Algoritmi per l'elezione di un leader nei Sistemi distribuiti: Bully e DKR

Lorenzo Grande
Università degli Studi di Roma Tor Vergata
Roma, Italia
Corso di studi magistrale in Ingegneria Informatica
lorenzo.grande@hotmail.it

***Abstract*—Nei contesti degli algoritmi dei sistemi distribuiti, si manifesta frequentemente l'esigenza di un processo atto a fungere da coordinatore o leader. Tale necessità richiede lo sviluppo di algoritmi che, coinvolgendo tutti i processi attivi del sistema, possano coordinarli efficacemente e procedere all'elezione di un unico coordinatore. L'utilità di tali algoritmi diventa evidente se si pensa che un processo di coordinazione potrebbe andare in crash in qualsiasi istante e, in assenza di tali algoritmi, il sistema si troverebbe senza un leader che lo coordini.**

***Index Terms*—Elezione, Leader, service, crash**

I. Introdzione

Un sistema distribuito è un insieme di computer indipendenti che appaiono agli utenti del sistema come un singolo sistema coerente. Questi computer comunicano tra loro tramite la rete e coordinano le loro azioni scambiandosi messaggi. [1] Molto spesso è necessario che, all'interno dei processi che costituiscono il sistema distribuito, ci sia un processo che svolga attività di leader/coordinatore. Nei sistemi distribuiti è quindi essenziale avere degli algoritmi di elezione distribuita che garantiscano la presenza costante di un leader all'interno del sistema. Tali algoritmi devono dunque assicurare che, in caso di guasto del nodo coordinatore, un nuovo nodo venga eletto come leader. In questo articolo, in particolare, ci concentreremo su due algoritmi di elezione: l'algoritmo Bully e l'algoritmo di Dolev-Klawe-Rodeh

II. Analisi del contesto

A. Modello di sistema distribuito

Prima di descrivere gli algoritmi di elezione specifici, è importante delineare il contesto e il modello su cui si basano tali algoritmi. Nel nostro scenario, consideriamo un sistema distribuito così fatto:

- $\mathbf{n}$ processi che comunicano tra loro
- comunicazione affidabile: i messaggi inviati tra i processi non vengono persi, duplicati o consegnati in ordine errato
- processi soggetti a fallimenti
- Ogni processo ha un Id unico e il processo non guasto con l'Id più alto è il processo che dovrà essere eletto come leader.

Inoltre, supponiamo che ogni processo abbia una conoscenza limitata dell'ambiente, in particolare riguardo allo stato degli altri processi. I processi possono quindi rilevare il fallimento di altri processi solo attraverso il mancato ricevimento di messaggi o attraverso segnalazioni esplicite di fallimento.

B. Server registry

Affinché ogni processo sia a conoscenza degli altri nodi nel sistema distribuito è stato realizzato un meccanismo di server discovery.
Quando un nuovo processo desidera entrare a far parte del sistema, questo contatterà il Server registry ad un indirizzo noto, invocando il servizio JoinNetwork, in modo da ottenere:

- Il proprio ID.
- La lista dei processi nel sistema e i relativi ID.

L'Id potrà essere assegnato in modo randomico o in maniera incrementale, l'importante è che ciascun processo abbia un Id che sia unico. Una volta che il nuovo processo è stato registrato, il server registry scorrerà la propria lista di processi e invierà ad ogni peer nella rete, tramite il servizio UpdateNetwork, le informazioni sul nuovo entrato.

C. Server registry e crash-recovery

Come precedentemente menzionato nella sezione subsection II-A, è possibile che, in un dato istante, un processo possa subire un guasto e passare successivamente allo stato di crash-recovery. Quando un nodo raggiunge tale stato, è fondamentale che ne sia consapevole. Pertanto, quando un peer si connette per la prima volta alla rete, memorizza l'ID assegnatogli dal server registry in un file. Questo permette di determinare lo stato attuale del processo semplicemente verificando l'esistenza di tale file al momento dell'avvio.
Quando un processo entra nello stato di ripristino a seguito di un guasto, esso contatta il server Registry per richiamare il servizio JoinNetwork. Durante questa operazione, vengono passati due parametri aggiuntivi:

- Una stringa identificativa: "Recovery"
- Un ID di ripristino, ottenuto dalla lettura del file

Quando il peer invoca il servizio sul server Registry per il ripristino del processo, quest'ultimo esegue una serie di operazioni cruciali:

1) **Identificazione del processo in stato di crash recovery**: Il server legge l'Id passato come parametro del servizio per identificare il processo coinvolto nel ripristino.
2) **Aggiornamento della lista mantenuta dal server**: Utilizzando la propria lista dei processi nella rete, il server verifica se l'indirizzo IP del processo in ripristino è cambiato. In caso affermativo, aggiorna la lista con le informazioni più recenti.
3) **Invio informazioni sulla rete al nodo in ripristino**: Il server invia al processo informazioni aggiornate riguardanti gli altri nodi presenti nella rete. Questo permette al processo di comprendere lo stato attuale della rete e di stabilire connessioni appropriate.
4) **Aggiornamento della rete se necessario:**Se le informazioni relative al nodo in ripristino sono cambiate, il server invoca il servizio UpdateNetwork per garantire che tutti i peer nella rete siano aggiornati.

D. Problemi con il server registry: replicazione e guasti

Come già visto, un peer che entra nella rete contatta un server registry per ottenere informazioni sui nodi connessi. Il server registry svolge quindi un ruolo cruciale, senza il quale un nuovo processo non potrebbe ottenere informazioni né comunicare con gli altri nodi. Avere un unico server registry rappresenterebbe un singolo punto di fallimento del sistema; è quindi opportuno gestire sia i possibili guasti del server, sia avere più repliche del server per distribuire il carico di lavoro. Nel progetto, questi due aspetti non sono stati trattati, ma nella seguente sezione approfondiremo come poterli gestire effettivamente.

Soluzione Proposta:

Una soluzione possibile consiste nel replicare i server registry e utilizzare un load balancer per distribuire le richieste tra di essi. Per non sovraccaricare il load balancer, quest'ultimo non dovrebbe gestire le risposte, ma semplicemente instradare le richieste ai server registry disponibili. Con questa soluzione, ciascun server registry potrà rispondere alle richieste dei peer che vogliono connettersi. Tuttavia, a causa della replicazione, la lista dei processi mantenuta da ciascun server potrebbe non essere consistente.Per risolvere questo problema, si propone l'utilizzo di Apache Kafka. Ogni registry sarebbe registrato a un topic in cui vengono pubblicati messaggi contenenti la lista aggiornata dei peer. Quando un server deve registrare un nuovo nodo in ingresso:

- **Verifica se ci sono messaggi sul topic:** In caso affermativo, consuma i messaggi e aggiorna la propria lista di processi.
- **Risponde al nodo:** Invia la lista aggiornata
- **Pubblica un nuovo messaggio:** Fornisce informazioni sul nuovo peer entrato.

Sarebbe inoltre opportuno gestire la fase di crash-recovery. Per gestire tale evenienza, si potrebbe definire un leader tra tutti i registry che ha il compito di fornire la lista più recente al server in fase di recupero.

E. Fallimento del leader

Nel caso di entrambi gli algoritmi che andremo ad analizzare, deve essere gestita la possibilità che il nodo coordinatore fallisca. Tuttavia, i nodi all'interno della rete non sono in grado di riconoscere quando un altro nodo fallisce; pertanto, è necessario implementare un meccanismo di failure-detection.In particolare, il meccanismo implementato è quello dell'"heartbeating", in cui ogni nodo della rete invia periodicamente segnali, noti come 'heartbeat', al nodo leader per verificarne l'attività. Quando un nodo invia un messaggio di heartbeat al leader, attende una risposta da quest'ultimo. In caso di mancato riscontro, supporrà che il nodo coordinatore sia fallito e avvierà una nuova elezione per eleggere un nuovo leader.

Implementazione della funzione di Heartbeat:

```
func Heartbeat(conn *grpc.ClientConn, err
    error) {
   for {
      log.Printf("Try to connect to leader(Id
          %d)\n", shared.LeaderId)
      time.Sleep(5000 * time.Millisecond)
      for _, leader := range shared.PeerList {
         if leader.Leader == true {
            shared.LeaderId = leader.Id
            conn, err = grpc.Dial(leader.Addr,
                grpc.WithInsecure())
            if err != nil {
               log.Fatalf("failed to connect
                   to registry: %v", err)
            }
            defer conn.Close()

            peer := registry.NewServiceClient(
                conn)

            _, leaderror := peer.HeartBeat(
                context.Background(), &
                registry.HeartBeatMessage{
                Message: "HeartBeat"})
            if leaderror != nil {
               log.Printf("Leader unreachable:
                    \n")
               if config.BullySelected == true
                   {
                  service.Bully()
               } else if config.DolevSelected
                   == true {
                  service.DolevStartElection()
               } else {
                  service.DKR()
               }

            } else {
               log.Printf("The leader is alive
                   \n")
            }
         }
      }
   }
}
```

III. Algoritmi di Elezione distribuita

In questa sezione, approfondiremo gli algoritmi di elezione distribuita e la loro implementazione.

