Map Reduce

Fault-tolerant and Locality-Aware implementation in GO

Andrea Di Iorio
Ingegneria Informatica Tor Vergata
matricola 0277550

ABSTRACT

In questo progetto ho realizzato un'implementazione del paradigma di computazione MapReduce applicato al problema di Word Count in un contesto distribuito in rete con simulazione di fallimenti utilizzando le api RPC del linguaggio GO.

KEYWORDS

MapReduce, Rpc, Fault Tollerant, Locallity Aware Routing, GO, WordCount.

1 Introduzione

Il problema Word Count è risolto dall'applicazione mediante l'esecuzione di funzioni Map e Reduce su dei processi Worker sotto la coordinazione di un processo Master a partire da una suddivisione dell'input del problema in Chunks.

La logica delle funzioni di Map e Reduce sostanzialmente riflette ciò che è esposto in [2] sfruttando una rappresentazione dei dati conveniente e delle facility offerte dal linguaggio GO.

Gli obbiettivi principali che hanno condizionato il design dell'applicazione sono:

1. Minimizzare la problematica relativa al possibile collo di bottiglia del processo Master dovuto a una quantità eccessiva di dati scambiati tra il Master e i Worker
2. Sfruttare la località dei dati durante il routing tra i processi
3. Gestire una possibile presenza di crash dei processi Worker e del processo master in momenti casuali durante l'esecuzione

La prossima sezione tratterà l'architettura dell'applicazione focalizzandosi su alcune sue viste rilevanti, in particolare:

1. Interazioni tra il Master e i Worker
2. Fault Tollerance
3. Locality Aware Routing

Seguirà poi, una descrizione implementazione, una descrizione del deployment ed infine qualche nota sulla configurazione, installazione ed esecuzione dell'applicazione

2 Architettura

L'applicazione è stata progettata con il pattern architetturale Master-Worker, dove il processo Master coordina la computazione distribuita sui i processi Worker in maniera molto simile a come esposto in [1].

Sostanzialmente il Master, stabilisce per ogni Worker su quali Chunk eseguire l'operazione Map, se prevedere che su un Worker sia possibile chiamare l'operazione Reduce e inoltrare le relative richieste.

I Worker si limitano ad eseguire le operazioni richieste dal Master, gestendo possibili scenari di guasto.

Nel seguito, indicherò un Worker che esegue un'operazione specifica di Map o Reduce dicendo rispettivamente che il Worker possiede un'istanza "*Mapper*" o un'istanza "*Reducer*".

Queste istanze sono identificabili rispettivamente dall'ID del Chunk su cui si effettua l'operazione Map e l'ID ottenuto dalla *partitioning function* (vedi [1]) applicata alle

chiavi dei dati intermedi prodotti dai Mapper, che discriminerà una specifica chiamata di Reduce a un *Reducer*.

2.1 Interazioni Master – Worker

Segue un SequenceDiagram esplicativo delle interazioni tra il Master e i Worker in assenza di fallimenti, dove le linee di vita rappresentano i diversi processi dell'applicazione, in particolare, $Worker_{map}$ e $Worker_{reduce}$ indicano rispettivamente generici Worker che eseguiranno rispettivamente le operazioni di Map e Reduce, (eventualmente possono essere coincidenti, dettagli a riguardo nelle prossime sezioni in particolare in 2.3)

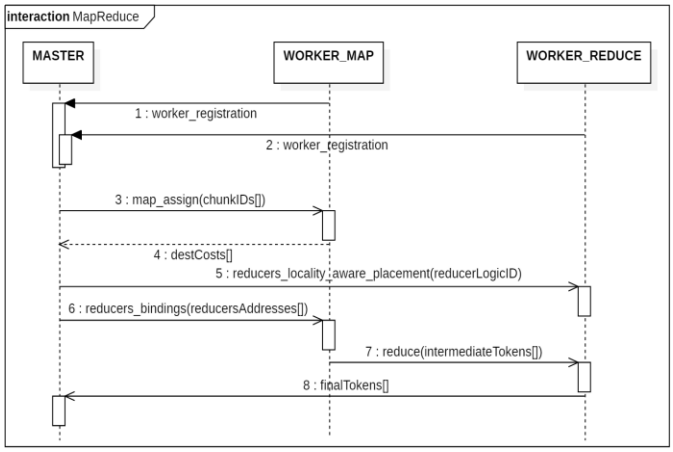

Figura 1: Sequence Diagram dei messaggi scambiati tra il Master e Worker tramite RPC

