Programmazione Concorrente e Distribuita
Assignment 2

Matteo Ragazzini
Simone Romagnoli

A.A. 2020/2021

Introduzione

Nel presente documento si descrivono le soluzioni ai problemi del secondo assignment del sopracitato corso realizzate dal nostro gruppo. Come linguaggio d'implementazione è stato utilizzato *Java*, usufruendo della libreria `org.apache.pdfbox` fornitaci assieme alle specifiche. In seguito, sono discusse separatamente le soluzioni ai tre problemi dell'assignment, dando per scontati gli aspetti di analisi del problema laddove siano già stati affrontati in precedenza (parte 1 e parte 3).

1 Word Counter: approccio a task

1.1 Introduzione

In questa parte dell'assignment è richiesta una soluzione simile a quella consegnata in precedenza, ma con approccio orientato ai task e utilizzo di Java Executors.

1.2 Design

Per effettuare al meglio il design del programma con approccio a task, si è partiti dall'analisi del problema descritta nell'assignment 1. Le 4 operazioni principali individuate nel problema (*strip*, *split*, *filter* e *count*) devono essere svolte sequenzialmente per ogni singolo documento e la prima risulta essere la più onerosa di tutte: per questo, è stato naturale implementare la *strip* come una `Recursive Action`, la quale lancia le altre operazioni prima di terminare.

I task che svolgono la computazione principale sono `Strip` e `SplitFilterCount`. Nello specifico, estendono il comportamento della classe `RecursiveAction` poiché sono indipendenti gli uni dagli altri e non restituiscono alcun risultato al chiamante: questo è stato fatto per favorire la reattività della GUI, aggiornando un monitor condiviso per i loro risultati. Come nell'assignment precedente, i monitor utilizzati sono un `OccurrencesMonitor` (che tiene conto della mappa parole-numero di occorrenze) e un `ElaboratedWordsMonitor` (che si occupa di tenere aggiornato il numero di parole elaborate).

Anche l'architettura MVC è stata leggermente modificata: la View è stata dotata di un task Viewer che aggiorna la GUI leggendo frequentemente il contenuto dei monitor; il Model è stato implementato come una `RecursiveAction` che crea i task descritti e ne attende la terminazione (a meno che non venga fermata prima). Perciò, come istanza di `ExecutorService` del sistema è stato utilizzato un `ForkJoinPool`: grazie ad esso, si è anche potuta esprimere al meglio l'importanza dell'ordine d'esecuzione dei task, pur mantenendo la loro indipendenza reciproca.

Metodo chiamato alla pressione del pulsante Start

```
public synchronized void started(File dir, File wordsFile, int limitWords) {
  this.stopFlag.reset();

  final ForkJoinPool executor = new ForkJoinPool(N_THREADS);

  final OccurrencesMonitor occurrencesMonitor = new OccurrencesMonitor(limitWords);
  final ElaboratedWordsMonitor wordsMonitor = new ElaboratedWordsMonitor();

  final Model model = new Model(this.stopFlag, dir, wordsFile, executor, occurrencesMonitor,
      wordsMonitor);
  final Viewer viewer = new Viewer(this.view, this.stopFlag, executor, occurrencesMonitor,
      wordsMonitor);

  executor.submit(model);
  executor.submit(viewer);
 }
```

Il sistema può essere rappresentato dalla rete di Petri in figura 1. Tale rappresentazione è un esempio del sistema provato con 5 pdf.

