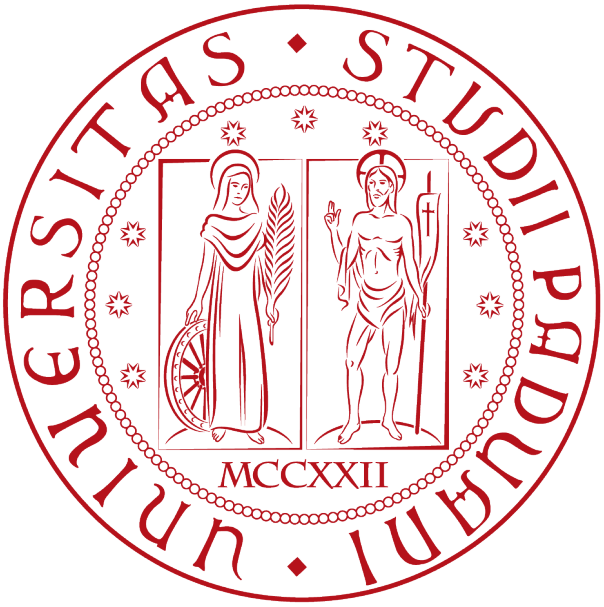

Risolutore di Puzzle

Parte 3

Stefano Munari mat. 1031243

Sommario:
Relazione del progetto di Programmazione concorrente e distribuita AA 2014-15

Indice

1 Note preliminari

Si è scelto di utilizzare alcuni costrutti presenti solamente dalla versione 7 di Java, pertanto per compilare il sorgente è necessario disporre di una versione >= Java7. In particolare è stato usato il costrutto try-with-resources perchè piú conciso e semplice da utilizzare con gli Stream.

1.1 Comandi per compilazione ed esecuzione del progetto

`Eseguire i comandi all'interno della cartella parte3`:

1. Per compilare il progetto lanciare il comando `make` da terminale;
2. Per eseguire il progetto, avviare prima il server scrivendo la riga seguente da terminale:
   `sh puzzlesolverserver.sh nome_server`
   è necessario passare un parametro, come descritto al punto 2, per la corretta esecuzione del programma;
3. Successivamente avviare il client scrivendo da terminale:
   `sh puzzlesolverclient.sh nome_file_input nome_file_output nome_server`
   è necessario passare tre parametri, nell'ordine indicato al punto 3, per la corretta esecuzione del programma.

Si nota inoltre che il programma accetta solamente puzzle risolvibili (gestendo anche puzzle mal formati), pertanto non vengono considerati input di puzzle con pezzi tra loro scorrellati.

2 Logica di comunicazione Client-Server

Per la comunicazione client-server è stato usato il modello RMI come richiesto nella specifica di progetto.
La comunicazione client-server è spiegata nei seguenti passaggi:

1. Il server istanzia un oggetto remoto che rappresenta l'algoritmo di risoluzione del puzzle;
2. Il server registra l'oggetto istanziato nel registro RMI invocando il metodo statico *rebind* della classe *Naming*;
3. Il client interroga il registro RMI attraverso il metodo statico *lookup* della classe *Naming*;
4. Il client esegue dunque un downcast al tipo interfaccia del riferimento restituito da *lookup*;
5. Il client invia un puzzle non ordinato serializzato al metodo remoto di risoluzione presente nel server;
6. Il server esegue l'algoritmo di risoluzione invocato remotamente dal client;
7. Il server ritorna un puzzle serializzato contenete i pezzi ordinati.

3 Robustezza del programma

Si è scelto di utilizzare una politica di sincronizzazione lato server al fine di rendere l'esecuzione dei client mutuamente esclusiva. A tal scopo il metodo *sort()* è stato dichiarato `synchronized`.

3.1 Client cerca un oggetto remoto non presente nel registro RMI

Viene sollevata un eccezione del tipo *NotBoundException*.
Il binding del nome dell'oggetto remoto è fallito perchè l'oggetto richiesto non esiste nel registro RMI.

3.2 Client cerca di invocare un metodo remoto di un oggetto che non è più nel registro RMI

In questo caso viene sollevata una eccezione di tipo *NoSuchObjectException*.
L'eccezione notifica il client dicendo che non esiste nessun oggetto corripondente al metodo invocato nel registro RMI.

3.3 Viene interrotto un thread Column in esecuzione

In questo caso viene sollevata un'eccezione del tipo *InterruptedException* internamente alla classe *PuzzleSort*, questa eccezione a sua volta solleva un eccezione di notifica per il client *InterruptNotifyException*. In questo modo si informa il client dell'interruzzione di un thread del server. A tal proposito la classe *InterruptNotifyException* estende *RemoteException* e i metodi *sort()* e *sortColumns()* contengono nella loro firma *throws RemoteException* rendendo possibile il flusso di azioni descritto precedentemente.

3.4 Client chiuso mentre il server risolve il puzzle

In questo caso non viene sollevata nessuna eccezione. Il server continua a procedere senza problemi, il puzzle risolto non verrà restituito al client ormai chiuso ma sarà invece il garbage-collector a deallocarlo dallo heap.

3.5 Client cerca di connettersi ad un server non attivo/esistente o la connessione al server viene troncata mentre il client è in attesa di risposta

Viene sollevata un eccezione di tipo *RemoteException*. Il client viene quindi notificato con la stringa *EXCEPTION: error while connecting to the server* seguito da un messaggio d'errore più specifico.

Per il server valgono alcune eccezioni identiche a quelle appena descritte, altre eccezioni legate al server sono descritte di seguito:

3.6 Server contenente un URL non valido

Viene sollevata un eccezione del tipo *MalformedURLException*. Sul server viene stampato il testo *EXCEPTION: the server URL is not valid* seguito da un messaggio d'errore più specifico.

