PCD Assignment 02

fabio.muratori2@studio.unibo.it

AA 2020/2021

1 Words Counter

Lo sviluppo del primo punto dell'assignment vede la rielaborazione del problema presentato precedentemente. È stato deciso di rielaborare aspetti implementativi già trattati nel primo assignment in favore di una maggiore chiarezza delle componenti interpellate e alla luce di problemi riscontrati.

1.1 Approccio con monitor (primo assignment)

Di seguito sono riportati i punti principali delle modifiche apportate alla struttura proposta nel primo assignment:

- è stata uniformata la comunicazione tra i componenti della struttura MVC: il Controller è ora un vero e proprio intermediario tra View e Model, aspetto che permette una più chiara comunicazioni tra questi ultimi. Inoltre nella comunicazione da Model a View è stata adottata una soluzione molto simile a quella proposta. Nella precedente versione infatti ogni aggiornamento dei dati nel Model comportava un aggiornamento della View con una frequenza non gestibile. Il Controller utilizza ora un thread dedicato al **polling** periodico dei risultati ottenuti dal Model. Tale thread è attivo solamente durante l'esecuzione del task.
- è stata raffinata la procedura di caricamento del contenuto di file PDF. Inizialmente ogni file veniva completamente caricato in memoria e processato indipendentemente dalla dimensione del documento. Si è preferito adottare l'approccio che separa un documento in pagine progressivamente caricabili. Ciò ha comportato la ristrutturazione del Buffer (in *pcd.ass1.model. ClosableBoundedBuffer*) e la suddivisione del task nei 3 componenti attivi:
    - scan di una directory del file system (in *pcd.ass1.model.FilesFinder*)
    - caricamento dei singoli chunk (in *pcd.ass1.model.TextLoader*)
    - processamento dei chunk (in *pcd.ass1.model.ChunkProcessor*)
- migliorata la gestione del life-cycle del Model manager (*pcd.ass1.model.Model*) tramite l'utilizzo opportuno di Barrier per l'inizializzazione del task e Latch per la terminazione.

1.2 Approccio a task

L'introduzione dell'approccio a Task ha permesso di semplificare notevolmente la struttura del Model del programma. Essendo i risultati dei task aggregati in maniera sincrona è stato possibile eliminare le strutture *ClosableBoundedBuffer* implementate nel primo assignment.
Considerando invece la comunicazione dei dati aggiornati tra Model e View, quindi nell'accesso alle strutture condivise (*pcd.ass2.part1.model.Dictionary*), rimane necessaria una gestione thread-safe con Monitor visto che il Controller (*pcd.ass1.part1.controller.Controller*) mantiene un proprio flusso di controllo a se stante, rappresentato da un task periodico: tale task permette l'aggiornamento dei dati della View considerando le stesse motivazioni individuate per la versione thread-oriented del sistema.

Nella realizzazione del task di conteggio dei termini si è voluto mettere a confronto i seguenti approcci:

- **Fixed thread pool** (in *pcd.ass2.part1.model.ModelThreadPool*): ad ogni documento DPF è associato un flusso di controllo ed un thread di esecuzione;
- **Fork-Join** (in *pcd.ass2.part1.model.ModelForkJoin*): si è tentato di ricorsivamente suddividere il carico di lavori su più task gerarchicamente relazionati (*FolderSearchTask*, *FileLoaderTask*, *WorkcounterTask* in *pcd.ass2.part1.model)*. In questo modo ogni task può essere eseguito dal primo thread libero.

Di seguito sono confrontati i tempi di calcolo due approcci.

