PCD Assignment #02 - Asynchronous Programming

PRIMA PARTE

Nella prima macro parte si richiede di affrontare il problema illustrato nel primo assignment (entrambi i punti) utilizzando un approccio asincrono basato su task (Framework Executors).

SOLUZIONE:

Per la risoluzione della prima parte siamo partiti dalla soluzione dell' Assignment-01 e abbiamo ritenuto corretto implementare le seguenti classi:

pcd/part1/simengine_conc/ExecutorStepService.java → questa classe ha preso il posto della vecchia architettura "master and worker" e viene richiamata dalla classe *AbstractSimulation,* in tal modo non abbiamo dovuto cambiare eccessivamente la soluzione precedente.

La classe *ExecutorStopService* ha il compito di, in ogni step, di istanziare l'executor con dimensione fissa di thread che grazie al metodo execute() gestirà l'esecuzione del metodo step() utilizzando il numero di thread fissi, newFixedThreadPool(nWorkers).

La sincronizzazione è stata gestita attraverso i metodi dell'esecutore:

```
executor.shutdown();
executor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
```

- *executor.shutdown()*: Questo metodo chiude l'ExecutorService. L'executor non accetterà ulteriori compiti, ma completerà quelli già accettati. I thread nel pool non verranno terminati dopo aver eseguito i compiti correnti, ma aspetteranno che vengano invocati e completati nuovi compiti.

- *executor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS*: Questo metodo blocca il thread corrente fino a quando tutti i compiti hanno completato l'esecuzione dopo una chiamata a shutdown() o fino a quando si verifica una interruzione, ovvero fino a quando l'executor non viene terminato. Il Long.MAX_VALUE in questo contesto specifica il tempo massimo di attesa, che è un valore molto grande, quindi praticamente significa "attendi indefinitamente". Questo metodo restituirà true se tutti i compiti sono terminati correttamente prima della scadenza del tempo massimo di attesa, altrimenti restituirà false.

Un altro cambiamento rilevante è stato effettuato nella gestione della GUI reattiva dove abbiamo inserito un executor con un single thread:

```
executor.execute( new Thread(() -> {
    simulation.setup();
    view.display();

    stopFlag.reset();
    simulation.run(nSteps, stopFlag, true);
    gui.reset();

}));
```

SECONDA PARTE

Si vuole realizzare una sistema che, una dato un indirizzo web, una parola e un valore p, restituisca un report contenente l'elenco delle pagine che contengono quella parola e l'occorrenza della parola nella pagine, a partire dall'indirizzo specificato e considerando le pagine collegate, ricorsivamente, fino a un livello di profondità pari al valore p.

SOLUZIONE VIRTUAL THREAD:

La soluzione con i Virtual Thread è stata sviluppata creando un'unica task chiamata "RecursiveWordCountTask" che implementa l'interfaccia "Runnable".
Essendo ricorsiva la prima istanza viene chiamata all'interno del metodo *getWordOccurences()* per poi richiamarsi e una volta finito il compito eseguire la *join(),* finché non avrà finito di ispezionare tutti i sottolink del problema.

```
Thread vt = Thread.ofVirtual().unstarted(new
RecursiveWordCountTask(entryPoint,word, depth,
result,pattern,monitor));

vt.start();
vt.join();
```

Rilevante per il funzionamento è il metodo *openConnection()* della libreria **java.net.HttpURLConnection** che essendo una funzione bloccante smonta il thread logico dal thread fisico per sostituirlo ad un'altro e cosi ricorsivamente, rendendo così il programma molto più performante.

SOLUZIONE EVENTI/EVENT-LOOP:

Per quanto riguarda la soluzione con l'utilizzo di event loop abbiamo scelto di utilizzare la libreria Vert.X di java. Sono state create due classi "verticle" (con il loro event loop dedicato) la prima ha il compito di trovare tutti i sottolink dopo di che attraverso l'event bus comunica con la seconda passandogli la lista di sottlink, quest'ultima inizierà a contare le parole di ogni pagina e inserire la coppia (link, ricorrenze della parola) all'interno del report che verrà poi restituito.
Il metodo che è stato utilizzato per rendere asincrona la soluzione è *executeBlocking()* e il rispettivo *onComplete()* Vert.X all' interno del quale in entrambi i verticle sono stati passati i metodi sotto forma di callable(runnable che restituiscono un dato), nel caso dei sottolink viene restituita una lista di stringhe mentre nel conteggio delle parole un intero.
Utilizzando pochi verticle (non ci sembrava corretto scomodare più event loop) abbiamo inserito un executor per rendere parallela e quindi più veloce la ricerca dei link.

