Sistemi Distribuiti e Cloud Computing - A.A. 2018/19

Progetto B2: A fault-tolerant MapReduce distributed application in Go

Di Simone Mesiano Laureani
Matricola 0278325
email: simesi@hotmail.it

I. DESCRIZIONE DETTAGLIATA DELL'ARCHITETTURA DELL'APPLICAZIONE E DELLE SCELTE PROGETTUALI EFFETTUATE

Si è implementata, nel linguaggio di programmazione Go, un'applicazione distribuita per risolvere il problema del WordCount partendo dall'applicazione data come esercizio durante il corso [1]. In particolare, l'applicazione è stata estesa con un meccanismo di tolleranza ai guasti dei worker e del master.

L'architettura adottata è quella del master-worker, che si traduce nella distribuzione del lavoro ai workers usando i canali RPC. Nello specifico è stato implementato un master che assegna i Map e Reduce tasks ai workers e assegna tali tasks dinamicamente in base alla grandezza dei dati in input e ai valori dei parametri configurabili.

La fase di Map consiste nella lettura dei file nella cartella di input e la suddivisione del loro contenuto in linee di testo. Le linee vengono processate in parallelo dai Map workers secondo uno schema distribuito. Inizialmente si pensava di poter permettere la divisione del file in byte invece che in linee di testo, ciò però avrebbe comportato la probabile presenza di "parole spezzate in due parti" se la divisione in byte fosse avvenuta su spazi non vuoti. Inoltre si è applicato anche un filtro alle linee di testo dei files per oscurare i caratteri speciali quali per esempio @,#,{,[,(e i numeri. In caso di presenza di uno di questi caratteri speciali nel testo, essi vengono rimpiazzati da uno spazio vuoto e quindi le parola suddivisa in più parti (ad esempio "simesi@hotmail" filtrata diventa "simesi hotmail"). In caso di lettura di linee di testo vuote esse vengono scartate e non inviate ai Map Workers. I risultati verranno inviati al Master per la loro aggregazione futura. L'ordine dei risultati non è di interesse per la corretta esecuzione del programma, quindi può succedere che il risultato di un Map Worker venga scritto al Master prima della scrittura del risultato precedente di altro Map Worker (invocato precedentemente). Il tipo di dato del singolo risultato sarà una map[string]int, ad esempio:

{"a":1,"all":1,"because":1,"before":2,"blessed":1,"breath":1,"breathed":1,"but":1,"caused":1,"created":2}

La fase di 'Shuffle and Sort' si colloca tra le fasi Map e Reduce e consiste nella raccolta dei risultati di tutti Map workers in modo da procedere al loro ordinamento e unione per chiave (parola). Quindi tutti i risultati vengono inseriti in un array di map, scritti su un file temporaneo inserito su un bucket S3 (con un nome scelto dinamicamente) con uno specifico tag ("beforeShuffle"). Tale file verrà utilizzato per il recover in caso di un fault del master nelle fasi successive. A questo punto il Master procede alla trasformazione da un tipo di dato quale array di map (una entry nell'array per ogni Map result) a una sola map con un array di int come valori (le parole/chiavi dei vari Map results vengono raggruppate e ordinate). Quest'ultima è stata la scelta progettuale preferita ad altre in quanto inizialmente si aveva pensato di modellare i risultati come tipi di dato List in modo da inserire in testa alla lista la parola (word) e successivamente le sue varie occorrenze trovate dai Map Workers; successivamente però tale scelta non è stata adottata in quanto le operazioni sul tipo di dato List risultano meno efficienti e meno affrontate online rispetto a quelle su map. Ci si è scontrati anche sul fatto che una singola map[string]int non possa avere più valori per chiave o chiavi duplicate quindi si è adottato il tipo di dato map[string][]int. I risultati dei Map Workers non contengono un campo per poter risalire a quale linea di testo li ha generati e quale file, ciò risulta infatti ininfluente nell'esecuzione del problema di Wordcount. Una volta terminata la trasformazione dei risultati, il file sul bucket S3 viene sovrascritto con i valori nel nuovo tipo di dato e viene aggiornato il tag in "afterShuffle". Al termine della fase Shuffle il tipo di dato sarà per esempio:

