Christopher Bussen
101657219
bussenc1@udayton.edu
CPS 475 - Secure Application Development
Dr. Phu Phung
Assignment 1 - A Secure and Robust Chat System in GoLang and Node.js

1. **Introduction**

   Bitbucket URL: https://bitbucket.org/bussenc1/secad-bussenc1/src/master/assignment1/

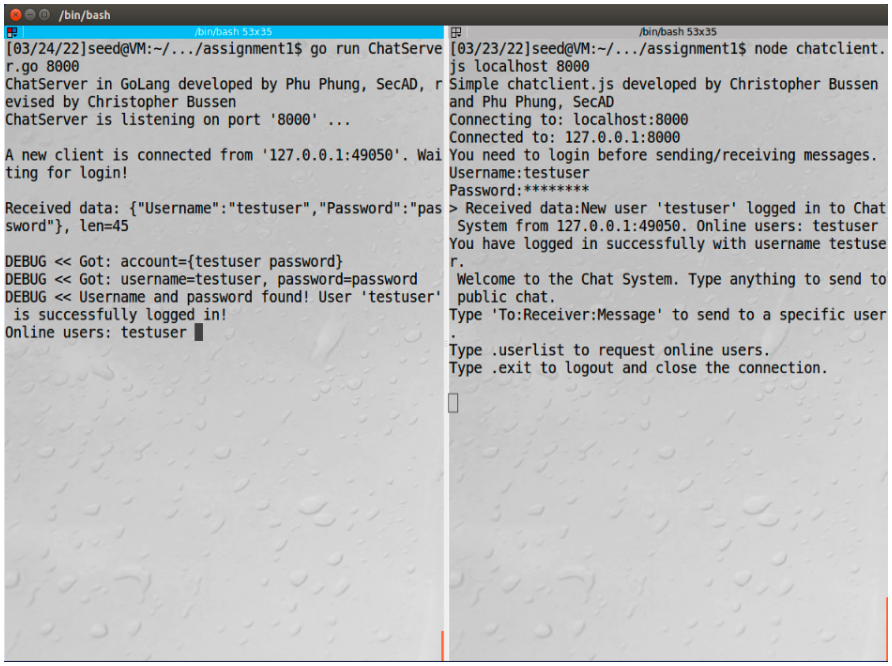
   In this assignment, I expanded on the BroadcastEchoServer.go program as well as the telnet.js program from lab 3. These programs were renamed to ChatServer.go and chatclient.js, and functionality was added to create a public and private chat system. In order to access either chat system, a user must log in to the server using a hard-coded username and password (which are established in the chat server program). As soon as a user connects to the server, they are prompted to enter a username and password. If the user enters incorrect login information, they are able to continue to attempt to login. Upon logging in, the client will see a printed menu with the different actions they can now perform. These actions include closing the connection, getting a list of the current logged in users, sending a private chat by specifying a receiver and a message, or sending a public chat to all of the logged in users by simply typing their message and hitting return. Additionally, upon a new user logging in, all logged in users will receive a message saying that a new user has logged into the system and will see the current list of logged in users. Finally, once a user exits the system, they are then removed from the list of logged in users.

