Mattia Sorgato, matricola 1004404

Relazione di Progetto di Programmazione Concorrente e Distribuita

Anno didattico 2014/2015

Terza consegna - Parte distribuita

Logica di Distribuzione

Nella realizzazione di questa parte finale del progetto e nell'utilizzo della tecnologia *rmi*, mi sono affidato ai concetti visti a lezione. Fondamentalmente, ho realizzato la classica infrastruttura adottata nel libro di testo, ovvero un'interfaccia condivisa tra Client e Server e la sua implementazione presente solamente nel Server. Quest'ultimo successivamente pubblicizza l'oggetto remoto istanziato per renderlo disponibile al Client, il quale otterrà il riferimento remoto tramite il registro *rmi* e invocherà i metodi necessari.

Package remote

Questo package contiene i sottopackage **client** e **server**, in aggiunta all'interfaccia remota **RemoteSolver** che sarà condivisa da entrambi.

Interfaccia remote.RemoteSolver

Questa interfaccia estende l'interfaccia **Remote** e contiene il metodo *sortPuzzle(List<Piece>)* che verrà invocato remotamente dal Client. Sarà poi l'implementazione di tale interfaccia, presente nel Server, che realizzerà effettivamente l'ordinamento del puzzle presente nel Client, per poi rispedire il puzzle ordinato al Client chiamante.

Package remote.server

Questo package contiene due classi: **ConcreteRemoteSolver** e **PuzzleSolverServer**. Il package contiene le funzionalità del server, ovvero l'implementazione dell'interfaccia remota e il gestore del Server stesso.

Classe remote.server.ConcreteRemoteSolver

Questa classe estende **UnicastRemoteObject** e implementa **RemoteSolver** con il suo metodo *sortPuzzle(List<Piece>)*. E' la classe che effettivamente esegue l'ordinamento di una lista di pezzi disordinati, eseguito nel Server e fornendo il puzzle ordinato da ritornare al Client. La sua implementazione è semplice: per ogni richiesta di ordinamento viene creata una nuova istanza di **ConcurrentSort**, la classe che implementa l'algoritmo di ordinamento concorrente che ho sviluppato durante la seconda parte del progetto. Creando localmente una nuova istanza per ogni richiesta, si evita il rischio di *race condition* e si possono servire tranquillamente più Client contemporaneamente.
E' ovviamente presente il costruttore obbligatorio che può sollevare **RemoteException**, sebbene abbia corpo vuoto. Il costruttore è stato marcato *protected* proprio per evitare che la costruzione di un'istanza di **ConcreteRemoteSolver** possa avvenire solamente all'interno del package **remote.server**.

Classe remote.server.PuzzleSolverServer

E' la classe che gestisce il Server in questa terza parte del progetto. Questa classe contiene il metodo *main* che esegue la pubblicazione dell'oggetto remoto utile al Client. L'esecuzione del *main* attraversa due fasi principali. Nella prima fase avviene un controllo preliminare dell'input passato alla funzione, per assicurarsi che l'argomento sia effettivamente uno solo (come espresso nella specifica) e che sia utilizzabile per la pubblicazione nel registro *rmi*. Nella seconda fase, invece, avviene la creazione dell'istanza di

ConcreteRemoteSolver e la sua pubblicazione del riferimento nel suddetto registro. Sia la creazione dell'istanza che la pubblicazione tramite il metodo *Naming.rebind(String, Remote)* sono rinchiuse nei necessari blocchi *try/catch*, per individuare eventuali problemi nella rete o con il registro *rmi.* Nel caso vengano rilevati problemi, l'utente viene notificato da messaggi di errore nello standard output del Server. Una volta avvenuta la pubblicazione del riferimento dell'oggetto remoto, il Client sarà in grado di ottenere tale riferimento per invocare il metodo remoto desiderato.

Package remote.client

Questo package contiene la classe **PuzzleSolverClient**, la quale gestisce le richieste del Client da inoltrare all'oggetto remoto contenuto nel Server. Questa classe contiene la gestione della lettura e scrittura dei file di input/output, oltre che ad ottenere il riferimento remoto per invocare il metodo di ordinamento del puzzle.
Sono stati quindi implementati i metodi di lettura e scrittura da e su file, oltre che al *main* necessario all'esecuzione del Client.
I metodi qui presenti sono:

- *readContent(String)*, il quale esegue la lettura (se possibile) del file archiviato nel percorso identificato nell'argomento, esegue un controllo preliminare sui pezzi contenuti nel file ed infine ritorna al chiamante la lista di **Piece** risultante;
- *writeContent(String, String)*, il metodo che esegue l'operazione duale, ovvero la scrittura su file del contenuto passato come secondo argomento.
- *puzzleStringBuilder(Piece[][])*, che prende come argomento il puzzle ordinato, crea e prepara la Stringa che verrà poi scritta in output, formattando il testo nello schema fornito nella specifica della prima parte del progetto.