{"secondo":[1],"semestre":[1],"sequenziali":[1],"serve":[1],"sessione":[3],"set":[1],"si":[2],"siano":[1,1],"simulator":[1],"sistemi":[1,1,1,2],"sito":[1,2]}

La fase di Reduce consiste nel sommare tutte le occorrenze di una parola. Nello specifico viene inviata una chiamata RPC a un Reduce Worker per ogni parola/chiave. Al termine della fase Reduce il risultato totale sarà del tipo:

{"secondo":1,"semestre":1,"sequenziali":1,"serve":1,"sessione":3,"set":1,"si":2,"siano":2,"simulator":1,"sistemi":5,"sito":3}

Le chiamate RPC sono asincrone e vengono gestite tramite la creazione di specifici threads per il tipo di operazione da effettuare. Il meccanismo di tolleranza ai guasti dei worker realizzato gestisce possibili guasti dei worker che possono avvenire durante la computazione del task assegnato. Poiché i worker sono stateless, per gestire la tolleranza ai guasti è stato sufficiente ri-assegnare il task di computazione eseguito sul worker rilevato come guasto ad un altro worker. Per rilevare il guasto di un worker, si è utilizzato un meccanismo di time-out della RPC. Tale scelta progettuale si è materializzata quindi nel creare un thread per ogni task assegnato a un worker, in questo modo il thread può tenere traccia del tempo residuo prima del time-out del worker e in caso rieseguire la chiamata RPC autonomamente.

Il meccanismo di tolleranza ai guasti del master realizzato gestisce un possibile crash del master che può avvenire durante la computazione. Poiché il master è stateful, per gestire la tolleranza ai guasti occorre salvare lo stato del master; a tale scopo è stato usato uno storage persistente esterno all'applicazione in Go quale è il bucket S3. In caso di rilevamento del guasto del master, un nuovo master ne prende il posto ripristinando lo stato recuperandolo dallo storage persistente nel bucket S3 e subentrando nella gestione. Il crash del master è accertato dal Master Controller che in questa applicazione è implementato come un thread ma che potrebbe benissimo essere un processo che esegue su una macchina virtuale. Il Master Controller invia periodicamente un tick (alias int) su un canale che il client ha creato e passato al master al suo avvio. I tick potevano essere mandati anche tramite chiamate RPC. Dopo essere trascorso del tempo (in base al parametro timeoutMaster), il Master Contoller verifica che il canale su cui ha inserito il tick sia vuoto, se non lo è allora dichiara il master come guasto e ne avvia un altro in base all'ultimo stato salvato sul bucket S3.

Il ripristino del Master viene implementato nella funzione RecoverMaster. Tale funzione verifica se ci sono file sul bucket S3 e in caso affermativo anche il loro tag. In base a queste informazioni la funzione avvia un nuovo Master passandogli oltre parametri soliti anche un intero che indicherà quali parti di codice il nuovo master può 'saltare'. Quando il master è dichiarato come guasto dal Master Controller, quest'ultimo inserisce un valore nel canale di tipo bool inizializzato all'invocazione del Master. Il canale risulta necessario per simulare il crash del master e quindi per ordinare al vecchio master di terminare l'esecuzione e lo stesso vale anche per i thread dedicati alla ricezione delle chiamate RPC.

Inizialmente si utilizzava una libreria [5] per l'implementazione dell'heartbeat del master, purtroppo dopo vari tentativi si è scoperto che non c'era modo di fermare l'heartbeat se non dopo che il programma che l'aveva chiamato avesse terminato l'esecuzione. Rendendoci conto di tale difetto si è optato per una soluzione meno sofisticata ma homemade utilizzo un canale con interi come tick.

La scelta di eseguire o meno un fallimento del master o del worker avviene tramite la randomizzazione di un intero ottenuta tramite *rand.Seed(time.Now().UnixNano())* e *rand.Intn(100)*, quindi si attua il re-seed ad ogni chiamata partendo dal valore attuale del tempo come seme.

L'interfaceRPC implementa i servizi di Map e Reduce, mentre il serverRPC è il programma che espone tali servizi per poter esser invocati dal master tramite procedura remota RPC.

