Implementazione di due algoritmi di elezione distribuita

Distributed Leader Election

ANDREA ANDREOLI 0350012

Agenda

Introduzione al concetto di leader in un sistema distribuito

Scelte progettuali

Presentazione degli algoritmi e della loro implementazione in Go

Accenni sulla progettazione effettuata in Docker

Introduzione al concetto di leader in un sistema distribuito

Ruolo del leader in un sistema distribuito

All’interno di un sistema distribuito molti algoritmi necessitano la presenza di un processo “speciale” che ha la responsabilità di svolgere funzioni utili agli altri processi nel sistema

Questo processo definito come “speciale” prende il nome di **leader** o **coordinatore**

Come viene scelto il processo leader?

Come detto in precedenza, il ruolo del processo leader in un sistema distribuito è di fondamentale importanza

Ma se questo processo per un qualsiasi motivo si guastasse, cosa succederebbe?

In caso di guasto del leader, non appena un qualsiasi altro processo nel sistema rileva la sua inattività, inizierà una nuova elezione. Da ciò derivano i cosiddetti **algoritmi di elezione**, utilizzati per scegliere un nuovo processo leader

In seguito ne tratteremo due:

- Algoritmo Bully
- Algoritmo di Chang-Roberts

Scelte progettuali

Assunzioni riguardanti il modello del sistema

Per lo sviluppo degli algoritmi sono state effettuate delle assunzioni circa il modello del sistema in esame:

- I processi nel sistema possono interrompersi
- La comunicazione è affidabile
- Ogni processo mantiene attiva una sola elezione per volta
- Ogni processo ha ID univoco, pertanto il processo con ID più alto sarà eletto come leader

Interruzione di un processo

Per simulare l’interruzione di un processo e dare via ad una nuova elezione si è implementata una **funzione** `StopNode(currentNode nodeINFO)`

L’interruzione avviene in modo del tutto casuale

```
48  func StopNode(currentNode NodeINFO) {  1 usage
49      minNum := 0
50      maxNum := 10000000
51
52      for {
53          randNum := Random(minNum, maxNum)
54          if currentNode.Id == randNum {
55              os.Exit( code: 1)
56          }
57      }
58  }
```

Comunicazione tra processi

Essendo la comunicazione tra processi affidabile per assunzione, si è optato per l'utilizzo del protocollo TCP

In ogni chiamata di funzione Dial sarà quindi specificato **tcp** come protocollo da utilizzare

Service Discovery

E’ stato implementato un meccanismo di service discovery per consentire ai processi nel sistema di essere consapevoli della presenza degli altri nel sistema

Un nuovo processo effettua una richiesta al Service Registry per ottenere:

- ID con cui p_i sarà riferito nel sistema
- Lista dei processi nel sistema

Algoritmo 1
Bully

Introduzione all'algoritmo Bully

Il processo p_i nota che il leader è inattivo e comincia una nuova elezione inviando un **messaggio di elezione** a tutti i processi con ID maggiore del suo, quindi $p_{i+1}, p_{i+2}, \ldots, p_N$

Possono quindi verificarsi due principali situazioni

Se nessuno risponde, p_i vince l'elezione ed annuncia la vittoria inviando un messaggio agli altri processi

Se p_k $(k > i)$ riceve un messaggio d'elezione, risponde con un messaggio OK e comincia una nuova elezione

Implementazione dell’algoritmo

L’algoritmo è stato realizzato in un package dedicato (*algorithm*) all’interno di un file chiamato *bullyAlgorithm.go*

Il processo effettua dei ping verso il leader (per verificarne lo stato) tramite una funzione chiamata *Bully*

