Sistemi Distribuiti e Cloud Computing
Algoritmi Di Elezione Distribuita*

*Implementazione in *Golang* degli algoritmi *Bully* e di *Chang & Roberts*

Gabriele Quatrana (0306403)
Università degli studi di Roma "Tor vergata"
Roma, Italia
gabriele.quatrana@alumni.uniroma2.eu

***Abstract*—Questo documento è una relazione relativa al progetto B3 del corso di *Sistemi Distribuiti e Cloud Computing* a.a. 2021/22.**

***Index Terms*—Algoritmo *Bully*, algoritmo di *Chang & Roberts*, *Docker*, *AWS EC2*, *Ansible*.**

I. Introduzione

Lo scopo del progetto è quello di realizzare, nel linguaggio di programmazione **Go** o **Python**, un'applicazione distribuita che implementi i due algoritmi di elezione distribuiti *Bully* e di *Chang & Roberts* visti a lezione.

L'applicazione, inoltre, deve:

- Offrire un servizio di registrazione per i processi che partecipano al gruppo di elezione.
- Offrire un servizio di monitoraggio (attraverso *heartbeat*) che permetta di identificare il crash di un processo.
- Supportare un flag di tipo *verbose* che permette di stampare le informazioni di debug con i dettagli dei messaggi inviati e ricevuti.
- Supportare un parametro *delay* che permette di specificare un ritardo, generato in modo casuale, per trasferire i messaggi.

L'applicazione deve prevedere anche l'esecuzione di tre casi di test:

a) Un solo processo subisce il crash e non è il leader.
b) Il processo leader subisce il crash.
c) Almeno due processi, uno dei quali è il leader, subiscono contemporaneamente il crash.

II. Architettura e Scelte Progettuali

In questa sezione viene descritta l'architettura dell'applicazione sviluppata e le motivazioni riguardo le principali scelte progettuali prese durante lo sviluppo.

L'applicazione è stata realizzata attraverso il linguaggio di programmazione **Go**. Questo linguaggio è stato scelto al posto di **Python** perché fornisce il package *net/rpc* (per realizzare un tipo di comunicazione basata su RPC) e le *goroutine* (che permettono la creazione di thread) che sono stati introdotti durante il corso.

Per avviare l'applicazione è stato ideato un piccolo script **Go** che permette di gestire i diversi parametri di esecuzione. Questi vengono inseriti in un file *.env* che può essere utilizzato, oltre che dall'applicazione, anche da **Docker** (attraverso il file *docker-compose.yml*) per stabilire il numero di nodi da inizializzare nella fase di avvio dei container.

I parametri che vengono inseriti nel file *.env* e necessari per eseguire l'applicazione sono i seguenti:

- Il numero dei nodi che compongono la rete.
- Il ritardo massimo di inoltro dei messaggi.
- La durata di un turno del servizio di *heartbeat*.
- L'algoritmo da utilizzare durante l'esecuzione del programma.
- I flag per abilitare la visualizzazione delle informazioni di debug dei nodi.
- Il flag per stabilire il tipo di test da eseguire.
- I nodi che dovranno subire il crash nel caso di esecuzione di uno dei test.

Per realizzare gli algoritmi di elezione è necessario che i nodi della rete scambino tra loro dei messaggi. Nell'applicazione sono disponibili quattro tipi di messaggi:

- *ELECTION*: per inviare un messaggio di elezione.
- *OK*: utilizzato solo in caso di algoritmo *Bully* per rispondere a un messaggio di elezione.
- *COORDINATOR*: per comunicare l'ID del leader dopo il termine di un'elezione.
- *HEARTBEAT*: per realizzare il servizio di monitoraggio dello stato dei nodi.

```
const (
  ELECTION = iota
  OK
  COORDINATOR
  HEARTBEAT
)
```

Ogni nodo ha associati ad esso un ID, un indirizzo IP e una porta sulla quale si mette in ascolto per ricevere messaggi dagli altri nodi della rete. Si è deciso di utilizzare il meccanismo RPC per scambiare i messaggi nella rete perché è uno degli argomenti principali affrontati nel corso: ogni nodo esporta il metodo *SendMessage* che permette di inviare e ricevere in risposta un messaggio che contiene uno o più ID e il tipo di messaggio inviato/ricevuto.

Quando tutti i nodi si sono registrati nella rete, solo il nodo con ID minore indice una nuova elezione per trovare il leader

iniziale. Quando si conclude un'elezione, tutti i nodi della rete ricevono un messaggio *COORDINATOR* che contiene l'ID del nuovo leader.

