Progetto B3: Distributed PageRank

Alessandro Lioi
Facoltà di Ingegneria Informatica
Università di Roma Tor Vergata
Rome, Italy
alessandro.lioi@students.uniroma2.eu

***Abstract*— Lo scopo di questo documento è quello di descrivere l'architettura, l'implementazione e le scelte progettuali effettuate per implementare l'algoritmo del PageRank distribuito con possibilità di deploy su Docker Compose e AWS EC2**

***Index terms*—PageRank, Docker Compose, AWS**

I. Introduzione

L'obiettivo del progetto è implementare l'algoritmo del PageRank tramite un'applicazione distribuita, seguendo il pattern *Map-Reduce*.

L'algoritmo del PageRank viene usato per assegnare un punteggio (*rank*) alle pagine web simulando la navigazione di un utente che naviga in maniera randomica tra le varie pagine collegate.

È possibile quindi astrarre l'insieme delle pagine web e i loro collegamenti tramite un grafo diretto con:

- nodi: rappresentano le pagine web
- arco: rappresenta un link tra due pagine

Per avere una simulazione più veritiera, si tiene inoltre conto che un utente possa navigare casualmente verso una pagina.

La regola di aggiornamento del rank è la seguente:

$$R_{i+1}(u) = c \sum_{v \in B_u} \frac{R_i(v)}{N_v} + (1-c)E(u) \qquad (1)$$

con:

- $R_i(u)$: rank del nodo u nell'iterazione i
- B_u: nodi entranti verso u
- N_v: numero di nodi uscenti da v
- $E(u)$: probabilità che corrisponde ad un nodo
- c: probabilità di seguire un outlink

Essendo un algoritmo iterativo, si stabilisce la convergenza se la differenza tra i rank dell'iterazione precedente e quella attuale è minore di una threshold.

Il pattern Map-Reduce permette di dividere la computazione in due fase:

- *Map*: si calcola la somma dei rank dei nodi di un sottografo
- *Reduce*: si calcola il nuovo rank, a partire dalle somme calcolate nella fase precedente

II. Architettura e Componenti

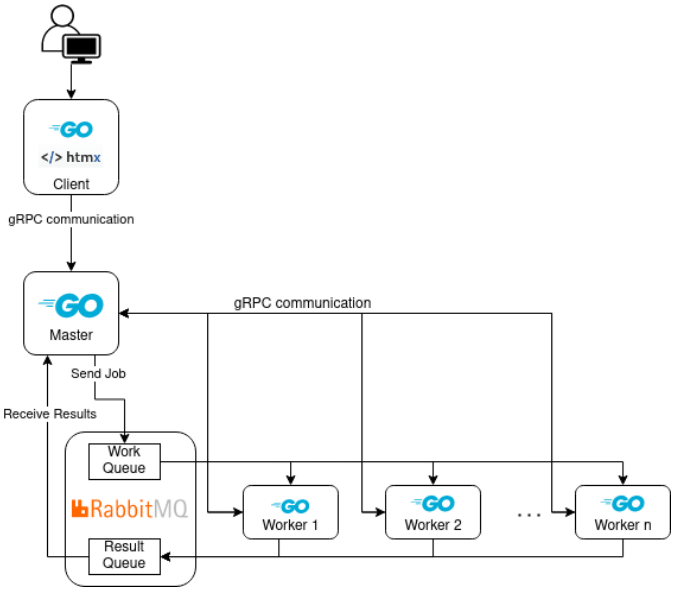

Figure 1: Architecture

A. Client

Il client è formato da un web server, come front-end per l'applicazione e un gRPC server, usato per comunicare con il master eseguiti entrambi all'avvio.

Il front-end è stato realizzato tramite il framework *Echo*, per gestire le richieste, *htmx* per gestire richieste AJAX (al server Echo) e gestire i Server Sent Events e *chota* come framework di CSS.

L'utente, collegandosi al sito, servito dal server Echo e dopo aver fornito i parametri di configurazione dell'algoritmo, può scegliere se caricare un file contenente le informazioni del grafo (formato `FromNode ToNode`) oppure generarne uno casualmente (fornendo il numero di nodi e il massimo numero di archi per nodo)

Una volta inoltrata questa richiesta, si instaura una connessione con il gRPC server per le API del nodo master (`proto/api.proto`) e si invia la richiesta di una nuova computazione.

Il front-end entra quindi in attesa di risposta dal Master con i risultati della computazione.

Una volta ricevuta la risposta dal Master, tramite un Server Sent Event, si aggiorna la pagina mostrando i risultati e il grafo usato.

B. Master

Il master è un nodo che ha il compito di organizzare la computazione tra i vari nodi, usando la coda di messaggi e gestire la comunicazione con il *client*.