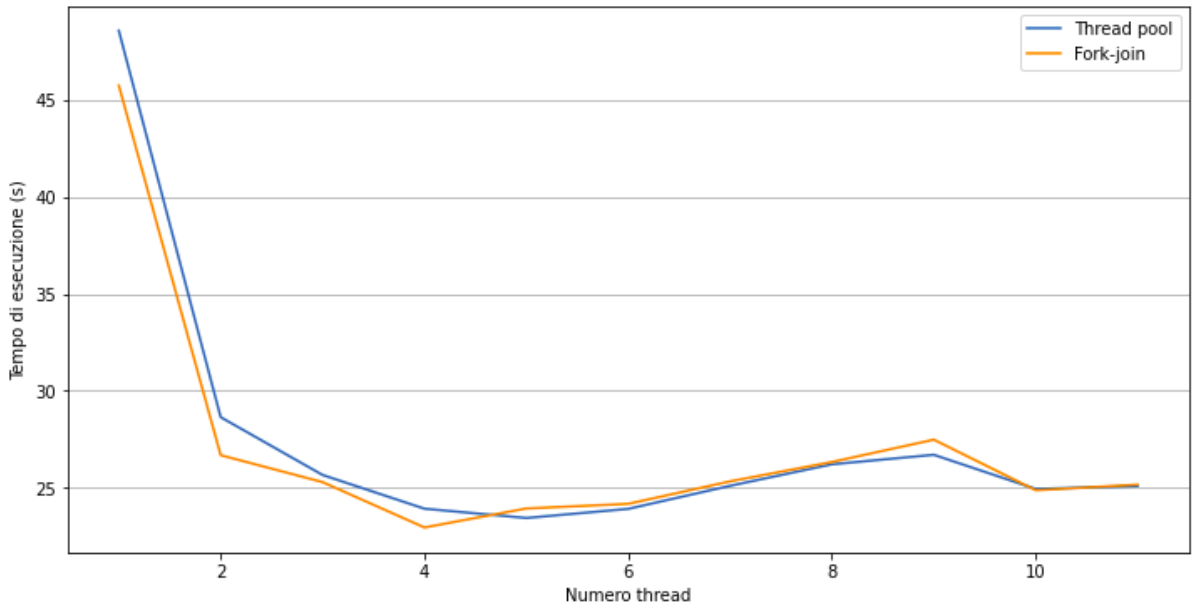

Tabella 1.1.1 - Confronto tra tempi di esecuzione metodi Fixed thread pool e Fork-Join.

Si osserva che l'approccio Fixed thread pool performa leggermente peggio della controparte, seppure non in maniera significativa. Questo è probabilmente dovuto in parte alla natura dei due metodi ma anche alla tipologia di test effettuati: utilizzando molteplici copie di uno stesso documento PDF, il tempo necessario a processare i singoli task è pressappoco identico, e una decomposizione in task ricorsivi con Fork-Join permette in determinate situazioni una migliore suddivisione del carico computazionale su più thread separati, mentre nell'approccio Fixed thread pool ciò non è possibile risultando nell'utilizzo di una parte dei processori a disposizione. Ciononostante si osserva anche come nell'approccio Fixed thread poolsi hanno minori sovraccarichi nella gestione degli executor da parte del sistema operativo.
Inoltre, un approccio Fork-Join sembra richiedere uno studio più dettagliato del task da realizzare per opportunamente selezionare i punti dove adottare una politica *divide et impera*.

La struttura generale del programma non ha avuto ulteriori modifiche sostanziali e le modifiche si sono concentrate prevalentemente nel Model.

1.3 Approccio reattivo

Nell'implementazione del punto 3 del assignment è stato fatto uso del framework JavaRX. Similmente all'approccio con worker, è stato possibile rielaborare il flusso di esecuzione del task minimizzando l'utilizzo di strutture concorrenti.

L'introduzione dell'approccio reattivo a visto la rielaborazione del flusso di esecuzione del programma:

- tramite l'utilizzo di PublishSubject di FlowableEvent (in *pcd.ass2.part3.Controller ViewFlowableEvent* e *ModelFlowableEvent*, classi wrapper di eventi generabili) è stato possibile implementare il meccanismo degli Observer per le comunicazioni tra View e Model;

- l’esecuzione parallela di più flussi di controllo è stata resa possibile da uno Scheduler
- è stato convertito il task generale in un Flowable ed una sequenza di operazioni applicate a stream di dati

In *pcd.ass2.part3.FlowableTask* è possibile osservare l’implementazione del Task adottando il principio di programmazione reattiva tramite JavaRX. È stato possibile monitorare l’esecuzione del task tramite la sottoscrizione di handler personalizzati al Flowable principale, consentendo sia l’aggiornamento dei dati condivisi ma anche della View tramite un apposito canale PublishSubject.
Non è stata tentata l’introduzione di un Flowable in sostituzione al PublishSubject per la gestione della policy di aggiornamento dei dati nella View in conseguenza all’avanzamento dei flussi lato Model. Idealmente tramite il meccanismo di back-pressure sarebbe stato possibile gestire opportunamente l'aggiornamento dei dati nella View in situazioni di stress (aggiornamento dei dati in un breve lasso di tempo).

1.4 Benchmark

Sono stati condotti esperimenti mirati al confronto delle varie versioni del progetto sviluppato. Ogni risultato è il prodotto di molteplici esecuzioni del codice fornito su una macchina con 4 core. Sono stati utilizzati 10 documenti PDF identici ciascuno di 500 pagine.
L’aggiornamento delle varie View è stato disabilitato durante i test per evitare interazioni con i dati condivisi tramite Monitor con potenziali chiamate bloccanti nel task in esecuzione.

Versione	**Tempo di esecuzione (s)**
sequenziale	46.40
thread oriented	23.14
task oriented (con Fork-Join)	22.95
reattiva	24.88

Tabella 1.4.1 - Tempo medio di esecuzione del task

Confrontando la versione sequenziale con quella task-oriented è possibile calcolare lo *speed-up*:

$$Speed\ Up = \frac{T_{serial}}{T_{parallel}} = \frac{46.40}{22.95} \simeq 2.02$$

2 Train API

Questa parte dell’assignment richiede l’implementazione di una API per ottenere informazioni in tempo reale riguardo allo stato di treni e stazioni. Nello specifico, sono state implementate una libreria che sfrutta VertX per mandare richieste a diversi web server e un’applicazione con un’interfaccia grafica rudimentale per permettere l’interazione con tale libreria.

