Sebastiano Valle - 1050123

Risolutore di puzzle – Parte 3
Programmazione concorrente e distribuita
Progetto A.A. 2014/2015

1 Logica di comunicazione client-server

Il precedente programma per la risoluzione del puzzle è diventato un programma che, dato un client che fornisce al server un insieme di stringhe contenenti i tasselli del puzzle da riordinare.

1.1 Client

A tal fine, il client inizialmente legge il file di input frammentando le righe rappresentanti i tasselli, dopodichè chiede ad un registro RMI condiviso con il server un oggetto di tipo *GruppoOrdinabile* che verrà riempito con l'insieme di righe appena ottenuto in input tramite una richiesta (**RMI**).

Successivamente, dopo aver richiesto l'ordinamento del mucchio di pezzi, il client richiede l'output da stampare su file corrispondente alla soluzione del puzzle (sempre grazie ad un meccanismo di **RMI**).

1.2 Server

Il server in questo programma si limita a registrare un oggetto di tipo *GruppoOrdinabile* che verrà cercato dal client nel registro RMI condiviso con il client. Dopo questa operazione, il server non svolge più alcuna operazione.

2 Robustezza del programma

2.1 Interruzione improvvisa del client

Se il client viene interrotto, necessiterà di essere riavviato dall'esterno e richiederà la risoluzione del puzzle dall'inizio.

2.2 Interruzione improvvisa del server o del canale di comunicazione

In caso di interruzione improvvisa del server, si provvede a notificare il malfunzionamento segnalando in che punto il programma si è arrestato.

Dopo aver rilevato l'errore, il client proverà ad eseguire ogni richiesta sull'oggetto remoto (riempimento, ordinamento e richiesta dell'output) fino ad un massimo di sette volte per richiesta, aspettando cinque secondi tra una richiesta e la sua successiva.

Se tutte le sette richieste per la stessa azione dovessero fallire, allora il client non eseguirà nemmeno

le seguenti azioni previste e segnalerà l'errore in console.

2.3 Anomalie nel registro RMI

Nel caso in cui venisse fornito un indirizzo sbagliato al client o se l'oggetto di tipo *Puzzle* dovesse essere ancora registrato nel registro RMI, l'utente verrà informato con un messaggio di errore e dovrà riavviare il client per ritentare. Allo stesso modo, il programma client verrà interrotto subito se durante il **lookup** viene interrotto improvvisamente il canale di comunicazione.

3 Variazioni

Le variazioni apportate al progetto sono le seguenti:

1. sono stati modificati i makefile nel progetto poiché potessero funzionare correttamente, così come gli script sh;
2. è stata introdotta una classe *PuzzleSolverClient*;
3. è stata introdotta una classe *PuzzleSolverServer*;
4. è stata introdotta una classe *RMIBinder*;
5. le interfacce *Gruppo* e *GruppoOrdinabile* sono state modificate;
6. le classi *Heap* e *Puzzle* sono state modificate;
7. la classe *Driver* (di utilità per le attività di test) è stata modificata.

3.1 Makefile e script sh

Ora il progetto è avviabile nelle modalità indicate nella sezione 4, ovvero come richiesto da specifiche.

3.2 PuzzleSolverClient (Classe)

La classe *PuzzleSolverClient* rappresenta un client che dispone del file di input e che vuole risolvere il puzzle contenuto in esso.

Inizialmente legge dal file di input l'insieme dei tasselli da riordinare e richiede ad un oggetto di tipo *GruppoOrdinabile* di essere riempito con questi tasselli.

Dopodichè, esegue la richiesta di ordinamento e scrive nel file di output il risultato del riordinamento, richiedendolo sempre tramite **RMI** al *GruppoOrdinabile* prima individuato.

3.3 PuzzleSolverServer (Classe)

La classe *PuzzleSolverServer* rappresenta un server che crea un *GruppoOrdinabile* e lo registra con **RMI** affinché questo sia a disposizione di un client per risolvere un puzzle.

3.4 RMIBinder (Classe)

La classe *RMIBinder* racchiude dei metodi di comune utilità a tutte le classi che fanno uso di **RMI**: con il metodo *richiedi()*, sia il server che il client possono effettuare una richiesta (rispettivamente

di **rebind** e **lookup**) per un oggetto riferito tramite **RMI**.

È stata presa questa decisione poiché si voleva evitare ripetizione di codice, in particolare per la gestione delle eccezioni comuni ad entrambe le classi server e client.

3.5 Gruppo, GruppoOrdinabile (Interfacce)

Queste interfacce, utilizzate per identificare i tipi riferiti comunemente tra client e server (è stata lasciata la possibilità al client di potersi riferire ad un gruppo non ordinabile), sono state marcate come remote. Per raggiungere tale risultato, sono state rese estensioni dell'interfaccia *java.rmi.Remote* ed è stato segnalato che i metodi offerti possono lanciare eccezioni di tipo **RemoteException**. Inoltre sono state modificate delle firme per consentire al client di passare una lista di stringhe per riempire il *GruppoOrdinabile* e per far sì che il client ricevesse la stringa da stampare sull'oggetto di output anziché lasciare questa responsabilità al server nel metodo *outcome()* (che nella parte di progetto precedente era *write(outputPath : Path)*).

3.6 Heap, Puzzle (Classi)

Le due classi sono state modificate coerentemente con le modifiche alle proprie interfacce, ovvero cambiando la firma dei due metodi modificati e aggiungendo la possibilità di lanciare eccezioni di tipo **RemoteException** per i metodi presenti nelle interfacce.

3.6.1 Modifiche relative solamente a Puzzle

Sono state aggiunte le clausole **catch** per i metodi invocati sul campo dati di tipo *Gruppo*, così da gestire le eccezioni di tipo remoto. Oltre a questo, *Puzzle* è diventata una classe estensione di **UnicastRemoteObject**, così da poter essere registrata e riferita tramite **RMI**.

3.7 Driver (Classe)

La classe *Driver* è stata modificata: ora avvia inizialmente un oggetto di tipo *PuzzleSolverServer* e successivamente un oggetto di tipo *PuzzleSolverClient*, ricevendo come input gli stessi parametri di un *PuzzleSolverClient*.

4 Note

4.1 Gestione della concorrenza tra client

Per semplicità si è supposto che solo un client alla volta richiedesse la risoluzione di un puzzle.

Se si fossero volute gestire più richieste da parte dei client in modo concorrente, sarebbe bastato aggregare i metodi *fill(rows : ArrayList<String>),sort()* e *outcome()* in un unico metodo *solve()* che restituiva la soluzione del puzzle.

Naturalmente sarebbero state necessarie due variazioni:

- Invocando il metodo *solve()* bisognerebbe prendere il lock sull'intero oggetto di tipo *Puzzle;*

- Porre a **null** gli elementi della prima riga e colonna della matrice dei tasselli in *Puzzle* prima di cominciare il metodo *sort().*

Non sarebbe stato necessario svuotare l'*Heap* di tasselli poiché questa operazione viene eseguita dal metodo *sort()*, con le estrazioni dei pezzi man mano che l'ordinamento avanza.

4.2 Avvio del programma

L'avvio del registro RMI è già contenuto in *puzzlesolverserver.sh*.

Per questo motivo, per avviare il programma è sufficiente digitare nel seguente ordine:

bash puzzlesolverserver.sh `nome_server`

bash puzzlesolverclient.sh `nome_input_file nome_output_file nome_server`

Come da specifiche.