Fault-tolerant MapReduce using Go and RPCs

Project B1 – Sistemi Distribuiti e Cloud Computing – A.A. 2024/2025

Dennis Mariani
Corso di Laurea Magistrale in Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
Matricola 0365494
dennis.mariani@students.uniroma2.eu

***Abstract* — Questo report descrive la progettazione e l'implementazione di un sistema MapReduce tollerante ai guasti, sviluppato in linguaggio Go per eseguire l'ordinamento distribuito di un insieme di numeri interi. L'architettura adotta il modello master-worker, dove il nodo master coordina il lavoro di mapper e reducer, utilizzando una comunicazione basata su RPC. Il sistema garantisce affidabilità rilevando guasti e riassegnando dinamicamente i task ai worker, grazie alla persistenza dello stato sia in locale sia su Amazon S3. Inoltre, è presente un meccanismo di monitoraggio che consente il riavvio automatico del master in caso di crash. Tutti i nodi coinvolti sono dei container Docker, orchestrati mediante Docker Compose, ed è possibile sia l'esecuzione in locale sia su un'istanza EC2 di AWS.**

***Keywords* — MapReduce, Distributed Systems, Fault Tolerance, Go, RPC, Docker, Docker Compose, AWS EC2, Amazon S3, Master-Worker Architecture**

I. INTRODUCTION

Il paradigma MapReduce è utilizzato in ambienti distribuiti in cui i dati sono elaborati da più nodi. Nello specifico, consente una scalabilità efficace e una gestione automatica di grandi volumi di dati mediante due operazioni fondamentali, chiamate *map* e *reduce*, che verranno presentate nel dettaglio nelle prossime sezioni.

La traccia B1 richiedeva di implementare tale paradigma utilizzando i servizi offerti dall'*AWS Learner Lab*, per dare vita ad un'architettura distribuita e containerizzata, con supporto alla tolleranza ai guasti, senza hardcoding dei parametri e utilizzando tecniche di service discovery. Dunque, mi sono posto l'obiettivo di realizzare un sistema robusto, facilmente configurabile e automatizzabile; ho altresì eseguito dei test di fault injection e valutazione delle prestazioni, variando anche il numero di nodi e il volume dei dati. Le principali sfide affrontate durante lo sviluppo del progetto riguardano la coordinazione del lavoro dei vari nodi coinvolti e la gestione dei fault con persistenza dello stato.

II. BACKGROUND

Per rispettare i requisiti richiesti dall'assegnazione è stata utilizzata un'istanza *Amazon EC2* (Elastic Compute Cloud), un servizio di *AWS* che consente di creare e gestire macchine virtuali nel cloud. Su questa istanza, il sistema è stato eseguito in modo distribuito sfruttando: *Docker*, una piattaforma che permette di creare ambienti isolati (*container*) per ogni componente del sistema (master, mapper, reducer); e *Docker Compose*, per gestire gruppi di container in modo coordinato tramite un file di configurazione. Infine, per garantire la persistenza dello stato e supportare il recovery dopo eventuali crash, è stato impiegato *Amazon S3 (*Simple Storage Service), un servizio di storage altamente disponibile.

Per concludere, come già accennato, il sistema sviluppato si basa sul modello master/worker, una tipica architettura per sistemi distribuiti. Difatti, il master ha capacità di coordinazione dell'elaborazione, divisione dei dati in chunk e successiva assegnazione ai nodi worker, che eseguiranno le operazioni di map e reduce. Tutta l'implementazione è stata sviluppata in linguaggio *Go*, che fornisce supporto nativo per *RPC* (Remote Procedure Call) tramite il pacchetto *net/rpc*, disponibile nella libreria standard. La comunicazione RPC consente a un processo di invocare metodi remoti su altri processi, come se fossero funzioni locali, passando strutture dati serializzate tra client e server. A differenza di altri framework RPC, net/rpc non richiede un file IDL separato, bensì i metodi remoti vengono definiti direttamente nel codice Go, rispettando alcune convenzioni.

III. SOLUTIONS DESIGN

A. Architettura Generale

Il sistema è organizzato secondo l'architettura master-worker, con quattro componenti principali:

1. **Master**, gestisce la coordinazione dell'intero flusso MapReduce e combina i risultati finali;
2. **Mapper**, riceve un chunk, dal master, lo ordina localmente e suddivide i valori in sotto-chunk da inviare ai reducer;
3. **Reducer**, riceve più sotto-chunk da diversi mapper, li unisce e scrive il risultato in un file locale;
4. **Standby Controller**, monitora periodicamente il master e lo riavvia automaticamente in caso di crash garantendo la continuità del servizio.

Tutte le componenti sono containerizzate con Docker e orchestrate con Docker Compose, eseguibili localmente o su AWS EC2.

