Assigment #2 - "Asynchronous Programming"

Martina Magnani
martina.magnani8@studio.unibo.it
Nicola Piscaglia
nicola.piscaglia2@studio.unibo.it
Mattia Vandi
mattia.vandi@studio.unibo.it

1 Analisi del problema

Progettare e implementare uno strumento per cercare corrispondenze di un'espressione regolare in una struttura ad albero o grafo di file (come un filesystem o un sito web).
Lo strumento deve avere il seguente comportamento:

- Input:
  - Percorso di base della ricerca (ad esempio, un percorso del filesystem o un URL della pagina web)
  - Espressione regolare
  - Profonditssima di ricerca
- Output:
  - Elenco dei file corrispondenti
  - Percentuale di file con almeno una corrispondenza (somma di file con corrispondenza/file totali);
  - Numero medio di corrispondenze tra i file con corrispondenze (somma di tutte le corrispondenze/numero di file con corrispondenze).

  Le uscite devono essere aggiornate in tempo reale con il procedere dell'elaborazione.

Esercizio 1:
Risolvere il problema usando task ed executor

- I file devono essere letti e analizzati contemporaneamente.
- Si puhe optare per parallelizzare l'analisi di file di grandi dimensioni.

Esercizio 2:
Risolvere il problema utilizzando la programmazione asincrona con Event Loop

- Cerca di riutilizzare il maggior numero possibile di codice dall'esercizio 1, ma anche di ripensare alla soluzione in base al nuovo punto di vista.

Esercizio 3:
Risolvere il problema utilizzando i flussi reattivi

- Ad esempio, i risultati dell'elaborazione possono essere reificati in un flusso di eventi che puere manipolato mediante tecniche di programmazione reattiva

Ulteriori note:

- L'interfaccia utente puere a riga di comando, grafica o basata sul web.
- Sei libero di utilizzare qualsiasi framework basato sugli eventi (ad es. Vertx, NodeJS) e la libreria del flusso reattivo (ad es. RxJava, Sodio).

2 Descrizione della soluzione proposta

Lo strumento realizzato serve per la ricerca di corrispondenze di un'espressione regolare tra i file, dato un percordo di partenza e la profonditssima.

2.1 Design

2.1.1 Dominio Applicativo

L'architettura del sistema ata realizzata in modo che il dominio applicativo fosse svincolato dalla logica di controllo e fosse condiviso fra le tre soluzioni.

Il dominio applicativo mposto dai seguenti elementi:

- `Document`: classe che astrae il concetto di documento, mposto da una lista ordinata di linee che rappresentano il suo contenuto e dal nome del documento.
- `Folder` classe che astrae il concetto di cartella di un filesystem, mposta da una lista di documenti e da una lista di sottocartelle.
- `SearchResult`: classe che rappresenta il risultato della ricerca delle occorrenze dell'espressione regolare (fornita in input) in uno specifico documento. Il risultato mposto dal nome del documento e dal numero di occorrenze trovate.
- `SearchStatistics`: classe che rappresenta le statistiche della ricerca in atto come richieste nell'Output specificato nell'analisi del problema **??**.

2.1.2 Esercizio 1: Task Executors

La soluzione proposta vede l'utilizzo del modello ForkJoin, questo ci permette di suddividere il problema in una gerarchia di compiti atomici. Analizzando i requisiti abbiamo individuato tre task principali:

- `FolderSearchTask`: ricerca delle occorrenze dell'espressione regolare in una cartella

- `DocumentSearchTask`: ricerca delle occorrenze dell'espressione regolare in un documento
- `SearchResultAccumulatorTask`: aggregazione dei risultati e calcolo delle statistiche

Dato un percorso e una profonditssima viene costruito un albero nel quale le cartelle rappresentano i rami e i documenti le foglie. Partendo dalla radice visitiamo tutte le sottocartelle e i documenti presenti: nel primo caso viene eseguito un nuovo `FolderSearchTask`, nel secondo caso viene eseguito un nuovo `DocumentSearchTask`.
Ogni risultato intermedio trovato mette in esecuzione una callback che notifica il `SearchResultAccumulatorTask`, eseguito su un executor dedicato.
`SearchResultAccumulatorTask` si occupa di aggiornare i risultati e di passarli ad una callback responsabile di aggiornare l'output.