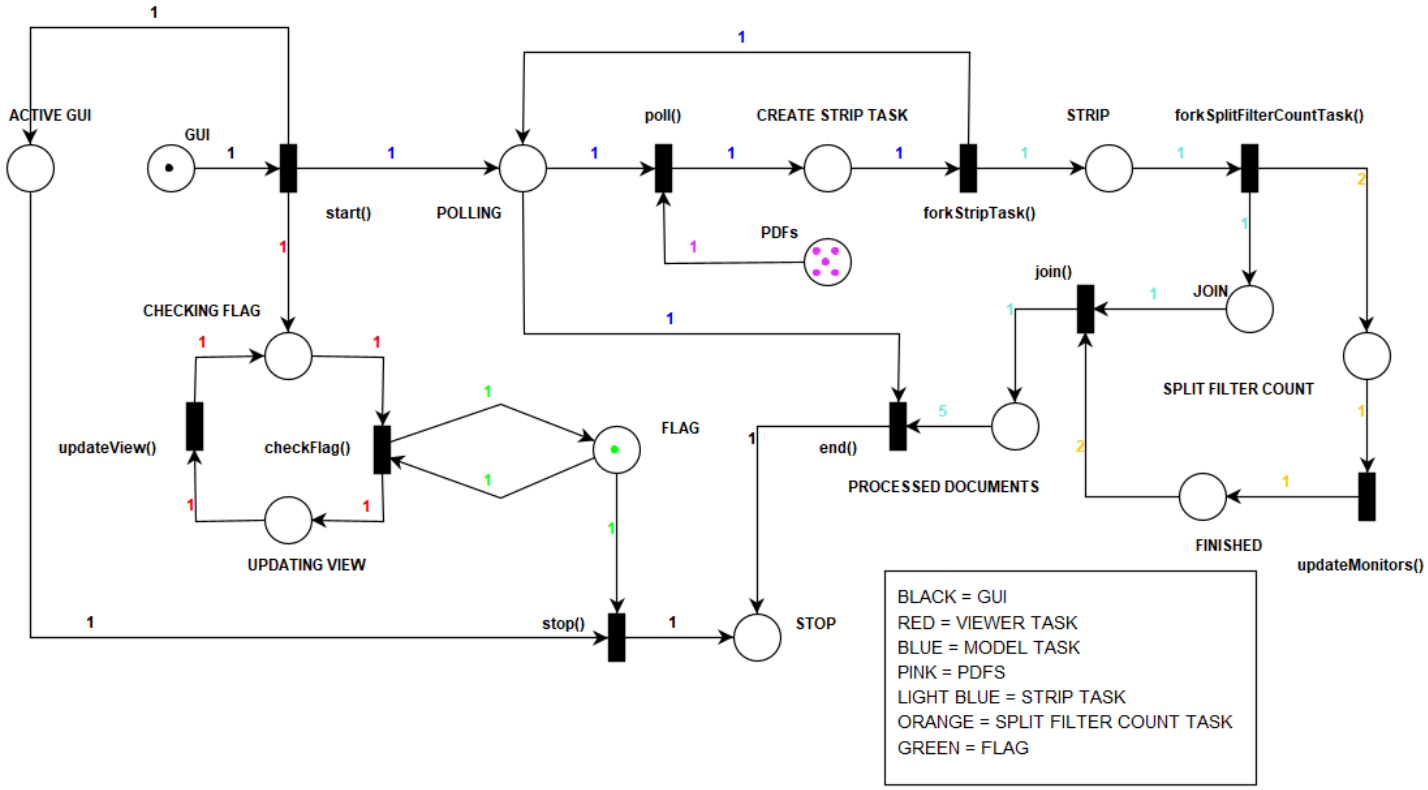

Figura 1: Rete di Petri rappresentante la soluzione al problema con approccio a task.

Il task del *Model* si occupa di prelevare un documento alla volta e creare una fork del task *Strip* per

ognuno di essi; vista la "leggerezza" delle operazioni restanti, ogni documento viene diviso in chunk (un chunk corrisponde a una pagina, nell'esempio si suppone che ogni documento ne abbia 2): per ogni chunk si crea una fork del task che esegue le operazioni di *split*, *filter* e *count*. La computazione termina quando il *ModelTask* effettua la join per ogni fork precedentemente lanciata.

1.3 Valutazione delle prestazioni

Per valutare la dimensione del ForkJoinPool, è stato provato il programma con pool di grandezze diverse: per ogni numero sono state fatte più misurazioni e ne è stata effettuata la media, quindi è stata mantenuta la dimensione che presentava le prestazioni migliori, in termini di tempo d'esecuzione e uso della CPU; i tempi d'esecuzione misurati sono mostrati nel grafico 2.