B. Flusso del Sistema

1. **Generate**: Il master genera N numeri casuali nel range [xi, xf], dove tali parametri sono definiti nel file di configurazione config.json.
2. **Split**: I dati sono suddivisi in maniera uniforme in M chunk (uno per ciascun mapper) e inviati ai mapper dal master.
3. **Map**: Ogni mapper ordina localmente il proprio chunk e partiziona i dati in base ai range dei reducer, inviando le sottoporzioni al reducer designato.

4. **Reduce**: Ogni reducer riceve i sotto-chunk, li unisce e scrive su file temporanei.
5. **Combine**: Il master raccoglie i file dei reducer e crea un file finale ordinato.

C. *Strategia di Partizionamento*

Per bilanciare il carico tra i reducer, il master applica una strategia basata su sampling:

- Viene estratto casualmente un 10% del dataset per formare un campione rappresentativo su cui determinare i boundary values, anziché gestire l'intero volume di dati (altrimenti non avrebbe più senso parallelizzare il lavoro richiesto e gestirlo in maniera distribuita).
- Il sample viene ordinato e da esso vengono estratti N–1 punti equidistanti.
- I valori ottenuti definiscono N intervalli disgiunti del dominio di partenza [xi, xf], ciascuno assegnato a un reducer, che riceverà solo i valori appartenenti al suo range.

Questa tecnica consente di ottimizzare il bilanciamento anche in presenza di dataset non uniformi riducendo, così, il rischio di congestione su un singolo reducer. Ovviamente gli effetti sono maggiormente apprezzabili all'aumentare dell'intervallo [xi, xf] e della quantità di dati generati.

D. *Fault Model*

- **Master**: è presente uno Standby Controller che verifica periodicamente la disponibilità del master tramite chiamate RPC e, in caso di crash rilevato per 3 tentativi consecutivi, esegue un riavvio automatico del container del master. Quest'ultimo, al riavvio, ripristina lo stato grazie ai file salvati localmente e/o su Amazon S3.
- **Mapper/Reducer**: in caso di mancata risposta entro un timeout, i task vengono riassegnati dinamicamente ad altri nodi attivi mediante un meccanismo di retry, che viene loggato negli appositi file per future analisi.

IV. SOLUTION DETAILS

A. *Master*

Il componente Master si occupa del coordinamento dell'intero processo MapReduce, operando come nodo centrale dell'architettura master-worker. Dopo aver configurato il logger (tramite utils.SetupLogger) ed effettuato la pulizia dei file di stato e output precedenti, il master carica la configurazione da file (config/config.json) e inizializza la propria struttura interna (Master{Workers, Settings}). Successivamente, il master espone la propria interfaccia RPC tramite il metodo rpcServer.Register(&master), rendendo i suoi metodi remoti accessibili agli altri nodi del sistema. In Go un metodo può essere esposto via RPC solo se rispetta le seguenti convenzioni obbligatorie:

- Il tipo su cui è definito deve essere esportato, cioè il nome deve iniziare con una lettera maiuscola (in questo caso type Master struct).
- Il metodo stesso deve essere esportato, quindi, il nome deve iniziare con una lettera maiuscola.
- Deve avere due argomenti:
    1. una struttura di input (in questo caso utils.WorkerConfig);
    2. un puntatore a una struttura di output (in questo caso reply *bool).
- Deve restituire un error.

Dopo la registrazione, il master apre un listener TCP sulla porta 9000 con net.Listen("tcp", ":9000") e accetta connessioni RPC in ingresso tramite rpcServer.Accept(listener). I worker (sia mapper che reducer) si registrano al master tramite una chiamata RPC al metodo Register, che verifica che l'indirizzo del worker non sia già presente e, se nuovo, lo aggiunge alla lista (m.Workers). Assistiamo poi a fasi successive in cui:

1. **Attesa Worker**: con WaitForWorkers(m, r) il master attende la registrazione di NumMappers e NumReducers (specificati all'avvio del sistema) monitorando il tempo massimo di attesa. Poi viene aggiornato il file workers.json, per supportare il recupero dello stato in caso di crash, con utils.SaveWorkerOnRegister().
2. **Generazione Dati**: con GenerateData(count, xi, xf) vengono generati count numeri casuali, nell'intervallo desiderato [xi, xf], e salvati in data.json.
3. **Suddivisione in Chunk**: i dati vengono suddivisi in M chunk tramite SplitData, e il risultato viene reso persistente in chunks.json con SaveChunksToFile.
4. **Partizionamento Reducer**: la funzione MapReducersToRanges esegue un sampling del 10% dei dati e calcola N-1 punti per creare intervalli bilanciati.
5. **Fase Map**: ogni chunk viene assegnato dinamicamente a un mapper disponibile attraverso chiamate RPC gestite in concorrenza. Per ciascun chunk, viene lanciata una goroutine, che effettua una chiamata al metodo remoto Worker.MapTask, usando la funzione CallWithFallbackMapBusy. Tale funzione implementa una logica di retry con fallback, in cui il master tenta di inviare la richiesta RPC al primo mapper disponibile, scartando quelli marcati come "occupati" (gestiti tramite la struttura thread-safe ThreadSafeMap). Se la connessione o l'invocazione fallisce, il master registra l'errore nel log e passa al mapper successivo. Per via del parametro maxRetries, questo processo viene ripetuto per un massimo di 5 tentativi in seguito i quali, per evitare il blocco dell'intero sistema, il task viene considerato fallito e loggato nel file worker_failed_tasks.log. La risposta attesa per ottenere un esito positivo è un oggetto di tipo MapReply che deve contenere Ack=true. In caso di conferma il mapper viene segnato come libero perché ha terminato il processo con successo e la goroutine termina.
6. **Fase Combine**: i file temporanei generati dai reducer vengono uniti nel file final_output.txt con CombineOutputFiles().