A. Algoritmo di Bully

[2] L'idea principale dell'algoritmo di Bully è che il nodo con l'ID più alto si fa strada prepotentemente verso la leadership. Supponiamo di avere un sistema con N processi e che il processo p_i si accorga che il coordinatore non sia più attivo. **Funzionamento:**

- p_i avvia l'elezione inviando un messaggio di **Election** a tutti i processi con ID più alto del suo.
- Se nessuno risponde, p_i vince l'elezione e diventa il coordinatore e invia a tutti i processi un messaggio **Coordinator**.
- Se un processo p_k con ID più alto riceve il messaggio **Election** da p_i, gli risponde con un messaggio **OK** che blocca l'elezione precedente. Il processo p_k avvierà quindi una nuova elezione ripetendo gli stessi passaggi sopracitati.

Nota: Affinché l'algoritmo funzioni è necessario che il sistema distributo sia sincrono. Infatti il processo p_i deve sapere quale è il tempo di attesa prima di essere sicuro che non ha ricevuto risposta dal processo p_j.

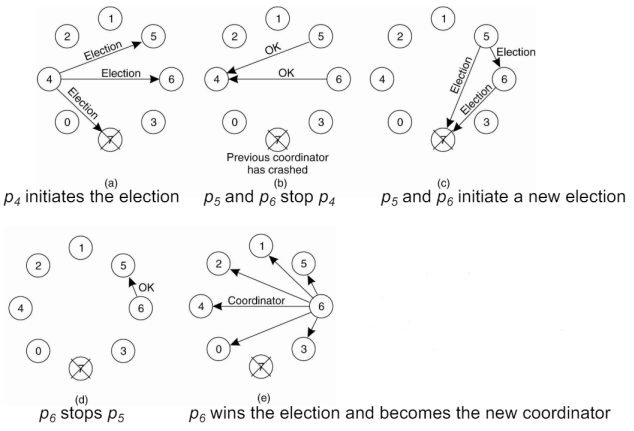


Fig. 1. Bully Election: [2]

Costo di comunicazione: Al fine di valutare l'efficienza di un algoritmo, possiamo valutare il numero di messaggi scambiati. Possiamo distinguere due casi:

- **Best case**: Supponiamo che sia il processo con il secondo ID più alto ad accorgersi che il leader è fallito. Il processo può selezionarsi come coordinatore e inviare quindi N-2 messaggi di coordinazione. **Costo di computazione:** $\mathcal{O}$(N)
- **Worst case:** Supponiamo che il processo con l'ID più piccolo avvia l'elezione. Lui invia N-1 messaggi ai processi, che a loro volta inizieranno l'elezione: ((N-1) + (N-2) + ...+ 1) + N-1(Messaggio di **Coordinator**). **Costo di computazione:** $\mathcal{O}(N^2)$

B. Implementazione dell'algoritmo di Bully

L'implementazione dell'algoritmo è suddivisa come segue:

- Una funzione Bully
- Un servizio BullyElection

Condizioni di ingresso:
L'algoritmo di bully viene invocato in due casi:

- All'avvio di ciascun peer, in moda da conoscere chi è il leader attuale nella rete;
- Quando un peer non riceve risposta da un messaggio di Heartbeat, come visto nella subsection II-E

Bully()
Funzione Bully:
Nella funzione Bully, il peer p_i che vi entra deve inviare un messaggio di **Election** a tutti i processi con ID maggiore del suo.

```
for _, server := range shared.PeerList {
      if server.Id > shared.MyId {
         //Invio messaggio Election:

         conn, err := grpc.Dial(server.Addr,
             grpc.WithInsecure())
         if err != nil {
            log.Fatalf("Failed to connect to
                Peer: %v", err)
         }
         defer conn.Close()

         client := pb.NewElectionClient(conn)

         reply, err := client.BullyElection(
             context.Background(), &pb.
             ElectionRequest{
            Election: "Election",
            ElectionId: shared.MyId,
         })
         if err != nil {
            continue
         }
         if reply.ElectionReply == "OK" {
            log.Printf("I can't became leader\
                n")
            return
         }
      }
   }
```