2. **Design**

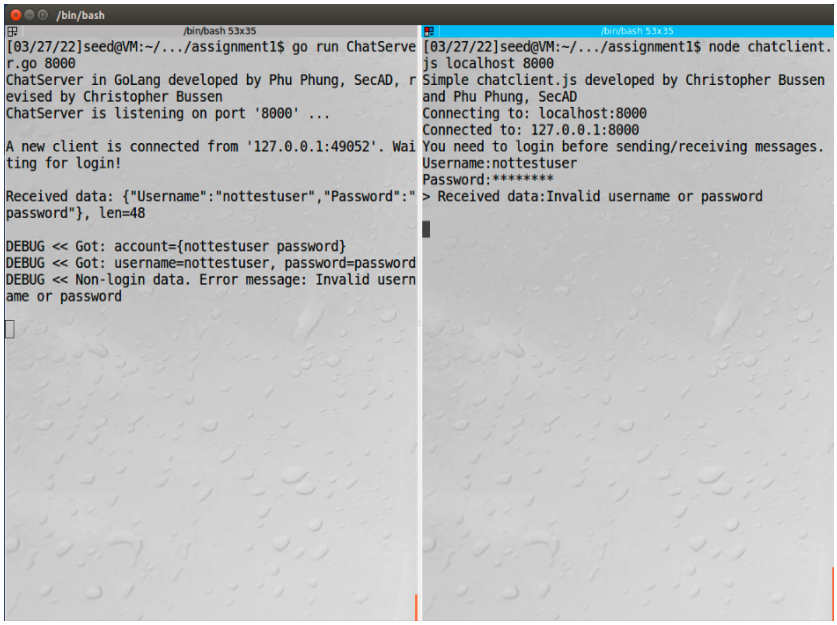
   When a user first connects to the server via the chat client, the client program will prompt the user to login. After the user enters a username and password, the chat client will check whether or not the input is valid (if it meets certain length requirements). If not, the user will be prompted to try to login again. If the input is validated, the chat client will format the username and password in the server's expected format ({"Username":"…", "Password":"..."}), and it will send this data to the chat server using client.write(login), where login is the formatted login data. At this point, the chat server will receive the data and parse through it for the username and password. After finding them, it will check if the username and password are equal to one of the sets of hard-coded login information provided in the checkaccount function. If they are equal, the checklogin function returns true and the user will be logged in within the login function. At this point, the server sends a message to all logged in users alerting them of a new login as well as sending a menu of options to the new user. The server does this by using the sendto function which utilizes the Write method to send this data to the chat client, which then prints the information on the client side. If the username and password are not equal to a hard-coded username and password, then the checklogin function will return false and the user will not be logged in (but may try to login once again). Once a user is logged in,

the goroutine continuously waits for new input from the user. In the chat client, the key function is used to allow the client to provide input within the terminal. The chat client once again uses client.write(input), where input is whatever the user typed, to send the data back to the chat server. The goroutine then uses the Read method to go through the data that is received from the chat client and store it in a variable. The server then converts this data into a string and checks the input to see what the client's intentions are (based on the options menu that was previously sent to them providing action prompts after logging in). If the client requests the user list, then the sendto function (and the Write method) is used to send the list of logged in users to only the client that requested it. If the client indicates that they want to send a private message, then the sendto function and Write method are used to send the client's message to the other user specified by the original client. Finally, if the client does not indicate any particular action, it is assumed that the input should be used as a public message, and a for loop is used with the sendto function and Write method to send the message to all logged in users. Additionally, within the call to the sendto function, the server will convert the user input back from a string to bytes in order to send the data to the client. Whenever a chat client receives data from the server, it uses client.on to print the received data to the client side so that the client can see it. These programs are also able to avoid data races as well as other security issues. One of these reasons is because the server synchronizes access to shared data. It does so by using channels in Go, which help goroutines prevent data races when multiple clients are connected to the server by ensuring that only one goroutine will access a shared resource at any given moment. Additionally, other security issues such as format string vulnerabilities are avoided by using more secure methods and formatting the strings as constants using arguments such as "%s". The authentication process also helps to limit security issues because only those with valid login information are able to access the main privileges of the programs. Finally, other security issues are avoided through the use of input validation throughout the programs. One place where this can be seen is in the client program within the inputValidated function for the login data length. For all of these reasons, the server and client programs are able to avoid data races and security issues.

