**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

**ADDIS ABABA INSTITUTE OF TECHNOLOGY, ADDIS ABABA UNIVERSITY**

Topic: Implement Paxos Algorithm in Go (Webservice, Kubernetes)

**Objective**: The lab activities aim to provide you with hands-on experience in implementing and deploying distributed systems. By the end of the lab series, students will have mastered the following key objectives:

**Activity 1: Implement Paxos Algorithm in Go**

**Objective:** Implement the Paxos consensus algorithm and simulate communication between nodes.

**Step-by-Step Instructions:**

1. **Define Paxos message structures.**

messages.go

package paxos

type Prepare struct {

ProposalNumber int

}

type Promise struct {

ProposalNumber int

AcceptedValue interface{}

}

type Accept struct {

ProposalNumber int

Value interface{}

}

type Accepted struct {

ProposalNumber int

Value interface{}

}

1. **Implement the Acceptor role.**

acceptor.go

package paxos

import "sync"

type Acceptor struct {

mu sync.Mutex

promisedNumber int

acceptedNumber int

acceptedValue interface{}

}

func (a \*Acceptor) HandlePrepare(p Prepare) Promise {

a.mu.Lock()

defer a.mu.Unlock()

if p.ProposalNumber > a.promisedNumber {

a.promisedNumber = p.ProposalNumber

return Promise{ProposalNumber: p.ProposalNumber, AcceptedValue: a.acceptedValue}

}

return Promise{}

}

func (a \*Acceptor) HandleAccept(ac Accept) Accepted {

a.mu.Lock()

defer a.mu.Unlock()

if ac.ProposalNumber >= a.promisedNumber {

a.promisedNumber = ac.ProposalNumber

a.acceptedNumber = ac.ProposalNumber

a.acceptedValue = ac.Value

return Accepted{ProposalNumber: ac.ProposalNumber, Value: ac.Value}

}

return Accepted{}

}

1. **Implement the Proposer role.**

proposer.go

package paxos

type Proposer struct {

ProposalNumber int

Value interface{}

}

func (p \*Proposer) Propose(value interface{}, acceptors []Acceptor) interface{} {

promises := 0

for \_, acceptor := range acceptors {

promise := acceptor.HandlePrepare(Prepare{ProposalNumber: p.ProposalNumber})

if promise.ProposalNumber == p.ProposalNumber {

promises++

}

}

if promises > len(acceptors)/2 {

accepted := 0

for \_, acceptor := range acceptors {

ack := acceptor.HandleAccept(Accept{ProposalNumber: p.ProposalNumber, Value: value})

if ack.ProposalNumber == p.ProposalNumber {

accepted++

}

}

if accepted > len(acceptors)/2 {

return value

}

}

return nil

}

1. **Simulate Paxos**

package main

import (

"fmt"

"paxos-lab/paxos"

)

func main() {

acceptors := []\*paxos.Acceptor{

&paxos.Acceptor{},

&paxos.Acceptor{},

&paxos.Acceptor{},

}

proposer := paxos.Proposer{ProposalNumber: 1, Value: "Distributed Systems"}

value := proposer.Propose("Distributed Systems", acceptors)

if value != nil {

fmt.Printf("Consensus reached on value: %s\n", value)

} else {

fmt.Println("Consensus not reached")

}

}

**Instructions for Testing:**

1. Create files (messages.go, acceptor.go, proposer.go, and main.go) in the paxis-lab project directory.
2. Run the application locally:

go run main.go

1. Verify the output, e.g., Consensus reached on value: Distributed Systems.

Exercise 1:

1. **Test Locally**:

* Modify the code to simulate 2k+1 instances (e.g., 5 acceptors, 1 proposer).
* Test with simulated failures (e.g., remove one or two acceptors) and observe consensus behavior.

1. **Test on Virtual Machines**:

* Run main.go on separate machines or virtual machines, modifying communication to use HTTP instead of local calls.

**Activity 2: Building a Paxos-Based Web Service**

**Objective:** Create a web service for distributed consensus using Paxos, allowing clients to propose values via HTTP.

**Step-by-Step Instructions:**

**1. Web Service with Paxos Integration (web-service.go):**

package main

import (

"encoding/json"

"fmt"

"net/http"

"sync"

"paxos-lab/paxos"

)

var (

acceptors = []\*paxos.Acceptor{{}, {}, {}}

mu sync.Mutex

)

func proposeHandler(w http.ResponseWriter, r \*http.Request) {

var body struct {

ProposalNumber int

Value string

}

json.NewDecoder(r.Body).Decode(&body)

proposer := paxos.Proposer{ProposalNumber: body.ProposalNumber, Value: body.Value}

mu.Lock()

value := proposer.Propose(body.Value, acceptors)

mu.Unlock()

if value != nil {

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Consensus reached: %s\n", value)

} else {

w.WriteHeader(http.StatusConflict)

fmt.Fprintf(w, "Consensus not reached\n")

}

}

func main() {

http.HandleFunc("/propose", proposeHandler)

http.ListenAndServe(":8080", nil)

}

**Instructions for Testing:**

1. Save as web-service.go and run:

go run web-service.go

1. Test with curl

curl -X POST -H "Content-Type: application/json" -d '{"ProposalNumber":1,"Value":"Hello"}' http://localhost:8080/propose

**Exercise**

1. Deploy the web service on multiple machines or Kubernetes pods.
2. Modify the client code to send proposals to all instances and verify consensus.

**Activity 3: Adding Fault Tolerance**

**Objective:** Enhance the Paxos web service to handle node failures gracefully.

**Tasks**:

1. Use Go's context package for timeouts.
2. Log failed communication attempts.
3. Retry failed proposals.

**Activity 4: Deploying on Kubernetes**

**Objective:** Deploy the Paxos web service on Kubernetes for fault tolerance and scalability.

**Steps to Deploy**

1. **Write a Deployment YAML**:

Deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: paxos-service

spec:

replicas: 3

selector:

matchLabels:

app: paxos

template:

metadata:

labels:

app: paxos

spec:

containers:

- name: paxos

image: your-docker-repo/paxos:latest

ports:

- containerPort: 8080

1. **Deploy to Kubernetes**:

kubectl apply -f deployment.yaml

**Exercise**

1. Simulate pod failures using kubectl delete pod.
2. Scale replicas dynamically

kubectl scale deployment paxos-service --replicas=5

**Submission Guidelines**

1. **Organize Your Work**:

Create a folder named <YourName>\_<LabTitle> (e.g., Wondimagegn\_PaxosLab).

1. Include all source code files in the code/ directory.
2. Provide **screenshots** of your work running on your system. Include:
   * Terminal output of the program execution.
   * Kubernetes dashboard or kubectl outputs for relevant labs.
   * HTTP responses for web services (e.g., using curl or Postman).
3. **Compress and Submit**