ESEMPIO DI PROGETTO MVC CON INTERAZIONI CLIENT-SERVER

Arianna Dragoni

Codice riadattato da un precedente esempio di Luca De Martini.

Lo scopo di questo esempio è illustrare il flusso delle interazioni tra client e server in un contesto MVC.

- L'organizzazione adottata nell'esempio non è vincolante. Pertanto, non è necessario seguire rigidamente la struttura proposta, poiché alcune assunzioni potrebbero risultare eccessivamente semplificate per il vostro progetto.
- Le eccezioni in questo esempio sono gestite tramite il meccanismo di throws, poiché la gestione dettagliata delle eccezioni è al di fuori dello scopo di questo esempio. Nel vostro progetto, è necessario implementare una gestione personalizzata delle eccezioni utilizzando i blocchi try-catch e una logica appropriata.
- Molte delle strategie implementative qui utilizzate non sono vincolanti: scegliete quella che meglio si adatta al vostro progetto, purché rispettosa dei paradigmi corretti.
- Alcune strategie implementate non sono considerate best practice. In questi casi, vi invitiamo a riflettere sulla soluzione più adeguata per il vostro progetto. L'inclusione di tali strategie nell'esempio ha l'obiettivo di stimolare la riflessione e di lasciare a voi la libertà di scegliere l'approccio migliore, senza fornire una risposta univoca, poiché esistono molteplici soluzioni corrette.

Il codice implementa un'applicazione client-server in cui il server gestisce un contatore, incrementandolo ogni volta che un client invia un numero intero. L'implementazione segue il pattern **Model-View-Controller (MVC)**.

Il **Model** rappresenta lo stato dell'applicazione tramite un valore intero, che può essere incrementato o resettato. Esso fornisce i metodi per la modifica dello stato.

Il **Controller** gestisce la logica applicativa, inoltrando le richieste di modifica dello stato ai metodi definiti nel Model. Poiché si tratta di un'applicazione **multi-client**, più client possono interagire simultaneamente con il Model, modificandone lo stato in parallelo. Per garantire la corretta gestione della concorrenza, i metodi del Controller sono **sincronizzati**. *(Nota: in questo esempio, la sincronizzazione avviene sull'intero Model, essendo costituito da un singolo valore intero. Nel vostro progetto, sarà opportuno valutare quali elementi sincronizzare per ridurre i blocchi e ottimizzare le prestazioni del sistema).*

La **View** è minimale e consiste in un'interfaccia a riga di comando (**CLI**) che consente all'utente di inserire un numero e visualizzare lo stato attuale del contatore. Per questa ragione, non è stato introdotto un package dedicato alla View: l'interfaccia è interamente implementata all'interno del client. Tuttavia, nel vostro progetto potrebbe essere necessario strutturare la View in un package separato e definire interfacce specifiche per supportare sia un'implementazione CLI che una GUI.

Il Model e, data la semplicità dell'esempio, l'intero Controller risiedono sul server, mentre la View è lato client. Nel vostro progetto, potrebbe essere utile spostare parte della logica del Controller sul client per ridurre il numero di comunicazioni con il server. Valutate attentamente in quali casi sia necessario coinvolgere il server e quando, invece, sia più efficiente gestire l'elaborazione direttamente lato client.

L'implementazione dell'architettura client-server dovrà essere implementata con **RMI** e/o **Socket.**

RMI: Invocazione di Metodi Remoti

Con RMI (Remote Method Invocation**)**, la comunicazione tra client e server avviene tramite l'invocazione di **metodi remoti**.

RMI fornisce un'astrazione che consente al client di invocare metodi come se fossero locali, anche se, in realtà, si tratta di metodi eseguiti su un **oggetto remoto**, cioè un oggetto che risiede su un altro processo o macchina. Sebbene il client interagisca con l'oggetto remoto come se fosse un oggetto locale, l'esecuzione dei metodi avviene effettivamente sul **server**, che gestisce la logica e restituisce i risultati al client.

RMI si occupa della gestione della comunicazione tra client e server, trasmettendo i dati necessari per l'invocazione del metodo remoto e il ritorno del risultato. Quando un client chiama un metodo su un oggetto remoto, non deve preoccuparsi dei dettagli del trasporto dei dati; il framework RMI gestisce automaticamente la comunicazione.

RMI è nativamente sincrono: Il client invoca il metodo e resta in attesa della risposta dal server. In risposta, il server può a sua volta invocare metodi remoti sul client, se necessario. Lo stesso vale lato server: quando il server invoca un

metodo remoto sul client, attende che il client abbia finito l'esecuzione e abbia risposto. Nel vostro progetto, dovrete gestire questo fatto cercando di rallentare l'intera applicazione il meno possibile (ad esempio, implementate un modello asincrono in cui il chi riceve un'invocazione remota risponde immediatamente con un ack prima di iniziare l'elaborazione effettiva del metodo).

