RELAZIONE ASSIGNMENT 2 DI PROGRAMMAZIONE CONCORRENTE E DISTRIBUITA

Bedei Andrea(matricola 0001126957)
Bertuccioli Giacomo Leo(matricola 0001136879)
Mazzotti Alex(matricola 0001146685)
Notaro Fabio(matricola 0001126980)

30 Aprile 2024

Indice

Capitolo 1

PARTE 1

1.1 Introduzione

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

1.2 Strategia risolutiva

La vecchia soluzione, molto brevemente, sfruttava:

- alcune barriere per garantire sincronizzazione e per accertarsi di rispettare l'ordine previsto dei vari step → prima step dei semafori, poi fase sense e decide delle macchine e solo dopo fase act delle macchine
- un monitor per fermare la simulazione in modo corretto.

Nella nuova versione sono stati eliminati i file relativi ai thread delle car e dei traffic light (`CarAgentThread` e `TrafficLightThread`), in quanto non più usati.

E' stato dunque sufficiente agire su `SimThreadsSupervisor`, nel quale non vengono più creati direttamente ed esplicitamente tutti i thread necessari ma ci si basa sugli executor.

In particolare il supervisor contiene un executor (di tipologia `newFixedThreadPool` con numero di thread specificato) al quale è demandata l'esecuzione dei cicli e dunque il susseguirsi delle iterazioni.
Più in particolare, osservando il metodo `cycle()` possiamo notare che per rispettare l'ordine degli step delle varie entità (sia macchinine che semafori) non sfruttiamo più le barriere ma sfruttiamo una tecnica basata sull'operazione `get`, che ricordiamo essere bloccante.

Dunque il ciclo del supervisor può essere riassunto nei seguenti passi:

- prima viene svolto lo step di tutti i semafori (parallelamente)
- si attende il completamento di tutti gli step di tutti i semafori tramite la `get` di tutte le future generate
- analogamente viene fatto per la fase di sense/decide di tutte le macchinine
- infine ugualmente viene fatto lo stesso anche per la fase di act di tutte le macchinine.

Mentre nell'approccio precedente occorreva coordinare più thread attraverso le barriere per completare un'iterazione, adesso è soltanto il supervisor che avvia tutti i task e passa all'iterazione successiva solo una volta che è passato un dato lasso temporale specificato dalla funzione `syncWithWallTime()`.

Per quanto invece concerne il monitor, esso veniva utilizzato nella precedente soluzione per implementare correttamente lo stop della simulazione. Avevamo inizialmente pensato di interrompere la simulazione direttamente attraverso lo shutdown dell'executor, tuttavia ciò avrebbe comportato di dover ricreare l'executor stesso ad ogni lancio della simulazione, pertanto abbiamo preferito mantenere il monitor, che con semplicità e correttezza garantisce mutua esclusione in fase di controllo e segnalazione dello stop della simulazione.

Per meglio comprendere il comportamento del sistema si faccia riferimento al diagramma seguente:

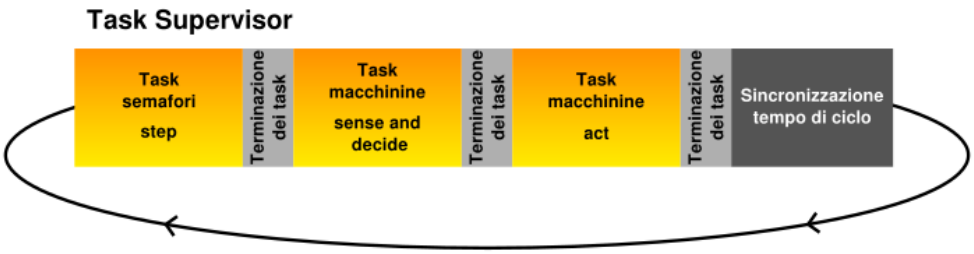

Figura 1.1: schema riassuntivo del ciclo svolto dal supervisor.

Capitolo 2

PARTE 2

2.1 Introduzione

La seconda parte dell'assignment prevedeva di realizzare un sistema che, dato un indirizzo web, una parola e un valore p, restituisse 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.
Tale problema doveva in particolare essere risolto sfruttando i seguenti approcci tipici della programmazione asincrona:

- Virtual Thread
- Eventi/Event-Loop
- Programmazione Reattiva.

