PCD ASS02 (2)

Federico Bravetti
federico.bravetti2@studio.unibo.it

Tommaso Patriti
tommaso.patriti@studio.unibo.it

Alex Testa
alex.testa@studio.unibo.it

I. Analisi del Problema

Il progetto in questione consiste nello sviluppo di un *Web Crawler* che a partire da un link e data una parola da ricercare esplora ricorsivamente le diverse pagine web fino ad una data profondità. L'implementazione è stata realizzata utilizzando tre diversi paradigmi: Programmazione Asincrona ad Eventi/Event-Loop, Programmazione Reattiva e Virtual Thread.

A. Concetti Comuni

Il pattern di esplorazione scelto consiste nella divisione del codice HTML delle diverse pagine in parole singole eseguendo uno split sul carattere spazio. Vengono poi selezionati tutti i collegamenti ipertestuali (attributo *href* nei tag *a*) filtrando i link *https* ed escludendo formati non testuali (e.g. .png, .msi, .mp3). Nel caso in cui uno stesso collegamento venga trovato più di una volta questo sarà comunque richiesto e analizzato.

B. La libreria

La libreria sviluppata consiste di un metodo asincrono *getWordOccurrences()*, di seguito il relativo pseudocodice

```
AsyncReport getWordOccurrences(String
    rootLink, String word, int depth){
   AsyncReport report = new
       AsyncReport();
   String[] links =
       getLinksFrom(rootLink);
   for(link : links){
      if(depth > 0){
         report.update(
            getWordOccurrencies(link,
                word, depth - 1)
         );
      }
   }
   return report;
}
```

C. Test

Per verificare la correttezza delle diverse implementazioni, dato che l'ambiente web è dinamico, è stato creato un server di test utilizzando Node.js che genera parole e link con la seguente API:

- **word**: parola generata
- **numberOfWords(nw)**: numero di parole generate in ogni pagina
- **numberOfLinks(nl)**: numero di link generati in ogni pagina
- **path**: utilizzato per generare link univoci

In questo modo è possibile calcolare il numero di parole complessive ad una determinata profondità utilizzando la seguente formula:

$$\sum_{k=0}^{depth} nw * nl^k$$

NOTA: Nonostante si usi un server locale nel momento in cui il numero di richieste diventa elevato alcune potrebbero essere rifiutate.

II. Event-Loop

Per questa versione del programma è stato deciso di utilizzare JavaScript come linguaggio e Node.js come esecutore. Inoltre, per l'implementazione della GUI, è stato utilizzato Express per l'hosting del servizio web e Socket.io per effettuare una comunicazione in tempo reale tra la pagina web e il crawler.

A. Strategia risolutiva

Sono state adottate due strategie risolutive differenti: una mirata alla semplicità implementativa e un'altra che permette maggior controllo, a discapito della chiarezza del codice.

Nella versione base, a cui non è stata attaccata nessuna GUI, viene implementata una singola funzione ricorsiva. Inizialmente, conta il numero di parole all'URL richiesto e incrementa una variabile globale. Successivamente, controlla tutti i link nella pagina e per ognuno di essi effettua una chiamata ricorsiva, che fa ripartire il loop da capo fino al raggiungimento della profondità massima richiesta. In questo caso, le chiamate sono tutte non bloccanti e, grazie alla natura dell'event-loop, non si rischiano problemi di inconsistenza. Il problema di questa implementazione si trova nell'impossibilità di conoscere la fine dell'esecuzione del programma. Per risolvere questo problema è stata implementata una seconda versione che, utilizzando una combinazione di *Promise* e la funzione *Promise.all()*, mantiene un controllo del flusso di esecuzione e permette di capire quando il programma termina. Inoltre per far fermare l'esecuzione è stato implementato un graceful stop, che da quel momento in poi farà generare solo Promise "vuote". Questo implica che la terminazione dell'esecuzione non sarà istantanea,

ma saranno prima elaborate tutte le Promise presenti in coda prima dello stop.

