Distributed Leader Election

Andrea Andreoli
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Roma, Italia
aandreo.2001@gmail.com

***Abstract*—Nei sistemi distribuiti viene spesso fatto riferimento ad un nodo "speciale", chiamato *coordinatore* o *leader*, che ricopre un ruolo particolare all'interno del sistema. Se il coordinatore si guasta, l'intero sistema si blocca. Per risolvere tale problema, occorre eseguire un algoritmo di elezione, o di *agreement*, in cui i processi ancora attivi raggiungono un accordo per eleggere un nuovo coordinatore.**

***Index Terms*—Container, elezione, leader, docker, discovery, crash, test, lista**

I. Introduzione

Un sistema distribuito è un insieme di nodi di rete indipendenti che non condividono memoria. Ciascun processore di ogni nodo ha una sua memoria e comunicano tra loro tramite la rete. La comunicazione in rete è implementata in un processo su una macchina che comunica con un processo su un'altra macchina.
Molti algoritmi utilizzati nei sistemi distribuiti necessitano la presenza di un *coordinatore* o *leader* che si occupa di svolgere le funzioni necessarie agli altri processi nel sistema. Per la scelta di un nuovo coordinatore sono stati appositamente ideati degli *algoritmi di elezione*.
Gli algoritmi di elezione scelgono un processo da un gruppo di processi per agire come coordinatore. Se il leader si interrompe per qualche motivo, ne viene eletto uno nuovo. L'elezione richiede ovviamente che venga raggiunto un **consenso distribuito** tra i nodi del sistema.
In questo report verrà affrontata la trattazione di due algoritmi di elezione distribuita:

- Algoritmo Bully
- Algoritmo di Chang-Roberts

II. Scelte progettuali

Per le implementazioni dei due algoritmi, sono state fatte delle assunzioni riguardanti il modello del sistema. In particolare:

- I processi nel sistema possono interrompersi.
- La comunicazione è affidabile.
- Ogni processo può mantenere attiva una sola elezione alla volta.
- Ogni processo possiede un ID univoco, pertanto il processo in funzione con l'ID più alto viene eletto come leader.

Inoltre, è stato utilizzato il linguaggio di programmazione Go per la realizzazione pratica dei due algoritmi menzionati. Di conseguenza, nel documento saranno inclusi esempi di codice in tale linguaggio.

A. Service Discovery

È stato realizzato un meccanismo di service discovery per consentire ai processi nel sistema di essere consapevoli degli altri processi nel sistema distribuito. In particolare, un nuovo processo che entra nel sistema effettuerà una registrazione al Service Registry (disponibile ad un indirizzo noto) per ottenere le informazioni fondamentali, quali:

- ID (assegnato in modo casuale con un numero da 0 a 20)
- Lista dei processi nel sistema

Il service registry comunicherà quindi in modo broadcast a tutti i processi già presenti nel sistema che è entrato un nuovo processo, così che possano aggiornare le rispettive liste di processi conosciuti.
Quest'operazione viene eseguita in un servizio implementato dal service registry attraverso diversi passi: inizialmente viene generato l'ID per il nuovo processo nel sistema

```
redo:
    newID := utils.Random(0, 20)
    for _, node := range nodes {
        /* Check if ID was already assigned */
        if newID == node.Id {
            goto redo
        }
    }
```

in seguito vengono mandate le informazioni utili al processo richiedente nella variabile di risposta

```
    nodeInfo.Id = newID
    nodeInfo.List.Nodes = nodeList.GetAllNodes
        ()
    nodeInfo.Leader = newNode.Leader
```

dove $newNode.Leader$ per default vale -1.
Infine viene effettuato l'aggiornamento delle liste degli altri processi nel sistema

```
    for _, node := range nodes {
        peer, err := rpc.Dial("tcp", node.
            Address)
        if err != nil {
            continue
        }

        err = peer.Call("PeerServiceHandler.
            UpdateList", newNode, nil)
        if err != nil {
```

```
            log.Fatal("List update failed: ",
                err)
        }

        err = peer.Close()
        if err != nil {
            log.Fatal("Closing connection
                error: ", err)
        }
    }
```

Una limitazione nel progetto è rappresentata dalla mancata replicazione del Service Registry.