Come si può vedere dal codice, basta una risposta da un processo con ID più alto per far sì che il nodo p_i non possa diventare leader.
BullyElection:
Quando un peer p_i riceve un messaggio **Election** da p_j, si limita semplicemente a rispondere e a avviare una nuova elezione invocando la funzione Bully() sopra menzionata.
Nel caso in cui, invece, p_i riceva un messaggio **Coordination**, significa che è stato eletto un nuovo leader. Il nodo p_i aggiorna quindi la conoscenza che ha sul leader.

C. Algoritmo DKR

[4] [5] A differenza dell'algoritmo di bully, che può operare in reti senza una struttura predeterminata, l'algoritmo

di elezione distribuita di Dolev-Klawe-Rodeh è stato appositamente concepito per reti organizzate in una struttura ad anello unidirezionale. Questo algoritmo si basa sull'idea di confrontare gli identificatori dei nodi con quelli dei loro vicini per individuare un massimo locale. Successivamente, i nodi condivideranno i loro massimi locali con l'obiettivo di determinare un massimo globale per l'intera rete.

Variabili dell'algoritmo:

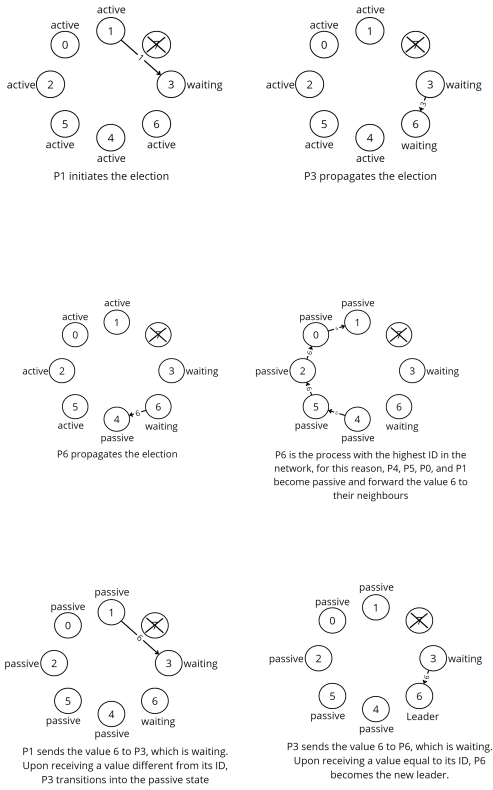
Ogni processo nella rete possiede le seguenti variabili:

- **StimateLeader**: Rappresenta la conoscenza che un nodo ha della rete. Ad ogni istante di tempo rappresenta il massimo locale conosciuto dal processo. All'inizio, ogni processo ha la variabile StimateLeader pari al proprio Id. Con lo scambio dei messaggi tra i vari processi, la variabile sarà aggiornata al valore $\max(StimateLeader, ReceivedId)$. Questa variabile è essenziale per determinare il massimo locale in fase di elezione e contribuisce al processo di individuazione del leader globale.
- **State**: Indica lo stato del nodo, che può essere:
  - **Active**: All'inizio di ogni elezione, ogni processo è in uno stato attivo. Indica che il processo non ha ancora informazioni sui propri vicini e rappresenta un possibile candidato per diventare leader.
  - **Waiting**: Indica che il processo ha ricevuto messaggi dai suoi vicini e, per il momento, il suo ID rappresenta il massimo locale. Il nodo rappresenta un possibile candidato per diventare leader.
  - **Passive**: Indica che il processo ha ricevuto messaggi dai suoi vicini e il suo ID è minore del massimo locale. In questo stato, il nodo non rappresenta un candidato per diventare leader.