Esegue due gRPC server:

- *API*: usato per la comunicazione con il client
- *Node*: usato per la comunicazione interna con i nodi

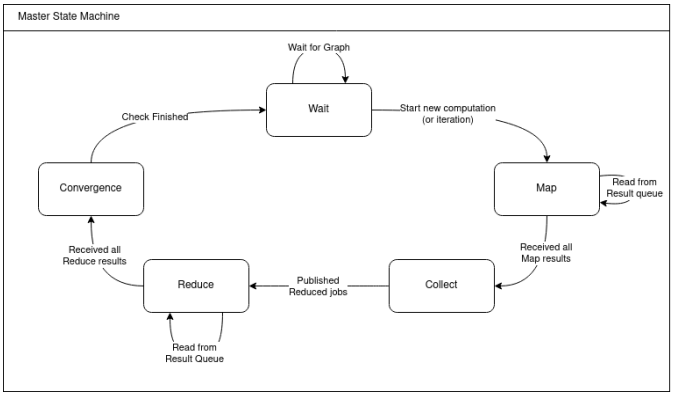

Figure 2: Master State Machine

È implementato tramite una macchina a stati divisa in varie fasi:

- Wait: il master è in attesa di una nuova computazione
  Una volta ricevuto un nuovo grafo, viene diviso in n sottografi con $n = \min(\#\text{ nodes}, \#\text{ workers})$ e, per ogni sottografo, si pubblica un *Map Job* sulla coda di messaggi, nella *Work Queue*. Si invia inoltre un aggiornamento di stato ai vari worker
- Map: si leggono i risultati della fase di Map, pubblicati dai worker nella *Result Queue*
- Collect: si sommano i risultati letti nella fase precedente e si pubblicano *Reduce Job* sulla coda di messaggi (nella *Work Queue*)
- Reduce: come per la fase di Map, si leggono i risultati dei worker dalla *Result Queue*
- Convergence: si aggiorna il valore del rank di ogni nodo e si decide se continuare con l'esecuzione dell'algoritmo per un'altra iterazione, oppure inviare al web-server i rank trovati
  La decisione avviene in base al numero di iterazione (massimo 100) e se si supera o meno una threshold (impostata dal web-server) che controlla il delta tra la somma dei rank dell'iterazione precedente e la somma dei nuovi rank appena calcolati.
  In entrambi i casi si ritorna nello stato di Wait

Nel caso in cui non ci siano worker nella rete, si esegue una versione su singolo nodo del PageRank in cui il master esegue l'intera computazione.

Ciò permette quindi di ridurre l'overhead introdotto dalla comunicazione con la coda di messaggi.

La lettura dalla *Result queue* e l'invio degli aggiornamenti di stato avvengono in due goroutine apposite, per non bloccare il thread principale.

C. Worker

I nodi worker hanno lo scopo di eseguire l'effettiva computazione dell'algoritmo.

Eseguono un gRPC server per comunicare con il nodo master e con gli altri worker se necessario.

Prendono dei job salvati su una coda di messaggi (*Work*) e, in base al contenuto del messaggio, eseguono la fase di *Map* o *Reduce*.

Una volta completata la computazione, salvano il risultato su un'altra coda (*Result*).

Periodicamente, tramite gRPC, inviano un *Health Check* al master, per verificare il corretto funzionamento di questo nodo.

D. Message Queue

Si usa RabbitMQ come broker di messaggi, tramite l'utilizzo di due code:

- *Work*: messaggi scritti dal master e letti dai worker
- *Result*: messaggi scritti dai worker e letti dal master

III. Implementazione e Scelte Progettuali

A. Gestione Grafo

L'applicazione prevede il caricamento di un grafo, effettuata dal master, secondo due modalità:

1. URL che punta ad un file di testo
2. generazione casuale

Per quanto riguarda la prima modalità, ci si aspetta un file di testo formato nel seguente modo: `FromNode ToNode` in cui si descrivono gli archi tra due nodi.

Per la seconda modalità, è possibile definire dei parametri: numero di archi e massimo numero di archi per nodo.

La generazione avviene scegliendo per ogni nodo un nodo randomico al quale deve essere collegato (escludendo connessioni con sè stessi). Per assicurare la connessione nel grafo, si collega ogni nodo con il suo successivo.

La struttura interna con cui è salvato il grafo (specificata in `proto/node.proto`) associa ad ogni nodo (identificato con un valore intero) ad una struttura `GraphNode` che salva il rank attuale, il valore di E e gli archi entranti nel nodo.

Per ogni arco entrante, si salvano le informazioni necessarie (specificata in `proto/common.proto`) per eseguire i vari

job di map e reduce: numero di archi uscenti dal nodo e il rank attuale.

