PCD Assignment 2

Introduzione al problema

Si richiede di implementare un tool che permetta, dato un insieme di file di testo, di contare le occorrenze di ogni parola presente all'interno di essi. L'operazione deve essere fatta parallelizzando la lettura e l'analisi dei file.

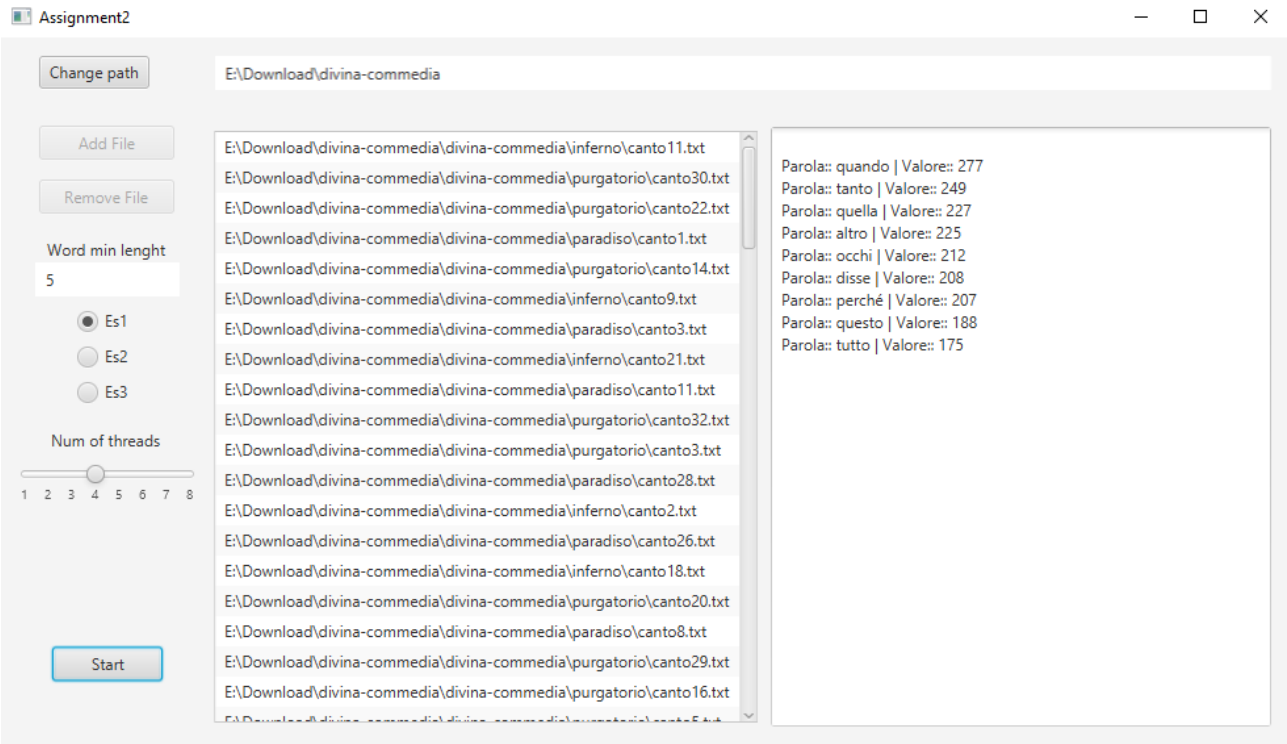
Durante la computazione si vuole inoltre mostrare, "in tempo reale" la lista aggiornata delle k parole più frequenti, con lunghezza > n, affiancate dal numero di occorrenze.

Si richiede di svolgere il lavoro in tre modalità diverse:

1. Utilizzando Task ed Executor.
2. Con la programmazione asincrona (Event loop).
3. Con la programmazione reattiva (Stream).

GUI

Si è preferita un'interfaccia grafica, rispetto ad una a linea di comando. La GUI permette di scegliere una cartella dalla quale iniziare la ricerca di file di testo e altri parametri di "setup", come l'esercizio, il numero di thread da usare, e la lunghezza minima delle parole da includere nel conteggio. Scelta la configurazione si può dare inizio alla computazione con il tasto ***start,*** che si trasformerà poi in ***stop*** in modo da poter essere premuto per interrompere l'esecuzione. Durante la computazione, inoltre, sarà possibile modificare il set di file mediante i tasti ***add file*** e ***remove file***.