Per implementare i test è stato necessario fare in modo che i nodi scelti, ad un certo punto, terminino la loro esecuzione. I nodi che subiranno il crash durante l'esecuzione vengono scelti casualmente e vengono arrestati non appena viene completata la prima elezione indetta nella rete in modo tale che, questi nodi, vengano terminati contemporaneamente.

Inoltre, nell'applicazione sono stati implementati i seguenti due servizi.

A. Servizio di Registrazione

Il servizio di registrazione permette ai nodi di registrarsi nella rete e di conoscere gli altri nodi che ne fanno parte. Il servizio assegna gli ID ai vari nodi in ordine crescente in base all'ordine di registrazione a partire da 0.

Il processo che gestisce il servizio di registrazione si mette in ascolto sulla porta *1234* (definita nel file *config.json*) ed esporta il metodo *RegisterPeer* tramite RPC. All'avvio, ogni nodo esegue una chiamata a questo metodo per registrarsi ed ottenere la lista dei nodi che fanno parte della rete. Per ogni nodo viene specificato ID, IP e porta.

B. Servizio di Heartbeat

Il servizio di monitoraggio attraverso *heartbeat* permette di rilevare la presenza di eventuali crash da parte dei nodi della rete.

Quando il servizio viene eseguito da un nodo, questo invia a ogni altro nodo della rete un messaggio *HEARTBEAT*. Se durante la comunicazione si verifica un errore (non riesce a collegarsi al nodo di destinazione o non riceve una risposta), viene rilevato il crash del nodo destinatario. Nel caso in cui questo nodo è il leader, viene inizializzata una nuova elezione da parte del nodo che sta eseguendo il servizio di *heartbeat*.

Il servizio viene eseguito, tramite una *goroutine*, periodicamente da ogni nodo a turno (la cui durata è definita da uno dei parametri di esecuzione dell'applicazione) in base al proprio ID.

I turni per il servizio di heartbeat sono stati introdotti durante lo sviluppo dell'applicazione per ridurre il numero di messaggi che vengono inviati tra i nodi contemporaneamente (facilitando anche la lettura delle informazioni di debug fornite dai nodi).

III. Implementazione degli Algoritmi

In questa sezione viene descritto come sono stai implementati, nell'applicazione realizzata, gli algoritmi di elezione distribuiti esaminati durante il corso.

A. Algoritmo Bully

In questa implementazione dell'algoritmo *Bully* il nodo che inizializza l'elezione invia sequenzialmente a tutti i nodi, che hanno ID maggiore del suo, un messaggio *ELECTION*:

- Se uno dei nodi destinatari riceve il messaggio, invia come risposta un messaggio *OK*.
- Se non conosce nodi con ID maggiore del suo, o non riceve il messaggio *OK* da questi nodi, si autoproclama leader.
- Se il nodo mittente riceve come risposta *OK*, esce dall'elezione.

Una volta che un nodo esce dall'elezione, non continua ad inviare altri messaggi *ELECTION*; in questo modo viene ridotto il numero di messaggi che vengono scambiati tra i nodi della rete.

B. Algoritmo di Chang & Roberts

L'algoritmo di *Chang & Roberts* si applica ad una rete con topologia ad anello in cui i nodi vengono ordinati in base ai loro ID. Ogni nodo, infatti, è collegato al primo con ID maggiore del suo. Il nodo con l'ID maggiore nella rete è collegato a quello con l'ID minore.

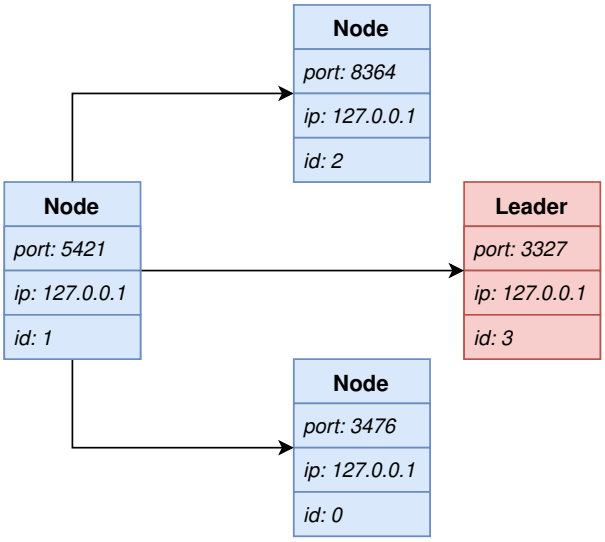

Fig. 1. Servizio di heartbeat eseguito dal nodo con ID 1.