Registrazione Worker
Il Master espone una porta e la pubblica con il suo indirizzo in uno storage accessibile anche ai processi Worker. I Worker recupereranno questo indirizzo e lo useranno per inviare un messaggio di registrazione al Master, così da poter partecipare alla computazione

Map
Il Master, dopo aver diviso l'input del programma in Chunk, stabilisce un assegnamento di questi, includendo della ridondanza, per ogni processo $Worker_{map}$ ed effettua le chiamate Map relative.
Tali Worker, eseguiranno le operazioni di Map richieste, aggregando i risultati separatamente per ogni Chunk, ritorneranno al Master informazioni di controllo relative al costo del routing dei dati intermedi generati verso i Reducers.

Data locality aware reducers placement
Il Master sfrutterà le informazioni ritornate dai $Worker_{map}$ per stabilire un assegnamento dei Reducers su dei processi Worker attualmente attivi (indicati con $Worker_{reduce)}$ in modo da sfruttare la località dei dati intermedi (dettagli in 2.3).
Successivamente il Master notificherà questi assegnamenti ai $Worker_{map}$ tramite la chiamata 6 e ai $Worker_{reduce}$ tramite la chiamata 5 della figura 1, notificando a quest'ultimi anche informazioni relative alla loro terminazione.

Reduce
Il Master comunica nella chiamata 5 ad ogni $Worker_{map}$ su quale porzione dei loro dati intermedi è necessario effettuare una chiamata di Reduce, con lo scopo di escludere la ridondanza precedente introdotta dal risultato finale.
I $Worker_{map}$ raggrupperanno la porzione indicata dei dati intermedi in loro possesso in base al risultato della partitioning function applicata alle chiavi di tali dati in insiemi separati per ogni Reducer (analogamente all'uso descritto della combiner function in [1]).
Verrà poi effettuata un'unica chiamata di Reduce (chiamata 7) per ogni Reducers da ogni $Worker_{map}$ tramite gl'indirizzi precedentemente comunicati.

Ritorno al master dei token finali aggregati
I Reducers, una volta che hanno ricevuto tutti i dati intermedi dai $Worker_{map,}$ ritornano i dati aggregati finali in loro possesso al Master mediante la chiamata 8 della figura 1
Successivamente il Master, ricevuti i dati aggregati da ogni Reducer, serializzerà i dati finali, comunicherà la terminazione della computazione a tutti i Worker e infine terminerà.

Come è possibile osservare dalla descrizione del Sequence Diagram, ho scelto di evitare un processo di raggruppamento intermedio alle fasi di Map e Reduce (Shuffle and Sort [2]) così da ridurre il traffico scambiato complessivo tra i Worker e il Master. In questo modo si evita un possibile collo di bottiglia nel Master in un'esecuzione con un grande numero di Worker e dei dati in ingresso all'applicazione molto grandi.
Inoltre, è possibile configurare l'applicazione per non includere i dati di un Chunk assegnato a un $Worker_{map}$, ma solo un suo identificativo in uno storage esterno.

In questo modo ad eccezione dell'ultima chiamata in Figura 1, tra il Master e i Worker verranno scambiate unicamente informazioni di controllo.

2.2 Fault Tollerance

Per avere una tolleranza ai guasti dei processi Master e Worker ho introdotto vari meccanismi per gestire dei crash in momenti arbitrari durante un'esecuzione sfruttando i dati presenti sui processi ancora funzionanti.