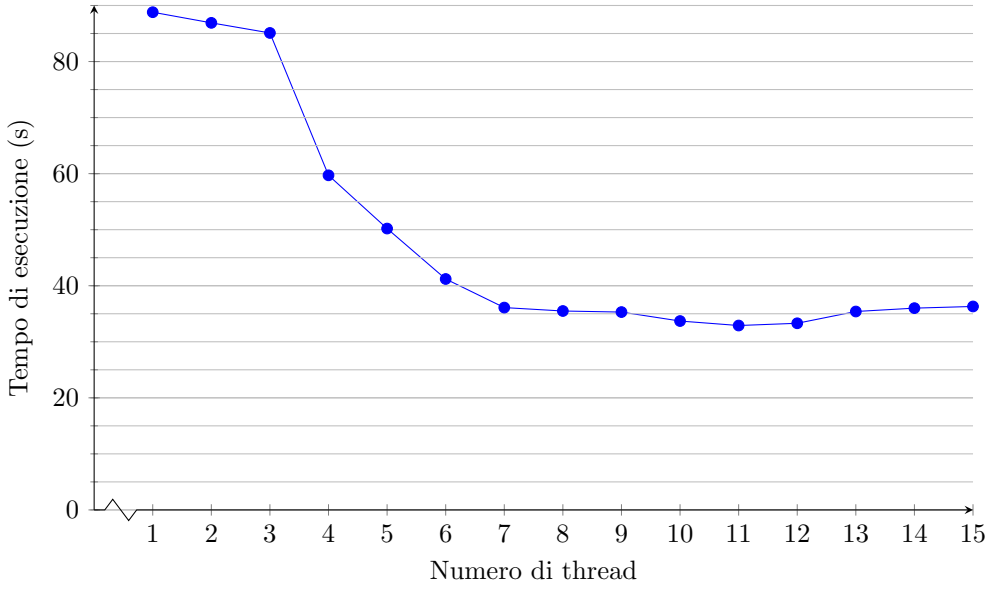

Figura 2: Grafico dei tempi al variare della dimensione del ForkJoinPool.

Nello specifico, per una macchina che dispone fino a 8 core logici, la dimensione migliore del pool è risultata essere di 11 thread; le misurazioni sono state effettuate con gli stessi documenti, ovvero un gruppo di 15 pdf da 900 pagine ciascuno. Comunque, tenendo conto del task *Viewer* e di quello della GUI, il numero totale di thread in gioco è leggermente maggiore. Confrontando la misurazione con 1 task e quella con 11, lo *speed-up* ottenuto risulta essere circa

$$Speed - up = \frac{T_{serial}}{T_{parallel}} = \frac{88.8}{32.9} \simeq 2.7$$

Lo *speed-up* rilevato non si avvicina ad uno *speed-up* lineare, ma è un risultato prevedibile, visto il *bottleneck* dovuto all'impossibilità di effettuare in parallelo l'operazione di *strip* su uno stesso documento.

2 Trains API: approccio a event-loop

2.1 Introduzione

Per questa parte dell'assignment è richiesta 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 che permette di usufruire di tale libreria per effettuare ricerche a piacimento. Sia il client per le richieste http, sia l'interfaccia grafica sono stati implementati in *Java*.

2.2 Design delle API

Per quanto riguarda l'*API*, è stata implementata una libreria che sfrutta la classe `WebClient` di *VertX*, con la quale si mandano delle richieste *http* per ottenere dati in formato *json*, riuscendo a mantenere un approccio a *event-loop*. I dati ottenuti vengono successivamente incapsulati in delle classi che ordinano al meglio i dati degli oggetti *json*, ad esempio formattando le date e gli orari. 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*.

