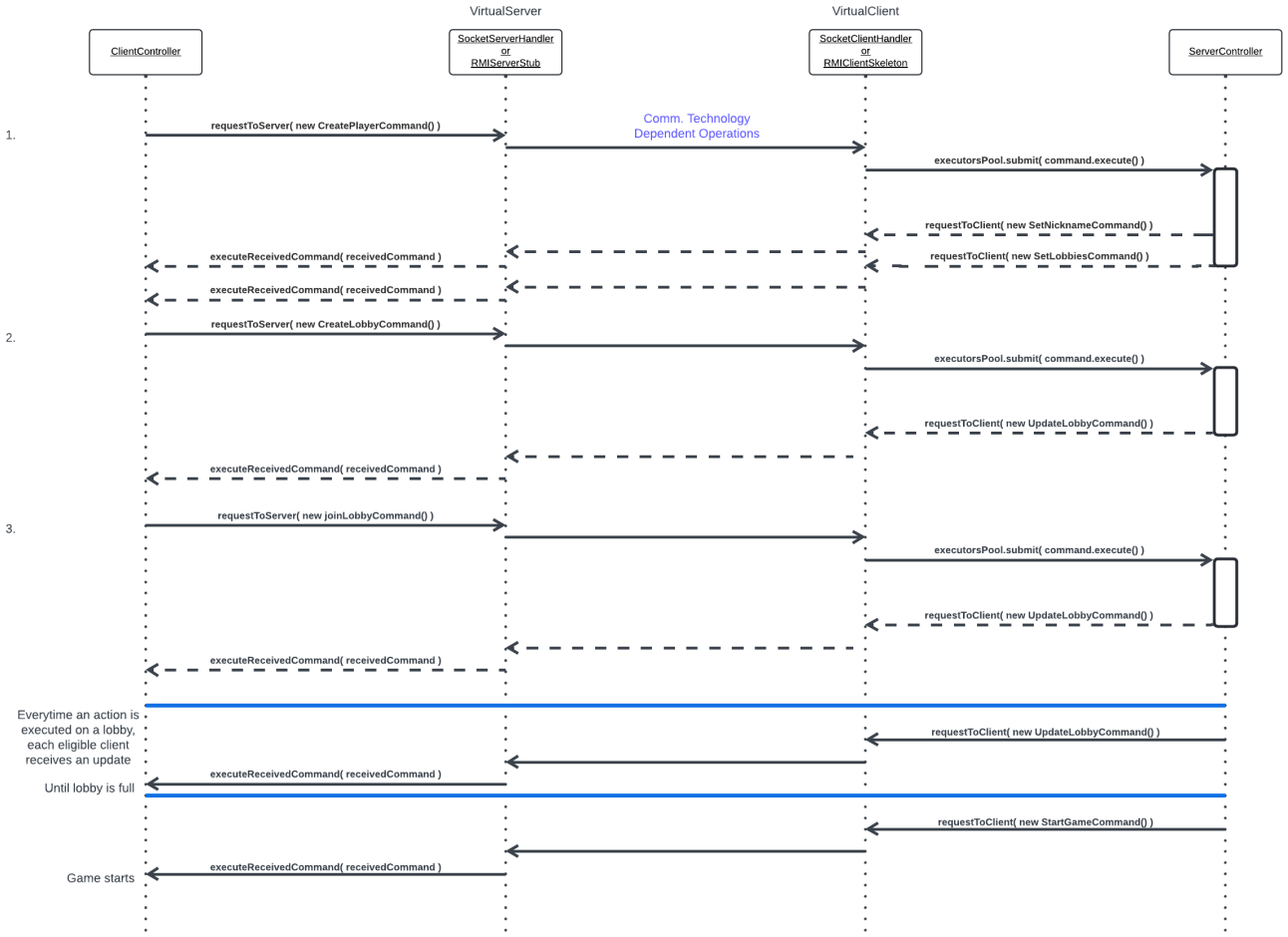
3 Resilienza alle disconnessioni

Per l'implementazione della funzionalità avanzata *Resilienza alle disconnessioni*, il progetto introduce un meccanismo di keepAlive, il quale è in grado di mantenere registrati solamente i client ancora connessi con il server. Il keepAliveCommand è un segnale periodico inviato dal client al server per notificarne la presenza. Tale scambio evita che il server consideri erroneamente il client come disconnesso.

Ogni qualvolta il server riceve un comando da un client, il timer di timeout a lui associato viene ripristinato. Se non vengono ricevuti comandi entro un certo intervallo di tempo, il server può assumere che il client sia disconnesso e quindi chiudere la connessione e rimuoverlo dalla lista dei client attivi. Il KeepAliveCommand è pensato per mantenere la connessione durante i periodi di tempo in cui non vi sono altri comandi da scambiare.

4 Sequence Diagram

Di seguito è rappresentato un Sequence Diagram che raffigura alcuni possibili scenari di interazione tra client e server.



4.1 Scenario 1: Registrazione del giocatore

- Il client invia il comando CreatePlayerCommand al server, chiedendogli di essere registrato.
- Il server riceve il comando, lo esegue tramite la propria thread pool di executors, e invia al client i comandi di risposta SetNicknameCommand (per confermare il nickname ricevuto) e SetLobbiesCommand (per fornire al client la lista delle lobby attualmente presenti).

4.2 Scenario 2: Creazione di una lobby

- Il client decide di creare una lobby e invia un CreateLobbyCommand.
- Il server processa il comando e invia un UpdateLobbyCommand a tutti i client non in partita per aggiornare la lista delle lobby attive e non piene.

4.3 Scenario 3: Entrata in una lobby già esistente

- Il client invia un JoinLobbyCommand per per chiedere di potersi unire a una lobby già esistente.
- Il server esegue il comando e invia a tutti i client non in partita l'aggiornamento della lobby tramite un UpdateLobbyCommand. Quando il numero di giocatori richiesto viene raggiunto, il server invia ai client coinvolti uno StartGameCommand in modo da inserirli in una partita, mentre invia a tutti i client non in partita la richiesta di cancellare quella lobby dalle loro liste, tramite un altro UpdateLobbyCommand.