Il modello di failure dell'applicazione è di tipo fail-silent dato che viene considerata sia l'eventualità di un crash che di una omissione di un Worker

2.2.1 Replicazione dei dati

Al fine di ridurre il costo del re assegnamento di operazioni perse in seguito a fallimenti nei Worker ho aggiunto una classe di processi Worker di backup utilizzati in situazioni di fallimento e un livello configurabile di ridondanza dei Chunk assegnati ai $Worker_{map}$.

Questi Worker di backup verranno utilizzati per rimpiazzare eventuali fallimenti dei Worker durante la registrazione e saranno impiegati anche loro nella fase di Map con un livello di ridondanza superiore a quello assegnato ai $Worker_{map}$.

2.2.2 Crash di un processo Worker

Al termine di ogni chiamata su un processo Worker, il Master rileva la presenza di un errore. Successivamente mediante un semplice servizio di ping, verranno identificati eventuali fallimenti di processi Worker e verranno riassegnate le operazioni perse a Worker funzionanti sfruttando la località dei dati ridondati.

In particolare, un crash durante la fase di Map comporta il solo reassegnamento dei Chunk persi dai Worker falliti non presenti tra i Chunk ridondanti dei Worker funzionati.

Un crash durante la fase di Reduce comporta un re assegnamento dei Chunk persi come sopra, ed eventualmente una reistanziazione di Reducers su processi Worker ancora funzionanti. Nel caso di rilevare fallimenti dopo la chiamata 5 di figura 1, viene evitato di aggregare i dati intermedi, dato che potrebbe verificarsi su un Reducer un accavallamento di dati intermedi provenienti da una nuova e una vecchia aggregazione.

2.2.3 Crash del processo Master

Per gestire un fallimento del processo Master in un momento arbitrario della computazione ho introdotto un processo Master di Backup che subentra nella coordinazione della computazione distribuita su necessita.

Il processo Master principale prima di ogni chiamata sui Worker effettua un backup del suo intero stato in uno storage esterno tollerante ai guasti.

Il processo Master di backup tramite un servizio di Heartbit Monitoring rileva un fallimento del Master principale e in caso subentra nella computazione interrotta, dopo aver eseguito alcune operazioni preliminari. In particolare, dopo aver recuperato lo stato salvato del vecchio Master, controllerà eventuali fallimenti anche nei processi Worker, avendo cura di aspettare che ogni Worker sia in uno stato di inattività prima di proseguire, mediante l'uso di messaggi di stato all'interno del servizio di ping (così da evitare eventuali race conditions nelle chiamate ai Worker immediatamente successive).

Successivamente, in base allo stato del master principale recuperato, il nuovo Master utilizzerà delle chiamate di recovery specifiche sui Worker funzionanti per gestire un eventuale assegnamento parziale delle operazioni nella precedente fase fallita.

2.3 Locality Aware

La località dei dati è tenuta in considerazione nelle decisioni del Master durante la fase di Reduce dell'applicazione e durante il reassegnamento di Chunk necessario al recupero di operazioni di Map.
Come esposto in [3] è possibile sfruttare la località dei dati durante il routing tra processi distribuiti per minimizzare l'overhead di rete dovuto al costo dell'invio di dati tra istanze di applicazioni eseguite su nodi distanti.
La modalità proposta in [3] per fare questo è quella di effettuare un placement intelligente delle istanze delle applicazioni che dovranno scambiarsi i dati, modellando l'intera applicazione distribuita come un grafo orientato pesato sugli archi dove un nodo rappresenta un'istanza di applicazione e un arco tra due nodi rappresenta la necessita di inviare dati tra questi nodi, con costo in base alla dimensione di tali dati.
Adattando quest'idea al modello di computazione MapReduce è possibile sfruttare la località dei dati intermedi dei $Worker_{map}$ nel successivo routing di tali dati verso i Reducers.