È stata sviluppata prima una soluzione senza eventBus, con un solo verticle, che però era interamente rivestita da un ciclo while dal quale si usciva solo una volta che tutti i link erano stati visualizzati.
Ma non è riuscita ad andare in porto però perché avendo le chiamate asincrone (*executeBlocking()*) all'interno di un while e i requisiti per uscirci all'interno dell'*onComplete()* sempre nel while, l'*onComplete()* non veniva mai eseguito perché evidentemente aveva bisogno di uscire dal while ma essendo il requisito per uscire al suo interno il tutto diventava un ciclo infinito.

SOLUZIONE PROGRAMMAZIONE REATTIVA
La soluzione reattiva è stata implementata grazie all'uso della libreria "*reactivex.rxjava3"* che mette a disposizione metodi e strutture dati per gestire flussi di dati asincroni che si innescano, come ad esempio (Observable, Observer, Operators, Schedulers, Backpressure).

Il metodo *getWordOccurrences(...)* all'interno crea un flusso di dati su sui è possibile sottoscrivere:

```
crawlRecursive(entrypoint, word, depth, result)
        .subscribeOn(Schedulers.io())
        .doFinally(() -> latch.countDown())
        .subscribe();
```

- *subscribeOn(Schedulers.io())*: utilizzato per specificare che il flusso di lavoro sarà gestito da uno scheduler I/O che utilizza un pool di thread dedicato per eseguire operazioni I/O-bound. Questo metodo garantisce che l'operazione di "ricerca e conteggio" sia eseguita in modo asincrono su thread separati, liberando il thread principale per altre attività.

- Per attendere il completamento del flusso è stato utilizzato un istanza di CountDownLatch e il thread di *getWordOccurency(...)* viene messo in wait fino a quando il contatore latch del flusso non è terminato.

- *Subscribe()* invece avvia l'esecuzione del flusso reattivo. Dopo aver sottoscritto al flusso, il codice nel flusso reattivo inizia ad eseguire.

CrawlRecursive() è un metodo privato che torna un Observale, quindi un flusso di List<String>, per ogni elemento emesso viene fatta una chiamata su un altro flusso reattivo che poi verrà ricombinato con i risultati degli altri flussi in un unico flusso principale.

```
Observable.fromCallable(() -> {..IMPLEMENTAZIONE..})
        .subscribeOn(Schedulers.io())
        .flatMap(subLinks ->
        Observable.just(subLinks)
                .concatWith(Observable.fromIterable(subLinks)
                        .subscribeOn(Schedulers.io())
                        .flatMap(subLink -> crawlRecursive(...))
                        .reduce(..in LISTA..))
```

All'interno di flatMap, vengono eseguite le seguenti operazioni:

- Si crea un flusso Observable (Observable.just(subLinks)) contenente i link trovati sulla pagina e l'occorrenza della parole.
- Per ciascun link, viene eseguita ricorsivamente la funzione crawlRecursive per esaminare i link di secondo livello.
- I risultati delle chiamate ricorsive vengono ridotti in un'unica lista.

IMPLEMENTAZIONE GUI:

Abbiamo sviluppato una piccola GUI con il metodo MVC e l'abbiamo resa reattiva creando un thread (Event dispacher Thread) che continuamente chiama il metodo update della model ogni 0.1s e aggiorna l'interfaccia utente della view.

