PuzzleSolver - Relazione Programmazione Concorrente e Distribuita, Parte 3

Tobia Tesan - #1051819

28 gennaio 2015

Sommario

La presente relazione dettaglia e motiva le scelte progettuali dei programmi `PuzzleSolverServer` e `PuzzleSolverClient`, che realizzano le specifiche della parte 3 del progetto di Programmazione Concorrente e Distribuita per l'A.A. 2014/2015.

Indice

1 Vista d'insieme

1.1 Premessa

Implementare una strategia di comunicazione client/server efficiente senza modifiche drastiche alla struttura delle classi non è stato del tutto triviale a causa delle peculiarità delle classi e strutture dati impiegate nelle parti 1 e 2.

In particolare, nella parte 1 si è concepita un interfaccia PuzzlePrinter con un metodo `print` che prendeva come argomento un IPuzzle risolto, la cui

implementazione stampava in console la soluzione del puzzle. Per utilizzarla occorre dunque una istanza di IPuzzle sul client.

D'altra parte l'implementazione di IPuzzle prevista in parte 1 e parte 2 si presta molto male ad essere serializzata e trasferita al client una volta risolta, poichè contiene una HashMap di BasicPuzzlePiece, ognuno dei quali contiene quattro riferimenti ai vicini che si perdono nella serializzazione.

Si è d'altronde preferito evitare di agire ridefinendo `readObject` e `writeObject` per correggere la situazione e si è invece agito come dettagliato in seguito. Si è anche voluto evitare fino alla fine di fare modifiche significative a classi esistenti per restare nello spirito della OOP.

Dovendo riorganizzare da capo il programma in vista del suo sviluppo concorrente e distribuito si riorganizzerebbero diversamente le classi, in particolare si avrebbe una classe serializzabile rappresentante una soluzione o si resituirebbe direttamente la soluzione come stringa.

1.2 Strategia di comunicazione

La strategia di comunicazione scelta è la seguente:

1. Il programma server istanzia un oggetto IRemotePuzzle.
2. Il programma client recupera un riferimento remoto all'oggetto IRemotePuzzle
3. Il programma client legge il file di input con PuzzleFileParser, immutato dalla prima versione, costruisce localmente un IPuzzlePiece e chiama il metodo `addPiece` sull'oggetto remoto.
4. Il programma client chiama `solve()` sull'oggetto remoto
5. Il programma client ottiene un oggetto locale IPuzzle rappresentante il puzzle risolto attraverso il metodo IRemotePuzzle.`freeze()`.
6. Il programma client stampa il risultato attraverso la classe PlaintextPuzzlePrinter, immutata dalla prima versione, passando l'oggetto locale recuperato al punto precedente.

Il passo 5 è reso necessario dal fatto che, come detto in 1.1, l'oggetto remoto non è serializzabile. Il metodo `freeze` restituisce un oggetto serializzabile che può essere stampato localmente.

1.3 Robustezza

Il programma client avvolge il riferimento remoto IRemotePuzzle con una classe ExponentialBackoffPuzzleWrapper, che prova a fare backoff esponenziale sulle chiamate ai metodi dell'oggetto remoto prima di lasciare risalire la RemoteException. Il programma client e il programma server gestiscono esplicitamente eventuali MalformedUrlException o ConnectException con un messaggio di errore informativo.

2 Organizzazione delle classi

È riportato di seguito l'elenco delle classi. Le nuove classi sono marcate da `*`

```
puzzlesolver
|-- client
|   `-- ExponentialBackoffPuzzleWrapper.java *
|-- core
|   |-- ArrayPuzzle.java *
|   |-- BasicPuzzlePiece.java
|   |-- BasicPuzzlePieceTest.java
|   |-- ConcurrentHashmapPuzzle.java
|   |-- ConcurrentHashmapPuzzleTest.java
|   |-- HashmapPuzzle.java
|   |-- IPuzzle.java
|   |-- IPuzzlePiece.java
|   |-- MissingPiecesException.java
|   `-- PuzzleNotSolvedException.java
|-- io
|   |-- IPuzzlePrinter.java
|   |-- MalformedFileException.java
|   |-- PlaintextPuzzlePrinter.java
|   `-- PuzzleFileParser.java
|-- PuzzleSolverClient.java *
|-- PuzzleSolverServer.java *
`-- server
    |-- FreezableHashmapPuzzle.java *
    |-- FrozenArrayPuzzle.java *
    |-- IRemotePuzzle.java *
    `-- RemoteHashmapPuzzle.java *
```

2.1 Nuove classi

2.1.1 `puzzlesolver.server.IRemotePuzzle`

L'interfaccia IRemotePuzzle offre gli stessi metodi di IPuzzle con l'aggiunta della clausola `throws` RemoteException ed è dunque idonea ad essere utilizzata con RMI.

2.1.2 `puzzlesolver.client.ExponentialBackoffPuzzleWrapper`

ExponentialBackoffPuzzleWrapper è un wrapper intorno a un RemotePuzzle che prova a fare backoff esponenziale un numero `MAX_RETRIES` di volte, entrambi specificati nel costruttore, prima di rilanciare l'eccezione RemoteException.

2.1.3 `puzzlesolver.server.FreezableHashmapPuzzle`

FreezableHashmapPuzzle estende ConcurrentHashmapPuzzle, immutato dalla prima versione, aggiungendo un metodo `freeze` che restituisce un *nuovo* IPuzzle risolto e "congelato".

2.1.4 `puzzlesolver.server.RemoteHashmapPuzzle`

RemoteHashmapPuzzle estende UnicastRemoteObject e implementa IRemotePuzzle. È sostanzialmente un wrapper intorno a un FreezableHashmapPuzzle.

2.1.5 `puzzlesolver.core.ArrayPuzzle`

ArrayPuzzle è una classe *astratta* che rappresenta un puzzle sotto forma di array.

2.1.6 `puzzlesolver.server.FrozenArrayPuzzle`

FrozenArrayPuzzle è un'estensione minimale di ArrayPuzzle. Solleva una RuntimeException se si tenta di effettuare operazioni in scrittura.

2.2 Modifiche a classi esistenti

È stato possibile evitare di fare modifiche a classi esistenti, con l'eccezione di avere esplicitamente dichiarato Serializable la classe BasicPuzzlePiece.

3 Compilazione e utilizzo

```
$ cd parte3/
$ make
$ rmiregistry 1099 &
$ ./puzzlesolverserver.sh localhost &
$ ./puzzlesolverclient.sh input.txt output.txt localhost
```

oppure

```
$ cd parte3/
$ make
$ rmiregistry 1099 &
$ java -jar PuzzleSolverServer.jar localhost &
$ java -jar PuzzleSolverClient.jar input.txt output.txt localhost
```

A Addendum alla parte 2

Nella parte 2 si è scelto di cambiare l'algoritmo risolutivo poichè l'attraversamento *breadth-first* si adatta molto male ad essere parallelizzato. Nello specifico, il fatto che la struttura dati scelta *non sia* un albero ma lo contenga solamente come sottoinsieme dei suoi spigoli e vertici, presentando dunque cicli e diversi percorsi tra due nodi, fa sì che sia molto difficile partizionare il lavoro in unità indipendenti e fare in modo che ciascun thread sappia quando fermarsi.