Sistemi Cooperativi e Reti Sociali

Lezione 2

Daniele Margiotta, Gabriele Prestifilippo, Luca Squadrone

19 marzo 2019

Indice

Capitolo 1

Introduzione

Proseguendo quanto detto nel C10K problem, la soluzione migliore possibile per il C10K problem sarebbe avere un singolo processo, e fare in modo che tutte le richieste vengano svolte nel minor tempo possibile. L'obiettivo di questa dispensa è analizzare una particolare soluzione che utilizza le **fibers** in go

Per realizzare ciò quando si hanno un numero elevato di richieste per un server, definendo come $\mathbf{T_c}$ è il tempo di gestire la comunicazione e $\mathbf{T_e}$ è il tempo di elaborazione della transazione allora, sapendo che possono arrivare al più un numero di connessioni pari alla **velocità di rete** (V_r), è necessario che:

$$T_c + T_e \leqslant V_r \tag{1.1}$$

Sapendo che T_c non può essere minore della V_r, possiamo idealmente pensare che $T_e = 0$, quindi bisogna che un server elabori tante connessioni al secondo quanto la velocità di rete.

Per fare un esempio, se la scheda di rete del server va a 1Gb/s e arrivano richieste di 100Kb/s allora sarà necessario gestire con 1 solo processo 10mila connessioni (socket) al secondo.

Quindi in altre parole è necessario passare da un modo di gestione sincrono e sequenziale delle connessioni, ad un mondo asincrono a flusso frammentato.

Capitolo 2

Tipi di comunicazione

In questo capitolo analizzeremo i vari tipi di comunicazione e vedremo alcuni esempi necessari per introdurre i concetti delle fibers e il loro utilizzo.

2.1 Comunicazione sincrone

La comunicazione sincrona è un tipo di comunicazione in cui il mittente invia il messaggio e rimane in attesa sin quando il ricevente non invia la risposta al mittente, anche se la risposta è solo acknoledgment.
Un esempio di comunicazione sincrona può essere il seguente:

```
canale C1(int);
canale C2(int);

Processo P1 {
   int n;
   ...
   synchSend C2(n);
   receive C1(n);
   ...
}

Processo P2 {
   int m;
   ...
   receive C2(m);
   synchSend C1(m+1);
   ...
}
```

Questo tipo di connessioni non sono per nulla veloci, infatti l'attesa della risposta solitamente è un tempo non ignorabile, al punto che in una connessione di rete su internet è necessario introdurre un timeout dopo il quale la connessione viene rifiutata. Inoltre in questo tipo di comunicazione le chiamate di invio e ricezione sono bloccanti per l'esecuzione del programma. Per migliorare la situazione si passa ad un tipo di comunicazione asincrono.

2.2 Comunicazione asincrone

La comunicazione asincrona è un tipo comunicazione in cui il mittente invia il messaggio e poi continua la propria esecuzione. l'asincronicità di questo tipo di comunicazione tra l'invio di un messaggio e la risposta al messaggio stesso permette di gestire più connessioni in un lasso di tempo, poichè non è necessario mantenere connessioni attive in attesa.

Un esempio di comunicazione asincrona può essere il seguente:

```
canale C(int);

Processo P1 {
   int n;
   ...
   send C(n);
   ...
}

Processo P2 {
   int m;
   ...
   receive C(m);
   ...
}
```

Il processo P1 invia sul canale C il valore della variabile n ed il processo P2 riceve sul canale C il valore della variabile n e la mette nella variabile m. Entrambi possono eseguire altre operazioni una volta inviata o ricevuta l'informazione, ma mentre la chiamata di invio di informazioni non è bloccante, la chiamata di ricezione può essere ancora bloccante, a seconda di come il programma è gestito.

2.2.1 Programmazione asincrona e callbacks

Un esempio di linguaggio in cui sono presenti le callbacks e siamo forzati ad utilizzare un singolo thread è il javascript, dove esiste anche la possibilità di eseguire le istruzione in modalità asincrona.

