

Assignment #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 un programma per cercare le corrispondenze di un'espressione regolare in una struttura ad albero o a grafo (come un filesystem o un sito web).

Il programma deve avere il seguente comportamento:

- Input:
 - Percorso di base della ricerca (ad esempio, un percorso del filesystem o l'URL di una pagina web);
 - Un'espressione regolare;
 - La profondità massima 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).

Gli output 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 può anche optare per parallelizzare l'analisi di file di grandi dimensioni.

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

- Cercare 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 può essere manipolato mediante tecniche di programmazione reattiva

Ulteriori note:

- L'interfaccia utente può essere a riga di comando, a finestre o basata sul web.
- È possibile utilizzare un qualsiasi framework basato sugli eventi (ad es. Vert.x, Node.js) o libreria per la manipolazione di flussi reattivi (ad es. RxJava, Sodium).

2 Descrizione della soluzione proposta

Il programma realizzato ricerca le corrispondenze di un'espressione regolare tra file, dato in input un percorso di partenza e la profondità massima.

2.1 Dinamica del Sistema

2.1.1 Dominio Applicativo

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

Il dominio applicativo è composto dai seguenti elementi:

- **Document**: classe che astrae il concetto di documento, è composto 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, è composta 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 è composto 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.
- **SearchResultAccumulator**: classe che implementa la logica di accumulazione dei risultati.

2.1.2 Esercizio 1: Task ed Executors

La soluzione proposta vede l'utilizzo del modello Fork-Join. L'adozione di tale modello a task 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 profondità massima viene costruito un albero nel quale le cartelle rappresentano i rami e i documenti le foglie. Partendo dalla radice si visitano tutte le sottocartelle e i documenti presenti: nel primo caso viene eseguito un nuovo **FolderSearchTask** per ogni sottocartella incontrata, nel secondo caso viene eseguito un nuovo **DocumentSearchTask** per ogni documento incontrato. 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.

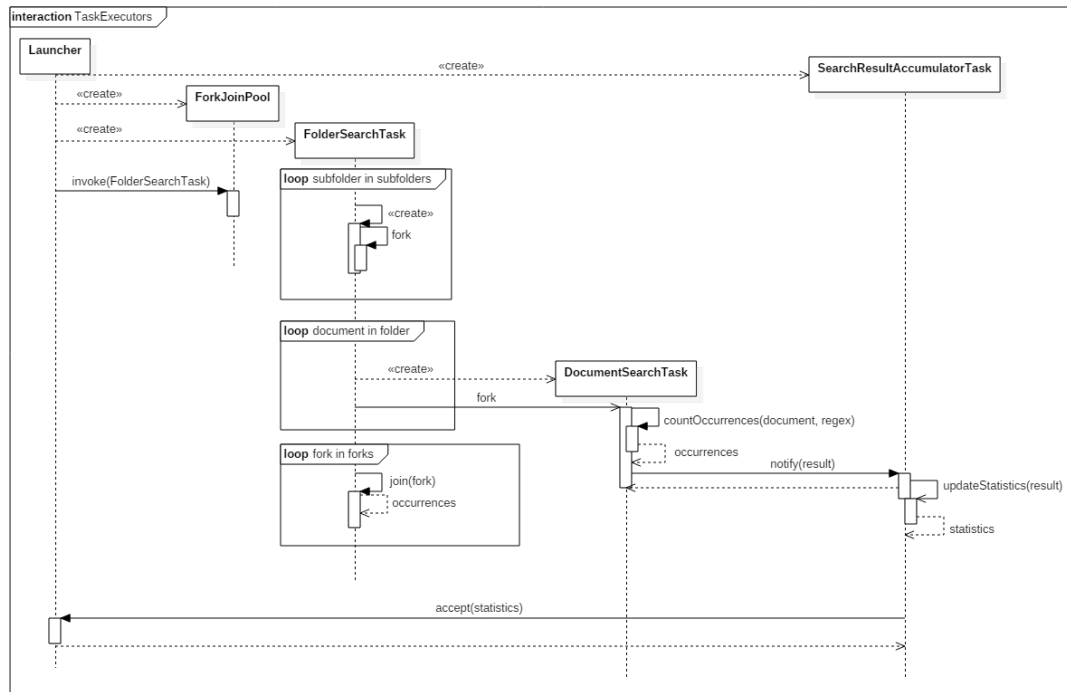


Figura 1: Diagramma di sequenza dell'Esercizio 1

2.1.3 Esercizio 2: Event Loop

Come implementazione del modello a Event Loop è stato utilizzato il framework Vert.x ¹. Analogamente all'Esercizio 1 il problema di partenza è stato scomposto in tre sottoproblemi, in questo caso risolti da tre tipologie di Verticle:

- **FolderSearchVerticle**: ricerca delle occorrenze dell'espressione regolare in una cartella
- **DocumentSearchVertice**: ricerca delle occorrenze dell'espressione regolare in un documento
- **SearchResultAccumulatorVerticle**: aggregazione dei risultati e calcolo delle statistiche

Si è scelto di mantenere un numero fisso di istanze per ogni tipologia di Verticle. Nel caso del `SearchResultAccumulatorVerticle` l'istanza è unica, mentre per le altre due tipologie di Verticle il numero di istanze può variare al fine di parallelizzare il carico di lavoro.

La comunicazione tra i Verticle si basa sullo scambio di messaggi attraverso un canale condiviso (Event Bus): ogni tipologia di Verticle ascolta su un canale dedicato.

¹<https://vertx.io>

La ricerca inizia inviando un messaggio contenente la radice dell'albero all'indirizzo dedicato alla ricerca all'interno di una cartella. Il Verticle che riceve il messaggio (**FolderSearchVerticle**) visita tutte le sottocartelle e, per ognuna di esse, invia un messaggio allo stesso indirizzo. In questo modo un altro **FolderSearchVerticle** provvederà alla ricerca nella sottocartella e così via. Analogamente, per ogni documento, viene inviato un messaggio ad un **DocumentSearchVerticle** che provvederà alla ricerca delle occorrenze dell'espressione regolare fornita in input nel documento inviatogli. Ogni **DocumentSearchVerticle**, dopo aver contato il numero di occorrenze il risultato, invia un messaggio al **SearchResultAccumulatorVerticle**. Quest'ultimo, dopo aver aggiornato le statistiche, chiama una callback responsabile di aggiornare l'output.

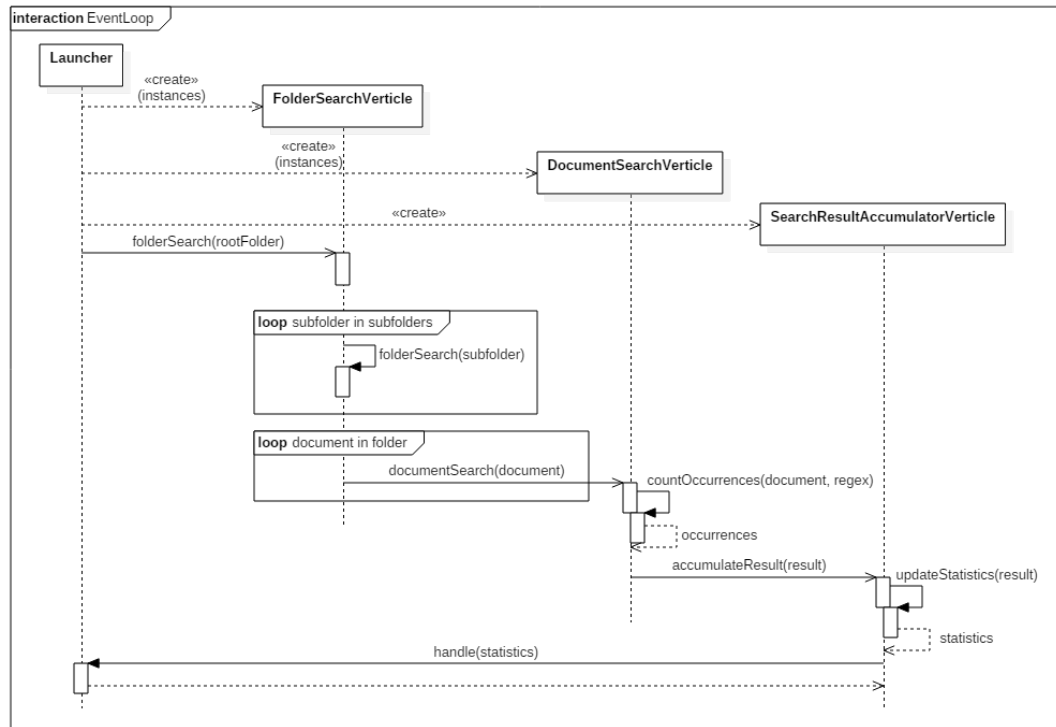


Figura 2: Diagramma di sequenza dell'Esercizio 2

2.1.4 Esercizio 3: Reactive Streams

Come implementazione dei flussi reattivi abbiamo utilizzato la libreria RxJava2². L'approccio dichiarativo fornito dall'utilizzo dei Reactive Stream permette di semplificarne la soluzione. In questo caso l'insieme di documenti è visto come flusso reattivo al quale è possibile applicare operazioni di trasformazione, di calcolo e di sottoscrizione. Abbiamo scelto di utilizzare il costrutto **Flowable** perché implementano i flussi reattivi³ ed il meccanismo di backpressure che permette di ottimizzare l'elaborazione dei flussi che emettono elementi ad una velocità maggiore di quella con cui vengono elaborati.