Funzionamento:

- p_i invia la sua variabile **stimateLeader** al nodo che lo sussegue in senso orario. Se il nodo non risponde, invio al successore del mio successore.
- Un nodo p_j che riceve un messaggio dal proprio vicino, verifica il proprio stato e si comporta come segue:
  - **Il processo p_j è nello stato active**: Confronta il valore ricevuto (ReceivedStimateLeader) con la sua variabile stimateLeader:
    - ***Se ReceivedStimateLeader > stimateLeader:*** Pongo stimateLeader = ReceivedStimateLeader ed entro nello stato **passive**.
    - **Altrimenti**, entro nello stato **waiting**

    In seguito, Propago al mio vicino l'informazione
- **Il processo p_j è nello stato passive**: Confronto il valore ricevuto (ReceivedStimateLeader) con la sua variabile stimateLeader, aggiornando quest'ultima se necessario (come nel caso descritto sopra) e propagando l'informazione.
- **Il processo p_j è nello stato waiting**: Confronta il valore ricevuto (ReceivedStimateLeader) con la sua variabile stimateLeader:
  - ***Se ReceivedStimateLeader = Id(p_j):*** p_j diventa Leader.
  - **Altrimenti**, il processo entra nello stato **passive**. aggiorna la propria variabile StimateLeader e propaga al proprio vicino l'informazione .



Costo di computazione:

In ogni caso il costo di computazione dell'algoritmo DKR è pari a $\mathcal{O}(n \log n)$

D. Implementazione dell'algoritmo DKR

L'implementazione dell'algoritmo è suddivisa, come nell'algoritmo di Bully in due metodi:

- Una funzione DKR
- Un servizio DKRElection

Come già citato nella subsection III-C, l'algoritmo utilizza due variabili inizialmente settate:

```
StimateLeader = MyId
State = "active"
```

Condizioni di ingresso:

Come già visto nell'implementazione dell'algoritmo di Bully, anche in tal caso un peer inizia l'elezione nel momento in cui un peer non riceve più risposta ai messaggi di HeartBeat.

Funzione DKR:

Nella funzione DKR, il peer p_i che vi entra deve inviare la propria variabile StimateLeader al proprio vicino p_{i+1}. La prima operazione che viene svolta è quindi il calcolo del nodo successivo e l'invio della suddetta variabile.

```
    nextNode := math.Mod(float64(shared.MyId+1)
        , float64(shared.NumNode))
```

Se tale operazione dovesse fallire, si proverà a contattare il successore del mio successore e così via finché non avremo inviato la nostra variabile StimateLeader.

DKRElection:

Quando il peer p_i riceve una variabile StimateLeader dal suo vicino p_{i-1}, quest'ultimo dovrà seguire lo schema elencato nella sezione III-C-Funzionamento. A seconda dello stato in cui si trova il nodo e della variabile `req.ElectionID`, svolgerà operazioni diverse come segue:

```
    func (s *Election) DKRElection(ctx context.
        Context, req *pb.ElectionRequest) (*pb.
        ElectionReply, error) {
    if req.Election == "Election" {
       //Ho ricevuto dal mio predecessore un
           stimateLeader, lo confronto con il
           mio:
       //Se sono nello stato di waiting e
           ricevo un valore pari al mio Id,
           devo divntare il leader
       if shared.State == "waiting" && req.
           ElectionId == shared.MyId {
          BecomeleaderDKR()
          return nil, nil
       }

       //Se l'id ricevuto, e' maggiore del mio
           stimateLeader, aggiorno il mio
           EstimateLeader:
       if req.ElectionId > shared.MyId {
          shared.StimateLeader = req.ElectionId
          shared.State = "passive"
       } else {
          shared.State = "waiting"
       }
       //Propago l'informazione
       DKR()
       return nil, nil
    }
    //Altrimenti ho un messaggio di Update:
    //Ripristino stato e StimateLeader:
    shared.StimateLeader = shared.MyId
    shared.State = "active"
    for i := range shared.PeerList {
       if shared.PeerList[i].Id == req.
           ElectionId {
          shared.PeerList[i].Leader = true
          shared.LeaderId = shared.PeerList[i].
              Id
       } else {
          shared.PeerList[i].Leader = false
       }
       log.Printf("After Update message peer: %
           d have leader: %t:", shared.PeerList
           [i].Id, shared.PeerList[i].Leader)
    }
    return nil, nil
}
```

IV. Tecnologie usate

Nella seguente sezione andremo ad esaminare le tecnologie utilizzate nel nostro progetto, partendo dal linguaggio di programmazione adottato e procedendo poi ad esplorare gli aspetti legati al deployment della stessa applicazione.