É stato usato il framework VertX ed un suo modulo WebClient, con il quale si mandano delle richieste http per ottenere dati in formato json. Vista l’architettura asincrona, i valori di ritorno dei metodi che

la libreria espone sono di tipo Future, e il loro contenuto è accessibile tramite le chiamate onSuccess oppure onComplete. I risultati in formato Json sono opportunamente filtrati e incapsulati in classi Java (in *pcd.ass2.part2.model* rispettivamente *TrainInfo*, *StationInfo*, *TravelSolution*) .

Il monitoraggio di una soluzione di viaggio ha richiesto l'introduzione di un Verticle separato. In particolare, per ogni treno identificato dalla soluzione di viaggio selezionata, viene periodicamente effettuata una chiamata API getRealTimeTrainInfo (in *pcd.ass2.part2.model.APIRequests*) il cui successo comporta l'aggiornamento del relativo treno. In questo modo è possibile sia mantenere un approccio ad event loop ma anche supportare richieste relative a molteplici treni (si pensi ad una soluzione di viaggio con più treni cambi).

Di seguito sono riportate le Petri net per le due parti del sistema. Si noti come alcune delle funzionalità offerte comportano in realtà molteplici chiamate all'API, in cascata (in Figura 2.1 si nota che per ottenere informazioni su un treno, l'API REST utilizzata necessita anche della stazione di partenza del treno specificato) oppure in parallelo (in Figura 2.2 ad ogni treno è associata una chiamata http, l'ordine è ininfluente)

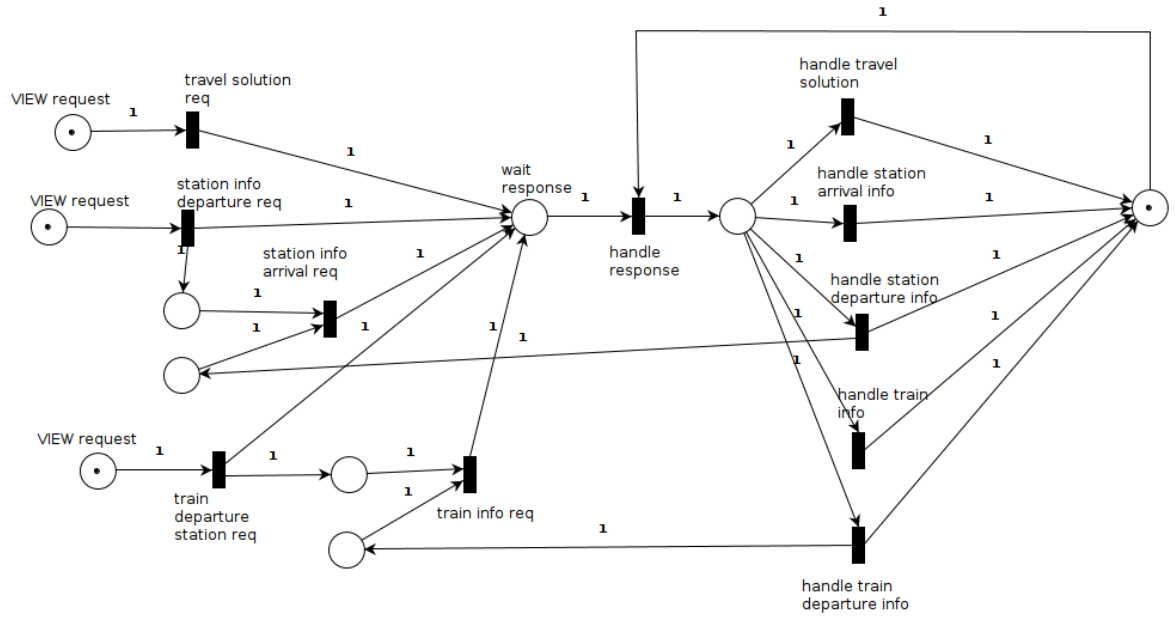

Figura 2.1 - Petri net gestione event loop principale

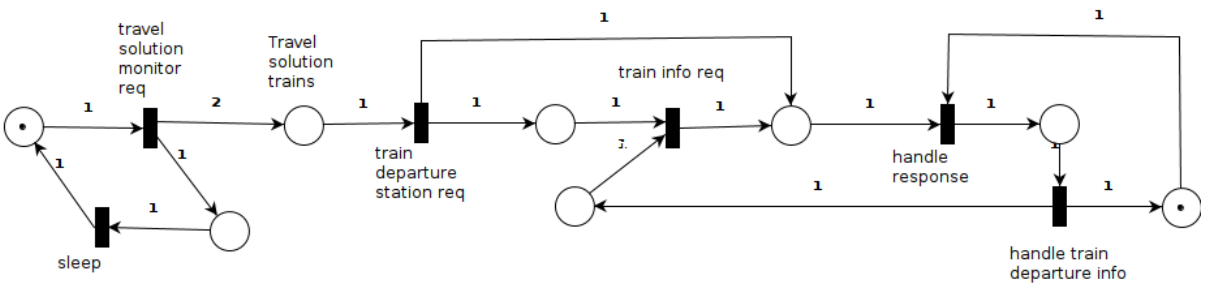

Figura 2.2 - Petri net per gestione event loop e monitoraggio di una soluzione di viaggio