2.2 Virtual thread

Per favorire il riuso di codice, abbiamo come prima cosa definito un'interfaccia comune per i tre esercizi denominata `WebCrawlerResult`, la quale permette di avere un tipo di ritorno comune per i risultati indipendentemente dall'approccio adottato.

2.2.1 Strategia risolutiva

Siccome non veniva specificato nella consegna quale implementazione di virtual thread utilizzare, noi abbiamo ritenuto opportuno che i virtual thread non venissero utilizzati esplicitamente e direttamente, ma abbiamo preferito utilizzarli all'interno di un executor di tipo
`newVirtualThreadPerTaskExecutor` (come comunque mostratoci in un esempio in classe).

Per quanto invece concerne nello specifico la nostra soluzione che usa i virtual thread, le due classi principali della soluzione sono:

- `WordCounterImpl` → ricopre il ruolo di gestore dell'executor, ovvero crea il primo task (di fatto avviando l'intero processo), lo sottomette, attende il suo completamento e lo può anche interrompere
- `WebCrawlerWithVirtualThread` → rappresenta il task ricorsivo di crawling della pagina web.

In particolare `WordCounterImpl` usa un executor di tipo `newVirtualThreadPerTaskExecutor`, il quale è un tipo particolare di executor in grado di avviare un nuovo virtual thread per ogni task e offrire un numero non limitato di thread disponibili.

Una volta creato l'executor, viene sottomesso il primo task (crawling dell'indirizzo iniziale) grazie ad un oggetto di tipo `WebCrawlerWithVirtualThread` ed in aggiunta è anche presente il metodo `stop()` che richiama il metodo `shutdown()` sull'executor, impedendogli di ricevere nuovi task ma comunque completando quelli già sottomessi.

Per quanto invece riguarda la classe `WebCrawlerWithVirtualThread`, essa implementa `Callable` in modo da poter essere usata come task dal metodo `submit()` dell'executor service.

Tale classe, partendo dal metodo `call()` permette di eseguire concretamente il crawling delle pagine ricorsivamente, operazione riassumibile nei seguenti passi:

- prima si aggiunge la pagina corrente ad un set condiviso che tiene traccia dei link già esplorati
- si esegue poi la connessione al web address desiderato
- si contano le occorrenze
- se il numero di occorrenze è maggiore di zero si utilizza un consumer per restituire il risultato
- se non siamo arrivati alla profondità massima richiesta vengono poi individuati tutti i link presenti nella pagina corrente e ciascuno viene sottomesso in un task differente all'executor → le future che conterranno i risultati vengono collezionate in una lista e l'operazione si ripete ricorsivamente
- una volta che tutti i task sono stati sottomessi (cioè è stata richiesta la visita di tutte le pagine) si fa la get per ogni elemento della lista di future, in modo da attenderne i risultati.

E' opportuno sottolineare che il set che tiene traccia di tutti i link già visitati dev'essere condiviso tra più task, pertanto abbiamo ritenuto vantaggioso implementarlo tramite le collezioni thread-safe che Java mette a disposizione.

Per meglio comprendere la struttura della strategia risolutiva si faccia riferimento allo schema sotto riportato:

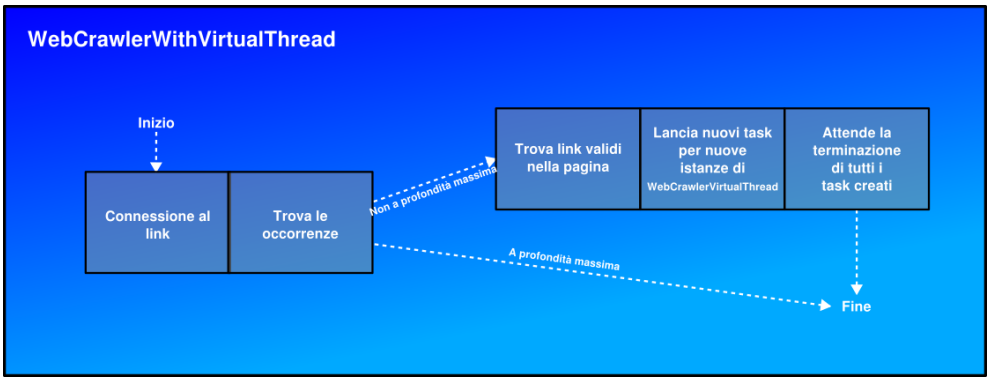

Figura 2.1: schema riassuntivo del task di crawling svolto da un oggetto WebCrawlerWithVirtualThread.