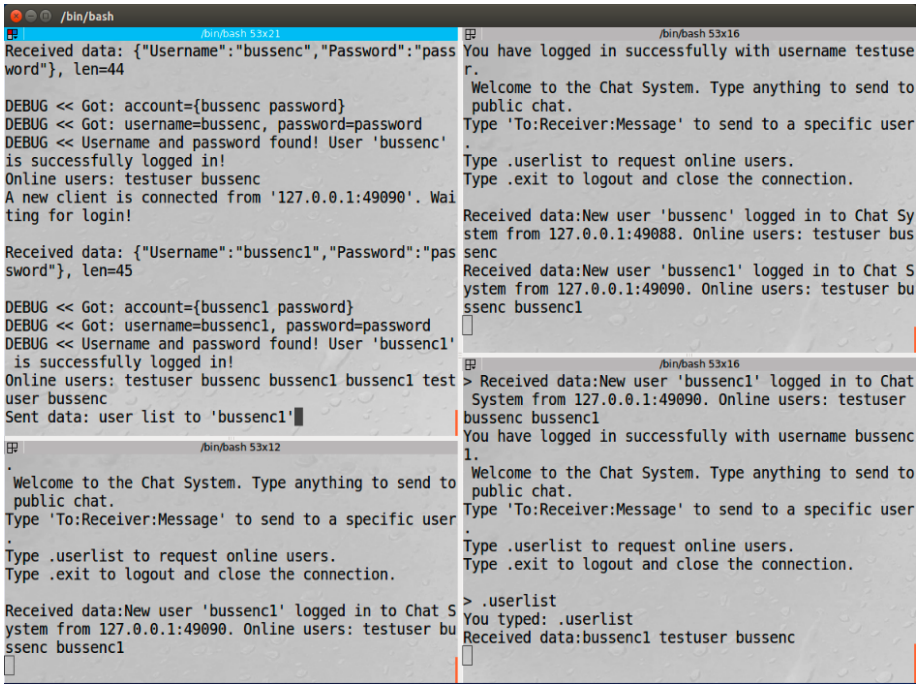
3. Test cases and Demo



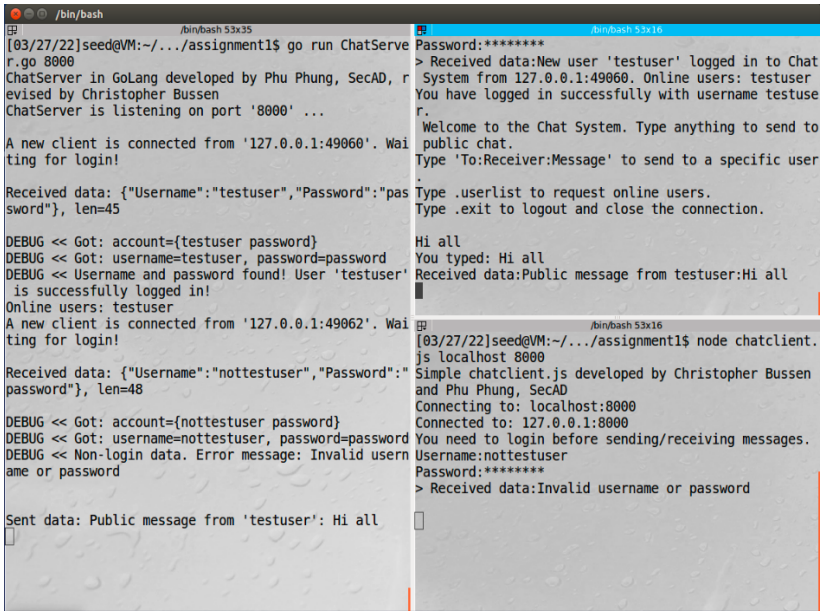
Test case 1 - this screenshot shows a user entering a correct username and password (which is masked) and logging into the chat system.