7. **Reset**: Una volta completata la fase di Combine, viene settato un flag di completamento tramite SaveCompletionFlag() per evitare duplicazioni in run successivi, e infine il sistema viene riportato allo stato iniziale con ResetState().

In tutto il codice main.go ci sono numerosi controlli condizionali per comprendere se il master è stato riavviato in seguito ad un crash (controllando i file di stato generati durante il flusso, data.json, chunk.json, workers.json e status.json) ed eventualmente continuare l'esecuzione dal punto in cui ci si era interrotto.

B. Mapper

Uno dei due ruoli che può assumere un nodo Worker è quello del Mapper, che si occupa di ordinare localmente i chunk assegnati dal master e di suddividerli per inviarli ai reducer. Dopo l'inizializzazione del logger e la lettura dei parametri, il mapper tenta di registrarsi al master con la funzione registerSelf(address, role, masterAddr), eseguendo ciclicamente una chiamata RPC che invoca il metodo remoto Register esposto dal master. Il tipo della richiesta è utils.WorkerConfig e specifica il ruolo ("mapper") e il proprio indirizzo, attendendo un booleano che conferma l'avvenuta registrazione. Per rendere il sistema resiliente a ritardi nell'avvio del container del master, se il master non è ancora attivo o non risponde, il worker aspetta 3 secondi e riprova.

Dopo la registrazione, il mapper crea un'istanza della struttura Worker e la registra presso il proprio server RPC locale con la chiamata server.Register(worker). In questo modo espone i propri metodi, nello specifico MapTask, che potrà essere invocato da remoto dal master e che, ovviamente, rispetta tutte le convenzioni precedentemente descritte. Dunque, il flusso del metodo MapTask è il seguente:

1. **Ordinamento del chunk**: il mapper ordina localmente il vettore di interi ricevuto nel campo req.Chunk tramite sort.Ints.
2. **Scrittura su file**: il chunk ordinato viene salvato su file temporaneo (temp_<hostname>.txt) per supportare eventuali analisi o debugging.
3. **Partizionamento per reducer**: utilizzando la mappa req.ReducerRanges, che specifica i range numerici assegnati a ciascun reducer, il mapper divide i numeri in sotto-chunk e li assegna al reducer corrispondente.
4. **Invio ai reducer**: per ogni sotto-chunk, viene invocata SendToReducerWithFallback, che cerca di inviare i dati al reducer primario. In caso di fallimento (es. timeout o errore RPC), tenta altri reducer disponibili, reagendo ad eventuali crash dei reducer.
5. **ACK al master**: una volta completato l'invio, il mapper imposta reply.Ack = true per segnalare al master che il task è stato eseguito con successo.

C. Reducer

Il secondo ruolo che può assumere un nodo Worker è quello di Reducer. Quest'ultimo è in grado di ricevere i sotto-chunk ordinati dai vari mapper, unirli e scriverli su file locali che verranno combinati successivamente dal master. L'avvio e la registrazione del reducer al master avvengono esattamente come per i mapper e anche il reducer espone i propri metodi RPC, in particolare, ReduceTask. Quest'ultimo viene invocato dai mapper al termine della fase di partizionamento e funziona nell'ordine seguente:

1. **Ricezione dei dati**: la struttura utils.ReduceRequest contiene il sotto-chunk da elaborare nel campo Chunks, mentre il campo Owner rappresenta l'indirizzo del reducer primario (quello a cui i dati erano inizialmente destinati), anche se il task viene eventualmente eseguito da un reducer di backup in seguito ad un eventuale crash del primario.
2. **Generazione del file temporaneo**: il reducer apre (in modalità append) un file con nome temp_<Owner>.txt, che quindi resta coerente anche se il task viene completato da un altro reducer.
3. **Scrittura del contenuto**: ciascun numero del sotto-chunk viene scritto su una riga separata del file utilizzando un writer buffered (bufio.NewWriter) per una scrittura efficiente e per evitare accessi frequenti al disco.
4. **ACK al master**: una volta completata la scrittura, viene impostato reply.Ack = true.