Inizialmente si pensò di utilizzare dynamoDB [2] al posto del bucket S3 come storage persistente. Tale scelta però fu accantonata sia per la difficoltà superiore che necessitava lo scrivere query apposite sul DB dynamo rispetto al semplice download del file su S3; ma specialmente perché dynamoDB non permetteva l'esecuzione di funzioni di aggregazioni quali sum o join [3][4] necessarie per il merge dei risultati dei workers.

Un accorgimento che è stato preso è stato quello di non chiamare rpc.DialHTTP("tcp", workerAddress) ad ogni assegnamento di un task al worker ma solo alla prima connessione al worker, altrimenti il programma per un numero di chiamate RPC troppo elevato poteva perdere funzionalità.

Le librerie degne di nota utilizzate sono:

- "encoding/json" per attuare il marshaling e unmashaling su file di testo dei risultati dei workers
- "math/rand" per attuare la randomizzazione necessaria alla esecuzione o meno di fault del master e dei workers
- "regexp" è utilizzato come filtro delle linee di testo per eliminare i caratteri speciali e i numeri dalle parole nei file
- "time" viene utilizzato per il calcolo del parametro di timeout del master e dei worker
- "bufio" è utilizzato per la 'scannerizzazione' dei file da processare
- "sync" viene utilizzato dai thread dedicati alla ricezione dei risultati dei Reduce Workers per poter scrivere su una variabile globale condivisa ai thread e la cui scrittura quindi necessita di sincronizzazione tramite lock.

II. Descrizione dell'implementazione

I soggetti attivi dell'applicazione sono: il client, il master, il master controller, i threads dedicati alla invio di chiamate RPC e alla ricezione dei risultati ed il server RPC .

A. Client

Il client si occupa di ottenere l'input dell'user tramite stdin. In particolare, verifica che sia stato passato esattamente un solo parametro all'avvio del programma. A questo punto lo logica del client interpreta tale parametro come un percorso alla cartella di files da elaborare e si occupa del 'marshaling' del percorso per soddisfare l'interfaccia di GO per i paths (nello specifico sostituisce il carattere '\' utilizzato da Windows per i percorsi con '/'). A questo punto la logica del client inizializza dei canali di comunicazione necessari per il corretto funzionamento del sistema. Nello specifico si crea un canale bool, che sarà condiviso con il thread Master Controller e con i futuri threads dedicati alla ricezione delle risposte dei workers, per rendere noto a tutti i listeners lo stato del Master. Viene creato anche un canale di tipo map[string] int dove verrà scritto il risultato finale della computazione da parte del Master al Client. Tale canale verrà passato anche al thread Master Controller per evitare che si perda in caso di fault del Master. Infine, si inizializzerà anche un canale di tick (nel nostro caso di tipo int) con capacità unitaria: tale oggetto verrà utilizzato per la periodica verifica dell'healthiness del thread Master da parte del thread Master Controller. A questo punto il Client fa partire due nuovi threads: lo Start Master (che noi abbrevieremo da qui in avanti in Master) e il Master Controller; a entrambi vengono passati come parametri: il percorso della repository di documenti (tradotto in logica GO), i tre canali descritti precedentemente e al Master anche un int che indica lo stato da cui farlo partire (si veda implementazione Master per informazioni aggiuntive).