B. Interruzione di un processo

Come già precedentemente specificato, si è assunto che i processi nel sistema possano interrompersi per un qualsiasi motivo in modo del tutto casuale. Per simulare tale assunzione è stata implementata una funzione chiamata *StopNode(·)*, che in modo del tutto casuale determinerà l'arresto dell'esecuzione di un processo.

```
func StopNode(currentNode nodeINFO) {
    /* Random range of numbers */
    minNum := 0
    maxNum := 10000000

    for {
        randNum := Random(minNum, maxNum)
        if currentNode.Id == randNum {
            os.Exit(1)
        }
    }
}
```

Dove la funzione *Random(minNum, maxNum)* è stata appositamente realizzata per estrarre un numero casuale all'interno di un dato intervallo numerico.
E' previsto che un processo interrotto possa riprendere di nuovo l'esecuzione con l'ID che possedeva prima del guasto.

C. Comunicazione tra processi

Poiché la comunicazione tra processi è affidabile, è stato scelto il protocollo TCP come protocollo di comunicazione.

D. File di configurazione

Per evitare codice *hard-coded* si è deciso di introdurre un file di configurazione con formato JSON in cui sono state specificate le seguenti informazioni:

- Hostname e porta del service registry
- Hostname e porta dei processi
- Algoritmo di cui si vuole effettuare il testing

E. Strutture dati

Sono state implementate varie strutture dati per poter realizzare lo scambio di dati durante le chiamate a procedura remota (in particolare si è utilizzato il meccanismo di RPC in Go). Tra le più rilevanti:

- Struttura per identificare il processo nel sistema

```
type Node struct {
    Id      int
    Address string
    Leader  int
}
```

- Struttura per realizzare la lista di processi all'interno del sistema

```
type NodeList struct {
    Nodes []Node
}
```

- Struttura che permette al processo di mantenere le informazioni correnti

```
type NodeINFO struct {
    Id      int
    Address string
    List    NodeList
    Leader  int
}
```

- Struttura per comporre il messaggio che viene scambiato durante l'esecuzione dell'algoritmo di Chang-Roberts:

```
type Message struct {
    SkipCount int
    MexID     int
    CurrNode  NodeINFO
}
```

III. Algoritmo Bully

Questo algoritmo è stato proposto da Garcia-Molina, con un idea principale: quando un processo nota che il coordinatore non risponde più alle richieste, allora comincia una nuova elezione.
Un processo p_i si accorge che il processo leader non risponde e comincia un'elezione, quindi invia un **messaggio di elezione** a tutti i processi con un ID maggiore rispetto al suo $(p_{i+1}, p_{i+2}, \ldots, p_N)$. A questo punto possono verificarsi vari scenari:

- Se nessuno risponde, p_i vince l'elezione e diventa il coordinatore. Annuncia la vittoria inviando un messaggio a tutti i processi
- Se p_k (con $k > i$) riceve il messaggio di elezione da p_i, allora risponde con un **messaggio OK**, e comincerà una nuova elezione

Quando p_i riceve un messaggio OK terminerà l'elezione. Nel caso in cui un processo nuovo o un processo che si riprende da un guasto entra nel sistema, comincerà una nuova elezione in quanto non conosce il leader.

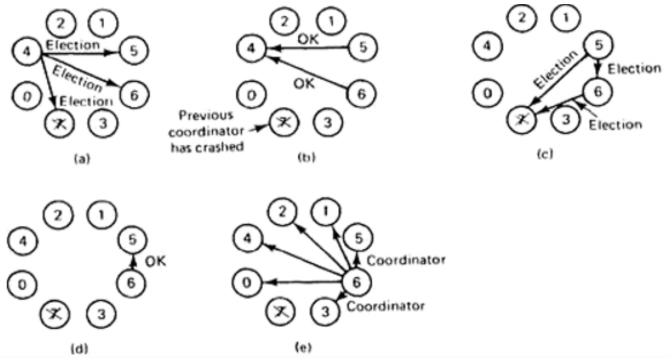

Fig. 1. Esempio di esecuzione dell'algoritmo

A. *Implementazione dell'algoritmo*

L'algoritmo è stato strutturato in due sezioni:

- Una funzione main chiamata *Bully(·)*
- Una funzione per gestire l'elezione chiamata *ElectionBully(·)*

Nella funzione *Bully(·)* verranno effettuati dei ping al processo leader per verificarne l'attività: in caso di inattività allora comincerà una nuova elezione. Il ping viene effettuato nel seguente modo:

```
peer, err := utils.DialTimeout("tcp",
    currNode.List.GetNode(currNode.Leader)
    .Address, 5*time.Second)
```