Il punto centrale di RMI è l'astrazione che permette di trattare gli oggetti remoti come se fossero locali. Per rendere tutto ciò possibile, RMI utilizza un **registry**, che funge da repository centrale per la gestione e localizzazione degli oggetti remoti. Il registry contiene le informazioni necessarie affinché il client possa trovare gli oggetti remoti esposti dal server. Quando un client vuole interagire con un oggetto remoto, interroga il registry per ottenere un riferimento all'oggetto. Se il server ha già registrato l'oggetto remoto, il registry restituisce uno **stub**, che è un proxy dell'oggetto remoto. Lo stub è un oggetto locale che agisce come rappresentante del server lato client. Quando il client invoca un metodo sull'oggetto locale (stub), l'invocazione non viene eseguita localmente, ma viene inoltrata al vero oggetto remoto sul server. Una volta che il server riceve l'invocazione e risponde, il risultato viene restituito al client tramite lo stub. Quindi, sebbene l'oggetto remoto sia fisicamente sul server, il client lo tratta come se fosse un oggetto locale.
Lo stesso principio si applica per le comunicazioni dal server verso il client. Tuttavia, in questo caso il client non si registra nel registry. Al contrario, si connette al server utilizzando lo stub ottenuto tramite il registry. Quando il server desidera inviare aggiornamenti o invocare metodi sul client, invoca i metodi remoti sull'oggetto stub del client, che è stato precedentemente registrato nel server.

Implementazione di RMI
Nel package `rmi` trovate l'implementazione della comunicazione client-server utilizzando RMI. Il package contiene:

- Interfacce per gli oggetti remoti che verranno utilizzati da client e server per comunicare. Per questo, queste interfacce estendono l'interfaccia `Remote` di RMI.
    - **VirtualServer**: Interfaccia che definisce i metodi del server che devono essere resi accessibili ai client per potersi connettere al server ed interagire con la logica del server (controller).
    - **VirtualView**: Interfaccia che definisce i metodi utilizzati dal server per notificare i cambiamenti di stato ai client.
- Classi lato server e lato client che rappresentano rispettivamente la logica del server e del client, e comunicano attraverso la rete. In questo semplice esempio abbiamo una sola classe per il server e una per il client, ma nel vostro progetto potreste aver bisogno di più classi.
    - **RMIServer**: gestisce le operazioni sullo stato del Model. Ha bisogno di un riferimento al controller per gestire la logica di gioco e invocare tali operazioni, e di un riferimento ad una lista con gli stub dei client connessi per notificargli i cambiamenti di stato.

      Il `main` del server crea ed esporta nel registro l'oggetto remoto corrispondente al server stesso. Gli assegna un nome canonico e lo inizializza su una porta. Per connettersi a quel server, i client dovranno usare il nome canonico e la porta per ottenere dal registro quell'oggetto remoto da utilizzare per le invocazioni.

      Inoltre, il server implementa l'interfaccia VirtualServer: implementa i metodi da eseguire quando vengono invocati dai client. Nell'implementare questi metodi, attenzione alla sincronizzazione! RMI associa automaticamente un thread ad ogni invocazione di un metodo; quindi, se i metodi non sono correttamente sincronizzati (o nel server o nel controller) potrebbero verificarsi delle data race. Ragionate sul corretto utilizzo delle strategie di sincronizzazione per sincronizzare quando necessario (senza esagerare).

      Infine, il server deve inviare aggiornamenti ai client quando lo stato del model cambia. Il codice attuale mostra un approccio che invia questi aggiornamenti in modo sincrono a tutti i client. Questa soluzione non è ottimale, perché causa un blocco del server fino a quando tutti i client non ricevono e completano l'invocazione del metodo. Nel vostro progetto dovete trovare una soluzione per gestire ed evitare questi blocchi (ad esempio con delle blocking queue: create un thread separato per l'invio degli aggiornamenti, che rimane in attesa di nuovi aggiornamenti – inseriti in una coda – e li invia non appena sono disponibili).
    - **RMIClient**: si occupa delle interazioni con l'utente e di mostrare lo stato corrente dell'applicazione all'utente. Ha bisogno di un riferimento allo stub del server per poter mandare gli input dell'utente.

      Implementa l'interfaccia remota VirtualView: implementa i metodi da eseguire quando il client riceve delle notifiche di cambio di stato dal server. Di conseguenza, questa classe rappresenta lo stub del client nel server, e deve poter passare attraverso la rete. Per farlo, implementa l'interfaccia di RMI `UnicastRemoteObject` (la stessa utilizzata per esportare l'oggetto remoto del server).

      Nel `main`, il client esegue il lookup nel registry per ottenere l'oggetto remoto del server al quale si vuole connettere. Per farlo, utilizza il nome canonico, l'indirizzo ip e la porta del server. Successivamente, il client crea una nuova istanza di sé stesso e avvia la logica nel metodo run.