D. Standby Controller

Il componente Standby Controller è utilizzato per garantire la tolleranza ai guasti anche del nodo master. Esso consiste semplicemente in un processo che monitora periodicamente lo stato del master e ne esegue il riavvio in caso di crash. All'avvio, il controller aspetta 10 secondi per consentire il bootstrap degli altri container, poi entra in un ciclo periodico in cui:

1. **Controlla il completamento della computazione**: se esiste il file completed.json, il controller comprende che il processo di MapReduce è terminato correttamente e invoca shutdownAll() per eseguire docker-compose down, arrestando l'intero sistema.
2. **Verifica lo stato del master**: ogni 7 secondi tenta di aprire una connessione RPC al master all'indirizzo master:9000. Se la chiamata fallisce, incrementa un contatore failCount.
3. **Gestione dei fallimenti**:
   - Al primo fallimento, attende altri 7 secondi e ricontrolla se nel frattempo la computazione è terminata;
   - Dopo 3 fallimenti consecutivi, considera il master effettivamente crashato ed esegue docker container start master tramite la funzione restartMaster() per riavviare il container corrispondente.

Questo meccanismo garantisce la disponibilità master anche in caso di crash isolati, senza necessità di intervenire manualmente, e permette di riprendere il flusso da dove era stato interrotto grazie al salvataggio dello stato effettuato dal master.

E. Utils

Il modulo utils/ racchiude tutte le funzioni di utilità che possono essere condivise tra i componenti del sistema.

Nel file *helpers.go* troviamo le strutture dati usate per la comunicazione RPC e la funzione LoadConfig(path) per leggere il file config.json contenente parametri di sistema e worker. Nello specifico, le strutture sono:

- **MapRequest**, l'oggetto che il master invia a ciascun mapper, che contiene: Chunk []int, ovvero il sottoinsieme di dati da ordinare; ReducerRanges map[string][2]int, una mappa dove a ogni indirizzo di reducer è associato un range di valori [min, max).
- **MapReply**, è la risposta del mapper che include un solo campo Ack bool indicante se il task è stato completato correttamente.
- **ReduceRequest**, si tratta della richiesta che il mapper invia al reducer, contiene: Chunks []int, ovvero i dati da salvare; WorkerAddress string, cioè l'indirizzo del reducer che riceve la chiamata; Owner string, identifica il reducer primario per cui il chunk era inizialmente destinato.
- **ReduceReply**, come MapReply, contiene solo Ack bool per confermare l'avvenuta scrittura.

Queste strutture rispettano le convenzioni di Go RPC, che richiedono che i tipi siano esportati (iniziano con la maiuscola). Poi abbiamo:

- Nel file *logging.go* troviamo le funzioni: SetupLogger(), per indirizzare l'output del logger verso un file specifico, e AppendToFile(), per scrivere una riga nel file specificato in modalità append.
- Il file *cleanup.go* contiene funzioni per la rimozione dei file di output finali e temporanei.
- Invece, *threadsafe.go* contiene la struttura ThreadSafeMap che permette di sincronizzare l'accesso a una mappa string → bool, usata dal master per marcare i worker occupati e prevenire race condition durante l'invio concorrente di RPC.

Infine, *state.go* include le funzioni per salvare, recuperare e sincronizzare lo stato del sistema. Abbiamo:

- SaveCompletionFlag() crea il file completed.json per indicare che l'elaborazione è stata completata con successo, mentre CompletionFlagExists() e RemoveCompletionFlag() lo rilevano o rimuovono.
- InitStatusFile(nChunks int) inizializza status.json, segnando ciascun chunk come "pending".
- SaveStatusAfterChunk(i int) aggiorna il chunk i come "done", riscrivendo l'intero file. La scrittura è thread-safe, protetta da mutex (statusMu).
- PhaseAlreadyDone() verifica se tutti i chunk risultano "done" in status.json per capire se la fase Map è già stata completata in run precedenti.
- SaveDataToFile() e SaveChunksToFile() salvano i dati generati e la divisione in chunk effettuata dal master. Poi DataFileExists() e ChunkFileExists() verificano la presenza dei rispettivi file. Infine, LoadDataFromFile() e LoadChunksFromFile() caricano eventualmente i file.
- SaveWorkerOnRegister() salva l'elenco dei worker registrati in workers.json mentre WorkersFileExists() verifica la presenza di tale file. Inoltre, RecoverWorkersFromFile() viene chiamata dal master ed è utile in caso di riavvio dopo un crash perché permette di ripristinare i worker già registrati.
- RecoverPendingChunks() osserva status.json e chunk.json per ricostruire solo i chunk ancora pendenti, in caso di crash a esecuzione iniziata.
- ResetState() elimina tutti i file di stato e rimuove anche le copie da S3.

Molte funzioni prevedono l'upload o il download su Amazon S3 se la variabile d'ambiente ENABLE_S3 è attiva.