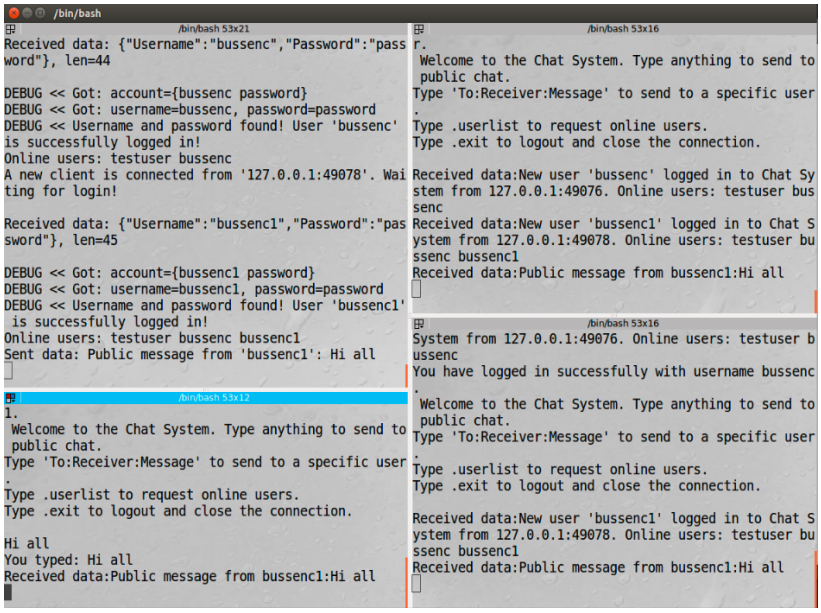
Test case 2 - this screenshot shows a user entering incorrect account credentials and thus being unable to login.



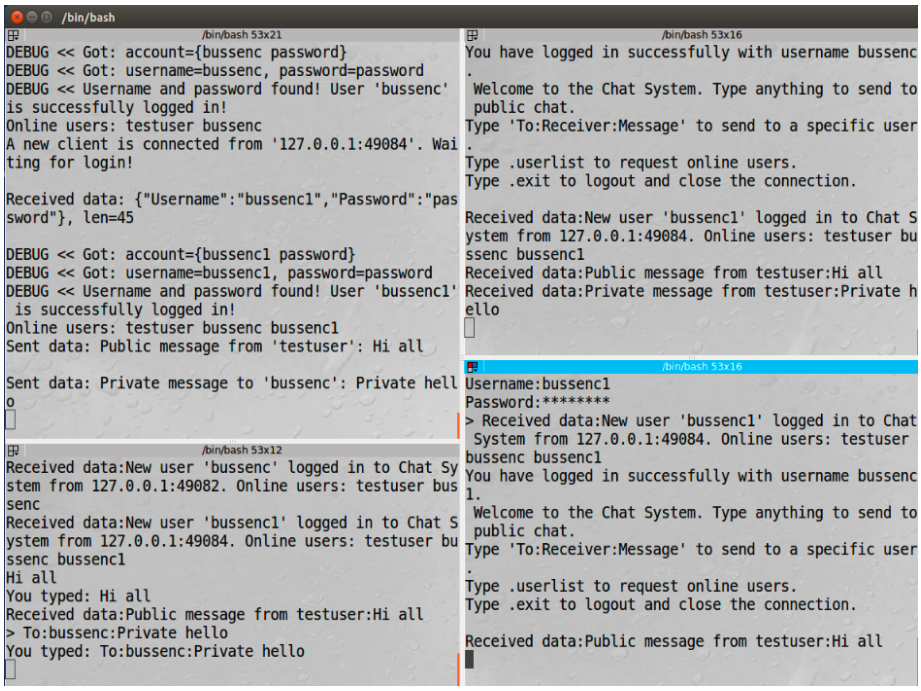
Test case 3 - this screenshot shows the user's ability to get a list of the logged in users using the command printed in the login menu.