Nel metodo `run`, per prima cosa il client si connette al server invocando il metodo `connect` del server. In altre parole, invia al server il proprio stub, informandolo che desidera ricevere gli aggiornamenti.

Anche in questo caso, è importante prestare attenzione alla sincronizzazione: poiché il client è un `UnicastRemoteObject`, può ricevere messaggi dal server in un thread separato in qualsiasi momento. Questo vuol dire che mentre con il metodo `run` sta aspettando un input dall'utente, potrebbe arrivare un aggiornamento (invocazione del metodo `showUpdate` con un altro thread) causando data race che interferiscono con l'input che l'utente sta fornendo.

Socket

Con i socket, la comunicazione tra client e server avviene attraverso lo scambio diretto di **messaggi** su una connessione di rete. I socket forniscono un meccanismo per stabilire una connessione tra due processi, anche su macchine diverse, consentendo loro di inviare e ricevere dati in modo bidirezionale. A differenza di RMI, che offre un'astrazione basata su metodi remoti, i socket si basano su un livello più basso di comunicazione: il client e il server devono gestire esplicitamente l'invio e la ricezione dei messaggi serializzati.

Il server crea un socket di ascolto su una porta specifica e attende le connessioni dai client. Il client deve conoscere l'indirizzo IP e la porta del server per avviare la connessione. Quando un client si connette, il server accetta la connessione e crea un socket dedicato per comunicare con quel client. A questo punto, entrambi possono inviare e ricevere messaggi attraverso i loro rispettivi socket.

Il protocollo di comunicazione tra client e server deve essere ben definito per interpretare correttamente i dati scambiati. Nello specifico, a differenza di RMI, dove il client interagisce con oggetti remoti attraverso metodi remoti, nei socket il client e il server devono gestire direttamente il formato e la struttura dei messaggi, oltre a sincronizzare l'invio e la ricezione per evitare problemi come deadlock o perdita di dati. Nel vostro progetto, dovrete scegliere il protocollo di serializzazione messaggi (XML, JSON, oggetti serializzati java, …), e gestire manualmente nel server la deserializzazione in base al tipo di messaggio ricevuto.

Inoltre, a differenza di RMI la comunicazione con i socket è asincrona: chi invia il messaggio non si mette automaticamente in attesa della risposta, ma deve implementare una logica esplicita per gestire la ricezione e l'elaborazione dei dati. Questo implica un'adeguata gestione della concorrenza tramite thread separati per l'invio e la ricezione, evitando situazioni di race conditions, deadlocks, …

Implementazione di Socket

Nota: per questo esempio, i messaggi scambiati sono stringhe, e si utilizzano gli oggetti `BufferedReader` *e* `PrintWriter` *per lo scambio. Nel vostro progetto, potreste aver bisogno di utilizzare tipi più complessi per il vostro protocollo di comunicazione.*

Nel package `socket` trovate l'implementazione della comunicazione client-server utilizzando i Socket. Il package contiene:

- Interfacce che verranno utilizzate da client e server per invocare i metodi dopo aver ricevuto un messaggio e/o per inviare messaggi. In questo caso, NON sono i metodi definiti in queste interfacce a passare attraverso la rete.
  - **VirtualServer**: Interfaccia che definisce i metodi del server che devono essere resi accessibili ai client per poter interagire con la logica del server (controller). Questa interfaccia verrà implementata per scrivere sul canale di output lato client il messaggio da mandare al server (che porterà ad un cambiamento di stato nel model).
  - **VirtualView**: Interfaccia che definisce i metodi utilizzati dal server per notificare i cambiamenti di stato ai client. Questa interfaccia verrà implementata per scrivere sul canale di output del server i messaggi da mandare al client (che notificano cambiamenti di stato nel modello). Verrà anche implementata lato client per mostrare il nuovo stato del modello all'utente.
- Classi che fungono da proxy per la comunicazione tra client e server. Sono l'analogo degli stub di RMI: implementano i metodi utilizzati per comunicare. La differenza è che, mentre con RMI questi sono i metodi remoti che attraversano la rete, con socket questi metodi creano il messaggio serializzato e lo inviano lungo il canale di comunicazione
  - **SocketClientHandler**: Funge da proxy del client lato server; quando il server utilizza un oggetto di questa classe, ha la percezione di comunicare direttamente con il client senza passare per la rete. Ha bisogno di un riferimento al controller, per invocare i metodi di modifica dello stato, di un riferimento al server per chiamare i metodi per notificare cambiamenti al client, e un riferimento ai canali di lettura e scrittura sulla rete.