L’elezione viene gestita tramite la funzione *ElectionBully*

```
12  func ElectionBully(currNode utils.NodeINFO) {  3 usages  Andrea041
13      for _, node := range currNode.List.GetAllNodes() {
14          if node.Id > currNode.Id {
15              peer, err := utils.DialTimeout( network: "tcp", node.Address, 5*time.Second)
16              if err != nil { continue }
19
20              var repOK string
21              err = peer.Call( serviceMethod: "PeerServiceHandler.ElectionMessageBULLY", currNode, &repOK)
22              if err != nil { log.Fatal( v...: "Election message forward failed: ", err) }
25
26              err = peer.Close()
27              if err != nil { log.Fatal( v...: "Closing connection error: ", err) }
30
31              /* Test if the successor node reply */
32              if repOK != "" { return }
35          }
36      }
37
38      for _, node := range currNode.List.GetAllNodes() {
39          peer, err := utils.DialTimeout( network: "tcp", node.Address, 5*time.Second)
40          if err != nil { continue }
43
44          err = peer.Call( serviceMethod: "PeerServiceHandler.NewLeaderBULLY", currNode.List.GetNode(currNo
45          if err != nil { log.Fatal( v...: "Leader update error: ", err) }
48
49          err = peer.Close()
50          if err != nil { log.Fatal( v...: "Closing connection error: ", err) }
53      }
54  }
```

Algoritmo 2
Chang-Roberts

Introduzione all'algoritmo di Chang-Roberts

Supponiamo che la rete sia organizzata in un anello unidirezionale con un canale di comunicazione che va da un processo al vicino in senso orario

L'idea è di far circolare tra i processi un messaggio contenente un ID, che sia *max(ID proprio, ID ricevuto)*

Ogni processo sarà inizialmente marcato come *non partecipante*, ed ogni volta che inoltra o invia un messaggio si marcherà come *partecipante*

Quando un processo riceve un messaggio contente il proprio ID allora sarà lui il nuovo leader e l'elezione si conclude

Il nuovo leader metterà in circolazione un messaggio contenente il proprio ID ed ogni processo si marcherà di nuovo come *non partecipante*