Esercizio 1: Task e Executor

Task

La fase di ricerca dei file di testo è stata gestita mediante un Task ricorsivo di nome ***FolderSearchTask***. Esso prende come parametro un ThreadPoolExecutor e, quando trova un file di testo, fa la *submit* di un ***ReadFileTask***, che si occupa della lettura e dell'analisi. Quando, invece, si imbatte in una sottocartella fa una *fork* su un FolderSearchTask.

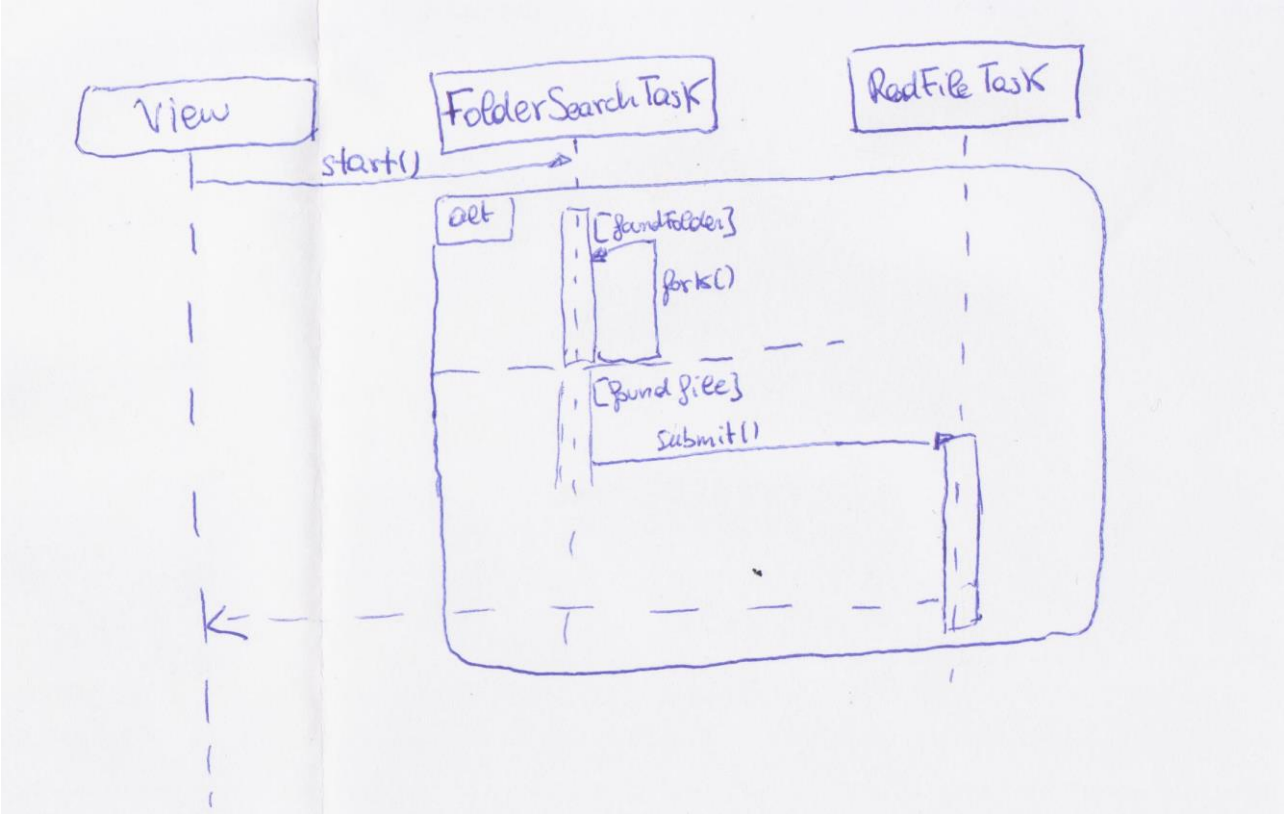
Il ReadFileTask prende come parametro il path di un file di testo; lo legge e ritorna una mappa che ha come chiave le parole più lunghe di n trovate nella porzione analizzata e come valore le occorrenze di queste ultime.

Executor

Sono stati utilizzati due Executor: uno di tipo ***ThreadPoolExecutor*** per la gestione dei readFileTask e uno di tipo ***ExecutorService*** che ha il compito di attendere i risultati dei task lanciati all'interno del threadPoolExecutor.