B. Master

Il Master inizialmente controlla il percorso della variabile d'ambiente $GOPATH che utilizzerà in seguito per crearci un file temporaneo da inserire sul bucket S3. Dopo inizializza una sessione per utilizzare i servizi offerti da aws (utilizzando le credenziali presenti nel file .aws/config). Poi attua una verifica dei bucket presenti e ne crea uno nuovo necessario per la conservazione dello stato del master. A questo punto il master scannerizza i file presenti nella cartella passata dallo user in input e ne filtra il contenuto (sostituisce i caratteri speciali e i numeri con spazi vuoti) lasciando comunque inalterate le parole accentate. Una volta scannerizzate *chunksLines,* il master inizializza una subroutine (tramite il comando 'go func') e continua la scannerizzazione e creazione di subroutine fino alla saturazione del contenuto dei file. A questo punto il master aspetta tramite select che tutti i threads dedicati alla ricezione dei risultati dei Map Workers scrivano su una variabile condivisa. Una volta che tutti i threads hanno scritto, viene creato un file temporaneo, che alla fine del programma sarà cancellato, sul percorso $GOPATH. In tale file vengono scritti i risultati dei workers in modalità append e alla fine viene inviato sul bucket S3 creato precedentemente con il tag "beforeShuffle". A questo punto si attua la trasformazione (fase shuffle and sort) da array di map ad una sola map con array di interi come valori. Dopodiché si procede alla sovrascrittura del file sul bucket S3 impostando questa volta come tag "afterShuffle". A questo punto il master inizializza un thread dedicato alla ricezione dei risultati dei Reduce Workers per ogni chiave/parola presente nella map prima citata. Come prima, il master rimane in select finchè tutti i risultati non vengono ricevuti e scritti su una variabile. A questo punto il master procede alla eliminazione del file temporaneo posto sul percorso $GOPATH e di quello presente nel bucket S3. Fatto ciò il master elimina anche il bucket stesso, inserisce il valore specifico di un intero all'interno del canale ascoltato dal Master Controller ed infine invia il risultato del problema di WordCount al Client.

C. Master Controller

Il Master Controller si preoccupa di inviare periodicamente al master un tick (implementato come un intero) su un canale e di aspettare fino al timeout del master. Se al timeout il tick è ancora presente nel canale e non è stato consumato, allora il master viene dichiarato come guasto e si procede all'esecuzione di una nuova istanza di master che partirà da punti specifici del codice in base a dei check effettuati sullo storage esterno. L'attività di heartbeat viene fermata se nel canale dove vengono inviati i tick il Master Controller trova un intero con un valore specifico, egli infatti lo interpreta come il successo delle operazioni del master e quindi della sua naturale terminazione.

D. Threads dedicati all'invio e alla ricezione dei risultati dei tasks

I threads vengono istanziati dal master in base alla mole di lavoro da effettuare. Essi prendono come input anche un canale utilizzato per sapere se il master è fallito o meno, che viene scritto dal Master Controller. In caso positivo essi terminano l'esecuzione.

E. Server RPC

Il server RPC rappresenta l'istanza del singolo worker ed attua la rpcInterface, cioè l'implementazione effettiva delle operazioni di Map e Reduce richieste. Il server è sempre in ascolto su una porta specifica (1234). Il server RPC può essere fatto girare anche in locale (su localhost) semplicemente eseguendo il relativo programma serverRPC.go. Una istanza del server è stata inserita in una macchina virtuale (istanza) EC2 e il relativo programma è eseguito all'avvio di tale macchina. L'istanza ha come indirizzo IP il seguente: "18.213.54.248:1234".

III. Limitazioni riscontrate

Tra le limitazioni riscontrate nello sviluppo di questo progetto si ha che:

- il peso computazionale di tanti threads potrebbe essere elevato, ma nella progettazione dell'applicazione si è preferito concentrarsi sul soddisfacimento dei requisiti funzionali piuttosto che sulle prestazioni.
- se il fallimento del Master avviene prima della raccolta di tutti i risultati dei Map Workers, lo stato non è salvato sul bucket S3 (perché non sarebbe consistente) e quindi

il master che ne prenderà il posto dovrà ripartire dall'inizio rimandando tutte le richieste Map RPC.