    Contiene la logica per leggere e deserializzare i messaggi che arrivano dal client. Dopo aver deserializzato il messaggio, chiama il metodo del controller per modificare lo stato e notifica ai client l'update (Attenzione alla

sincronizzazione!!). La deserializzazione in questo progetto è fatta con uno switch per identificare il tipo di messaggio e chiamare il corretto metodo del controller per modificare lo stato. Questo non è un buon pattern di programmazione. Nel vostro progetto, dovrete trovare un modo per evitare lo switch creando un protocollo di serializzazione che sfrutta il paradigma object oriented.

Inoltre, la classe implementa i metodi definiti in VirtualView per inviare al client update sullo stato del modello: scrive i messaggi di modifica dello stato nel canale di comunicazione.

- **VirtualSocketServer:** Astrae la comunicazione del client verso il server. Quando il client chiama questi metodi, ha la percezione di avere il server in locale, ma in realtà questi metodi mandano dei messaggi al server descrivendo gli input del client.

  Implementa i metodi di VirtualServer per inviare al server l'input del client che porterà a modifiche dello stato. Nell'implementazione, viene creato e scritto il messaggio nel canale di output del client.

- Classi lato server e lato client che rappresentano rispettivamente la logica del server e del client, e comunicano attraverso la rete. Come prima, in questo semplice esempio abbiamo una sola classe per il server e una per il client, ma nel vostro progetto potreste aver bisogno di più classi.

  - **SocketServer**: gestisce le operazioni sullo stato del Model. Ha bisogno di un riferimento al controller per gestire la logica di gioco e invocare tali operazioni. Inoltre, deve rimanere in ascolto delle connessioni in arrivo, e quindi si tiene un riferimento ad una istanza di `ServerSocket`, che funge da socket di ascolto e accetta le connessioni dai client. Infine, mantiene una lista di stub con i client che si sono connessi. In particolare, ogni volta che un client si connette, il `SocketServer` crea una istanza di SocketClientHandler e la salva nella lista per poter notificare gli aggiornamenti (metodo `broadcastUpdate`, invocato a seguito di modifiche dello stato).

    Il `main` del server inizializza il socket di ascolto passandogli la porta sul quale vuole ascoltare le nuove connessioni. Poi fa partire il server, passandogli il socket per ascoltare le nuove connessioni e il controller, che inizializza. Il controller viene inizializzato per ogni server perché nel caso di più partite ogni partita avrà il proprio controller, che gestirà il proprio stato del model.

    Con l'uso di socket, è necessario gestire esplicitamente l'ascolto di nuovi client. Questo avviene nel metodo `run`, dove definiamo il comportamento del server in risposta a una nuova connessione. Quando un client si collega, il server utilizza il suo `ServerSocket` per accettare la connessione, salvandola in una variabile di tipo Socket. In questa implementazione, per leggere e scrivere sul canale utilizziamo un `InputStreamReader` e un `OutputStreamWriter`. (nella vostra implementazione potreste aver bisogno di altri tipi di oggetto, in base al vostro protocollo di comunicazione).Per interagire con il client appena connesso, utilizziamo una istanza di `SocketClientHandler` (che salviamo nella lista di client). Per gestire la ricezione e l'elaborazione dei messaggi in arrivo (metodo `runVirtualView` del proxy) senza bloccare il thread principale del server, avviamo esplicitamente un nuovo thread.

  - **SocketClient**: gestisce le interazioni con l'utente, mostrando lo stato attuale del model e ricevendo gli input dall'utente. Mantiene un riferimento al canale dove arrivano gli input dall'utente e un riferimento al proxy del server attraverso il quale invia i messaggi contenenti gli input ricevuti.

    Implementa i metodi dell'interfaccia VirtualView per visualizzare lo stato del model all'utente. Quando implementate questi metodi, attenzione a sincronizzare correttamente per evitare le data race!

    Il `main` del client si connette al socket del server. Per farlo, utilizza l'host e la porta del server ed apre una socket passando host e porta. Poi istanzia il client, passando i canali di input e output che serviranno per scrivere e leggere messaggi, e fa partire la logica del client.

    La logica del client è implementata nel metodo run. La logica del client è implementata nel metodo `run`. A differenza di RMI, dove l'avvio del thread era gestito automaticamente, con i socket dobbiamo gestirlo esplicitamente. Quindi, all'avvio del client, è necessario creare un thread dedicato alla gestione della comunicazione con il server. Il thread ha il compito di ascoltare i messaggi in arrivo e, in base alla loro tipologia, eseguire le funzioni appropriate per aggiornare e mostrare le informazioni all'utente. Questa logica è definita nel metodo `runVirtualServer`. Anche in questo caso, dovrete trovare un modo per evitare lo switch creando un protocollo di serializzazione che sfrutta il paradigma object oriented. Infine, il metodo run mostra l'interfaccia (in questo caso la CLI) all'utente (anche qui, attenzione alla sincronizzazione!).