**Laborator Nr. 8**

**Problema Alegerii Liderului (Leader Election)**

Problema **alegerii liderului (Leader Election)** este utilizată în sisteme distribuite și presupune identificarea unui nod unic (lider) dintre mai multe noduri, astfel încât acel nod să fie responsabil de luarea deciziilor sau coordonarea activităților din sistem. Liderul este esențial pentru coordonarea eficientă, evitarea conflictelor și menținerea coerenței în sistem.

Sunt anumite caracteristici principale:

1. **Distribuire**. Toate nodurile din sistem sunt egale la început, iar procesul trebuie să desemneze unul dintre ele drept lider. Nodurile pot fi distribuite pe mai multe mașini fizice sau virtuale.
2. **Fără conflicte**. Trebuie garantat că la finalul procesului, există exact un lider desemnat.
3. **Detectarea defectelor**. Dacă liderul cade, sistemul trebuie să detecteze acest lucru și să aleagă un nou lider.
4. **Comunicare**. Nodurile comunică între ele prin mesaje pentru a negocia și a decide liderul.
5. **Toleranță la erori**. Procesul trebuie să fie rezilient la pierderi de mesaje, latențe sau defectarea unor noduri.

Considerăm următoarele exemple de scenarii pentru alegerea liderului:

1. **Coordonarea într-un cluster de baze de date**. Liderul gestionează scrierile, iar ceilalți noduri replică datele.
2. **Sisteme distribuite de fișiere**. Liderul este responsabil de gestionarea metadatelor.
3. **Sisteme distribuite de stocare**. Liderul coordonează alocarea resurselor și replicarea.

Există și provocări peste care trebuie să trecem în implementarea acestora, precum:

1. **Failover**. Dacă liderul cade, nodurile rămase trebuie să detecteze rapid acest lucru și să repornească procesul de alegere.
2. **Latență**. Comunicarea între noduri poate întârzia procesul, iar algoritmii trebuie să gestioneze acest lucru.
3. **Concurență**. Nodurile pot începe simultan procese de alegere, ceea ce poate duce la conflicte sau rezultate eronate.
4. **Rețele partitionate**. În cazul unei partitionări a rețelei, fiecare subgrup poate alege un lider, ducând la probleme de consistență.

Algoritmii utilizați în problema alegerii liderului sunt:

1. **Algoritmul Bully**

* Nodurile cu ID-uri mai mari pot înlocui liderii cu ID-uri mai mici.
* Toate nodurile cunosc lista completă a celorlalte noduri.
* Nodul cu cel mai mare ID devine lider.

1. **Alegerea pe inel (Ring Election)**

* Nodurile sunt aranjate logic într-un inel.
* Fiecare nod propune un lider și transmite această propunere următorului nod, comparând valorile propuse.

1. **Protocolul Raft**

* Utilizat în sisteme distribuite pentru replicarea jurnalelor (log replication).
* Un nod devine lider prin obținerea unui număr majoritar de voturi.

1. **Paxos**

* Un protocol mai complex, care asigură consistența distribuită prin propuneri și acceptări de mesaje.

**Implementarea în limbajul de programare Go a Algoritmului Bully**

Algoritmul Bully este folosit pentru alegerea liderului într-un sistem distribuit. Nodul cu cel mai mare ID devine lider. Nodurile cu ID-uri mai mici acceptă acest lucru, iar liderul coordonează activitățile sistemului.

package main

import (

"fmt"

"sync"

"time"

)

// Node reprezintă un nod într-un sistem distribuit.

type Node struct {

ID int // Identificator unic

IsAlive bool // Indică dacă nodul este activ

IsLeader bool // Indică dacă nodul este lider

Cluster \*Cluster // Referință către cluster

mu sync.Mutex

ElectionOngoing bool // Flag pentru a preveni multiple alegeri simultane

}

// Cluster reprezintă un grup de noduri.

type Cluster struct {

Nodes []\*Node

mu sync.Mutex

}

// NewNode creează un nou nod activ.