Consideriamo lo scenario in cui dobbiamo effettuare una richiesta ad un server.In modalità sincrona avremmo ad esempio:

```
richiesta = preparaRichiesta();
risposta = mandaRichiestaSincrona(richiesta);
visualizza(risposta);
```

Se il server impiega 10 secondi a rispondere, il nostro codice è bloccato alla seconda istruzione, non si possono eseguire altre istruzioni nel frattempo. Sfruttando invece le closures è possibile riscrivere il codice in questo modo, realizzando quella si definisce una **asynchronous callback**:

```
richiesta = preparaRichiesta();
mandaRichiestaAsincrona(richiesta, function(risposta) {
  visualizza(risposta);
});
visualizza("Siamo in attesa di risposta dal server ...");
```

In questo caso, la seconda istruzione è asincrona e non bloccante, dopo la sua esecuzione il nostro programma continua con l'istruzione successiva: quando il

server risponderà, verrà eseguita la funzione di callback passata come argomento alla funzione asincrona.

Tecnicamente, il supporto alla programmazione asincrona è implementato, nell'interprete javascript, tramite l'utilizzo di:

1. una coda di messaggi:
2. un ciclo principale di gestione eventi.

Tutte le istruzioni che devono essere eseguite dal programma vengono inserite in una coda di messaggi che viene gestita da un ciclo principale che svuota ed esegue in un unico thread le istruzioni presenti nella coda.

Nel caso di una funzionalità asincrona, dipendente da un evento esterno al programma, un componente del runtime di javascript aggiunge alla coda dei messaggi la funzione di callback da eseguire, con i relativi parametri, non appena l'evento si è verificato.

Le funzioni di callback non sono mai interrompibili, così come qualsiasi funzione in javascript: in questo modo non esisterà mai un problema di accesso ad aree di memorie condivise da thread diversi, visto che il ciclo principale viene eseguito in un unico thread.

Il linguaggio di programmazione Go(o golang) è strutturato a grandi linee stesso modo di Javascript per quanto riguarda la gestione delle callback.

2.3 Continuation call

Una *Continuation call* è un modo per rappresentare lo stato di esecuzione di un programma ad un preciso istante e punto. Molti linguaggi hanno un costrutto che permette di salvare lo stato corrente di esecuzione per poi riprendere l'esecuzione a partire da questo stato in un momento successivo. Questa tecnica è tipica della programmazione funzionale ma sono molti i linguaggi di programmazione che presentano questa caratteristica con varie denominazioni come per esempio:

- Lisp: call/cc;
- C: setcontext.

La Continuation call si utilizza per "saltare" da una parte all'altra del programma a seconda delle necessità. Funziona in modo analogo al comando goto presente nel BASIC con la differenza che il comando goto è sconsigliato perché rende la manutenzione del codice difficile mentre la continuazione non rende ardua la modifica del codice quando questo diventa di elevate dimensioni. La continuazione non salta tra linee di codice ma tra stati. Si può vedere il programma come un insieme di stati che evolvono a seconda delle azioni dell'utente e la continuazione consente di saltare tra stati diversi a seconda delle necessità. Se le continuation call può essere usata solo per uscire dal contesto corrente, si dice il linguaggio supporta l'*escape continuation*. Molti linguaggi che non

supportanto direttamente la continuazione, hanno il supporto per la gestione delle eccezioni, che è equivalente all'escape continuation e può essere usata per gli stessi scopi. I costrutti C setjmp e longjmp sono di questo tipo: possono essere usati solo per srotolare lo stack e non per ripristinare una precedente continuazione salvata.