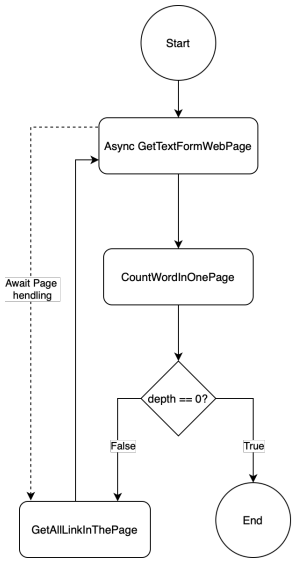

Figure 1. Diagramma degli stati valido per Event-Loop e VirtualThread

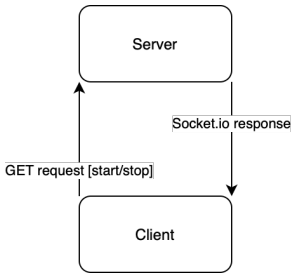

Figure 2. Comunicazione tra Client e Server (Event-Loop)

B. Criticità

Socket.io mantiene un buffer lato client per non aggiornare la grafica troppo velocemente, allo stop quindi, viene chiusa la connessione per fermare l'aggiornamento dell'interfaccia. Questo determina un ritardo tra l'invio dei dati da parte del server e l'effettiva visualizzazione di questi da parte del client.

III. Programmazione Reattiva

Per questa versione del programma è stato utilizzato il framework *JavaRx* e quindi il linguaggio Java. La programmazione reattiva rappresenta un approccio innovativo nell'ambito dello sviluppo software, offrendo una soluzione sofisticata per gestire la complessità dei sistemi moderni. Caratterizzata da una tempestiva risposta agli eventi e una gestione efficiente di dati e stato, inoltre si distingue per i suoi numerosi vantaggi nell'agevolare lo sviluppo di applicazioni altamente performanti e scalabili.

Uno dei principali, risiede nella sua capacità di gestire in modo efficace flussi di dati asincroni, rendendo più agevole lo sviluppo di applicazioni che forniscono un'esperienza utente più fluida.

A. Strategia risolutiva

Essenzialmente, la classe *SearchHandler* si assume la responsabilità di gestire i componenti reattivi che si distinguono in tre flussi distinti. Il flusso denominato *searchObservable* è dedicato all'esecuzione della computazione effettiva, che comporta la richiesta delle pagine, il conteggio delle occorrenze e la visita dei sotto-link fino alla profondità specificata. La ricerca avviene mappando il flusso di link di un livello nel flusso di link del livello successivo, come mostrato in Figura 3, fornendo un *SearchReport*. Quest'ultimo contiene informazioni dettagliate sullo stato attuale della ricerca, includendo i link visitati, il numero di occorrenze individuate, il totale delle parole trovate e i link della pagina. Gli altri due flussi sono il *searchReportSubject* e l'*errorReportSubject*, dove rispettivamente vengono pubblicati i SearchReport e gli eventuali ErrorReport, la struttura di questi flussi è mostrata in Figura 4.

In particolare il flusso searchObservable è stato implementato come flusso "cold", gli altri due invece sono flussi "hot".

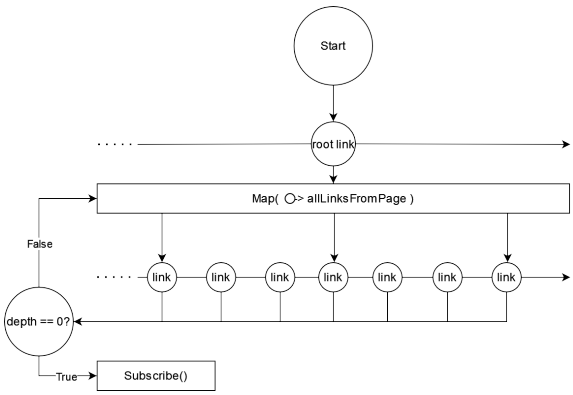

Figure 3. Funzionamento del flusso searchObservable

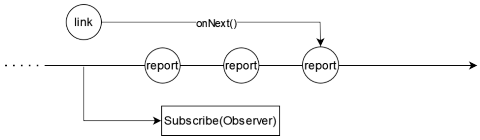

Figure 4. Funzionamento dei flussi di notifica (searchReportSubject e errorReportSubject)

B. Criticità

Il sistema non presenta particolari criticità.