func NewNode(id int, cluster \*Cluster) \*Node {

return &Node{

ID: id,

IsAlive: true,

Cluster: cluster,

}

}

// NewCluster creează un cluster cu noduri distribuite.

func NewCluster(size int) \*Cluster {

cluster := &Cluster{}

for i := 1; i <= size; i++ {

cluster.Nodes = append(cluster.Nodes, NewNode(i, cluster))

}

return cluster

}

// StartElection inițiază procesul de alegere a liderului.

func (n \*Node) StartElection() {

n.mu.Lock()

if n.ElectionOngoing {

n.mu.Unlock()

return // O alegere este deja în desfășurare

}

n.ElectionOngoing = true

n.mu.Unlock()

fmt.Printf("Nodul %d inițiază o alegere.\n", n.ID)

highestID := n.ID

for \_, node := range n.Cluster.Nodes {

if node.ID > n.ID && node.IsAlive {

fmt.Printf("Nodul %d notifică nodul %d despre alegere.\n", n.ID, node.ID)

highestID = node.ID

node.StartElection()

}

}

if highestID == n.ID {

n.BecomeLeader()

}

}

// BecomeLeader setează nodul curent ca lider.

func (n \*Node) BecomeLeader() {

n.mu.Lock()

defer n.mu.Unlock()

fmt.Printf("Nodul %d devine lider.\n", n.ID)

for \_, node := range n.Cluster.Nodes {

node.mu.Lock()

node.IsLeader = false

node.mu.Unlock()

}

n.IsLeader = true

n.ElectionOngoing = false

}

// SimulateFailure simulează defectarea unui nod.

func (n \*Node) SimulateFailure() {

n.mu.Lock()

n.IsAlive = false

n.IsLeader = false

n.mu.Unlock()

fmt.Printf("Nodul %d a căzut!\n", n.ID)

}

// MonitorCluster detectează dacă liderul cade și inițiază o nouă alegere.

func (c \*Cluster) MonitorCluster() {

for {

time.Sleep(2 \* time.Second) // Monitorizare periodică

c.mu.Lock()

var leaderExists bool

for \_, node := range c.Nodes {

node.mu.Lock()

if node.IsLeader && node.IsAlive {

leaderExists = true

}

node.mu.Unlock()

}

if !leaderExists {

fmt.Println("Liderul a căzut. Inițierea unei noi alegeri.")

for \_, node := range c.Nodes {

node.mu.Lock()

if node.IsAlive {

go node.StartElection()

node.mu.Unlock()

break

}

node.mu.Unlock()

}

}

c.mu.Unlock()

}

}

func main() {

// Creăm un cluster cu 5 noduri

cluster := NewCluster(5)

// Pornim monitorizarea clusterului

go cluster.MonitorCluster()

// Pornim procesul inițial de alegere

cluster.Nodes[0].StartElection()

// Simulăm defectarea liderului

time.Sleep(5 \* time.Second)

for \_, node := range cluster.Nodes {

node.mu.Lock()

if node.IsLeader {

go node.SimulateFailure()

}

node.mu.Unlock()

}

// Așteptăm câteva secunde pentru a permite clusterului să proceseze

time.Sleep(10 \* time.Second)

}

**Documentația codului sursă**

**Principalele concepte și structuri folosite în codul de mai sus sunt :**

1. **Node**:
   * Reprezintă un nod din sistemul distribuit.
   * ID: Identificatorul nodului (unic).
   * IsAlive: Indică dacă nodul este activ.
   * IsLeader: Indică dacă nodul este lider.
   * ElectionOngoing: Previne inițierea mai multor procese de alegere simultan.
2. **Cluster**:
   * Reprezintă un grup de noduri distribuite.
   * Contine metode pentru monitorizarea nodurilor și detectarea defectelor.