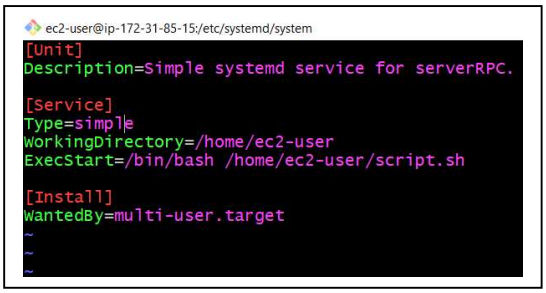
- se il fallimento del Master avviene prima della raccolta di tutti i risultati dei Reduce Workers, il Master che ne prende il posto dovrà rieseguire tutte le chiamate di Reduce RPC, anche ai workers che avevano risposto prima del fault.
- si voleva collegare un load balancer ad un indirizzo DNS per poter chiamare i worker in maniera trasparente da quale di essi avrebbe effettivamente risposto alla chiamata; tale funzionalità però non è attuabile per chi ha la versione AWS Educate Starter Account [6].
- si è rilevato un difetto non da poco nell'API S3 per il linguaggio di programmazione GO [7]. Nello specifico si è riscontrato che la funzione *s3.CreateBucket(input)* non ritorna un errore se il bucket che si vuole creare già esiste; per questo motivo si è aggirato il problema utilizzando una funzione per listare tutti i bucket e poi fare il controllo direttamente nel programma su quali bucket name era possibile usare.
- si voleva creare una chiave di accesso per user in modo di poter lasciare eseguire anche ad altri utenti il programma qui descritto in tutta sicurezza, purtroppo lo Starter Educate Account non permette le generazione di chiavi di accesso e quindi si ha la limitazione non da poco di dover passare ad altri le proprie chiavi di accesso per admin che oltretutto sono anche temporanee (durano due ore poi cambiano).

IV. Piattaforma software utilizzata per sviluppare l'applicazione

La piattaforma software utilizzata per sviluppare l'applicazione è stata Atom [8] munita di alcuni importanti packages quali:

- Go-plus [9] per poter lavorare su Atom con il linguaggio GO
- Script [10] per poter girare i programmi direttamente su Atom
- Go-signature-statusbar [11] per fare il display delle funzioni sotto il cursore nella barra di stato e dedurne i parametri necessari

Occasionalmente si è dovuto usare la shell di Git [16] per potersi connettere in SSH all'istanza EC2 che esegue i tasks. Tramite questo strumento è stato possibile aggiornare il software dell'istanza, installare i packages desiderati e inserire in */etc/systemd/system* un nuovo servizio, raffigurato in seguito, per avviare il server automaticamente all'avvio:

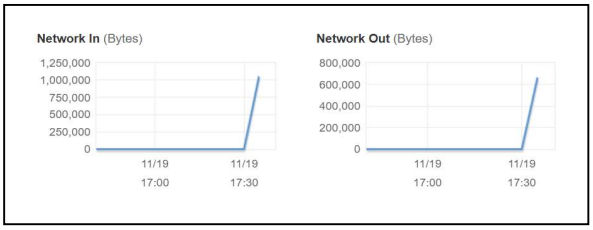


Di seguito è mostrato il contenuto dello script:

```
#!/bin/sh
export GOPATH="/home/ec2-user/go"
go run serverRPC/serverRPC.go
```

V. Test dell'applicazione

Passando come input un repository di file di testo grande quasi 4 MB (come file di esempio è stata presa la versione di The King James della Sacra Bibbia in lingua inglese [17]) è stato calcolato un tempo di esecuzione per il problema di WordCount di circa 80 secondi.



Passando un file di input di poche parole (circa 10) si ottiene un risultato in 4 secondi. Il support hardware come CPU del client durante il test è stato un processore Intel i5-8400 a 2.80 GHZ e con 8 GB di RAM installata.

VI. Howto per l'installazione, la configurazione ed esecuzione

Per una corretta installazione dell'applicazione occorre innanzitutto scaricare la libreria GO [12] più recente e settare opportunamente i percorsi delle variabili d'ambiente GOROOT e GOPATH; quest'ultima variabile verrà utilizzata direttamente dalla nostra applicazione (per maggiori informazioni si veda il paragrafo relativo all'implementazione del Master). Oltre al package di GO, l'applicazione necessita la libreria per l'utilizzo delle API GO specifiche per AWS; quindi bisogna aprire una shell di comando e digitare "go get -u github.com/aws/aws-sdk-go".