La funzione *DialTimeout(·)*, che si trova nel pacchetto *utils*, è una variante della classica *rpc.Dial(·)* realizzata per consentire la scelta di un timeout personalizzato.
La variabile *err* non sarà *nil* nel caso in cui non sia possibile stabilire una connessione con il processo leader entro un intervallo di tempo Δt. Al superamento di tale intervallo, il processo verrà considerato inattivo e, di conseguenza, verrà avviata una nuova elezione.
Nella funzione *ElectionBully(·)*, viene gestita l'elezione che è avviata dal processo p_i che ha rilevato l'inattività del leader. La funzione è divisa in due cicli *for*:

```
for _, node := range currNode.List.
    GetAllNodes() {
    if node.Id > currNode.Id {
        peer, err := utils.DialTimeout("
            tcp", node.Address, 5*time.
            Second)
        if err != nil {
            continue
        }

        var repOK string
        err = peer.Call("
            PeerServiceHandler.
            ElectionMessageBULLY",
            currNode, &repOK)
        if err != nil {
            log.Fatal("Election message
                forward failed: ", err)
        }

        err = peer.Close()
        if err != nil {
            log.Fatal("Closing connection
                error: ", err)
        }

        /* Test if the successor node
            reply */
        if repOK != "" {
            return
        }
    }
}

for _, node := range currNode.List.
    GetAllNodes() {
    peer, err := utils.DialTimeout("tcp",
        node.Address, 5*time.Second)
    if err != nil {
        continue
    }

    err = peer.Call("PeerServiceHandler.
        NewLeaderBULLY", currNode.List.
        GetNode(currNode.Id), nil)
    if err != nil {
        log.Fatal("Leader update error: ",
            err)
    }

    err = peer.Close()
    if err != nil {
        log.Fatal("Closing connection
            error: ", err)
    }
}
```

- Nel primo ciclo vengono inviati messaggi a tutti i processi con ID $k > i$, e successivamente viene verificato se rispondono con un messaggio OK. Al primo messaggio OK ricevuto, il processo che ha avviato l'elezione esegue un *return*, evitando così di procedere al secondo blocco in cui viene aggiornato l'ID del precedente leader con il vincitore dell'elezione
- Nel secondo ciclo vengono inviati messaggi a tutti i processi nel sistema per aggiornare l'informazione sul processo leader. Questo ciclo *for* viene eseguito solo dal nodo che diventa il nuovo leader, e sarà quindi lui a comunicare l'esito agli altri processi

Il processo nel sistema utilizza due servizi:

- *NewLeaderBULLY(·)*: in questo servizio viene aggiornato l'ID relativo al leader
- *ElectionMessageBULLY(·)*: questo servizio gestisce la risposta con un messaggio OK, e il processo in questione avvierà una nuova elezione

IV. Algortimo di Chang-Roberts

Supponendo che la rete sia organizzata in un anello unidirezionale con un canale di comunicazione che va da ogni processo al vicino in senso orario, l'idea dietro questo algoritmo è quella di far scambiare in modo circolare tra i processi un messaggio contenente un ID, che sia *max(ID proprio, ID ricevuto)*. Se un processo riceve un messaggio contenente

il proprio ID allora sarà lui il nuovo leader e l'elezione è conclusa.
Di seguito un esempio ipotizzando che il processo con ID 6 cominci una nuova elezione:

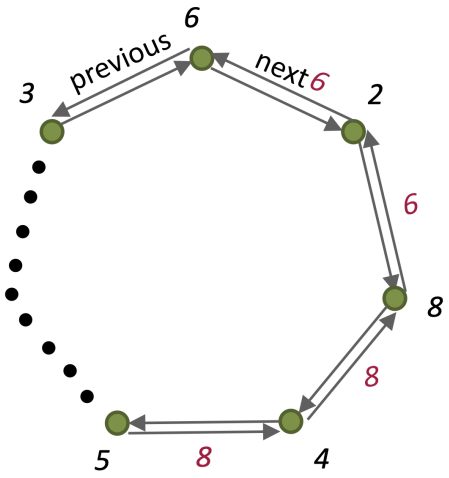

Fig. 2. Esempio di esecuzione dell'algoritmo

Quindi, il nodo 6 inoltra il suo ID a *6.next*, che è il nodo 2. A questo punto, il nodo 2 prende il massimo tra il suo ID (2) e quello ricevuto (6) e lo inoltra a *2.next*, che è il nodo 8. Successivamente, il nodo 8 prende il massimo tra il suo ID (8) e quello ricevuto (6) e lo inoltra a *8.next*, e così via. Il valore 8 continuerà ad essere inoltrato lungo tutto l'anello (supponendo che l'ID 8 sia il più grande all'interno del sistema) fino a quando non verrà inoltrato al processo 8. A quel punto, il processo 8 saprà di essere il processo con l'ID più alto e si dichiarerà come leader, inviando un messaggio all'interno della rete per comunicare di essere il nuovo leader. Se il processo p riceve il messaggio d'elezione m con $m.ID = p.ID$, allora si dichiarerà come leader ed imposterà $p.leader = p.ID$. Successivamente, invierà un messaggio con $p.ID$ al $p.next$. Ogni altro processo q che riceve questo messaggio imposterà $q.leader = p.ID$ e inoltrerà il messaggio a $q.next$.