Modellando i $Worker_{map}$ e i Reducers in un grafo orientato si ottiene un grafo bipartito dove gli archi sono pesati sul costo del routing dei dati intermedi su cui eseguire l'operazione Reduce.
È quindi possibile "contrarre" alcuni archi di questo grafo risparmiando il costo di tali archi dal costo complessivo del routing da eseguire.
Una buona euristica per eseguire questa operazione è ordinare gli archi di questo grafo per peso decrescente in una lista e successivamente selezionare gli archi da contrarre scorrendo questa lista mantenendo il vincolo di non contrarre più archi verso lo stesso Reducer.
Il Master tramite questa euristica seleziona su quali Worker attivare dei Reducer,
Al fine di evitare un possibile sbilanciamento del carico dovuto ad un eccessivo numero di istanze posizionate su un singolo Worker, viene rispettato un limite configurabile relativo al numero di archi contraibili su uno stesso Worker e viene considerata un'altra classe di Worker gestiti dal Master, adibiti ad eseguire unicamente le operazioni di Reduce.

3 Descrizione dell'implementazione

Segue una descrizione di alcuni parti rilevanti dell'implementazione.

- ## Rappresentazione dei Worker Master-Side

Ogni Worker che si registra presso il Master rientra in una di queste categorie: $Worker_{mapreduce}$, $Worker_{onlyreduce}$, $Worker_{backup}$ rispettivamente se è un Worker che potrà potenzialmente ospitare istanze sia di Mapper sia di Reducers, un Worker che potrà ospitare solo istanze di Reducers o un Worker che potrà ospitare istanze generiche, che verrà utilizzato nella fase di Map con fattore di replicazione dei Chunk assegnati maggiore ed eventualmente nella fase successiva in presenza di guasti.

- ## Rappresentazione dei dati intermedi

Il modello di computazione MapReduce prevede che da un'operazione di Map applicata ad un Chunk si ottengono dei Token intermedi (coppie chiave valore) che poi vengono aggregati insieme agli altri nelle chiamate di Reduce.

Ho deciso di rappresentare questi token intermedi tramite una hashmap così da avere una aggregazione di tali token a livello di singolo Chunk direttamente nell'esecuzione.

- ## Crash simulation

Per simulare fallimenti in momenti arbitrari dell'esecuzione dell'applicazione ho implementato una funzione eseguita su una go routine specifica che va in sleep per una quantità di tempo casuale in un intervallo e al risveglio chiude l'intero processo. Inoltre, sui Worker è scelto casualmente se eseguire questa logica di simulazione a partire da una probabilità configurabile.

- ## Ping e Heartbit monitoring service

Il rilevamento di un guasto su un processo è eseguito mediante un servizio di ping, che prova per un numero configurabile di volte ad inviare dei messaggi a una porta UDP configurabile. Il processo destinatario è considerato funzionate se risponde con un messaggio di pong rappresentativo del suo stato attuale nella computazione distribuita, altrimenti è considerato fallito.

Il servizio di Heartbit monitoring è implementato sui processi Master e Master backup.

La replica del Master proverà ad intervalli regolari a invocare il servizio di ping sul Master principale fino a rilevare un fallimento o lo stato relativo alla terminazione della computazione, o rileva un fallimento.

- ## Concorrenza interna

A livello di singolo processo sono presenti alcune funzioni eseguite correntemente grazie alle facility di programmazione esposte dal linguaggio GO.

In particolare, vengono eseguite correntemente su più Go routine:

1. la suddivisione degl'input in Chunk,
2. l'eventuale download o upload dei Chunk dallo storage esterno,
3. l'esecuzione della funzione Map sui Chunk assegnati ad un Worker,

- ## Worker Recovery RPC

Dato che il processo Master può fallire durante un ciclo di chiamate ai Worker è possibile avere uno stato di parziale inconsistenza tra lo stato dei Worker recuperato dalla replica del master e il reale stato dei Worker.

Per questo motivo, la replica del Master, una volta atteso lo stato Idle su ogni worker ancora funzionante, utilizza chiamate di Recovery sui Worker prima di proseguire l'esecuzione della logica del Master dall'ultimo punto da lui raggiunto.