2.4 Modello Produttore/Consumatore

Abbiamo visto che i processi possono comunicare scambiando messaggi. Il modello a scambio di messaggi si presta, di fatto, ad applicazioni strutturate come il produttore/consumatore: un processo produce dati e l'altro li utilizza.
Nel modello a scambio di messaggi lo schema produttore/consumatore si realizza immediatamente come segue:

```
port A;

Produttore() {
  while(1) {
    /* produce d */
    send(A,d); // invia d su A
  }
}

Consumatore() {
  while(1) {
    receive(A,&d); // riceve d da A
    /* consuma d */
  }
}
```

Se utilizziamo una send asincrona (bloccante quando il buffer è pieno) e una receive sincrona, otteniamo una soluzione soddisfacente: il consumatore attende se non ci sono dati da consumare, il produttore attende se non c'è spazio nel buffer.

Capitolo 3

Meccanismi di comunicazione e gestione thread avanzata go

In questo capitolo parleremo di una modalità di comunicazione che Go implementa, i **canali**, e di un importante miglioramento riguardo la gestione dei thread con l'utilizzo delle **fibers** (o thread leggeri).

3.1 Goroutines

Poiche dalla stessa documentazione del linguaggio Go il concetto di canale è strettamente legato al concetto di goroutine, definiamo brevemente cosa effettivamente sia una **goroutine**.

Una goroutine è un thread "leggero", ed è un costrutto dell'ambiente del linguaggio Go. Le goroutine operano in **multitasking cooperativo**, ovvero i thread non sono mai costrette a interrompere la loro esecuzione (come succede nel preemptive multitasking, ovvero il metodo di gestione classico dei thread) ma spontaneamente rilasciano la CPU se essi sono in attesa o in deadlock, permettendo a tutti gli altri processi di continuare la loro esecuzione. Perchè questo tipo di scheduling funzioni è necessario che tutti i processi cooperino, e nel caso delle goroutines la cooperazione è assicurata dall'ambiente del linguaggio Go(e non può essere specificato dal programmatore). In questo tipo di gestione dei thread lo scheduler si limita a assegnare la CPU ad un nuovo processo ogni volta che il processo stesso rilascia la CPU.

In linguaggio Go, una volta definita una funzione $f(arg1, arg2)$, è possibile chiamare una goroutine che esegue quella funzione come:

```
1 go f(arg1, arg2,...)
```

3.2 Canali Go

I canali Go sono dei pipe che consentono alle gorutines di inviare messaggi su di essi, di leggere i messaggi in essi contenuti e consumarli, o sincronizzazione di due goroutines

Normalmente i *Channel Go* sono privi di buffer, ovvero accettano nuovi messaggi solo se c'è un processo pronto a riceverli, tuttavia è possibile specificare la grandezza del buffer durante la creazione del channel.

Supponiamo di voler creare un channel contenente stringhe, il codice in linguaggio go è il seguente:

```
1 channel_name := make(chan string, 2)
```

3.3 Fibers

Una **fibers** è un thread *"leggero"*, come le goroutines, e anche esse operano in multitasking cooperativo. Tuttavia le fibers non sono implementate in maniera nativa in go.

Mentre le goroutines lavorano a livello logico come costrutti associati all'enviroment go, le fibers sono viste come costrutti associato al sistema operativo e la cooperazione non è assicurata dall'ambiente del linguaggio go, tuttavia delle regole di cooperazione possono essere specificate dal programmatore.

Poichè in letteratura le fibers sono sempre associate alle cooroutines per completezza si fornisce anche anche una breve digressione su di esse. Una cooroutine in letteratura è descritta con gli stessi concetti di una fibers e la distinzione, se effettivamente esiste, è che le cooroutines sono costrutti a livello del linguaggio utilizzato, mentre le fibers sono costrutti a livello del sistema operativo, che possono essere visti come thread che non operano in parallelo. Come è evidente la letteratura non è ben chiara sull'argomento, al punto tale che alcune fonti citano le fibers come un implementazione delle cooroutines, altre come un substrato sulle quali sono implementate le cooroutines.