**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

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

Topic: Remote Procedure Calls(RPC)

**Objective**: The goal of this lab is to familiarize you with RPC in Go and to help you build a simple distributed system

## **What is RPC (Remote Procedure Call)?**

RPC allows a program to call functions or procedures on a remote machine as if they were local. In a distributed system, RPC hides the underlying network communication, making it easier to interact with remote services.

### **Key Features of RPC in Golang:**

* Synchronous Calls: Clients wait for the server to respond to a method call.
* Remote Method Invocation: Methods are invoked on remote servers as if they were local.
* Built-in RPC Support: Go’s standard library (net/rpc) makes it easy to implement RPC.

**Activity 1: Build a Simple RPC Server and Client**

**Objective:** Create a basic RPC server and client where the server provides a Multiply method that the client can invoke remotely.

**Step-by-Step Instructions:**

**Server Code( rpc\_server.go)**

package main

import (

"errors"

"fmt"

"net"

"net/rpc"

)

// Args holds the arguments for multiplication

type Args struct {

A, B int

}

// Calculator provides methods for arithmetic operations

type Calculator int

// Multiply multiplies two integers and returns the result

func (c \*Calculator) Multiply(args \*Args, reply \*int) error {

if args.A == 0 || args.B == 0 {

return errors.New("multiplication by zero is not allowed")

}

\*reply = args.A \* args.B

return nil

}

func main() {

// Register the Calculator service

calc := new(Calculator)

rpc.Register(calc)

// Start listening for incoming RPC connections

listener, err := net.Listen("tcp", ":1234")

if err != nil {

fmt.Println("Error starting RPC server:", err)

return

}

fmt.Println("RPC server is listening on port 1234...")

rpc.Accept(listener) // Block and serve clients

}

**Client Code (rpc\_client.go)**

package main

import (

"fmt"

"log"

"net/rpc"

)

// Args holds the arguments for multiplication

type Args struct {

A, B int

}

func main() {

// Connect to the RPC server

client, err := rpc.Dial("tcp", "localhost:1234")

if err != nil {

log.Fatal("Error connecting to RPC server:", err)

}

// Prepare the arguments and call the Multiply method

args := Args{A: 3, B: 5}

var reply int

err = client.Call("Calculator.Multiply", &args, &reply)

if err != nil {

log.Fatal("Error calling RPC:", err)

}

fmt.Printf("Result of %d \* %d = %d\n", args.A, args.B, reply)

}

**Instructions for Testing**:

1. Open one terminal to start the server

go run rpc\_server.go

1. Open another terminal to run the client

go run rpc\_client.go

1. Expected Output

Result of 3 \* 5 = 15

**Activity 2: Concurrent RPC Calculator with Multiple Clients**

**Objective:**

Modify the server to handle concurrent clients using goroutines. The server will provide multiple arithmetic operations (add, subtract, divide).

**Modify Server Code**

* Add More Methods
  + Add
  + Substract
  + Divide

**Modify Client Code to call different RPC Methods**

* + Add
  + Substract
  + Divide

## **Activity 3: Handle Errors and Timeouts in RPC**

### **Objective**:

1. Handle RPC Errors Gracefully.
2. Implement timeouts for long-running RPC calls to avoid blocking clients.

### **Client Code with Timeout Handling**:

package main

import (

"fmt"

"log"

"net/rpc"

"time"

)

// Args holds the arguments for arithmetic operations

type Args struct {

A, B int

}

func main() {

client, err := rpc.Dial("tcp", "localhost:1234")

if err != nil {

log.Fatal("Error connecting to RPC server:", err)

}

args := Args{A: 10, B: 0} // Division by zero to trigger an error

var reply int

// Call RPC method with a timeout

call := client.Go("Calculator.Divide", &args, &reply, nil)

select {

case <-call.Done:

if call.Error != nil {

log.Println("RPC error:", call.Error)

} else {

fmt.Printf("Result: %d\n", reply)

}

case <-time.After(2 \* time.Second):

log.Println("RPC call timed out")

}

}

## **Activity 4: Exercise – Extending the RPC Calculator for Persistent State**

### **Objective**:

In this exercise, students will:

1. Extend the calculator to maintain state (e.g., store the result of the last operation).
2. Implement a method to retrieve the last result from the server.

Exercise Steps:

1. Add a stateful method to store the last result on the server.

type Calculator struct {

lastResult int

mu sync.Mutex

}

func (c \*Calculator) GetLastResult(args \*Args, reply \*int) error {

c.mu.Lock()

defer c.mu.Unlock()

\*reply = c.lastResult

return nil

}

2. Modify the client to call GetLastResult after performing operations.

## **Submission Requirements:**

1. Code Submission:
   * Submit both server and client code for Activity 4.
2. Testing Evidence:
   * Provide screenshots showing multiple clients interacting with the server.
3. Reflection Report:
   * Discuss how RPC simplifies communication compared to socket programming.
   * Describe challenges encountered (e.g., handling timeouts, concurrent clients).