2.2.2 Descrizione GUI

Per quanto concerne la versione con GUI dell'approccio con virtual thread occorre evidenziare principalmente due aspetti:

- il bottone stop consente di interrompere la ricerca richiamando il metodo `stop()` della classe `WordCounterImpl`
- grazie all'utilizzo del consumer in `WordCounterImpl`, possiamo specificare un differente comportamento a seconda delle due modalità (console o GUI) → nella prima il consumer stampa le occorrenze, nella seconda invece le mostra nella GUI (anche attraverso il metodo `invokeLater()` che permette reattività).

2.3 Eventi/Event-Loop

Utilizziamo Vertx per implementare la nostra soluzione.
Tale framework si basa sull'event loop, il motore principale di Vertx, responsabile della gestione degli eventi e dell'esecuzione dei verticle.
Quando un evento si verifica, esso viene messo in coda affinchè l'event-loop possa gestire l'evento.

2.3.1 Strategia risolutiva

La prima cosa da fare è stata la creazione di un Vertx, su cui viene fatto il deploy di un verticle(inteso come un pezzo di codice reattivo e orientato agli eventi, utilizzato per gestire task specifici all'interno dell'applicazione), implementato nella classe `VerticleSearch`.
A tale classe passiamo i soliti parametri della ricerca che si vuole svolgere ed un consumer (sempre per specificare l'azione da fare all'arrivo di un risultato).
Tale classe avvia la ricerca attraverso il metodo `start()`, chiamato implicitamente con il deployment del verticle, in cui è presente un consumer in ascolto sull'event-bus (più precisamente sull'address *my-topic*) che manda in esecuzione il metodo `crawl()` per ogni messaggio pubblicato sull'event-bus.

La ricerca parte quindi pubblicando il primissimo link sull'event-bus.
Il metodo `crawl()` richiama il metodo `executeBlocking()` sul vertx corrente in modo che il codice al suo interno venga eseguito in modo concorrente, anche se in questo caso avendo un solo verticle l'esecuzione è fatta su un singolo thread dunque non vi è parallelismo.
Per ogni ricerca viene decrementato il numero di ricerche rimanenti e vengono fatte le stesse operazioni degli approcci precedenti.
La differenza sostanziale rispetto agli approcci precedenti consiste nel fatto che si chiama il consumer passandogli come risultato il numero delle occorrenze e per ogni link trovato nella pagina corrente viene incrementato il numero di ricerche rimanenti e pubblicato sull'event-bus.
Se il verticle si accorge che non ci sono più ricerche da fare esegue la complete della promise, che causa la terminazione del programma.

2.3.2 Versione alternativa

Si noti che nella versione di `VerticleSearch` sopra descritta tutti i messaggi riguardanti la scoperta dei link da visitare sono pubblicati sullo stesso address sull'event-bus (*my-topic*) e vi è un unico verticle che legge e scrive su di esso.
Ciò può comportare lentezza e performance limitate.

Pertanto, nel tentivo di esplorare una possibile alternativa, abbiamo realizzato una seconda versione di `VerticleSearch` chiamata `VerticleSearch_v2`, la quale cambia la struttura affinchè ci siano più verticle che comunicano attraverso diversi address sull'event-bus, in modo che essi possano spartirsi l'esplorazione dei link che mano a mano emergono.
Implementando la seconda versione con più verticle e conseguentemente con piu thread (siccome ciascun verticle sfrutterà un thread differente), otteniamo un miglioramento di performance notevole (diminuzione del tempo di

esecuzione di 6 volte utilizzando 5 verticle invece che uno solo).

Per meglio comprendere la differenza tra le due versioni si faccia riferimento allo schema seguente:

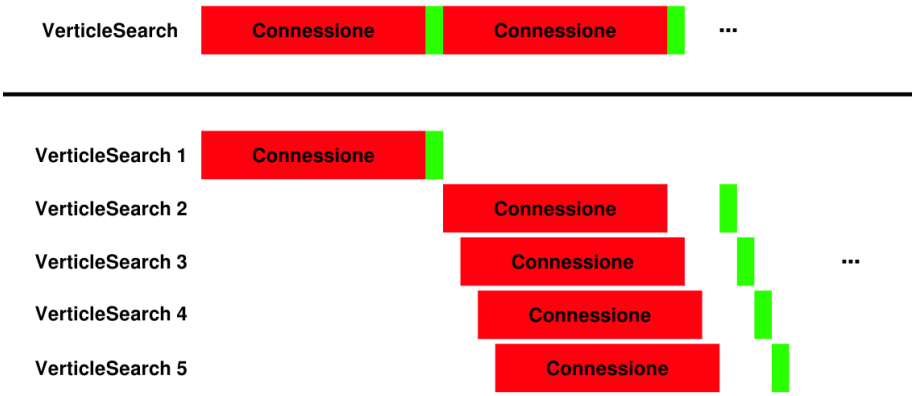

Figura 2.2: schema riassuntivo che mostra le differenze tra le due versioni di VerticleSearch, si noti che le operazioni evidenziate di rosso sono quelle eseguite concorrentemente in quanto lente.