In particolare, dopo la fase Map verrà controllato se ogni Worker possiede l'output dell'operazione Map sui Chunk a lui precedentemente assegnati, e in caso negativo verrà innescata una logica di recupero dei Chunk eventualmente mancati.

Dopo la fase Reduce, verrà controllato se le istanze Reducer possiedono già tutti i token aggregati finali, altrimenti verrà rieseguita la fase.

- **Assegnamento flessibile delle porte**

Al fine di poter supportare una esecuzione locale su un singolo nodo, è presente un servizio di assegnamento dinamico di porte che, a partire da una richiesta di una specifica porta, procederà a testare porte disponibili fino a trovarne una assegnabile per un generico servizio.

Al fine di sfruttare questo assegnamento dinamico i numeri delle porte assegnate ai servizi interni dell'applicazione vengono inserite in fondo a specifici messaggi di controllo.

4 Deployment

Segue un diagramma di deployment dell'applicazione dove i nodi indicano istanze EC2

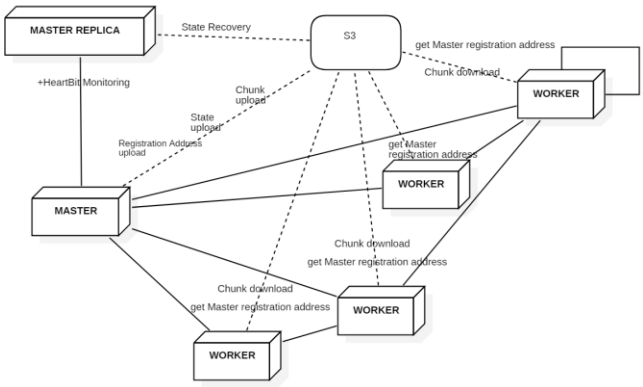

Figura 2: Deployment Diagram finale dell'applicazione

Nel deployment dell'applicazione ogni processo Worker o Master rappresenta un'istanza EC2 di AWS, ad eccezione della replica del master che per semplicità è un processo interno allo stesso nodo del Master principale.

Ogni nodo interagisce con S3 scaricando e caricando file da un bucket.

il Master salva su S3: il suo indirizzo di registrazione che espone ai Worker, eventualmente la suddivisione dell'input in Chunk e il suo stato serializzato tramite gob e codificato in base64 prima di ogni chiamata ai Worker. La replica del Master tramite S3 potrà accedere a quest'ultimo in seguito a un errore sollevato dal Heartbit monitoring del Master principale.

Tutti i Worker scaricano da S3 l'indirizzo di registrazione, precedentemente caricato dal Master, mentre solo quelli che dovranno eseguire una operazione di Map scaricheranno dei Chunk.

Il Master interagisce con ogni Worker e alcuni Worker interagiscono tra di loro secondo la semantica illustrata in 2.1. Inoltre, dato il placement dei Reducers per sfruttare la località dei dati, esistono Worker che interagiscono con loro stessi, ovvero dove un Reducer è inserito sullo stesso nodo dove sono presenti dei Mappers.

5 Configurazione

Segue una descrizione dei principali parametri configurabili dell'applicazione tramite il file *config.json*

1. *LoadChunksToS3:* permette di caricare i Chunk generati dall'input del programma su S3 nel master, e di scaricarli nei Worker.
   In caso settato a false i Chunk saranno allegati dal Master nelle richieste di Map ai Worker
2. *CHUNK_SIZE:* la dimensione massima di ogni Chunk
3. *CHUNKS_REPLICATION_FACTOR:* numero di Chunk ridondanti da assegnare ad ogni $\text{Worker}_{\text{mapreduce}}$
4. *CHUNKS_REPLICATION_FACTOR_BACKUP_WORKERS:* numero di chunk da assegnare ai $\text{Workers}_{\text{backup}}$
5. *WORKER_NUM_MAP:* numero di $\text{Worker}_{\text{mapreduce}}$
6. *WORKER_NUM_ONLY_REDUCE:* numero di $\text{Worker}_{\text{onlyreduce}}$
7. *WORKER_NUM_BACKUP_WORKER:* numero di $\text{Worker}_{\text{backup}}$