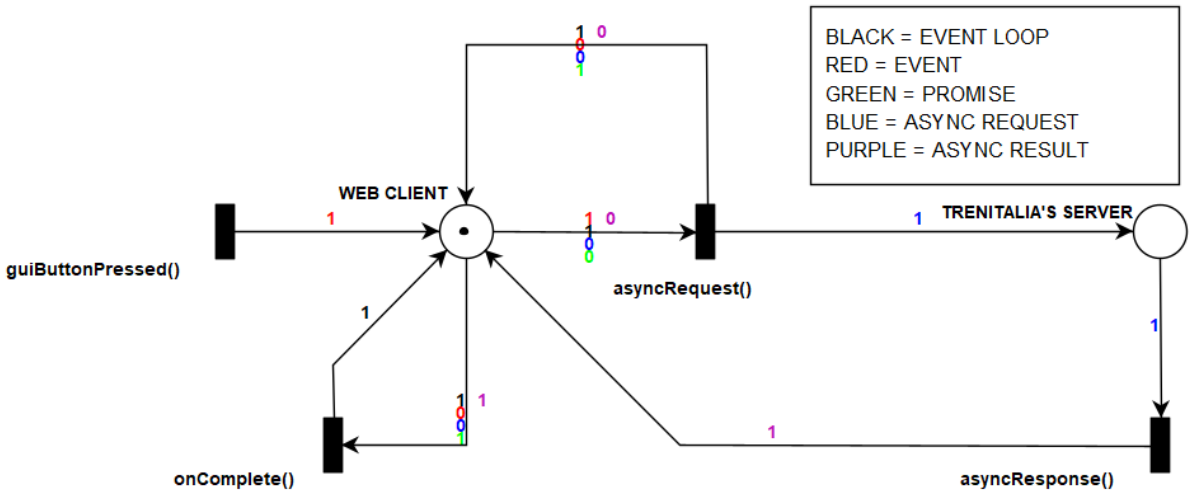

Figura 3: Rete di Petri rappresentante l'event-loop principale implementato con un web client di *VertX*.

Inoltre, è stato utilizzato un Verticle per implementare il monitoraggio in tempo reale delle soluzioni di viaggio; in particolare, è stata realizzata una classe `MonitoringVerticle` che estende `AbstractVerticle` e che lancia periodicamente richieste per ottenere dati in tempo reale sullo stato dei treni (ritardo e ultimo avvistamento).

Metodo *start()* del `MonitoringVerticle`

```
@Override
public void start() {
    getRealTimeInfo();

    this.id = this.getVertx().setPeriodic(MINUTE, (t) -> {
        getRealTimeInfo();
    });
}
```

2.3 Design della GUI

L'interfaccia grafica, sviluppata con *Swing*, permette di osservare ordinatamente fino a 5 soluzioni di viaggio a partire da una stazione di partenza, una di arrivo, una data e un orario. Una volta che le soluzioni di viaggio sono state visualizzate correttamente, è possibile sceglierne una per visualizzarne i dettagli o monitorare lo stato dei treni coinvolti in tempo reale. Nello specifico, tramite i dettagli si possono vedere le stazioni presso cui effettuare eventuali scali e i relativi orari di partenza e arrivo; mentre con il monitoraggio, vengono frequentemente effettuate richieste per aggiornare lo stato dei treni coinvolti, osservandone posizione attuale e ritardo (da cui ricavare l'orario d'arrivo previsto). La figura 4 mostra l'interfaccia grafica sviluppata: nella sezione di sinistra si possono ricercare delle soluzioni di viaggio o controllare lo stato di un treno a partire dal suo codice; nella sezione di destra si possono controllare le soluzioni di viaggio consigliate, visualizzarne i dettagli e monitorarle in tempo reale (i valori monitorati sono il ritardo e l'ultima stazione per cui è passato uno specifico treno). Il tempo d'aggiornamento del monitoraggio è di 1 minuto.