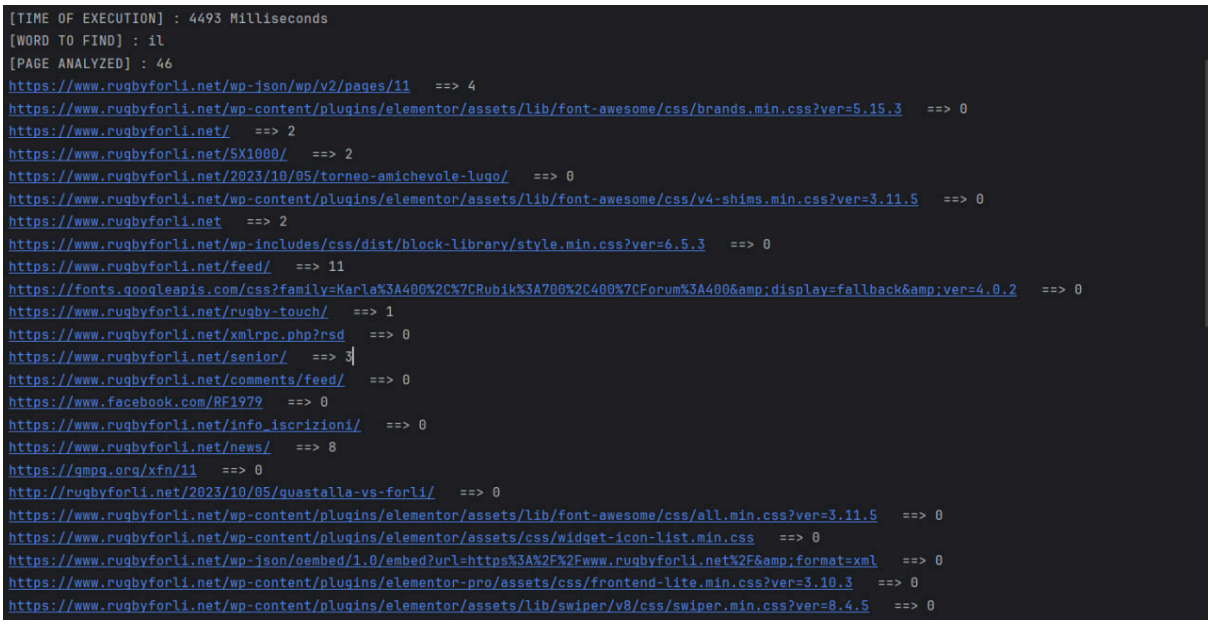
Si è scelto di creare due package distinti per la soluzione cli e GUI perchè nella prima come richiesto dalla consegna viene restituito un Report mentre per quanto riguarda la GUI dovendo essere reattiva i record vengono stampati sull' interfaccia appena sono disponibili.

PERFORMANCE:

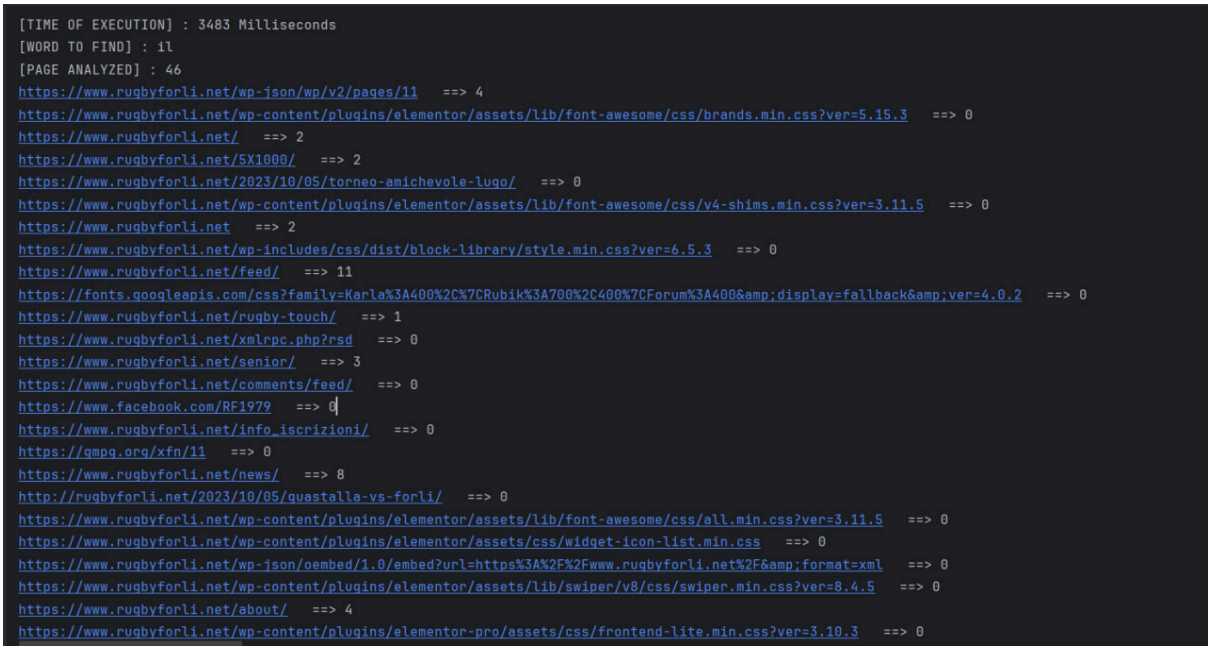
Le performance sono state testate solamente a linea di comando, connettendosi sempre allo stesso entrypoint con profondità "1".

I risultati ottenuti sono stati i seguenti, partendo da una **versione sequenziale** che ha impiegato **37s** :

La versione **Virtual Thread** ha impiegato **4s** registrando uno speed up di circa 9.



La versione **Reattiva** ha impiegato **3s** registrando uno speed up di circa 12.



La versione a **eventi/event-loop** ha impiegato **20s** registrando uno speed up di circa 2.