L'utilizzo di questa struttura dati richiede una fase di pre-processamento computazionalmente più intensiva nel caricamento del grafo.

Permette però di inviare ai worker per la fase di *Map* solamente le informazioni necessarie per il calcolo delle somme parziali.

Infatti, per ogni nodo, sono richiesti i nodi entranti e il numero di outlinks, già pre-calcolati.

I valori del vettore di probabilità E sono calcolati nella fase di pre-processamento. Viene assegnato un valore casuale (usando la libreria standard `math/rand` fornita da Go), normalizzato per avere una distribuzione di probabilità.

B. Coda di Messaggi e Job

I messaggi pubblicati sulla coda di messaggi rappresentano due tipi di job: Map e Reduce.

Per quanto riguarda il formato dei messaggi, si utilizzano i Protocol Buffers in modo tale da riutilizzare le stesse strutture dati con cui il grafo viene salvato.

Un job (specificato in `proto/jobs.proto`) contiene le informazioni sul tipo di Job (Map o Reduce) e in base al tipo di messaggio il lavoro che deve svolgere:

- *Map Job*: si lavora su un sottografo
- *Reduce Job*: si lavora sulle somme ottenute aggregando i risultati parziali della fase di Map

C. State Update

Ad ogni iterazione dell'algoritmo, il master invia il proprio stato ai worker.

Lo stato (definito in `proto/node.proto`) è formato da:

- grafo
- parametri `c` e `threshold`
- stringa di connessione con il gRPC server del client
- numero di iterazione
- lista di worker

Ciò permette quindi di avere un recupero delle informazioni e una ripresa dell'esecuzione in caso di crash del master.

Si è scelto di inviare un aggiornamento di stato solo ad ogni iterazione, invece che ad ogni cambiamento di fase, per diminuire il numero di messaggi da dover inviare.

L'invio ad ogni fase inoltre avrebbe introdotto problemi per quanto riguarda la lettura dalla coda. Infatti, nel caso in cui il nodo master andasse in crash nella fase di Map o Reduce, non sarebbe possibile sapere quali e quanti messaggi sono già stati letti dalla *Result queue*.

Ad ogni nuova iterazione, il sistema si trova in uno stato con entrambe le code vuote, i worker in attesa di messaggi e i valori di rank dei nodi del grafo già aggiornati.

D. Worker Join

All'avvio di un nuovo worker, il nodo cerca di contattare il master ad un indirizzo specificato in una variabile d'ambiente.

Se non ottiene risposta, si identifica come nuovo master

Altrimenti, esegue una richiesta di *join*, ricevendo dal master le informazioni per contattare le code, lo stato del master e l'ID univoco per il nuovo nodo.

Il master inoltre invia agli altri nodi della rete un aggiornamento dello stato riguardante la sola lista di worker, in modo tale che ogni worker abbia la lista completa degli altri nodi nel sistema.

Ciò permette di evitare problemi nell'elezione di un nuovo master.

Se il master incorre in un crash nella prima fase di wait (in attesa di un nuovo grafo), nessun worker sarebbe a conoscenza degli altri worker. Ogni worker, a seguito dell'*Health Check*, inizierebbe una nuova elezione e, poichè non ha altri nodi da contattare, diventerebbe il nuovo master. Si ottiene quindi una partizione completa della rete.

Con questo aggiornamento di stato ridotto, ad ogni join, ogni worker ha le informazioni più aggiornate sulla rete.

Anche questo aggiornamento di stato avviene in una goroutine per non bloccare il main thread

E. Health Check e Elezione Master

Periodicamente (con una frequenza basata sul parametro *health_check*), i nodi worker contattano il master per controllare se quest'ultimo è ancora in esecuzione.

Il master risponde con un messaggio vuoto se il worker era presente nella lista dei worker noti, altrimenti invia il proprio stato (ciò avviene nel caso in cui non è stato possibile contattare questo worker nel precedente aggiornamento di stato).

Nel caso di crash del master, i vari worker si rendono conto di questa condizione se non ottengono risposta entro il timeout impostato nella chiamata.

I worker che rilevano il crash del master avviano una nuova elezione, seguendo l 'algoritmo di elezione *Bully*.

Ogni worker invia la propria candidatura come master agli altri nodi worker:

- se riceve un ACK positivo da tutti i worker di cui è a conoscenza, è stato eletto come nuovo master
- se riceve un ACK negativo da almeno un worker, abbandona la propria candidatura e aspetta di ricevere la candidatura del worker con ID minore

Ogni worker tiene traccia dell'ID del master corrente e rifiuta un nuovo worker se riceve una candidatura con ID minore.