L'applicazione necessita anche dell'installazione di AWS CLI e della sua configurazione mediante il comando "aws configure" [13]. Purtroppo, **il servizio di AWS Educate Account non permette la generazione di chiavi di accesso aggiuntive** per gli utenti. Di seguito è riportato l'avviso fornito da AWS: *"[…], you cannot attach a login profile or associate keys with the users you create. This means that additonal users you create cannot log into your account. If you need to use your credentials, please click the Account Details on the right*". Per questo motivo per poter eseguire l'applicazione occorre richiedere al sottoscritto le credenziali di accesso (cioè aws_access_key_id e aws_secret_access_key). Tali credenziali una volta generate hanno una validità di circa due ore (dopodiché vengono invalidate). Le credenziali fornite andranno inserite durante l'esecuzione del comando "aws configure" da shell oppure nel file C:\\Users*name-user*\.aws\config.

Successivamente occorre inserire la cartella "Progetto" [14] nella cartella $GOPATH/src e a questo punto è possibile eseguire il programma. Se lo si vuole eseguire tramite shell occorre posizionarsi sul percorso in cui si trova il file Progetto/client.go e digitare sulla shell "go run client.go *percorso-directory*" e attendere il calcolo del risultato del problema di WordCount.

Alcuni parametri possono essere configurati diversamente dai valori di default, per farlo occorre aprire il file Progetto\Master\master.go con un qualunque editor di testo o IDE e modificare i valori dei parametri.

I valori dei parametri modificabili sono:

- timeoutMaster: una volta prodotto un tick, il numero di secondi da aspettare prima di dichiarare il master come guasto non avendo consumato il tick è dato da questo parametro
- timeoutWorker: numero di secondi da aspettare per la ricezione della riposta di un worker prima di tentare un'altra chiamata RPC
- chuncksLines: numero di linee di un file da far elaborare a un singolo Map Worker. Nell'esercizio assegnato durante il corso [1] il valore di questo parametro era 1. Per motivi di efficienza in questa applicazione è stato impostato il valore 20 come default.
- faultProbability: è la probabilità che il master abbia un fault durante l'esecuzione (intesa come percentuale)
- faultProbabilityWorker: è la probabilità che un worker fallisca durante l'esecuzione (intesa come percentuale)
- workerAddress: è l'indirizzo del server RPC e la porta da contattare. Tale valore può essere "localhost:1234" se si vuole eseguire il programma su rete locale, quindi eseguendo oltre a client.go anche serverRPC.go; altrimenti si può immettere come valore l'indirizzo **"18.213.54.248:1234"** che corrisponde all'indirizzo IP elastico [15] fornito al sottoscritto da AWS. Su tale indirizzo è collegata un'istanza EC2 che esegue il programma serverRPC.go.
- tmpFile: è il nome del file temporaneo che verrà creato sul percorso dato dalla variabile d'ambiente $GOPATH. Il file conterrà la somma dei risultati della fase Map e successivamente il loro 'ordinamento' (operazione Sort and Shuffle). Tale file risulta necessario in quanto non si è riusciti a poter inserire un file sul bucket S3 (tramite PutObject()) senza prima averno uno su storage locale.

REFERENCES

[1] V. Cardellini. Elective exercise using Go and RPC, 2019. http://www.ce.uniroma2.it/ courses/sdcc1819/slides/EsercizioGo.pdf

[2] https://aws.amazon.com/it/dynamodb/.

[3] https://stackoverflow.com/questions/44150156/how-to-do-basic-aggregation-with-dynamodb

[4] https://www.reddit.com/r/aws/comments/8rbnxw/achieving_sum_like_aggregate_functions_with/

[5] https://github.com/Syncbak-Git/heartbeat.

[6] Domain name registration not supported https://forums.aws.amazon.com/thread.jspa?threadID=262406

[7] https://docs.aws.amazon.com/sdk-for-go/api/

[8] https://atom.io/

[9] https://github.com/joefitzgerald/go-plus

[10] https://github.com/rgbkrk/atom-script

[11] https://atom.io/packages/go-signature-statusbar

[12] https://golang.org/

[13] https://docs.aws.amazon.com/it_it/cli/latest/userguide/cli-chap-configure.html

[14] scaricata da https://github.com/simesi/Progetto

[15] https://docs.aws.amazon.com/it_it/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

[16] https://git-scm.com/downloads

[17] http://corpus.canterbury.ac.nz/descriptions/