**Funcționalitățile principale sunt:**

1. **Alegerea liderului**:
   * Un nod inițiază procesul de alegere.
   * Nodurile notifică toate nodurile cu ID-uri mai mari pentru a participa la alegere.
   * Nodul cu cel mai mare ID devine lider.
2. **Gestionarea defectelor**:
   * Dacă liderul cade, clusterul detectează acest lucru și inițiază o nouă alegere.
3. **Monitorizarea clusterului**:
   * Clusterul verifică periodic dacă liderul este funcțional.
   * Dacă liderul este defect, se inițiază o alegere.
4. **Simularea defectării nodurilor**:
   * Nodurile pot fi marcate ca defecte (inactivate), iar clusterul reconfigurează alegerea liderului.

**La rulare sunt simulate următoarele:**

1. Un cluster cu 5 noduri este creat.
2. Nodurile participă la procesul de alegere, iar cel cu ID-ul cel mai mare devine lider.
3. Liderul este "căzut" artificial, iar clusterul detectează acest lucru și inițiază o nouă alegere.

**Exerciții**

* 1. Rulați programul de mai sus
  2. Studiați output-ul rulării programului și codul sursă
  3. Toate rutinele go sunt în sleep după prima rulare și se ajunge în deadlock. Ceea ce înseamnă un blocaj între rutinele go. Rezolvați această situație, astfel încât să nu se ajungă într-un ciclu închis de așteptare reciprocă.

**Implementarea în limbajul de programare Go a algoritmului de alegere pe inel (Ring Election)**

Algoritmul **Alegerea pe Inel** presupune organizarea nodurilor într-un inel logic. Nodurile comunică prin transmiterea de mesaje, iar fiecare nod propagă un mesaj care conține un ID propus pentru lider. La final, nodul cu cel mai mare ID devine lider.

package main

import (

"fmt"

"sync"

"time"

)

// Node reprezintă un nod într-un sistem distribuit.

type Node struct {

ID int // Identificator unic

IsLeader bool // Indică dacă nodul este lider

Next \*Node // Referință către următorul nod din inel

mu sync.Mutex

}

// Cluster reprezintă un inel logic format din noduri.

type Cluster struct {

Nodes []\*Node

mu sync.Mutex

}

// NewNode creează un nou nod.

func NewNode(id int) \*Node {

return &Node{

ID: id,

IsLeader: false,

}

}

// NewCluster creează un cluster cu noduri aranjate în inel.

func NewCluster(size int) \*Cluster {

cluster := &Cluster{}

for i := 1; i <= size; i++ {

cluster.Nodes = append(cluster.Nodes, NewNode(i))

}

// Conectăm nodurile în inel

for i := 0; i < size; i++ {

cluster.Nodes[i].Next = cluster.Nodes[(i+1)%size]

}

return cluster

}

// StartElection inițiază alegerea liderului folosind algoritmul Ring Election.

func (n \*Node) StartElection() {

fmt.Printf("Nodul %d inițiază alegerea.\n", n.ID)

proposedID := n.ID

n.PropagateMessage(proposedID)

}

// PropagateMessage propagă un mesaj cu ID-ul propus către următorul nod din inel.

func (n \*Node) PropagateMessage(proposedID int) {

time.Sleep(1 \* time.Second) // Simulează o latență în comunicare

n.Next.mu.Lock()

defer n.Next.mu.Unlock()

fmt.Printf("Nodul %d primește mesajul cu ID-ul propus: %d.\n", n.Next.ID, proposedID)

if n.Next.ID == n.ID {

// Mesajul s-a întors la inițiator

fmt.Printf("Nodul %d finalizează alegerea. Liderul este nodul cu ID-ul: %d.\n", n.ID, proposedID)

n.DeclareLeader(proposedID)

return

}

// Propagăm ID-ul maxim

if proposedID > n.Next.ID {

n.Next.PropagateMessage(proposedID)

} else {

n.Next.PropagateMessage(n.Next.ID)

}

}

// DeclareLeader declară liderul pentru toate nodurile din inel.

func (n \*Node) DeclareLeader(leaderID int) {

current := n

for {

current.mu.Lock()

current.IsLeader = (current.ID == leaderID)

current.mu.Unlock()

if current.Next.ID == n.ID {

break

}

current = current.Next

}

fmt.Printf("Liderul a fost ales: Nodul %d.\n", leaderID)