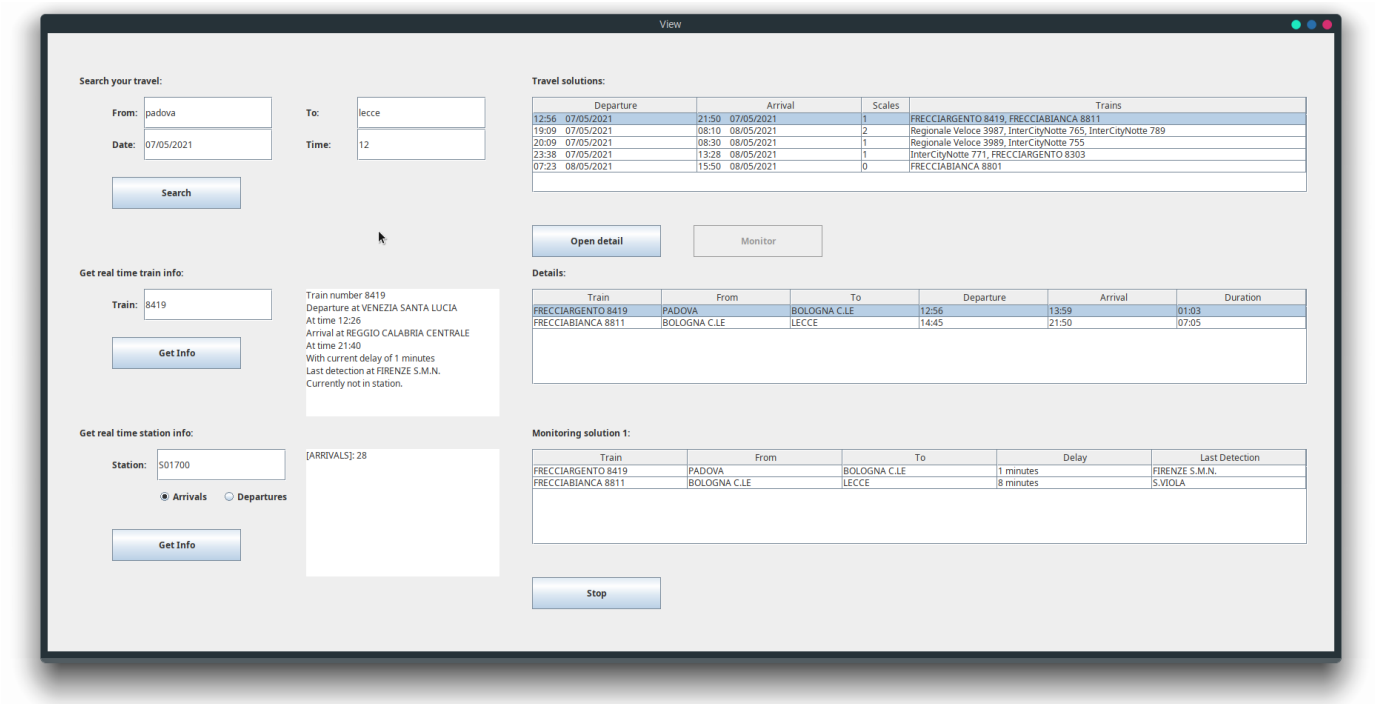

Figura 4: Screenshot della GUI del programma.

3 Word Counter: approccio reattivo

3.1 Introduzione

In questa parte dell'assignment è richiesta una soluzione simile a quella consegnata in precedenza, ma che sfrutti la programmazione reattiva, in particolare il framework JavaRX.

Il programma si presta particolarmente all'approccio reattivo in quanto è possibile identificare in maniera chiara il **reactive stream**, ossia il flusso asincrono di elaborazioni che, partendo dai singoli file, porta ad ottenere le n parole più frequenti con il relativo numero di occorrenze.

3.2 Design

Per favorire la leggibilità della soluzione e massimizzare le prestazioni, si è deciso di non utilizzare il pattern MVC alleggerendo l'architettura complessiva, che presenta 4 componenti principali:

- **Flow** che contiene il codice ad *assembly-time*, ossia tutta la componente dichiarativa che spiega come i dati vengono elaborati e trasformati lungo la catena di operatori;
- **FlowableOperations** che è la classe statica contenente tutte le operazioni da effettuare sui dati;
- **ViewListener** che è il componente che funge da listener per la View e da *Subsciber* per il flow. Alla pressione del pulsante Start, istanzia un nuovo Flow e richiede il Flowable al quale si sottoscrive;
- **View** che viene costantemente aggiornata dal ViewListener ad ogni emissione di dato dalla sorgente.

La gestione dei dati scambiati fra i componenti descritti (ad esclusione di `FlowableOperations`, i cui metodi vengono semplicemente invocati come chiamate asincrone all'interno del flow) viene rappresentata dal diagramma di sequenza in fig. 5.

