**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

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

Topic: Socket Programming

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

**What is Socket Programming?**

Socket programming is a way to enable communication between devices over a network using standard communication protocols like TCP/IP or UDP. In distributed systems, socket programming is used to enable inter-process communication, data sharing, and real-time coordination between different components of the system.

**How Go Supports Socket Programming**:

Go's net package provides easy-to-use APIs for both **TCP** (Transmission Control Protocol) and **UDP** (User Datagram Protocol) communication. Go's concurrency model (goroutines) makes it perfect for handling multiple client connections simultaneously, which is essential in building distributed systems.

* **TCP** is a reliable, connection-oriented protocol used for data transmission between clients and servers.
* **UDP** is a connectionless protocol, often used for broadcasting or when speed is more critical than reliability.

**Activity 1: Simple Client-Server Communication using TCP**

**Objective:** You will create a simple client-server application using TCP sockets in Go. The server will listen for incoming connections, and clients will send a message to the server, which echoes the message back.

**Step-by-Step Instructions:**

1. **Server**:
   * The server will listen on a specific port and accept incoming client connections.
   * When a client sends a message, the server will read the message, print it, and send a response back to the client.
2. **Client**:
   * The client will connect to the server, send a message, and wait for the response from the server.

**Server Code:**

package main

import (

"bufio"

"fmt"

"net"

"os"

)

// Simple TCP server that listens for incoming connections and responds to messages

func main() {

// Start the server and listen on port 8080

listener, err := net.Listen("tcp", ":8080")

if err != nil {

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

return

}

defer listener.Close()

fmt.Println("Server is listening on port 8080...")

for {

// Accept a client connection

conn, err := listener.Accept()

if err != nil {

fmt.Println("Error accepting connection:", err)

continue

}

// Handle the client connection in a new goroutine (allowing multiple clients)

go handleClient(conn)

}

}

func handleClient(conn net.Conn) {

defer conn.Close()

// Read message from client

message, \_ := bufio.NewReader(conn).ReadString('\n')

fmt.Print("Message received:", string(message))

// Send response back to client

conn.Write([]byte("Message received: " + message))

}

**Instructions for Testing**:

1. Run the server program in one terminal window:

go run server.go

1. Open another terminal window and run the client program:

go run client.go

1. Enter a message in the client terminal and observe how the server responds.

**Activity 2: Creating a Multi-Client Chat Application**

**Objective:**

Build a simple multi-client chat application where multiple clients can send messages to the server, and the server broadcasts the messages to all connected clients. This helps simulate a distributed system where nodes (clients) communicate via a central server.

**Step-by-Step Instructions:**

1. **Server**:
   * The server will maintain a list of connected clients and broadcast messages received from one client to all other clients.
2. **Client**:
   * Each client will connect to the server and be able to send and receive messages from other clients via the server.

**Server Code (Multi-Client Chat):**

package main

import (

"bufio"

"fmt"

"net"

"sync"

)

// Slice to hold all connected clients

var clients = make(map[net.Conn]bool)

var mu sync.Mutex // Mutex to ensure safe concurrent access to client list

func main() {

listener, err := net.Listen("tcp", ":8080")

if err != nil {

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

return

}

defer listener.Close()

fmt.Println("Chat server is listening on port 8080...")

for {

conn, err := listener.Accept()

if err != nil {

fmt.Println("Error accepting connection:", err)

continue

}

// Add new client to client list

mu.Lock()

clients[conn] = true

mu.Unlock()

// Handle each client in a separate goroutine

go handleClient(conn)

}

}

func handleClient(conn net.Conn) {

defer func() {

mu.Lock()

delete(clients, conn) // Remove client from list when they disconnect

mu.Unlock()

conn.Close()

}()

reader := bufio.NewReader(conn)

for {

// Read message from client

message, err := reader.ReadString('\n')

if err != nil {

fmt.Println("Client disconnected:", err)

return

}

fmt.Printf("Broadcasting message: %s", message)

// Broadcast message to all connected clients

broadcastMessage(message, conn)

}

}

// Send the message to all clients except the sender

func broadcastMessage(message string, sender net.Conn) {

mu.Lock()

defer mu.Unlock()

for client := range clients {

if client != sender {

client.Write([]byte(message))

}

}

}

**Client Code (Multi-Client Chat)**:

package main

import (

"bufio"

"fmt"

"net"

"os"

)

func main() {

conn, err := net.Dial("tcp", "localhost:8080")

if err != nil {

fmt.Println("Error connecting to server:", err)

return

}

defer conn.Close()

// Start a goroutine to listen for incoming messages

go receiveMessages(conn)

// Read messages from stdin and send to server

for {

message, \_ := bufio.NewReader(os.Stdin).ReadString('\n')

fmt.Fprintf(conn, message)

}

}

// Function to receive messages from the server

func receiveMessages(conn net.Conn) {

for {

message, \_ := bufio.NewReader(conn).ReadString('\n')

fmt.Print("Message from server: ", message)

}

}

**Instructions for Testing:**

1. Run the server program.
2. Open multiple terminal windows and run the client program in each.
3. Type a message in one client, and observe how it gets broadcasted to all other connected clients.

**Lab Activity 3: Distributed Task Processing**

**Objective:**

Build a simple distributed task processing system where the server assigns tasks to clients. Clients perform tasks and send the results back to the server. This models a distributed processing system.

**Step-by-Step Instructions:**

1. **Server**:
   * The server will generate tasks and send them to clients.
   * After a client completes a task, it sends the result back to the server.
2. **Client**:
   * Each client will receive a task, perform it (e.g., squaring a number), and send the result back to the server.

**Server Code (Task Processing)**:

package main

import (

"bufio"

"fmt"

"net"

"strconv"

"sync"

"time"

)

// List to hold active clients

var clients = make(map[net.Conn]bool)

var mu sync.Mutex

func main() {

listener, err := net.Listen("tcp", ":8080")

if err != nil {

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

return

}

defer listener.Close()

fmt.Println("Server is ready to assign tasks...")

for {

conn, err := listener.Accept()

if err != nil {

fmt.Println("Error accepting connection:", err)

continue

}

mu.Lock()

clients[conn] = true

mu.Unlock()

// Handle each client connection

go handleClient(conn)

}

}

func handleClient(conn net.Conn) {

defer func() {

mu.Lock()

delete(clients, conn)

mu.Unlock()

conn.Close()

}()

for {

// Generate a random number as a task

task := time.Now().Unix() % 100

// Send task to client

fmt.Fprintf(conn, "%d\n", task)

// Receive result from client

response, \_ := bufio.NewReader(conn).ReadString('\n')

fmt.Println("Received result from client:", response)

time.Sleep(5 \* time.Second) // Simulate task intervals

}

}

**Client Code (Task Processing)**:

package main

import (

"bufio"

"fmt"

"net"

"strconv"

"strings"

)

func main() {

conn, err := net.Dial("tcp", "localhost:8080")

if err != nil {

fmt.Println("Error connecting to server:", err)

return

}

defer conn.Close()

for {

// Receive task (number) from server

task, \_ := bufio.NewReader(conn).ReadString('\n')

task = strings.TrimSpace(task)

// Perform task (square the number)

num, \_ := strconv.Atoi(task)

result := num \* num

// Send result back to server

fmt.Fprintf(conn, "%d\n", result)

}

}

**Instructions for Testing:**

1. Run the server.
2. Run one or more client instances.
3. Observe how the server assigns tasks (random numbers) to clients and receives the squared results.