IV. Virtual Thread

Per questa versione del programma, essendo i Virtual Thread un'implementazione specifica di Thread Lightweigth, è stato utilizzato il linguaggio Java. I

Virtual Thread sono stati introdotti con lo scopo di semplificare la programmazione multithread preservando la stessa interfaccia di utilizzo dei Thread di sistema.

A. Strategia risolutiva

Concettualmente il programma consiste di un *SearchState* e di vari componenti attivi chiamati *PageHandler*. Ogni PageHandler ha il compito di richiedere e analizzare una pagina web, quindi conta le occorrenze della parola desiderata e avvia ulteriori PageHandler nel caso in cui la ricerca non è completata. Poiché i Virtual Thread sono collocati ad un livello di astrazione superiore rispetto ai Thread è stato possibile associare ad ogni componente attivo un Thread logico, questo di fatto semplifica di molto la modellazione.

Il funzionamento complessivo, mostrato in Figura 1, è analogo a quello utilizzato nel paradigma ad eventi. In particolare in questo caso per mantenere il controllo del flusso di esecuzione ogni componente attivo esegue una *join()* sui componenti attivi che ha istanziato.

La principale differenza tra i due paradigmi riguarda la proprietà di mutua esclusione che in questo caso non è garantita *by design* ed è quindi stato necessario utilizzare un monitor per gestire eventuali corse critiche.

Infine sono state implementate due modalità di stop della ricerca. La prima (graceful stop) interrompe la ricerca e attende la terminazione dei Thread attivi, la seconda (brute stop) oltre a interrompere la ricerca interrompe anche l'ascolto degli ultimi aggiornamenti. Nel secondo caso l'applicazione risulta più responsive dal punto di vista dell'utente.

B. Criticità

L'architettura teorica non presenta problemi particolari se non la necessità delle classiche accortezze legate alla gestione di applicazioni multithread. Le criticità maggiori però si manifestano nella pratica, in particolare nel collegamento della GUI al sistema. La GUI è stata sviluppata con il framework *Java Swing*. In questo framework gli eventi di aggiornamento sono gestiti tramite una coda, però a causa della gestione interna di quest'ultima nel momento in cui vengono aggiunti un numero massiccio di eventi questi non sono processanti correttamente portando ad un blocco della GUI. Per questo motivo un'architettura che usa *listener* per la gestione di tutti gli eventi di notifica non è di fatto utilizzabile.

Per porre rimedio a questo problema si è passati ad un approccio più attivo, aggiungendo un refresh rate alla GUI implementato attraverso un timer. Anche in questo caso si sono presentati problemi di reattività teoricamente dovuti all'elevato numero di accessi concorrenti al monitor condiviso che è diventato il nuovo *bottleneck* dell'applicazione.

Per ovviare a questa nuova problematica le informazioni necessarie alla GUI sono state separate dallo stato globale della ricerca e sono a loro volta aggiornate in modo temporizzato.

Dato l'elevato numero di risorse necessarie per eseguire i test e dato il loro tempo di esecuzione la problematica rimane per il momento ancora aperta.

NOTA: Anche nella soluzione proposta utilizzando il paradigma reattivo si utilizza una GUI sviluppata con Java Swing e non sembra presentarsi la problematica, forse questo è dovuto anche da una migliore gestione delle risorse da parte del framework JavaRx.

V. Performance

Per quanto riguarda le performance si è utilizzato il server creato in precedenza caricato su una macchina remota, questo è stato fatto per cercare di ricreare un ambiente statico che fosse il più possibile simile al normale web.

I parametri utilizzati sono stati:

- **depth**: 5
- **nw**: 50.000
- **nl**: 3

E complessivamente si hanno un numero di parole pari a 18.200.000 parole distribuite equamente in 364 link.

Questi parametri sono stati scelti sopratutto per evitare che il server faccia da bottleneck.

Il programma è stato testato utilizzando un processore I5-1035G1 4 core 8 thread e facendo la media con 10 esecuzioni. Risultati:

- **Event-Loop** : 13.2 s
- **Programmazione Reattiva** : 12.8 s
- **Virtual Thread** : 14.2 s

Come si può osservare le performance sono molto simili, probabilmente questa differenza si sarebbe potuta accentuare aumentando la grandezza della ricerca ma questo, come detto in precedenza, non è stato possibile, poiché il server non è in grado di gestire correttamente un numero elevato di richieste.