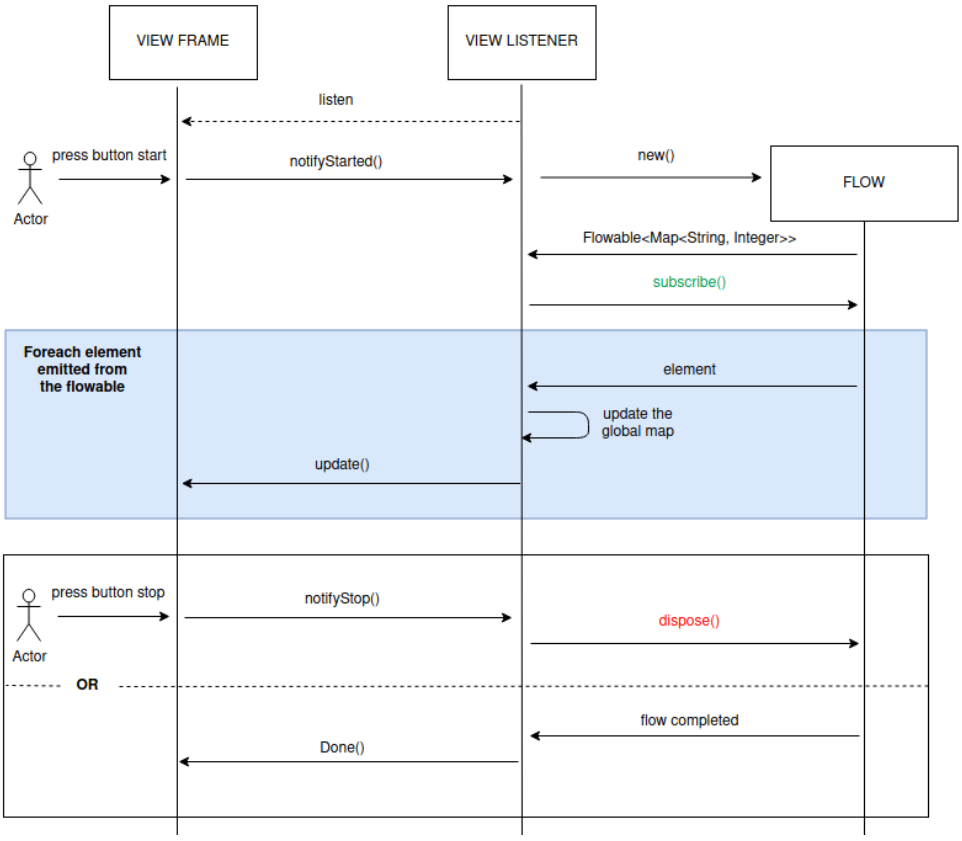

Figura 5: Diagramma di sequenza rappresentante le interazioni fra i 3 componenti principali del programma

3.2.1 Flowable Design

Per quanto riguarda il flusso generato all'interno di `Flow`, si è deciso di realizzare un Flowable *cold*, che quindi inizia ad emettere dati solo nel momento in cui qualcuno si sottoscrive al flusso. Questo permette di iniziare e stoppare la computazione semplicemente effettuando le chiamate *subscribe()* e *dispose()*, come mostrato in fig. 5.

Si riporta in fig. 6 una rappresentazione grafica di come i dati vengono trasformati lungo la catena di operatori.