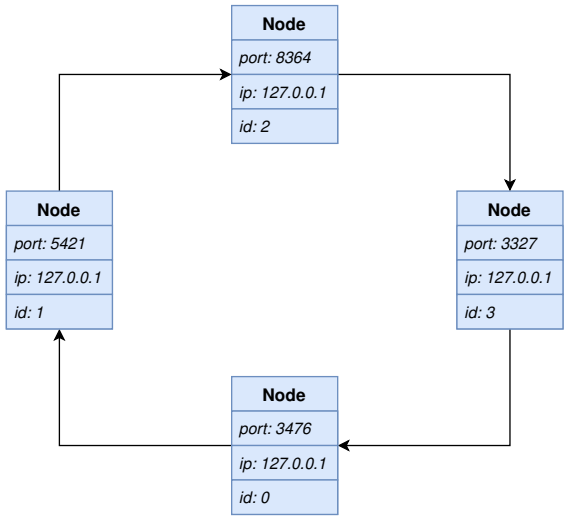

Fig. 2. Esempio di rete con topologia ad anello.

Quando un nodo indice una nuova elezione, invia un messaggio *ELECTION* al nodo successivo nell'anello allegando il suo ID. Ogni qualvolta un nodo riceve un messaggio *ELECTION*, inserisce nella lista di ID, presente nei messaggi di elezione, il proprio e lo invia al nodo successivo nell'anello.

Quando il nodo che ha indetto l'elezione riceve il messaggio *ELECTION* che aveva generato, sceglie come leader il nodo con ID maggiore presente nella lista del messaggio e inoltra a tutti i nodi attivi della rete un messaggio *COORDINATOR* che contiene l'ID del leader.

IV. Limitazione Riscontrate

In questa sezione vengono presentate le limitazioni riscontrate durante la fase di testing e debug dell'applicazione sviluppata. La limitazione principale è relativa al servizio di *heartbeat*. Infatti, quando un nodo subisce un crash questo non fa più parte effettivamente della rete ma continua ad avere il suo turno nel servizio di *heartbeat*. Questo significa che un nodo che è ancora attivo deve attendere anche il turno dei nodi guasti.

Un'altra limitazione è data dalla natura seriale dell'applicazione. Infatti, sarebbe possibile introdurre diverse *goroutine* che permettano di rendere parallela l'applicazione per quanto riguarda l'invio dei messaggi e loro gestione quando vengono ricevuti. Questo tipo di lavoro non è stato effettuato in quanto non è strettamente collegato con gli argomenti affrontati nel corso.

V. Distribuzione

Ogni nodo della rete viene eseguito in un container **Docker** e attraverso **Docker Compose** viene automatizzato il processo di creazione dei container e di una rete dove questi possono comunicare tra loro.

È possibile eseguire l'applicazione su un istanza di **AWS EC2**. Per automatizzare la procedura di deploy dell'applicazione è stato utilizzato **Ansible** che permette (tramite il playbook *deploy_sdcc.yaml*) di installare **Go** e **Docker** e di trasferire il codice sull'istanza **EC2** che viene utilizzata.

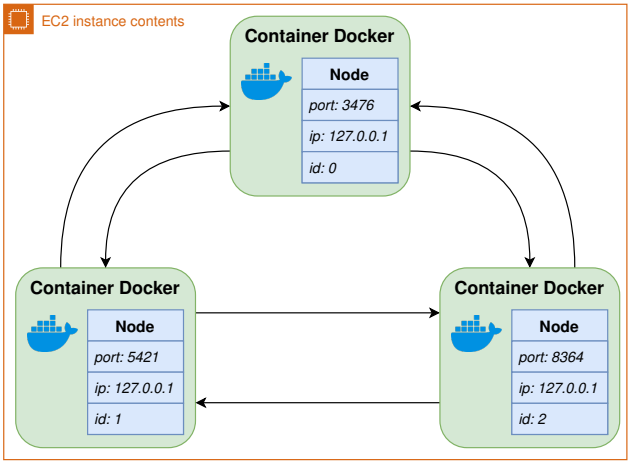

Fig. 3. Deploy con un'istanza **AWS EC2** e container **Docker**.

VI. Piattaforma e Librerie per lo Sviluppo

In questa sezione vengono indicati e descritti i software e le librerie usati per lo sviluppo dell'applicazione. Sono stati utilizzati:

- **Golang**: come linguaggio di programmazione.
- **Goland**: come IDE per la scrittura del codice.
- **Docker Desktop**: per installare **Docker** e **Docker Compose** in *Windows*.

Oltre a quelle di default di **Go**, sono state utilizzate le seguenti librerie:

- *godotenv*: permette di definire e accedere facilmente alle variabili di ambiente contenute nel file *.env*
- *freeport*: permette di trovare una porta TCP aperta e non utilizzata.