Di seguito riportiamo le caratteristiche principali di questa versione alternativa:

- abbiamo alcune strutture concorrenti condivise tra i diversi verticle (ciò sarebbe un'eccezione siccome l'approccio ad eventi prevederebbe solo comunicazione tramite eventi, tuttavia per semplicità, leggibilità e dato lo scopo educativo del progetto abbiamo mantenuto questo approccio)
- vengono creati più verticle e dunque occorre anche una lista di `Future`
- a ciascun verticle vengono passati non solo i parametri di ricerca ma anche le strutture condivise citate sopra
- la differenza con l'approccio precedente risiede nel fatto che una volta individuati i link in una pagina, essi vengono pubblicati sullo stesso event-bus ma con address differente, in modo tale da distribuire equamente il carico di lavoro tra i vari verticle
- in questa versione cambia totalmente la modalità di completamento della promise → mentre nella versione precedente utilizzavamo un contatore che teneva traccia del numero di ricerche rimanenti da effettuare, tale approccio è inadatto quando si hanno più verticle (siccome tale

contatore verrebbe aggiornato solo localmente) → pertanto è stato necessario introdurre altri address dedicati all'incremento e decremento delle ricerche rimanenti e per la verifica che esse siano pari a zero.

2.3.3 Descrizione GUI

Sia la GUI che la versione console di default implementano la versione con multipli verticle (è comunque possibile passare a qualla con un singolo verticle decommentando il relativo codice).

Nella GUI occorre mantenere la lista di tutti i verticle, in modo che una volta che viene premuto il pulsante stop sia possibile eseguire l'undeploy su tutti i verticle appartenenti a tale lista.

2.4 Programmazione reattiva

2.4.1 Strategia risolutiva

In questo approccio invece viene creata un'istanza di `WebCrwalerWithReactiveProgramming` passandole i soliti parametri di ricerca (indirizzo iniziale, parola e profondità massima).
Su tale oggetto viene richiamato il metodo `crawl()`, il quale restituisce un `Observable<WebCrawlerResult.Result>` su cui poi richiamiamo il metodo `blockingSubscribe()`, che attende sul thread corrente per stampare ogni elemento pubblicato sul flusso.

Analizziamo ora più nel dettaglio la classe `WebCrawlerWithReactiveProgramming`.
In essa inizialmente viene creato un flusso a partire dal link iniziale da visitare e poi per ciascun livello di profondità viene applicata una flatMap nella quale per ciascun elemento del flusso viene creato un ulteriore flusso su cui applichiamo ancora una flatMap con operazione specificata dalla funzione crawler.
Tale funzione ha tipo `Function<Search, Flowable<Search>>` e svolge sostanzialmente i seguenti passi:

- pubblica sul flusso il risultato della pagina correntemente esplorata
- e restituisce il `Flowable<Search>`, ovvero un altro flusso composto dai nuovi link trovati nella pagina corrente che dovranno essere analizzati.

Occorre dunque notare che all'inizio il flusso è di un solo elemento (il link di partenza da esplorare).
A tale flusso viene poi applicata la flatMap che lo trasforma in un flusso di link al livello di profondità successivo.

Per meglio comprendere l'organizzazione dei flussi si faccia riferimento allo schema seguente:

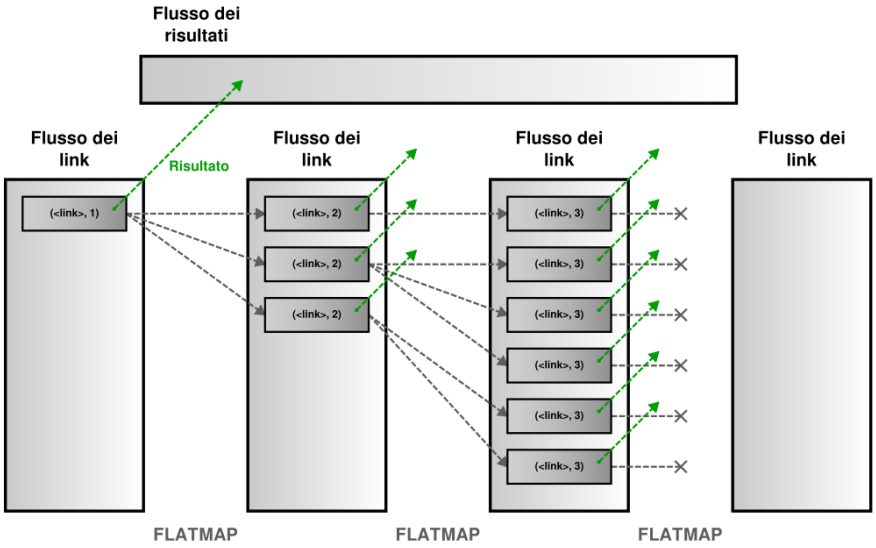

Figura 2.3: schema riassuntivo che mostra la struttura e l'organizzazione generale dei vari flussi usati in un esempio di ricerca con profondità 3.

2.4.2 Descrizione GUI

La GUI non ha alcuna particolarità ma occorre evidenziare che, per aggiornare l'interfaccia con i risultati che mano a mano vengono prodotti, ci si mette in ascolto sul flusso restituito dal metodo `crawl()` attraverso il metodo `subscribe()`.

Capitolo 3

PROVE DI PERFORMANCE

Per verificare le tempistiche dei vari approcci (e dunque individuare il più performante), abbiamo svolto vari test, che prevedevano ricerche a differenti profondità .
I risultati ottenuti sono consultabili nella tabella seguente:

Tabella 3.1: tempi di esecuzione (in millisecondi).

TIPO	**PROFONDITA'**		
	3	**4**	**5**
Virtual Thread	7665	19379	35173
Events Programming (5 verticles)	9735	195485	/
Reactive Programming	8146	27731	136652

Occorre precisare che, per quanto le prove siano state ripetute, tali rilevazioni sono influenzate da vari fattori incontrollabili quali latenza di rete, calo di prestazioni, siti che rifiutano connessioni per motivi di sicurezza ecc, pertanto esse sono da ritenersi solamente indicative.

Comunque dalla tabella possono essere tratte le seguenti considerazioni interessanti:

- sicuramente i virtual thread raggiungono le tempistiche migliori, anche perchè sfruttano al massimo le risorse e la capacità di calcolo della macchina
- come si nota nel caso della programmazione a eventi, anche aumentando di poco la profondità richiesta il tempo necessario per completare la ricerca cresce esponenzialmente → ciò potrebbe essere causato da qualche nostra imprecisione di implementazione o design, ma considerando che la concorrenza è comunque limitata dall'event loop è ragionevole pensare che questo sia l'approccio più lento e meno scalabile.

Capitolo 4

CONCLUSIONI

In conclusione, l'implementazione di questo progetto ci ha permesso di esplorare e confrontare diverse tecniche di programmazione concorrente e asincrona.

Attraverso l'analisi e l'implementazione dei tre approcci distinti per risolvere il problema delle ricerche, abbiamo acquisito una comprensione più approfondita delle caratteristiche, delle sfide e delle prestazioni di ciascun approccio che ci erano state anticipate ed introdotte a lezione.

Una delle principali lezioni apprese è stata l'importanza di adattare l'approccio di programmazione alla natura del problema e ai requisiti specifici di prestazioni, anzichè cercare di adattare le soluzioni già implementate.

Abbiamo poi osservato come l'utilizzo dei virtual thread possa offrire prestazioni superiori in termini di scalabilità e gestione della concorrenza, mentre l'approccio basato sugli eventi e la programmazione reattiva possono fornire un modello più elegante e astratto.

Inoltre, durante lo sviluppo del progetto, abbiamo avuto modo di sperimentare direttamente le sfide pratiche legate alla concorrenza, alla gestione delle risorse e alla progettazione di soluzioni funzionanti per i diversi approcci. Questa esperienza ci ha fornito preziose lezioni su come affrontare problemi reali in contesti complessi e dinamici.

Il fatto che il modo di progettare e implementare lo stesso problema possa variare significativamente a seconda dell'approccio di programmazione utilizzato rappresenta sicuramente una caratteristica interessante e peculiare dell'esperienza affrontata.