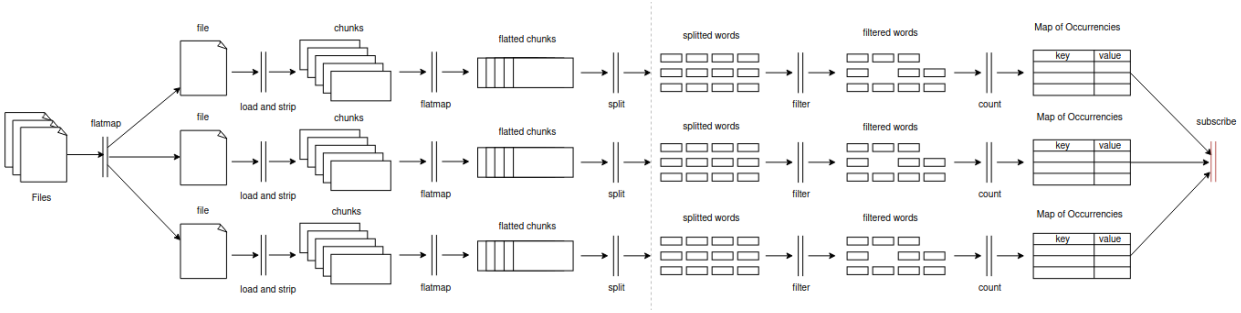

Figura 6: Flow dei dati

Come si evince dallo snippet di codice sottostante e dalla fig. 6, la sorgente viene creata a partire dalla lista di File. Successivamente, grazie alla *flatMap* e all'operatore *just()* si vanno a creare tanti Flowable quanti sono i file, permettendo di parallelizzare tutte le operazioni successive, sfruttando il pool di thread **Schedulers.computation()**.

Come per il primo assignment, le azione di *strip*, *split*, *filter* e *count* vanno eseguite nell'ordine specificato, perciò le ritroviamo in questo ordine anche nella dichiarazione del flow. In particolare, tramite la *loadAndGetChunk()* si suddivide il file in chunks ottenendo quindi un `Flowable<List<String>>` che successivamente viene "appiattito" sfruttando la flatMap.

Dopodiché, ogni stringa viene splittata in singole parole tramite la *split()*, le quali vengono poi filtrate secondo i criteri stabiliti nel file scelto in fase di configurazione.

Infine, tramite l'operazione di *count()*, per ogni chunk, viene creata una `Map<String, Integer>` con il numero di occorrenze relativo a ciascuna parola trovata nei pdf.

Flusso dichiarativo delle operazioni di trasformazione dei dati

```
public Flowable<Map<String,Integer>> getMapsFlowable() {
    Flowable<File> source = Flowable.fromIterable(documents);

    return source.flatMap(s->Flowable.just(s)
            .subscribeOn(Schedulers.computation())
            .map(FlowableOperations::loadAndGetChuncks)
            .flatMap(Flowable::fromIterable)
            .map(FlowableOperations::split)
            .map(FlowableOperations::filter)
            .map(FlowableOperations::count));
}
```

3.2.2 Subscriber

Il Subscriber, nel nostro caso il `ViewListener`, riceve dalla sorgente il dato "confezionato" ossia le mappe delle occorrenze di ogni chunk. A questo punto, il suo compito è quello di raggruppare queste mappe in una mappa globale, ordinare le varie entry in base al numero di occorrenze e, se si sono verificati degli aggiornamenti nelle prime n parole, aggiornare la GUI.

In caso di errore, questo viene notificato all'utente, mentre in caso di completamento viene stampato il messaggio "DONE" e chiamato il corrispondente metodo sulla `View` che va a disabilitare il pulsante di stop.

3.3 Valutazione delle prestazioni

Per valutare le prestazioni di questa versione con programmazione reattiva, si è deciso di confrontarla con le versioni precedentemente sviluppate e fare un confronto finale.

