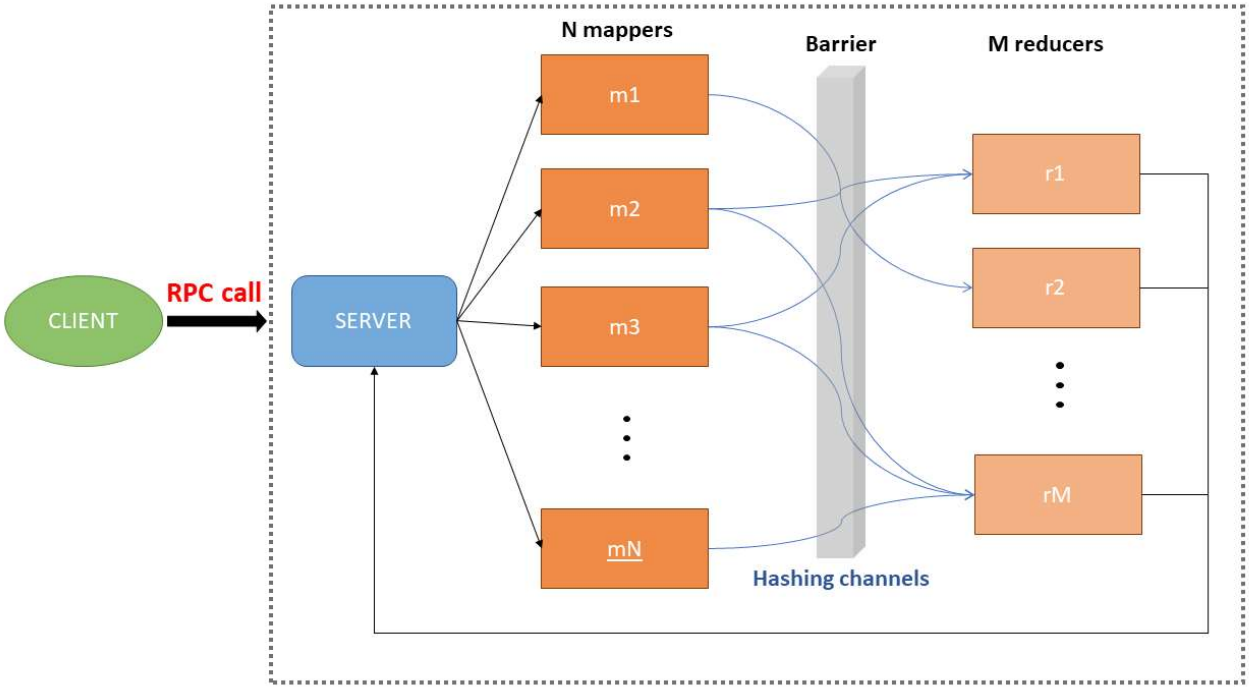
Wordcount in Go

Worgo: an implementation

Diletta Lagomarsini 0266667, dilettalagom@gmail.com
Federica Montesano 0265577, federica.montesano42@gmail.com

Architettura



Il caso di studio realizzato consiste nel Wordcount semplice, dunque il conteggio delle occorrenze di ogni parola in un file. L'architettura di base è di tipo master-worker e implementa una versione dell'algoritmo Map-Reduce.

Era richiesto di utilizzare la chiamata RPC, che nel nostro caso è sincrona, e consiste nel client che richiede al server RPC di effettuare il Wordcount sui file contenuti nella directory selezionata.

Gli argomenti della chiamata RPC sono una struttura

```
type Args struct {
      File    string
      N       int
      M       int
}
```

In cui File è il path relativo del file, N il numero di mapper e M il numero di reducer.

Il parametro restituito dalla chiamata RPC è `type Result map[string]int,` che rappresenta l'insieme di coppie chiave-valore dell'intero file.

Le fasi di Map e di Reduce sono state modellate con delle goroutines:

- **Map,** che consiste in N goroutines mapper (con N configurabile) il cui compito è quello di analizzare chunk di file e creare coppie chiave-valore del tipo <parola, frequenza>. Inoltre svolge una funzione di aggregazione della stessa parola con la sua frequenza, ad esempio una situazione del tipo <parola, 1>, <parola, 1>, <parola, 1> viene considerata <parola, 3>. Questo viene fatto per diminuire il carico di lavoro della fase successiva. Infine l'insieme delle coppie chiave-valore viene inviato ai reducer attraverso dei canali.

- **Reduce,** che consiste in M goroutines reducer (con M configurabile) il cui compito è quello di contare globalmente le frequenze relative ad una stessa parola ricevuta da ogni worker. Infine le coppie vengono restituite al client attraverso RPC.

Implementazione

Il client effettua una chiamata RPC per ogni file contenuto nella directory passata come parametro al programma e poi attende i risultati. Il procedimento descritto in seguito viene ripetuto per ogni file.

Il master si occupa del partizionamento del file in chunk utilizzando come separatore di parola lo spazio (“ “) e della creazione di mapper e reducer e i relativi canali. I mapper vengono creati dopo il partizionamento, mentre per i reducer si effettua un preforking.

Ogni mapper riceve una slice di `words_to_read` parole tratta dall’array `list_of_words`. Il numero di parole che ogni mapper deve leggere è calcolato nel modo seguente:

```
words_to_read = min{number_of_words,[len(list_of_words)-i*int(number_of_words)]}
```

in cui `number_of_words` è la media di parole che ogni mapper teoricamente dovrebbe ricevere e `i` va da 0 a N. In particolare, se il numero di parole non è multiplo di N, si hanno situazioni di sbilanciamento che sono risolte distinguendo ciò che legge l’ultimo mapper da tutti gli altri.

Tutti i mapper ricevono `list_of_words[offset:(i+1)*words_to_read]`, in cui `offset` = `i*number_of_words`, tranne l’ultimo che invece riceve l’array da `offset` fino alla fine.

Due elementi fondamentali per garantire sincronizzazione sono i channel e la barrier, rispettivamente usati per la comunicazione tra mapper e reducer e per attendere il più lento dei mapper.

Ogni reducer inizia il proprio ciclo di vita in attesa su una barrier per permettere ai mapper di eseguire la loro routine e per ricevere i risultati della loro computazione. La ricezione avviene tramite un vettore di M canali creato tra mapper e reducer dal padre, sul quale si garantisce bilanciamento del carico attraverso l’uso di una funzione di hashing calcolata sulla chiave della coppia chiave-valore sommando i caratteri della chiave e dividendo modulo M il risultato, ottenendo quindi un indice compreso tra 0 e M-1 con cui indicizzare il vettore di canali. Esempio: su <casa, 4> e avendo M=10 viene calcolata la funzione hash 99+97+115+99 % 10 = 410 % 10 = 4, dunque la coppia <casa, 4> verrà mandata al canale di indice 4.

Infine abbiamo scelto di utilizzare un timeout per permettere ai reducer di chiudere il canale e restituire il valore ottenuto al master. Questo timeout è un timeout assoluto, ovvero parte al momento di attraversamento della barrier. Non è una soluzione ottimale sia poiché potrebbe chiudere il canale troppo presto se il timeout è piccolo, sia per l’overhead causato dal tempo di idle del reducer se il timeout è troppo grande. Una soluzione più efficiente

sarebbe stata quella di chiudere il canale solo quando ogni mapper avesse inviato le coppie <chiave, valore>, ma essendo i mapper indipendenti tra loro, non si può predire quanti di loro invieranno sullo stesso canale, dunque si è ritenuto più semplice adottare la soluzione del timeout.