F. Other Files

Oltre ai componenti presentati ci sono dei file aggiuntivi per gestire il deploy, la configurazione, la gestione dei container Docker e la simulazione dei fault. Primo fra tutti è docker-compose.yml che definisce l'intera architettura containerizzata ed è accompagnato da un Dockerfile per ciascun componente. Poi abbiamo il file config/config.json, che, come già accennato, specifica numero di mapper/reducer utilizzati in caso di default e i parametri della generazione dati (xi, xf, count). Infine, .env contiene le credenziali temporanee AWS necessarie per caricare/scaricare file di stato su Amazon S3, indicando anche il nome del bucket. Per concludere abbiamo diversi script di automazione:

- run_EC2.sh aggiorna config.json, costruisce le immagini e avvia i container in base al numero di mapper/reducer specificato.
- init_env.sh crea un file .env vuoto da popolare con credenziali AWS.
- kill_master.sh simula il crash del master (usato per testare il recovery da parte di standby).
- kill_random_mapper.sh e kill_random_reducer.sh selezionano e terminano un mapper/reducer casuale, utile per test di fault injection.
- clean_cache.sh rimuove cache e immagini Docker.
- view_output.sh visualizza il contenuto del file final_output.txt prodotto a fine esecuzione.
- view_master_log.sh mostra il log del master.

V. RESULTS

A. Esecuzione

Di seguito viene mostrata un'esecuzione completa e corretta del sistema MapReduce, senza fault. In particolare, il run è stato avviato con i seguenti parametri:

- Intervallo di generazione: xi = 1, xf = 50
- Numero di valori generati: count = 100
- Numero di mapper/reducer: 4×4

Possiamo osservare i dati generati in data.json:

```
[1, 33, 46, 41, 21, 37, 22, 16, 9, 31, 7, 1, 15, 19, 14, 27, 38, 44,
46, 15, 27, 9, 22, 38, 25, 42, 9, 23, 28, 14, 38, 4, 33, 39, 50, 39,
45, 18, 5, 5, 31, 16, 1, 28, 26, 25, 22, 19, 40, 24, 26, 43, 3, 18,
3, 38, 39, 5, 5, 46, 29, 22, 26, 21, 13, 7, 33, 38, 12, 17, 17, 17,
38, 1, 36, 13, 44, 11, 6, 36, 31, 22, 50, 9, 26, 41, 37, 23, 28, 26,
5, 24, 47, 46, 1, 24, 32, 28, 40, 50]
```

Per poi passare alla divisione in chunk nel file chunks.json:

```
[
  [1, 33, 46, 41, 21, 37, 22, 16, 9, 31, 7, 1, 15, 19, 14, 27, 38, 44, 46, 15, 27, 9, 22, 38, 25],
  [42, 9, 23, 28, 14, 38, 4, 33, 39, 50, 39, 45, 18, 5, 5, 31, 16, 1, 28, 26, 25, 22, 19, 40, 24],
  [26, 43, 3, 18, 3, 38, 39, 5, 5, 46, 29, 22, 26, 21, 13, 7, 33, 38, 12, 17, 17, 17, 38, 1, 36],
  [13, 44, 11, 6, 36, 31, 22, 50, 9, 26, 41, 37, 23, 28, 26, 5, 24, 47, 46, 1, 24, 32, 28, 40, 50]
]
```

E infine al file di output final_output.txt:

```
1
1
5
7
9
12
13
14
15
16
1
3
5
9
16
1
1
3
5
6
9
11
15
4
5
5
7
9
13
14

17
18
19
23
19
22
22
17
17
18
21
21
22
22
22
23

24
25
26
26
26
24
25
26
24
26
```

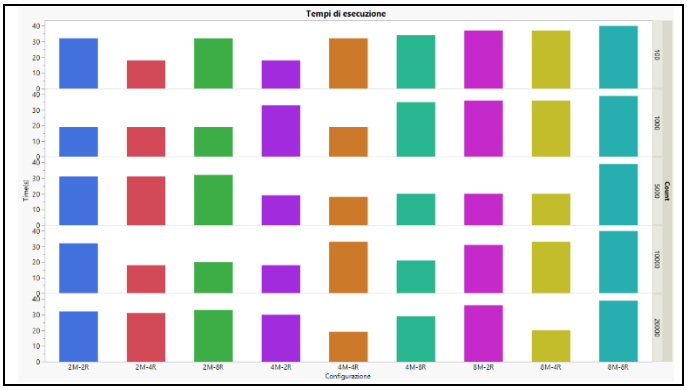
```
31
31
33
38
40
41
47
50
50
28
28
29
33
37
38
38
39
43
46
50
28
31
32
36
36
37
38
38
40
41
44
44
46
46
27
27
28
33
38
39
39
42
45
46
```