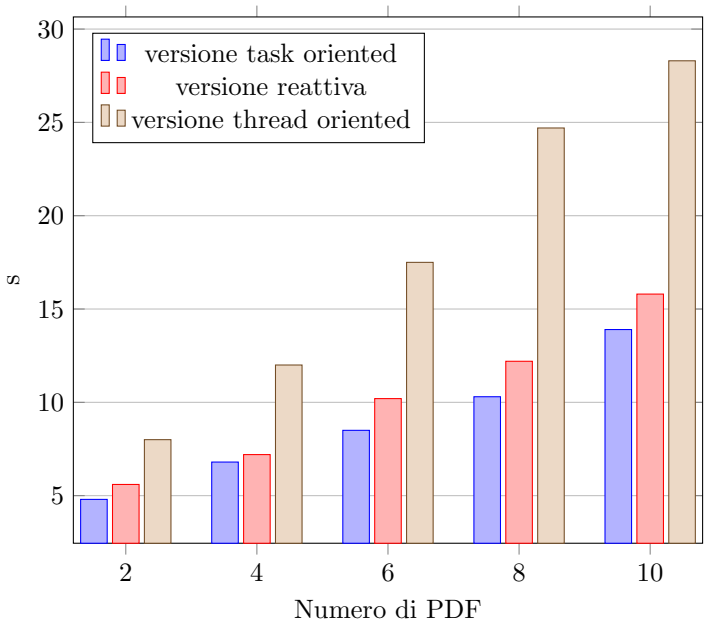

Figura 7: Confronto dei tempi delle tre versioni realizzate

Come si evince dal grafico, i tempi ottenuti con programmazione a task e reattiva sono decisamente migliori rispetto a quelli ottenuti con approccio a thread. Questo incremento delle prestazioni è da imputare sia alla natura stessa degli approcci asincroni che alla migliore scomposizione del lavoro che è stata introdotta nelle due nuove versioni. Inoltre, considerare ogni pagina del pdf come un chunk ha portato ad una maggiore reattività dell'interfaccia grafica.

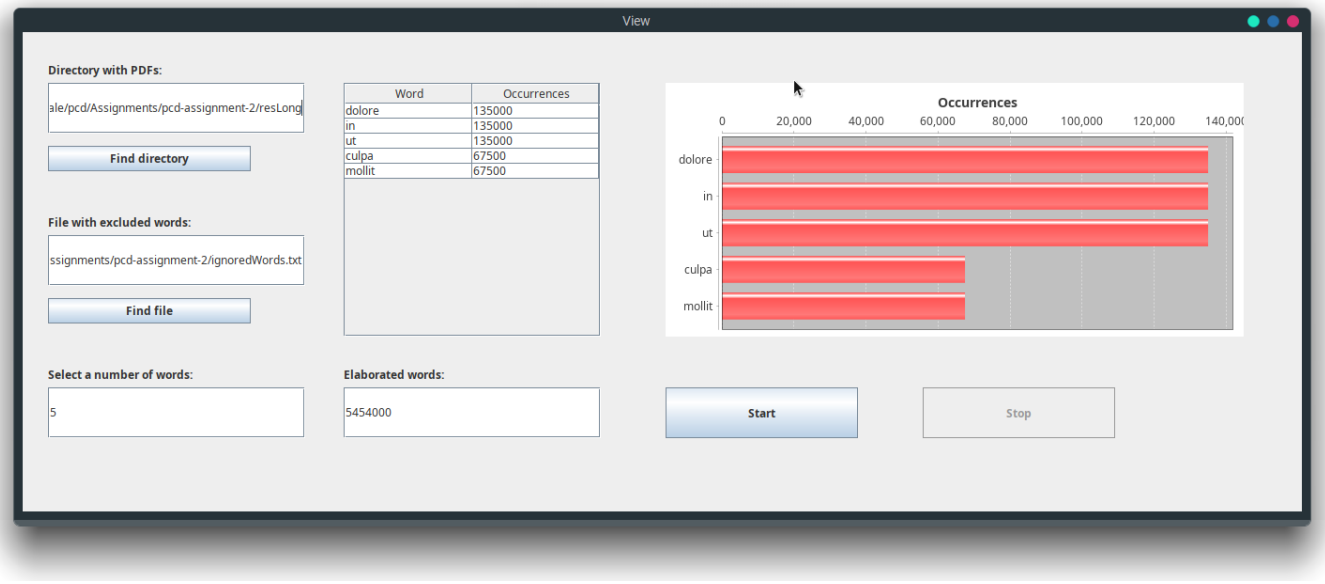

Figura 8: Schermata della GUI dei punti 1 e 3.

Si riporta infine, in fig. 8 una schermata dell'interfaccia grafica relativa ai punti 1 e 3 dell'assignment.

Conclusioni

In questo assignment sono stati esplorati 3 diversi approcci asincroni, a task, a event-loop e reattivo. Durante lo svolgimento dell'elaborato, sono emerse le potenzialità di ognuno di essi, e si è appreso quando è più opportuno adottare l'uno o l'altro approccio. Per ulteriori analisi, il codice sorgente dell'elaborato è reperibile nel repository pubblico `https://github.com/SimoneRomagnoli/pcd-assignment-2` .