2.1.3 Esercizio 2: Event Loop

2.1.4 Esercizio 3: Reactive Streams

Come implementazione dei flussi reattivi abbiamo usato la libreria RxJava2 [1]. L'approccio dichiarativo fornito dall'utilizzo dei Reactive Stream permette di semplificare la soluzione. In questo caso l'insieme di documenti sto come flusso reattivo al quale ssibile applicare operazioni di trasformazione, di calcolo e di sottoscrizione. Abbiamo scelto di utilizzare i Flowable perchplementano i reactive stream e il meccanismo di backpressure che permette di ottimizzare l'elaborazione del flusso di dati.
Lo stream di dati ato creato a partire dalla radice dell'albero di cartelle: con il metodo `getDocuments()` otteniamo ricorsivamente i documenti di ogni cartella e sottocartella e li mappiamo in un unico stream di documenti.
Generato lo stream specifichiamo su quale Scheduler verrano eseguite le manipolazioni.
Per ogni documento dello stream contiamo il numero di occorrenze dell'espressione regolare in input e lo trasformiamo in un `SearchResult`.
A questo punto sottoscriviamo `SearchResultAccumulator` al flusso utilizzando `blockingSubscribe()`: questa scelta vuta al fatto che vogliamo sospenderci sul thread corrente finchtto lo stream non ato processato.
`SearchResultAccumulator` implementa l'interfaccia `Subscriber`, che gli permette di essere notificato quando un nuovo elemento del flusso sponibile e della sua terminazione. All'arrivo di un nuovo dato si occupa di aggiornare i risultati e della creazione delle statistiche. Quest'ultime vengono passate ad una callback responsabile dell'aggiornamento dell'output.

3 Dinamica del sistema

La dinamica del sistema ata rappresentata formalmente con le Reti di Petri. Le piazze (*places*) del livello corrispondono ai `Worker` che concorrono all'aggiornamento della scacchiera di gioco; in questo esempio ato scelto di mostrare il sistema nel caso se ne utilizzino 4.

[1] `https://github.com/ReactiveX/RxJava`

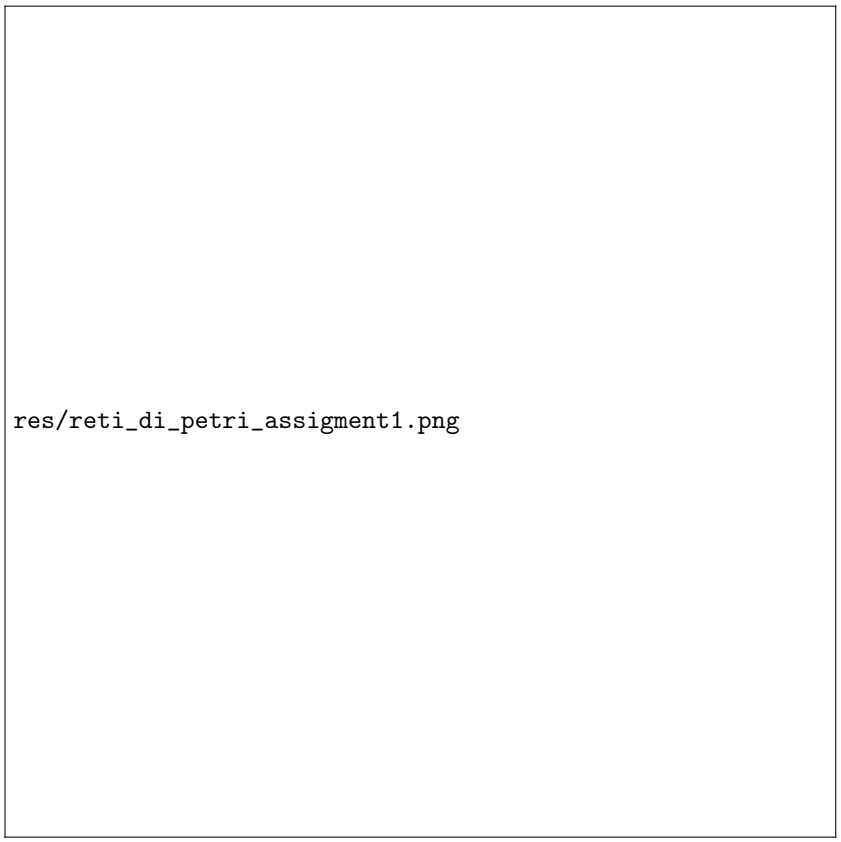

Figura 1: Petri Net

4 Analisi delle prestazioni

	2	3	4	5	6	7	8	9	10
Min.	1.77	2.43	3.04	2.93	2.89	2.90	2.86	2.85	2.85
Max.	2.11	2.87	3.41	3.22	3.40	3.26	3.38	3.06	3.45
Avg.	1.81	2.50	3.06	2.94	2.93	2.91	2.92	2.85	2.94

Tabella 1: Speedup considerando schacchiera di gioco 5000x5000: sulle colonne sono specificati il numero di worker utilizzati per la parallelizzazione dell'algoritmo, sulle righe i valori statistici (minimi, massimi e medi) degli speedup calcolati. Gli speed-up sono stati calcolati su un Apple Macbook Pro (Retina, 15", met15) dotato di un processore Intel Core i7 quad-core a 2.2GHz.