L'ultimo metodo presente è il *main* di **PuzzleSolverClient**, il quale esegue diverse operazioni.
La prima operazione, come nel *main* del Server, è di controllo degli argomenti di invocazione, che devono essere obbligatoriamente tre: *nome_file_input, nome_file_output* e *nome_del_server*.
Esegue quindi la lettura del file di input tramite il metodo *readContent(String)*, salvando localmente i pezzi da mandare poi al Server per l'ordinamento. Esegue quindi un controllo sommario degli indici presenti nel puzzle disordinato, per poi passare all'ottenimento del riferimento remoto.
A questo punto avviene la consultazione del registro *rmi* per ottenere il riferimento remoto tramite il metodo *Naming.lookup(String)*, a cui verrà passato il nome del Server fornito dall'argomento di invocazione del *Main*. Questa operazione può lanciare diversi tipi di eccezione, a seconda del problema che può presentarsi. Per questo il *lookup* deve essere provvisto di giuste clausole *try/catch* per gestire i vari casi ed informare l'utente se uno di questi si presenta. A seconda che il Server non sia raggiungibile, che il nome fornito non sia compatibile o che al nome utilizzato per il *lookup* non sia associato nessun oggetto remoto, all'utente viene notificato ogni caso con una stampa sullo standard output del Client.
Se l'ottenimento del riferimento remoto va a buon fine, viene richiamato il metodo remoto per l'ordinamento, anch'esso circondato da un blocco *try/catch*. Questo è indispensabile, in quanto, sebbene si sia ottenuto il riferimento remoto all'oggetto pubblicizzato, per un qualche motivo il Server potrebbe non essere più raggiungibile, per cause dipendenti dalla rete o da un problema interno al Server, come un crash dell'applicazione. Anche in questo caso l'utente viene notificato se questo problema si presenta prima dell'invocazione del metodo o durante lo stesso.
A questo punto, se la chiamata del metodo remoto ha avuto successo, il Server avrà ritornato il puzzle ordinato, il quale verrà passato a *puzzleStringBuilder(Piece[][])* per ottenere la Stringa da scrivere su file con il metodo *writeContent(String, String)*.

Cambiamenti dalle versioni precedenti

Rispetto alle versioni precedenti, visto che il package **remote** è stato introdotto *ex novo*, gli unici cambiamenti adottati sono stati l'adattamento alla versione distribuita del codice già presente. La differenza che balza subito all'occhio è che ora la classe **Piece** implementa l'interfaccia **Serializable**, per consentire la trasmissione su rete della classe che rappresenta un pezzo del puzzle.
Inoltre, nella stesura della classe **PuzzleSolverClient** mi sono trovato a riscrivere molto del codice già presente in **PuzzleSolver**, la classe principale delle due consegne precedenti. Ne posso dedurre quindi che forse avrei potuto spostare l'implementazione di tali metodi in una classe a parte, invocabile poi anche da **PuzzleSolverClient** per aumentare il riutilizzo del codice.
Per una questione di semplicità, ho abbandonato i costrutti di Java7 presenti nella gestione degli input per passare a quelli più familiari di Java6, anche per una maggiore compatibilità con i sistemi del laboratorio.

Bash

Nel root del mio progetto ho creato i due file *bash* richiesti dalla specifica di questa parte del progetto. Diversamente dalle consegne precedenti, questi non richiamano più il comando make, anche se questo continua ad essere presente e richiamabile, bensì eseguono le chiamate dei *main* associati con gli argomenti passati al *bash* stesso. Nel caso di errata invocazione dello script, verrà notificato sullo standard output l'errore presente per correggere l'invocazione.

Nota: Se viene eseguito il comando **make***,* i compilati vengono posizionati ordinatamente nella cartella **bin.**

L'esecuzione dei due *bash* richiede comunque che sia attivo il registro *rmi* per la pubblicazione e l'ottenimento dei riferimenti remoti. Il comando va lanciato all'interno della cartella **bin**, per far sì che il *classpath* usato nella compilazione sia visibile al registro stesso.

Se non si volessero utilizzare i *bash* per eseguire il progetto, una volta compilati i file necessari, occorre posizionarsi nella cartella dei compilati (che nel caso del *bash* è **bin**) e chiamare il comando

java -cp . puzzlesolver.remote.server.PuzzleSolverServer nome_del_server
e successivamente
java -cp . puzzlesolver.remote.client.PuzzleSolverClient path_input path_output nome_del_server

Note

Sono state commentate le stampe dei vari Thread di ordinamento del puzzle immesse nella consegna precedente. L'unica stampa che viene eseguita durante il corretto funzionamento, è la stampa di conferma dell'avvenuta pubblicazione e avvio del Server.