²<https://github.com/ReactiveX/RxJava>

³<http://www.reactive-streams.org>

Il flusso di dati è stato creato a partire dalla radice dell'albero di cartelle: cottenendo ricorsivamente i documenti di ogni cartella e sottocartella e trasformandoli in un unico flusso di documenti. Generato il flusso specifichiamo su quale **Scheduler** verranno eseguite le manipolazioni. Per ogni documento del flusso contiamo il numero di occorrenze dell'espressione regolare fornita in input e lo trasformiamo in un **SearchResult**. A questo punto sottoscriviamo **SearchResultSubscriber** al flusso utilizzando il metodo **blockingSubscribe()**: questa scelta permette di sospenderci sul thread corrente finché tutto il flusso non è stato processato, in caso contrario il thread corrente terminerebbe prima che tutti gli elementi emessi nell flusso reattivo siano processati.

SearchResultSubscriber implementa l'interfaccia **Subscriber**, permettendogli di sottoscrivere al flusso reattivo e di essere notificato quando il prossimo elemento è disponibile e quando il flusso è terminato. All'arrivo di un nuovo elemento si occupa di aggiornare i risultati e della aggiornamento delle statistiche, che vengono passate ad una callback responsabile dell'aggiornamento dell'output.

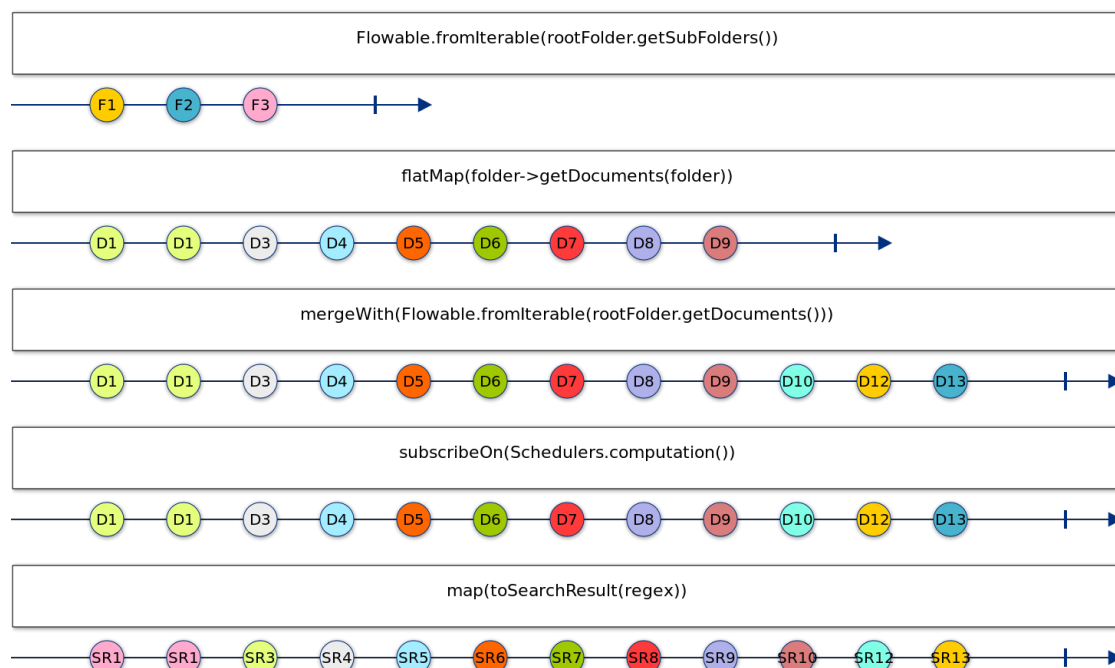


Figura 3: Marble Diagram dell'Esercizio 3

3 Analisi delle prestazioni

Le prestazioni sono state valutate considerando i tempi di esecuzione di ogni esercizio. Il calcolo è stato automatizzato e

	Executors	Event Loop	Reactive Streams
Min	1.766	2.43	3.04
Max	2.11	2.87	3.41
Avg	1.81	2.50	3.06

Tabella 1: Tempo di esecuzione.