A. Linguaggio di programmazione: gRPC in Go

[6] [7] gRPC è un framework di comunicazione remota sviluppato da Google. È progettato per supportare una vasta gamma di esigenze di connettività tra servizi, rendendolo ideale per i sistemi distribuiti. Le sue principali caratteristiche sono:

- **Supporto per applicazioni poliglotte**: Una delle caratteristiche principali di gRPC è la possibilità di scrivere i diversi programmi in differenti linguaggi di programmazione, mantenendo però la capacità di comunicazione fra di essi. Tale framework si presta per taòe motivo a connettere microservizi poliglotti che utilizzano la comunicazione in stile richiesta-risposta.
- **Comunicazione affidabile**: gRPC utilizza per la comunicazione HTTP/2, offrendo così la possibilità di avere streaming bidirezionale, consentendo a client e server di leggere e scrivere in modo asincrono.
- **Utilizzo di Protocol Buffer**: gRPC utilizza Protocol Buffers sia come IDL (Interface Definition Language), che permette la generazione automatica del client/server stub, sia come formato di scambio di messaggi, rendendo più piccolo il payload e più efficiente la connessione.

Sono proprio le caratteristiche sopra enunciate il motivo della scelta di tale linguaggio di programmazione.
Inoltre, l'utilizzo con Go offre anch'esso i suoi vantaggi: Go è noto per il suo modello di concorrenza basato su goroutine e canali. Questo si adatta bene con gRPC e rende la gestione delle connessioni simultanee molto più semplice.

B. Docker

Docker è una tecnologia di virtualizzazione basata su container. Un container Docker contiene un'applicazione e le sue dipendenze, isolandole dal resto del sistema e condividendo solo il kernel del sistema operativo con altri container. Questa architettura ci consente di creare, distribuire e gestire facilmente i container delle nostre applicazioni. L'utilizzo di Docker porta numerosi vantaggi. Innanzitutto, ci consente di isolare le nostre applicazioni e le relative dipendenze, garantendo che esse siano eseguite in ambienti containerizzati rendendo più agevole il processo di deployment su diversi ambienti. Nel nostro progetto, abbiamo implementato due DockerFile per la creazione dei container. Uno è dedicato al server registry, mentre l'altro gestisce i processi nel sistema. Inoltre, per orchestrare e avviare l'insieme di container, abbiamo adottato Docker Compose. Abbiamo creato un file denominato 'compose.yaml', dove sono specificati i servizi necessari e le configurazioni per comporre e avviare i container.

C. Istanza EC2

Amazon Elastic Compute Cloud (EC2) è un servizio di calcolo basato su cloud offerto da Amazon Web Services (AWS). Consente agli utenti di eseguire istanze virtuali con

vari sistemi operativi, dimensioni e configurazioni di rete, offrendo così una capacità di calcolo scalabile e flessibile. Le istanze EC2 forniscono un'infrastruttura affidabile e on-demand per l'esecuzione di applicazioni, consentendo agli sviluppatori di distribuire facilmente le proprie soluzioni software su un ambiente cloud sicuro e gestibile.

Esecuzione su istanza EC2:

Dopo aver completato lo sviluppo del nostro codice, abbiamo proceduto al deployment dell'applicazione su un'istanza EC2. Inizialmente, abbiamo configurato un'istanza EC2 selezionando le specifiche desiderate, come dimensioni, sistema operativo e configurazioni di rete. Successivamente, abbiamo trasferito i file DockerFile, il file di composizione e il codice dell'applicazione sull'istanza EC2 tramite il comando git clone. Una volta trasferiti con successo, abbiamo installato Docker sull'istanza EC2 per abilitare l'esecuzione dei container. Una volta completata l'installazione di Docker, abbiamo utilizzato i Dockerfile e il file di composizione per creare e avviare i container necessari per l'esecuzione dell'applicazione. Abbiamo verificato il corretto funzionamento dell'applicazione sull'istanza EC2, assicurandoci che tutti i componenti fossero operativi e funzionali. EC2.

REFERENCES

[1] M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.

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

[3] G.L. Peaterson, An $\mathcal{O}(n \log n)$ unidirectional algorithm for the circular extrema problem

[4] Dolev, D., Klawe, M., and Rodeh, M. An $\mathcal{O}(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. IBM Research Rep. RJ3185, IBM Corp., San Jose, Calif., July 1981.

[5] TP2 MVFA: Peterson mutual exclusion and DKR leader election

[6] Valeria Cardellini, "Communication in Distributed System: RPC", University of Rome Tor Vergata.

[7] gRPC