}

// PrintClusterStatus afișează starea actuală a clusterului.

func (c \*Cluster) PrintClusterStatus() {

fmt.Println("Starea clusterului:")

for \_, node := range c.Nodes {

role := "Participă"

if node.IsLeader {

role = "Lider"

}

fmt.Printf("Nod %d: %s\n", node.ID, role)

}

}

func main() {

// Creăm un cluster cu 5 noduri

cluster := NewCluster(5)

// Afișăm starea inițială a clusterului

cluster.PrintClusterStatus()

// Inițiem procesul de alegere de la un nod

time.Sleep(1 \* time.Second)

go cluster.Nodes[2].StartElection()

// Așteptăm câteva secunde pentru ca alegerea să fie completă

time.Sleep(10 \* time.Second)

// Afișăm starea finală a clusterului

cluster.PrintClusterStatus()

}

**Documentația codului sursă**

**Principalele structure și concepte folosite sunt:**

1. **Node**:
   * Reprezintă un nod din sistemul distribuit.
   * ID: Identificator unic al nodului.
   * IsLeader: Indică dacă nodul este lider.
   * Next: Referință către următorul nod din inel.
2. **Cluster**:
   * Reprezintă un grup de noduri conectate într-un inel logic.

**Funcționalități principale sunt:**

1. **Inițierea alegerii**:
   * Un nod începe procesul de alegere propagând un mesaj care conține propriul ID.
2. **Propagarea mesajelor**:
   * Fiecare nod primește mesajul și îl propagă mai departe, trimițând ID-ul maxim observat.
3. **Determinarea liderului**:
   * Când mesajul ajunge înapoi la inițiator, liderul este stabilit ca nodul cu cel mai mare ID.
4. **Anunțarea liderului**:
   * Toate nodurile sunt actualizate pentru a recunoaște liderul.

**La rulare sunt simulate următoarele:**

1. Se creează un cluster cu 5 noduri conectate într-un inel logic.
2. Un nod inițiază procesul de alegere.
3. Mesajul circulă în inel până când se determină liderul.
4. Starea finală a clusterului este afișată, cu liderul marcat.

**Exerciții:**

1. **Exercițiul 1: Implementarea unui inel dynamic. Ne vom concentra pe înțelegerea structurii inelului și pe adaptarea algoritmului la modificări dinamice.** Extinde implementarea algoritmului astfel încât să permită adăugarea și eliminarea dinamică a nodurilor în inel în timpul execuției.Scopul constă în averifica modul în care sistemul poate gestiona modificările structurii inelului și asigură stabilitatea procesului de alegere.
   * Considerăm următoarele indicații:
     + Creează funcții pentru adăugarea și eliminarea nodurilor.
     + Asigură-te că toate nodurile rămase sunt conectate corect în inel.
2. **Exercițiul 2: Gestionarea nodurilor defecte. Obiectivul acestui exercițiu este de a testa capacitatea de a gestiona defectele, un aspect esențial în sistemele distribuite**.Modifică algoritmul pentru a include detectarea și excluderea nodurilor care devin inactive (ex. simularea căderii unui nod) prin testarea rezilienței algoritmului în cazul defectării unuia sau mai multor noduri.
   * Considerăm următoarele indicații:
     + Adaugă o funcție pentru simularea defectării unui nod (IsAlive = false).
     + Modifică propagarea mesajelor astfel încât să treacă peste nodurile inactive.
3. **Exercițiul 3: Simularea cu latențe variabile. Acest exercițiu include concepte de latență și performanță, apropiind implementarea de condițiile reale din rețele**. Adaugă latențe variabile pentru comunicarea între noduri, simulând condițiile reale dintr-o rețea distribuită, scopul constând în observarea efectului latențelor asupra timpului total necesar pentru finalizarea alegerii liderului.
   * Considerăm următoarele indicații:
     + Folosește funcția time.Sleep pentru a adăuga întârzieri aleatorii în propagarea mesajelor.
     + Măsoară timpul total necesar pentru alegerea liderului și afișează rezultatele.