Per completezza mostriamo il file di log prodotto dal master:

```
[MASTER] 2025/07/14 12:40:24 [STATE] Scaricato completed.json da S3
[MASTER] 2025/07/14 12:40:24 [STANDBY] completed.json rilevato correttamente
[MASTER] 2025/07/14 12:40:24 File output/final_output.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_3922f07581ae.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_406c56e95c83_9001.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_43c800c0dbe1_9001.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_631f1d7630f1_9001.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_806184549356.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_b3ea2861305d_9001.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_b759cc02e01e.txt rimosso.
[MASTER] 2025/07/14 12:40:24 File temporaneo output/temp_c921104cbbec.txt rimosso.
[MASTER] 2025/07/14 12:40:24 [STATE] completed.json rimosso
[MASTER] 2025/07/14 12:40:25 completed.json rimosso da S3: s3://sdcc-mapreduce-recovery/state/completed.json
[MASTER] 2025/07/14 12:40:25 Master RPC server in ascolto su :9000 per registrazioni
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: abe7c1985845:9001 (reducer)
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: e427d7366649:9001 (mapper)
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: d67dd156220c:9001 (reducer)
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: 5060929d2e81:9001 (mapper)
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: 343e9c5719ba:9001 (reducer)
[MASTER] 2025/07/14 12:40:25 Registrato nuovo worker: da804ec0e55a:9001 (mapper)
[MASTER] 2025/07/14 12:40:26 Registrato nuovo worker: 4de7a101aaa1:9001 (reducer)
[MASTER] 2025/07/14 12:40:26 Errore download status.json da S3: exit status 1
Output: fatal error: An error occurred (404) when calling the HeadObject operation: Key "state/status.json" does not exist
[MASTER] 2025/07/14 12:40:26 [STATE] Nessun file di stato trovato: skip PhaseAlreadyDone.
[MASTER] 2025/07/14 12:40:26 Attendo la registrazione di 4 mapper e 4 reducer...
[MASTER] 2025/07/14 12:40:26 Registrati finora: 3 mapper, 4 reducer
[MASTER] 2025/07/14 12:40:27 Registrati finora: 3 mapper, 4 reducer
[MASTER] 2025/07/14 12:40:27 Registrato nuovo worker: 5b1dcb00c0ba:9001 (mapper)
[MASTER] 2025/07/14 12:40:28 Registrati finora: 4 mapper, 4 reducer
[MASTER] 2025/07/14 12:40:28 Tutti i worker sono registrati, si può partire.
[MASTER] 2025/07/14 12:40:28 [STATE] workers.json salvato correttamente.
[MASTER] 2025/07/14 12:40:29 Upload workers.json su S3 riuscito: s3://sdcc-mapreduce-recovery/state/workers.json
[MASTER] 2025/07/14 12:40:29 [STATE] Dati salvati in state/data.json
[MASTER] 2025/07/14 12:40:30 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/data.json
[MASTER] 2025/07/14 12:40:30 [STATE] Chunk salvati in state/chunks.json
[MASTER] 2025/07/14 12:40:31 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/chunks.json
[MASTER] 2025/07/14 12:40:31 [STATE] Stato inizializzato in state/status.json
[MASTER] 2025/07/14 12:40:32 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:32 sample = [1 37 25 50 45 41 44 6 19 6]
[MASTER] 2025/07/14 12:40:32 sample ordinato= [1 6 6 19 25 37 41 44 45 50]
[MASTER] 2025/07/14 12:40:32 ranges= [6 25 41]
[MASTER] 2025/07/14 12:40:32 ranges = [6 25 41]
[MASTER] 2025/07/14 12:40:32 Reducer abe7c1985845:9001 gestisce l'intervallo [1, 6]
[MASTER] 2025/07/14 12:40:32 Reducer d67dd156220c:9001 gestisce l'intervallo [6, 25]
[MASTER] 2025/07/14 12:40:32 Reducer 343e9c5719ba:9001 gestisce l'intervallo [25, 41]
[MASTER] 2025/07/14 12:40:32 Reducer 4de7a101aaa1:9001 gestisce l'intervallo [41, 51]
[MASTER] 2025/07/14 12:40:32 Reducer effettivamente utilizzati: 4
2025/07/14 12:40:37 [MAP-01] Completato con successo da 5060929d2e81:9001
```

```
[MASTER] 2025/07/14 12:40:37 MAP-01 completato
[MASTER] 2025/07/14 12:40:37 [STATE] Stato aggiornato: chunk 1 → done
2025/07/14 12:40:37 [MAP-00] Completato con successo da 5b1dcb00c0ba:9001
2025/07/14 12:40:37 [MAP-02] Completato con successo da da804ec0e55a:9001
[MASTER] 2025/07/14 12:40:37 MAP-00 completato
[MASTER] 2025/07/14 12:40:37 MAP-02 completato
2025/07/14 12:40:37 [MAP-03] Completato con successo da e427d7366649:9001
[MASTER] 2025/07/14 12:40:37 MAP-03 completato
[MASTER] 2025/07/14 12:40:38 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:38 [STATE] Stato aggiornato: chunk 0 → done
[MASTER] 2025/07/14 12:40:39 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:39 [STATE] Stato aggiornato: chunk 2 → done
[MASTER] 2025/07/14 12:40:40 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:40 [STATE] Stato aggiornato: chunk 3 → done
[MASTER] 2025/07/14 12:40:41 Upload su S3 riuscito: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:41 Fase di Map completata.
[MASTER] 2025/07/14 12:40:41 Unisco il file temporaneo: output/temp_abe7c1985845_9001.txt
[MASTER] 2025/07/14 12:40:41 Unisco il file temporaneo: output/temp_d67dd156220c_9001.txt
[MASTER] 2025/07/14 12:40:41 Unisco il file temporaneo: output/temp_343e9c5719ba_9001.txt
[MASTER] 2025/07/14 12:40:41 Unisco il file temporaneo: output/temp_4de7a101aaa1_9001.txt
[MASTER] 2025/07/14 12:40:41 Output finale scritto in: output/final_output.txt
[MASTER] 2025/07/14 12:40:41 [STATE] Flag completamento salvato e flushato su disco.
[MASTER] 2025/07/14 12:40:42 Upload completed.json su S3 riuscito: s3://sdcc-mapreduce-recovery/state/completed.json
[MASTER] 2025/07/14 12:40:42 [STATE] File state/status.json rimosso
[MASTER] 2025/07/14 12:40:42 [STATE] File state/data.json rimosso
[MASTER] 2025/07/14 12:40:42 [STATE] File state/chunks.json rimosso
[MASTER] 2025/07/14 12:40:42 [STATE] File state/workers.json rimosso
[MASTER] 2025/07/14 12:40:43 File rimosso da S3: s3://sdcc-mapreduce-recovery/state/status.json
[MASTER] 2025/07/14 12:40:44 File rimosso da S3: s3://sdcc-mapreduce-recovery/state/data.json
[MASTER] 2025/07/14 12:40:45 File rimosso da S3: s3://sdcc-mapreduce-recovery/state/chunks.json
[MASTER] 2025/07/14 12:40:46 File rimosso da S3: s3://sdcc-mapreduce-recovery/state/workers.json
```

