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 between a client and server can be seen through a three-step process:

1. **Client Initiation**: The client initiates the connection by calling `net.Dial("tcp", "localhost:8080")`. If successful, it receives a `net.Conn` object.
2. **Error Handling**: If the connection fails, the client prints an error message and exits.
3. **Data Transmission**: Once connected, the client can use `conn.Write` to send data and `bufio.NewReader(conn).ReadString('\n')` to read responses from the server.

Part (II):
The server manages incoming connections as follows:

1. **Listening for Connections**: The server binds to port 8080 using `net.Listen("tcp", ":8080")` and starts listening for connections.
2. **Accepting Connections**: Inside an infinite loop, the server uses `listener.Accept()` to block until a client connects, returning a `net.Conn` object.
3. **Concurrent Handling**: For each accepted connection, the server spawns a new goroutine with `go handleClient(conn)` to allow continued acceptance of new connections.
4. **Processing Client Messages**: The handleClient function reads messages from the client and responds 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 managing multiple clients, a server faces challenges such as:

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

Part (II)
Go's concurrency model addresses these challenges effectively:

1. **Goroutines**: Each client is handled in a separate goroutine, enabling the main loop to accept new connections without delay.
2. **Mutex**: A mutex ensures safe concurrent access to shared resources, preventing race conditions when modifying client lists.
3. **Buffered I/O**: The bufio package allows efficient reading and writing, improving communication without blocking the server.
4. **Broadcasting Messages**: The broadcastMessage function sends messages to all connected clients while maintaining consistency through locking.

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 on port 8080 and adds connected clients to an active list.
2. **Handling Client Connections**: Each connection is processed in a separate goroutine via the handleClient function.
3. **Task Generation**: A task is generated based on a random number derived from the current Unix timestamp modulo 100.
4. **Sending Tasks**: The server sends the task to the client using `fmt.Fprintf(conn, "%d\n", task)`.
5. **Receiving Results**: After receiving the task result from the client, the server prints it using `bufio.NewReader(conn).ReadString('\n')`.
6. **Task Interval**: The server simulates task intervals by pausing for 5 seconds before sending the next task.

Part (II)
This model resembles a **distributed task processing system**, commonly used in various real-world applications, facilitating efficient handling and processing of tasks across multiple clients. Or like **Grid Computing** (Scientific computing projects) where complex computations are broken down into smaller tasks and distributed to multiple computers for parallel processing.