A. *Implementazione dell'algoritmo*

L'algoritmo è stato strutturato in tre sezioni:

- Una funzione main chiamata *ChangAndRoberts(·)*
- Una funzione per gestire l'inoltro del messaggio d'elezione al processo $p.next$, chiamata *ElectionChangAndRoberts(·)*
- Una funzione per gestire l'inoltro del messaggio contenente l'ID del nuovo leader al processo $p.next$, chiamata *WinnerMessage(·)*

Nella funzione *ChangAndRoberts(·)* verranno effettuati dei ping al processo leader per verificarne l'attività: in caso di inattività, allora sarà avviata una nuova elezione. Il ping viene effettuato come precedentemente descritto nell'algoritmo Bully. Nella funzione *ElectionChangAndRoberts(·)*, viene gestita l'elezione che viene avviata dal processo p_i che ha rilevato l'inattività del leader. Inoltre, viene anche gestito il guasto di uno o più processi calcolando il "passo" da effettuare nell'inoltro del messaggio, evitando i processi non attivi. Il "passo" effettuato sarà comunicato a scopo informativo al processo scelto (che sarà il primo utile attivo).

```
skip := startIndex
i := 0
for {
    i++
    pass := (startIndex + i) % len(
        currentNode.List.Nodes)
    if pass == skip-1 {
        info.MexID = currentNode.Id
        peer, err = utils.DialTimeout("tcp
            ", currentNode.List.GetNode(
            currentNode.Id).Address, 5*
            time.Second)

        err = peer.Call("
            PeerServiceHandler.NewLeaderCR
            ", info, nil)
        if err != nil {
            log.Printf("Leader update
                error: %v", err)
        }

        err = peer.Close()
        if err != nil {
            log.Fatal("Closing connection
                error: ", err)
        }

        return
    }

    peer, err = utils.DialTimeout("tcp",
        currentNode.List.GetNode(
        currentNode.List.Nodes[pass].Id).
        Address, 5*time.Second)
    info.SkipCount = i
    if err != nil {
        continue
    } else {
        break
    }
}
```

Nella funzione *WinnerMessage(.)*, viene gestito l'inoltro del messaggio che comunica l'ID del nuovo leader, come già precedentemente anticipato.
Il processo nel sistema utilizza i seguenti servizi:

- *NewLeaderCR(·)*: il servizio si occupa di gestire l'aggiornamento dell'ID del leader conosciuto dal processo, quindi effettua la seguente operazione:

```
if mex.MexID != currentNode.Id {
    currentNode.Leader = mex.MexID
} else {
    return nil
}
```

  Successivamente, inoltra il messaggio al processo successivo
- *ElectionMessageCR(·)*: il servizio gestisce il confronto tra l'ID del processo e quello ricevuto nel messag-

gio di elezione ed inoltra il messaggio con il risultato del confronto. Nel caso in cui dal confronto si abbia un'uguaglianza, il processo sarà il nuovo leader e imposterà il suo leader con il suo ID. Inoltre, invierà un messaggio contenente il suo ID che dovrà circolare nell'anello per comunicare la presenza di un nuovo leader.

V. Docker

La distribuzione del codice è stata simulata utilizzando Docker, un software libero progettato per eseguire processi informatici in ambienti isolabili, minimali e facilmente distribuibili. In particolare, sono stati realizzati due Dockerfile:

- Un Dockerfile per i processi nel sistema
- Un Dockerfile per il service registry

Inoltre, è stato realizzato un file *compose.yaml* per comporre i container Docker. I test sono stati condotti utilizzando 6 repliche dei processi all'interno del sistema.

References

[1] H. Sun, "Leader election & Failure detection", School of Informatics, University of Edinburgh.

[2] "Unit 3 - Election Algorithm", Department of Information Technology, KDK College of Engineering, Nagpur.

[3] Valeria Cardellini, "Mutua Esclusione ed Elezione nei Sistemi Distribuiti", University of Rome Tor Vergata.