B. Prestazioni



Dalle analisi condotte mediante il file benchmark_test.sh notiamo alcune cose interessanti:

- All'aumentare dei dati (parametro count) il tempo di esecuzione non cresce in modo lineare, e tende a rimanere abbastanza stabile e contenuto. Questo mostra un buon grado di parallelismo e distribuzione del carico.
- All'aumentare del numero di mapper e reducer non si ottiene un miglioramento, anzi notiamo dei leggeri aumenti dei tempi di esecuzione, probabile sintomo di overhead di coordinamento. È probabile che con dataset di dimensioni maggiori si riuscirebbero a percepire meglio i benefici della scalabilità orizzontale.
- In alcune configurazioni il tempo è contro intuitivamente più basso di altre esecuzioni con count minore. Questo potrebbe essere dovuto a bilanciamenti particolarmente fortunati del carico ottenuti mediante il sampling.
- Nessuna esecuzione ha terminato in timeout, il che indica che il sistema è robusto, anche in presenza di carichi importanti.

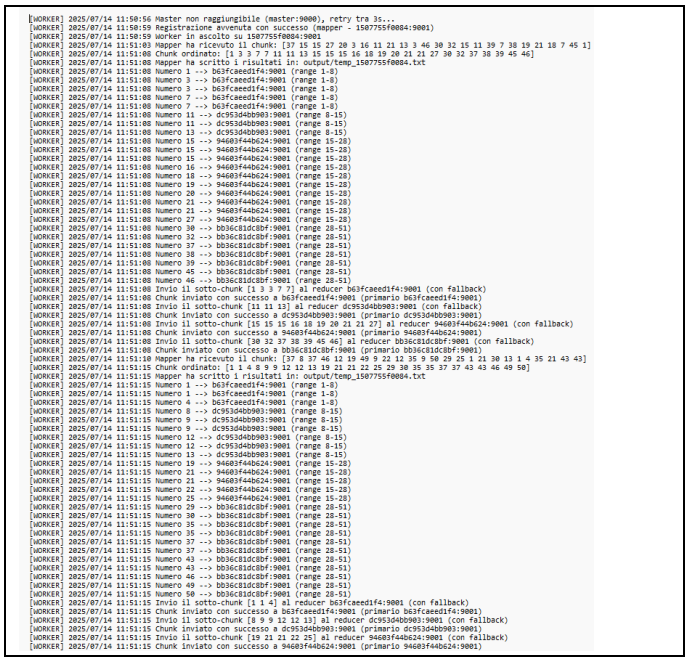
VI. DISCUSSION

Il sistema è stato progettato per gestire crash sia del master che dei worker, garantendo il recupero automatico. In tutti i test, è stato in grado di completare con successo l'elaborazione, di seguito riportiamo dei brevi estratti.

