**Addis Ababa Institute of Technology**

**Distributed Systems Lab 3: Answer to Reflection Questions**

**By: Nathnael Getachew**

**Id: UGR/8932/13  
  
1: Explain how the TCP connection is established between the client and server. How does the server handle incoming connections?**

**Answer:**

**Part (I)**

The TCP connection is established between the client and server in the 3 steps as Follows:

1. **Client Initiation**: The client calls `net.Dial("tcp", "localhost:8080")`, which attempts to establish a TCP connection to the server running on `localhost` at port `8080`. If the connection is successful, a `net.Conn` object is returned, which represents the connection to the server.

2. **Error Handling**: If there is an error during the connection attempt, the client prints an error message and exits.

3. **Data Transmission**: Once the connection is established, the client can send data to the server using the `conn.Write` method, and it can also read responses from the server using `bufio.NewReader(conn).ReadString('\n')`.

**Part (II)**

The server handles incoming connections as follows:

1**. Listening for Connections**: The server starts by calling `net.Listen("tcp", ":8080")`, which binds to port `8080` and begins listening for incoming TCP connections. If successful, it will accept connections from clients.

2. **Accepting Connections**: Inside an infinite loop, the server calls `listener.Accept()`, which blocks until a client connects. When a connection is accepted, it returns a `net.Conn` object representing the connection to the client.

3. **Concurrent Handling**: To handle multiple clients simultaneously, the server spawns a new goroutine for each accepted connection by calling `go handleClient(conn)`. This allows the server to continue accepting new connections while processing existing ones.

4. **Processing Client Messages**: The `handleClient` function reads the message sent by the client using `bufio.NewReader(conn).ReadString('\n')`. After processing the message (in this case, simply printing it), the server sends a response back to the client using `conn.Write`.

**2: What challenge does the server face when handling multiple clients, and how does Go’s concurrency model help solve this problem?**

**Answer:**

**Part (I)**

When a server handles multiple clients, it faces several challenges, including:

1. **Concurrency**
2. **Resource Management**
3. **Message Broadcasting**
4. **Client Disconnection**

**Part (II)**

Go's concurrency model, primarily based on goroutines and channels, provides effective solutions to these challenges:

1. **Goroutines**: Each client connection is handled in a separate goroutine. When a new client connects, the server spawns a new goroutine to handle that client's communication, allowing the main loop to continue accepting new connections.
2. Mutex ensures safe concurrent access to shared resources, such as the clients map that holds all connected clients. By locking the mutex when modifying the client list, the server prevents race conditions that could lead to inconsistent states or crashes.
3. **Buffered I/O**: The bufio package is used for efficient reading and writing of messages. This helps in managing input and output operations without blocking the server unnecessarily, allowing for smoother communication between clients and the server.
4. **Broadcasting Messages**: The broadcastMessage function iterates over all connected clients and sends messages to each one except the sender.This is done within a locked section to ensure that the client list remains consistent while messages are being sent .

**3: How does the server assign tasks to the clients? What real-world distributed systems scenario does this model resemble?**

**Answer:**

**Part (I)**

Tasks are assigned to clients as follows:

1. **Listening for Connections**: The server listens for incoming TCP connections on port 8080. When a client connects, the server accepts the connection and adds the client to a list of active clients.
2. **Handling Client Connections**: Each client connection is handled in a separate goroutine using the handleClient function. This allows the server to manage multiple clients concurrently.
3. **Task Generation**: Inside the handleClient function, the server generates a task by calculating a random number based on the current Unix timestamp modulo 100. This number serves as the task that the client needs to process.
4. **Sending Tasks**: The server sends the generated task to the client using `fmt.Fprintf(conn, "%d\n", task)`. This sends the task as a string followed by a newline character.
5. **Receiving Results**: After sending the task, the server waits for the client to send back the result of the task. The server reads the response using `bufio.NewReader(conn).ReadString('\n')` and prints it to the console.
6. **Task Interval**: The server simulates task intervals by sleeping for 5 seconds before generating and sending the next task. This allows for a controlled flow of tasks to the clients.

**Part (II)**

**Real-World Distributed Systems Scenario**

This model resembles a **distributed task processing system**, which is commonly used in various real-world applications