Quando l'ExecutorService riceve un risultato si provvede ad aggiornare la mappa principale delle occorrenze e il set dei file analizzati. Dopodiché si aggiorna la GUI visualizzando la nuova lista delle k parole più frequenti e il relativo conteggio.

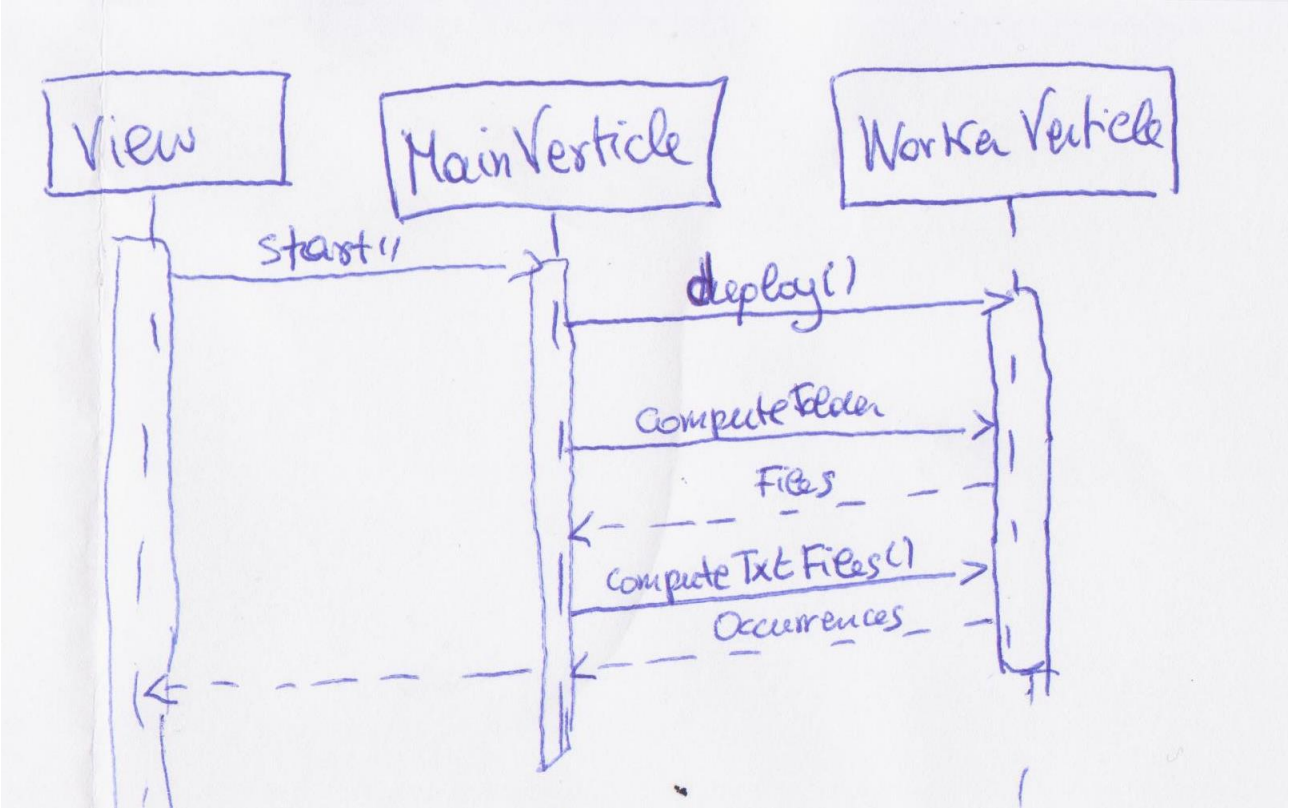
Nel caso della AddFile e della RemoveFile il comportamento è analogo, con l'unica differenza che nel caso della remove, quando si aggiorna la mappa principale tramite la merge, le occorrenze vengono sottratte anziché sommate.



Esercizio 2: VertX Event loop

La struttura del programma ruota attorno a due Verticle: mainVerticle e workerVerticle.

All'avvio dell'applicazione viene creato il mainVerticle, che gestisce l'event loop. Esso, a sua volta, crea un workerVerticle, che ha al suo interno un pool di thread sotto forma di WorkerExecutor. Successivamente il mainVerticle chiama un metodo all'interno del workerVerticle che procede con la ricerca dei file di testo e restituisce, tramite Future, i path dei file individuati. Per ogni file di testo individuato, viene chiamato un altro metodo di workerVerticle che lo elabora e ritorna una mappa delle occorrenze. Man mano che riceve i risultati, il mainVerticle procede ad aggiornare il model e la view.

