ITMD:469/569 - Final Project: Creating a Concurrent TCP Client and Server that use TLS

Travis Smith

Project Summary

This final project represents the culmination of our class. You have two options to conclude this course: complete this project (which provides an opportunity for extra credit), or take the 20-question final exam along with a 1-page write-up posted on Blackboard.

The project's requirements are outlined below, designed as a guiding framework. If you wish to adopt a less conventional approach, feel free to do so, ensuring you meet all the requirements. For graduate students, a mandatory requirement is the implementation of message forwarding; when a client dispatches a message to the server, it should be forwarded to all connected clients.

Your grade will be determined by the successful execution of this functionality, the correct application of TLS, effective error handling, and adherence to best practices in Go programming. All aspects of this project are covered by in-class examples, so use them to your benefit. Check the project rubric on Blackboard for grading criteria.

Project Requirements

Your task involves creating two programs: a TCP server and a TCP client. Both programs should be coded in separate Go files.

Your TCP server should:

- Load a pem-encoded server certificate and a server private key from a file. This can be created with OpenSSL or a Go program you develop.
- Initiate a TLS listening connection over TCP using the loaded certificate and key. This can be done by creating a tls.Config.
- Continuously listen for incoming client connections.
- Upon accepting a connection, handle it concurrently in a separate function. Read the accepted connection contents and echo them back to the client.
- For graduate students, whenever a client connects, save the connection. Also, whenever a client sends a message, forward that message to all clients.

Your TCP client should:

- Load a pem-encoded client certificate and a client private key from a file.
- Establish a TLS connection to the server using the certificate and the key. This should be done with a tls.Config, similar to the server.
- Send a message to the server.
- Read the server's reply and print it to the console.
- For an additional 5 percent extra credit, read user input from the terminal as the messages to send to the server, as though this was a chat client.
- For graduate students, given that the server is saving every client connection, ensure your client remains open to keep accepting connections to the server. Implement this functionality.

Part 1: Creating the Certificates

You have two options to create the client and server certificates. You can either use OpenSSL to create the pem certificates and the keys for both the client and server and then load them into your server and client using tls.LoadX509KeyPair(), or you can create them within a Go program. Week 7 examples include a bash script that creates the server pem, server key, client pem, and client key for use in the server and client. If you opt to create the certificates via Golang, you can earn 20 percent extra credit.

Part 2: Building the TCP Server

This is a conventional TCP server that is secured with TLS. Steps to start the server connection using the server certificate and key are provided in the requirements section. Ensure to manage client connections concurrently and store and access client connections safely when concurrent access is needed (for instance, Mutex locks can be used).

Part 3: Creating the TCP Client

Similar to the server, this is a standard TCP client that connects to a server securely using TLS. The necessary steps are outlined in the requirements section. Consider reading in messages from the command line for extra credit, simulating a chat client scenario.

Submission

Please compress your source code into a ZIP file and upload it to Blackboard. Additionally, attach a separate screenshot demonstrating your code running successfully.