3.7 Qualsiasi altra eccezione sottotipo di Exception

Vale sia per il server sia per il client. Viene sollevata una qualsiasi altra eccezione sottotipo di Exception non catturata precedentemente, quest'ultima viene indicata come eccezione generica ed è seguita dal messaggio d'errore specifico.

3.8 Altre eccezioni legate ai file di input/output oppure agli argomenti passati al main

3.8.1 Parametri del main non validi

Se il numero di parametri passati al metodo main non rispetta il numero indicato nella specifica di progetto viene sollevata un'eccezione di tipo *InvalidArgumentException* e stampato il seguente messaggio *EXCEPTION: the number of arguments is not valid.*

3.8.2 Il file di input del client non possiede il numero di dati necessari per costruire il puzzle

Viene sollevata un'eccezione del tipo *InvalidParameterSizeException* indicante la riga del file di input contenente l'errore.

3.8.3 Il file di input del client è malformato

Viene sollevata un'eccezione del tipo *InputException*. Il client viene notificato con un messaggio di file non valido.

4 Modifiche rispetto alla versione precedente

4.1 File aggiunti ed eliminati

Rispetto alla versione precedente sono stati aggiunti i seguenti file corrispondenti a classi o interfacce:

1. Package algorithm:
    - PuzzleSortInterface.java
    - InterruptNotifyException.java
2. Package client:
    - PuzzleSolverClient.java

3. Package server:
    - InvalidArgumentException.java
    - PuzzleSolverServer.java

É stato invece eliminato il seguente package ed il suo contenuto:

1. Package solver:
    - PuzzleSolver.java

Nella terza versione del progetto è richiesto di separare la parte algoritmica di risoluzione del puzzle dalla parte di gestione dei file di input ed output, per questo motivo PuzzleSolver è diventato PuzzleSolverClient mentre la logica dell'algoritmo viene gestita interamente da PuzzleSolverServer.

4.2 Puzzlesort

Durante l'implementazione dell'interfaccia PuzzleSortInterface ci si è resi conto che PuzzleSort esponeva un metodo *toList()* inutile che è stato quindi eliminato.
É stato aggiunto un campo dati *tPoolCached* di tipo *ExecutorService* per la gestione del thread-pool.
Inoltre il campo dati result è stato eliminato e spostato come variabile locale interna al metodo sort(), permettendo così di riutilizzare la classe più comodamente per ordinare altri puzzle.
A tal proposito è stato aggiunto un blocco *finally* dopo l'istruzione *catch(InterruptedException)*, interna a *sortColumns()*, che ripulisce *result* e *monitor* perchè i valori contenuti non servono più. Inoltre il blocco finally chiude *tPoolCached*.
L'unico metodo pubblico che si è ritenuto sensato mantenere è *sort()*, è infatti l'unico metodo che viene reso disponibile al client attraverso l'intefaccia *PuzzleSortInterface*.
Così facendo l'implementazione dell'algoritmo utilizzato risulta nascosta al client.
Il grado di accoppiamento tra client e algoritmo è molto basso (collegati solamente dalla chiamata del metodo *sort()*).
PuzzleSortInterface estende *Remote* perchè si vuole utilizzare come interfaccia remota.
PuzzleSort implementa l'interfaccia appena descritta delegando la gestione delle eccezioni alla classe utilizzatrice.
PuzzleSort estende *UnicastRemoteObject* definendo quindi TCP/IP come protocollo di comunicazione.

4.2.1 tPoolCached

Il campo dati *tPoolCached* della classe *Puzzlesort* gestisce il pool di thread attivi in maniera automatica.
A tal proposito è stata modificata la classe Column, ora implementa *Runnable*.
Gli oggetti Column vengono eseguiti da tPoolCached che si occupa anche della loro gestione.

4.3 Puzzle

Sono stati dichiarati *Serializable* sia *Puzzle* sia *Piece* perchè si vuole passare una copia del puzzle al server, inoltre si nota come *Piece* non sia un sottotipo di *Puzzle* e debba quindi essere dichiarato *Serializable* a sua volta.
Non è stato necessario definire *Serializable* i tipi contenuti nelle due classi citate in quanto *tutti i tipi primitivi, String, HashMap<T, K>, ArrayList<T>*[1] sono già *Serializable*.

4.4 InterruptNotifiedException

Eccezione correlata al package algorithm. Estendendo RemoteException permetto all'algoritmo di risoluzione presente sul server di notificare il client nel caso in cui uno dei thread Column venga interrotto.

4.5 InvalidArgumentException

Eccezione gestita sia dal Server sia dal Client. Per entrambe le classi viene controllato che il numero di parametri passati al metodo main() dei 2 programmi sia quello che ci si aspetta, in caso contrario viene lanciata l'eccezione *InvalidArgumentException*.

4.6 PuzzleSolverClient

Legge il file di input, invoca il metodo remoto di risoluzione del puzzle, scrive sul file di output la soluzione del puzzle.
Il main della classe gestisce le possibili eccezioni che possono essere sollevate durante l'esecuzione del programma client (vedi sezione *Robustezza del programma*).

4.7 PuzzleSolverServer

Attende che venga avviato il registro RMI. Istanzia l'oggetto remoto *PuzzleSort*. Registra l'oggetto remoto nel registro RMI, se questo non è attivo viene sollevata un eccezione di tipo *RemoteException*. Se si riesce a registrare l'oggetto remoto nel registro RMI allora il server è pronto ed esegue una stampa per indicare la corretta esecuzione del *main()*.

[1] tipi dei campi dati appartenenti alle classi Puzzle e Piece