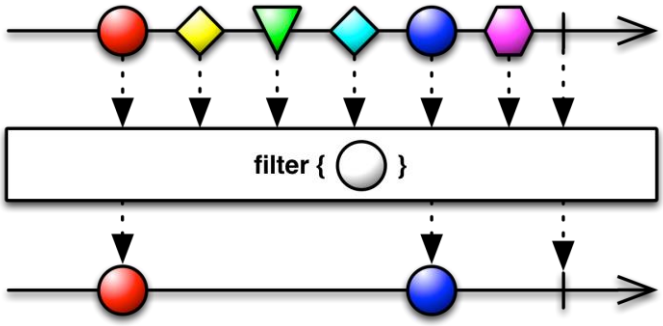


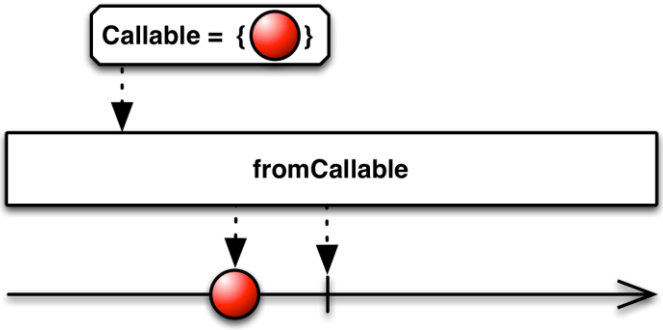
Esercizio 3: Reactive programming - RxJava

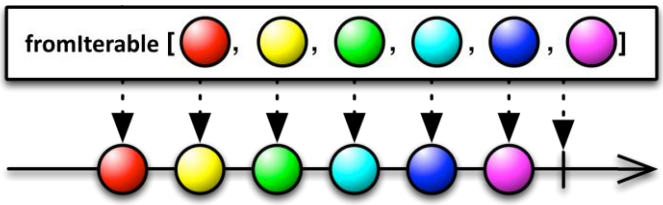
Alla pressione del pulsante start, viene creato un ExecutorService. Esso viene utilizzato per parallelizzare l'elaborazione dei flussi asincroni di dati che porta all'individuazione dei file di testo presenti nell'albero di cartelle che ha come radice la cartella selezionata. L'elaborazione viene eseguita mediante una chiamata ricorsiva a SearchTextFiles, che genera uno stream di Path.

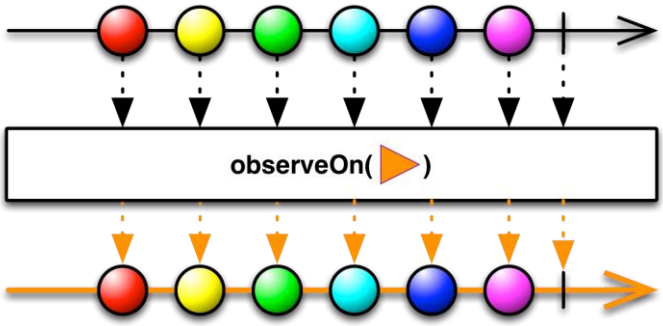
Per ogni Path si procede (sempre in maniera asincrona e utilizzando il pool di thread) ad analizzare i file ricavando le occorrenze, a estrarre le parole più frequenti e ad aggiornare la view di conseguenza.

Riportiamo tutti i diagrammi di Marble dei relativi operatori utilizzati durate lo sviluppo della terza parte dell'Assignment.


filter { }


Callable = { }
fromCallable


fromIterable [, , , , ,]


observeOn( )

Tempi di esecuzione e speedup

Sono state effettuate prove sullo stesso sottoalbero di cartelle, ed è stata calcolata una media dei tempi di esecuzione per ogni esercizio. Ne è risultato che l'implementazione che utilizza Task ed Executor è la più veloce, mentre con la programmazione reattiva si riesce ad ottenere il valore più alto di speedup.

	Single core ex time (ms)	**8 cores ex time (ms)**	**Speedup**
ES 1	12,7106	6,6054	**1,92**
ES 2	15,0160	4,7700	**3,15**
ES 3	13,4080	2,8085	**4,77**