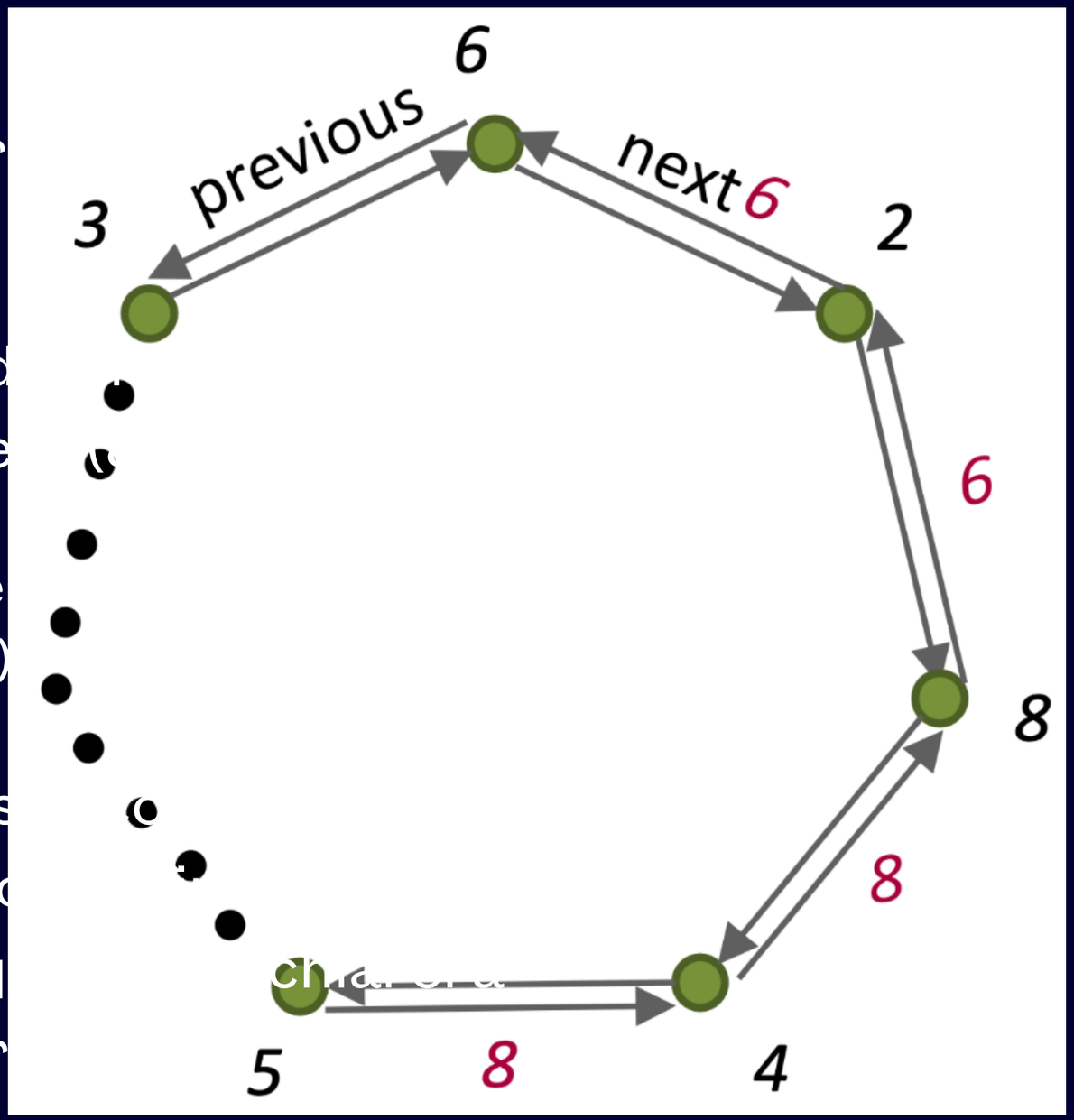
Applicazione dell'algoritmo

Il processo 6 inoltr
(2)

Il processo 2 prend
6 e lo inoltra a 2.ne

Il processo 8 prende
lo inoltra a 8.next (4)

Se 8 fosse il proces
allora il suo ID circo
tornerà a lui, che q
come nuovo leader



Implementazione dell’algoritmo

L’algoritmo è stato realizzato in un package dedicato (*algorithm*) all’interno di un file chiamato *chang&robertAlgorithm.go*

Il processo effettua dei ping verso il leader (per verificarne lo stato) tramite una funzione chiamata *ChangAndRoberts*

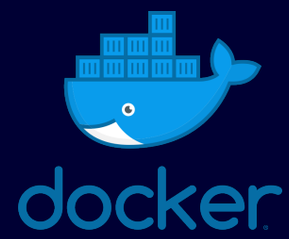
L’elezione viene gestita tramite la funzione *ElectionChangAndRoberts*

La circolazione del messaggio con l’ID del leader viene gestita nella funzione *WinnerMessage*

```go
func ElectionChangAndRoberts(currentNode utils.NodeINFO, mexReply int) {  4 usages  Andrea041 *
    var info utils.Message
    info = utils.Message{SkipCount: 1, MexID: mexReply, CurrNode: currentNode}

    startIndex := currentNode.List.GetIndex(currentNode.Id)
    nextNode := currentNode.List.Nodes[(startIndex+1)%len(currentNode.List.Nodes)]

    peer, err := utils.DialTimeout( network: "tcp", currentNode.List.GetNode(nextNode.Id).Address, 5*time.Second)
    if err != nil {
        /* Calculate skip for node inactivity */
        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( network: "tcp", currentNode.List.GetNode(currentNode.Id).Address, 5*time.Second)

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

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

                return
            }

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

    err = peer.Call( serviceMethod: "PeerServiceHandler.ElectionMessageCR", info,  reply: nil)
    if err != nil { log.Fatal( v...: "Election message forward failed: ", err) }

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

Docker
Accenni sulla progettazione effettuata

Docker



La distribuzione del codice è stata realizzata utilizzando Docker, un software open-source

Sono stati implementati due Dockerfile, in particolare:

- Uno per i processi nel sistema
- Uno per il service registry

Per la composizione dei container è stato realizzato un file *compose.yaml*

I test sul progetto sono stati effettuati con l'utilizzo di 6 repliche dei processi all'interno del sistema