IV. Limitazioni

- Si assume che il master non incorra in crash nella fase di Wait prima di essere contattato dal client: non sarebbe possibile avere l'indirizzo IP del nuovo master da contattare
- Non è possibile renderizzare il grafo se il numero di nodi è > 50 (la fase di rendering richiede troppo tempo).

V. Tecnologie e Librerie Utilizzate

Il linguaggio di programmazione utilizzato è **Go** (v1.20.8) sia per lo sviluppo dei nodi (master e worker) che per il web-server

Il frontend è stato sviluppato in HTML e CSS tramite l'utilizzo delle librerie HTMX (libreria UI che estende HTML e permette di eseguire chiamate Ajax) e Chota (framework CSS minimale)

La comunicazione tra i vari nodi e il web-server è stata implementata tramite Protocol Buffers e gRPC.

Il container engine utilizzato è stato Docker, e quindi Docker Compose per eseguire ed orchestrare i vari container.

Per il deploy su AWS è stato utilizzato Terraform (definire l'infrastruttura) e script bash per il deploy dei vari componenti.

Le librerie utilizzate nell'applicazione sono le seguenti:

- `grpc` e `protobuf`: per la comunicazione tra nodi
- `amqp091-go`: per la comunicazione con la coda RabbitMQ
- `echo`: per facilitare la scrittura del web-server
- `godotenv`: per caricare le variabili d'ambiente da un file `.env`
- `go-graphviz`: per renderizzare il grafo, mostrato nel front-end
- `go-nanoid`: ID generator

VI. Deployment

L'applicazione prevede una configurazione tramite variabili di ambiente.

Per facilitare il deploy tramite varie modalità, è possibile configurare il file `config.json`: porte utilizzate, utenza RabbitMQ, frequenza *Health Check*, numero di workers e informazioni per AWS (chiave SSH da usare e tipo di istanza)

A. Locale

Tramite l'esecuzione dello script Python in `deploy/local.py`, vengono stampati a schermo i comandi necessari per eseguire in locale le varie parti dell'applicazione.

Per quanto riguarda RabbitMQ, il deploy avviene tramite Docker, come suggerito dalla documentazione ufficiale e si espongono le porte necessarie per la comunicazione (in questo caso la porta di default è la 5672).

Per il resto dell'applicazione invece si esegue direttamente i binari compilati.

Lo script assegna automaticamente (con valori contigui) le porte per ogni nodo, in modo da evitare conflitti.

B. Docker Compose

Eseguendo lo script `deploy/compose.py`, è possibile generare un file `compose.yaml` per l'esecuzione dell'applicazione tramite Docker Compose, partendo dalla configurazione presente in `config.json`.

L'esecuzione avviene eseguendo il comando `docker compose up --build`.

Vengono esposte all'host le seguenti porte:

- 15672 dal servizio RabbitMQ: UI per la gestione della coda
- 80 dal servizio client: frontend

Poichè Docker Compose crea automaticamente una rete interna, non è necessario esporre altre porte per permettere la comunicazione tra i nodi.

C. AWS

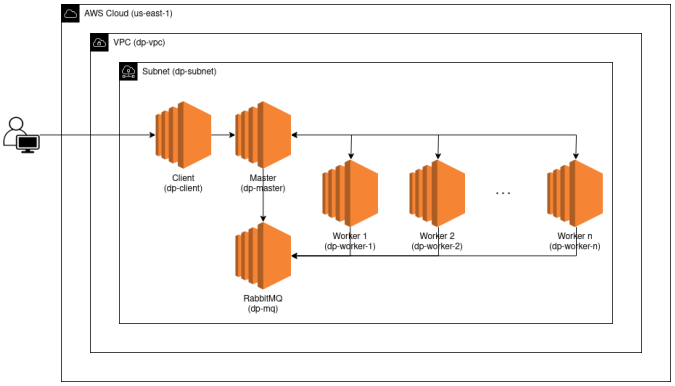

Figure 3: AWS Infrastructure

Tramite l'esecuzione di `deploy/aws.py`, è possibile creare l'infrastruttura AWS e effettuare il deploy dei vari componenti.

Lo script utilizza Terraform per creare l'infrastruttura e degli script bash ad-hoc per il deploy effettivo dell'applicazione.

Non è stato possibile usare Ansible a causa di un problema riscontrato con il deploy di RabbitMQ: nella fase di configurazione delle utenze di RabbitMQ, un comando non termina, portando quindi ad un'attesa infinita.

Gli script copiano i file necessari all'applicazione, effettuano il build e creano il servizio *systemd* necessario per avviare l'applicazione.

A seguito dell'esecuzione dello script, vengono stampati a schermo gli indirizzi IP dai quali è possibile accedere all'applicazione.