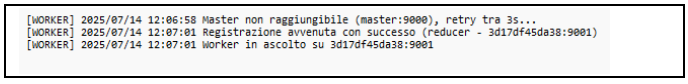
Nel caso di kill di un mapper notiamo dai log dei worker che c'è il mapper interrotto che non esegue la fase di map:

```
[WORKER] 2025/07/14 11:50:56 Master non raggiungibile (master:9000), retry tra 3s...
[WORKER] 2025/07/14 11:50:59 Registrazione avvenuta con successo (mapper - 5ab7d92c19c8:9001)
[WORKER] 2025/07/14 11:50:59 Worker in ascolto su 5ab7d92c19c8:9001
[WORKER] 2025/07/14 11:51:03 Mapper ha ricevuto il chunk: [37 8 37 46 12 19 49 9 22 12 35 9 50 29 25 1 21 30 13 1 4 35 21 43 43]
```

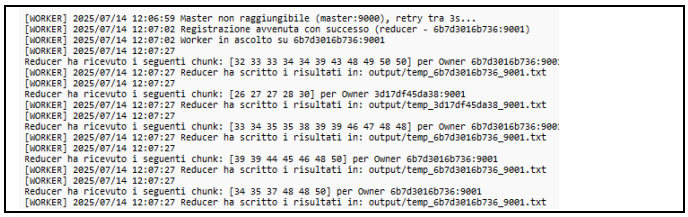
Ma il suo chunk viene correttamente gestito da un altro mapper che aveva terminato la fase map della sua porzione:



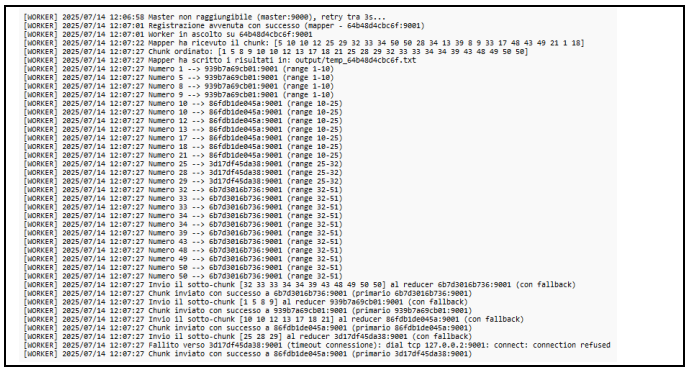
In maniera simile quando interrompiamo un reducer, notiamo dai log che quest'ultimo non esegue la fase di reduce:



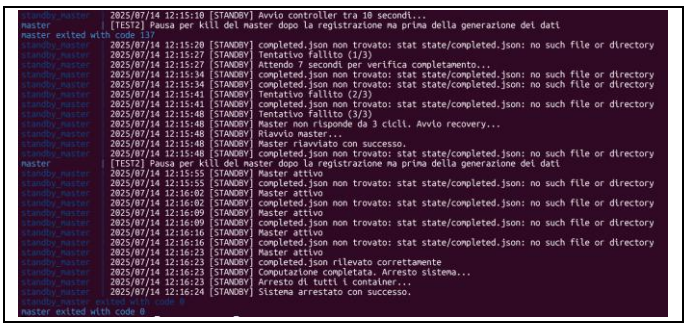
Ma gli altri reducer sono in grado di gestire la sua porzione, come mostrato nel seguente log:



Inoltre, i mapper comprendono automaticamente di dover reindirizzare i dati in caso di tentativi falliti verso il reducer killato:



Infine, mostriamo che, in caso di kill del master, il controller riesce a riavviare il master e l'esecuzione termina correttamente con exit code 0:



VII. CONCLUSIONS

Questo progetto mi ha permesso di sviluppare un sistema di ordinamento distribuito basato sul paradigma MapReduce, utilizzando il linguaggio di programmazione Go, la comunicazione RPC e la containerizzazione Docker. I punti di forza sono: la fault tolerance e il recovery automatico testati in scenari reali di crash di master, mapper e reducer; la scalabilità parametrica del numero di mapper/reducer e il logging dettagliato. Sicuramente alcuni timeout e ritardi potrebbero essere perfezionati, in quanto basati sull'esperienza empirica condotta durante lo sviluppo del sistema. Inoltre, il sistema potrebbe essere esteso con un deploy diverso, in cui ogni worker è collocato su una EC2 distinta. Per finire, il progetto ha rappresentato un'importante occasione per approfondire l'utilizzo degli strumenti di AWS, fondamentali nello sviluppo del software odierno basato sul cloud, affidabilità e scalabilità.

REFERENCES

Link Github:
https://github.com/Denni-02/sdcc-mapreduce.git

[1] https://aws.amazon.com/it/

[2] V. Cardellini. *Slides del corso*, 2024-2025, http://www.ce.uniroma2.it/courses/sdcc2425/

[3] G. Russo Russo. *Hands-on Cloud Computing Services*, 2024-25, http://www.ce.uniroma2.it/courses/sdcc2425/

[4] Docker. *Docker Documentation*. https://docs.docker.com/

[5] Go Team. *The Go Programming Language Documentation*. https://golang.org/doc/

[6] Go Standard Library. *Packages documentation*. https://pkg.go.dev/std