8. *MAX_REDUCERS_PER_WORKER:* numero Massimo di instanze Reducers piazzabili su un $Worker_{mapreduce}$
9. *SIMULATE_WORKERS_CRUSH:* abilita la simulazione dei Guasti in momenti arbitrari su tutti i Worker
10. *BACKUP_MASTER:* abilita la logica di serializzazione e upload dello stato del Master
11. *SIMULATE_WORKERS_CRUSH_NUM:* numero medio di Worker casuali su cui eseguire la simulazione di guasto

Sono inoltre presenti altri campi relativi ai timeout e alle porte da assegnare ai vari servizi.

6 Installazione ed esecuzione

A partire da un clone del repository dell'applicazione da `https://andysnake96@bitbucket.org/andysnake96/mapreduceextended.git` è possibile accedere a diversi script che permettono di eseguire l'applicazione, in particolare *startEC2istances.sh*.

Con vari argomenti che si possono passare allo script in oggetto è possibile interagire con le varie istanze EC2 tramite awscli, i principali sono:

- Con argomento *master* verrà avviata un'istanza EC2 con un semplice script di inizializzazione per scaricare l'intero codice e compilare il codice del master tramite comandi di go racchiusi in un Makefile

- Con 1 argomento numerico *n* si istanziano *n* istanze EC2 con un LaunchTemplate che contiene un semplice script di inizializzazione per scaricare l'intero codice e compilare la parte relativa al Worker tramite comandi di go racchiusi in un Makefile.

  Successivamente verrà avviato un processo Worker e un altro processo in backgroud che tramite l'ascolto di nuove comunicazioni da una porta configurabile permette di riavviare nuovamente un processo Worker, caricando su S3 un log relativo a una redirezione dell'esecuzione precedente

- Con argomento *relay:* si inizializza un ambiente d'esecuzione generico per il debugging. Sostanzialmente viene avviata un'istanza EC2 con un LaunchTemplate che contiene uno script che esegue operazioni di aggiunta al file di configurazione del demone *sshd* e pubblica il suo indirizzo ai Worker mediante S3.

  Successivamente viene instaurato un *reverse ssh tunnel* per ogni porta che il master espone ai Worker. In questo modo è possibile eseguire il Master su una macchina locale che si connette ad internet tramite una doppia Nat o una singola non configurabile, dato che tutte le connessioni che il Master ha bisogno di ricevere le vengono inoltrate dall'istanza di relay attivata.

- Con argomento *restart:* tramite un file temporaneo di appoggio (*instances)* è possibile riavviare i processi Worker sulle istanze EC2

- Con argomenti *spawn* e *spawn_ssh_to* rispettivamente: viene aperta una shell ssh per ogni instanza EC2 il cui hostname publico è presente all'interno di un file di appoggio *(instances)* o una singola shell ssh verso l'istanza EC2 passata come secondo argomento

- *get_hostnames* permette di recuperare tutti gli hostnames pubblici delle istanze EC2 attualmente attive

Limitazioni Riscontrate

In configurazioni particolarmente sfavorevoli di fallimento durante l'operazione di Reduce è possibile riscontrare un ritardo nell'applicazione, probabilmente dovuto alla gestione di GO di varie go-routine relative a numerose chiamate RPC in contrasto con la sincronizzazione necessaria nei Reducers durante l'operazione Reduce (dato l'update di strutture dati che la documentazione indica come da proteggere da particolari accessi concorrenti)

REFERENCES

[1] MapReduce:SimplifiedDataProcessingonLargeClusters https://static.googleusercontent.com/media/research.google.com/it//archive/mapreduce-osdi04.pdf

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

[3] Locality-Aware Routing in Stateful Streaming Applications https://matthieu.io/dl/papers/storm-locality-middleware-2016.pdf