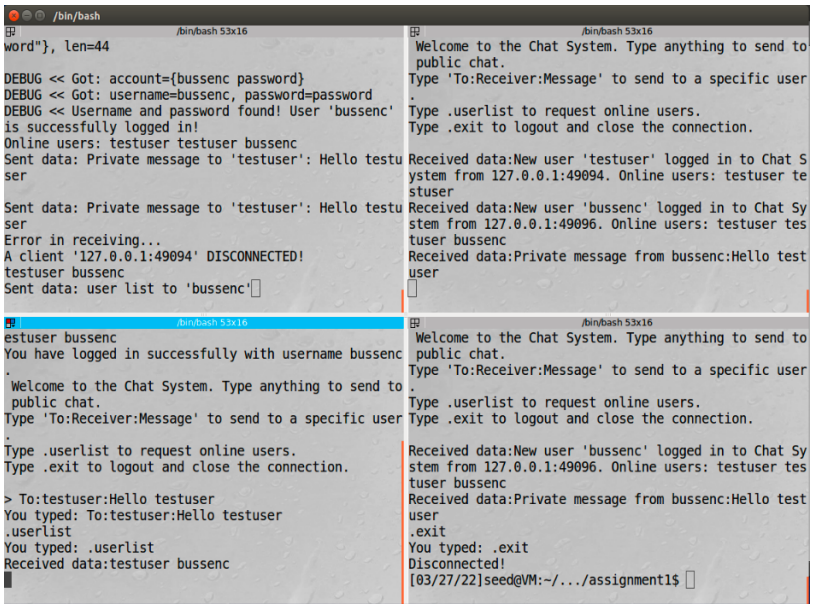
Test case 4 - this screenshot shows a logged in user in the top right able to send and receive public chats, but the non-logged in user (bottom right) is unable to send/receive public chats.



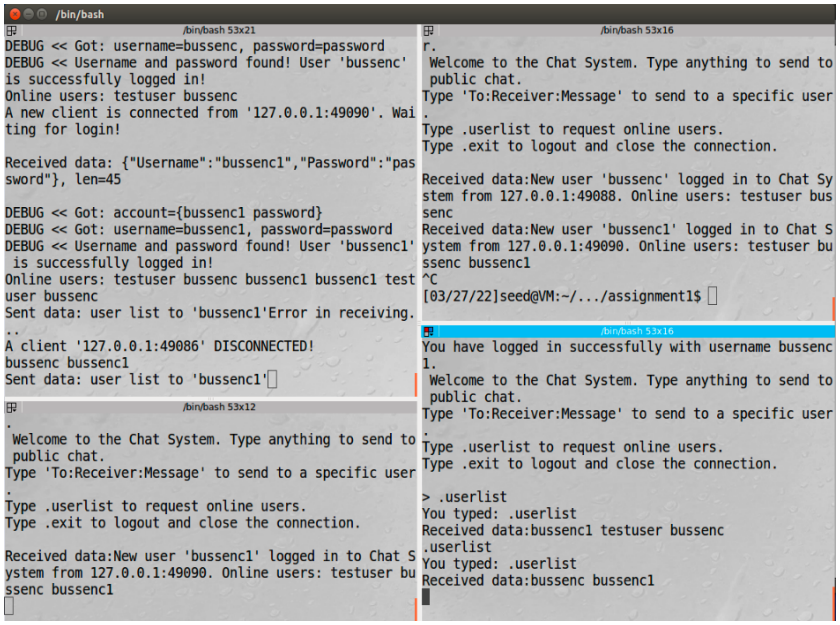
Test case 5 - this screenshot shows that when a new user is logged into the chat system, all clients receive a message that has an updated user list. Additionally, it displays how public chats are sent to all logged in users.



Test case 6 - this screenshot shows that private chats are sent only to the intended user, not to all logged in users.



Test case 7 - this screenshot shows how a user can login from two different terminals and can still receive private messages. Additionally, if one terminal connection is closed, the same user is still in the user list as they have another terminal connection still open.



Test case 8 - this screenshot shows how the user list is updated after a logged in user logs out and closes their connection.

Appendix

ChatServer.go Code:

```
/* Simple ChatServer in GoLang by Phu Phung, customized by Christopher
```

```
Bussen for ChatServer in SecAD*/
package main

import (
	"fmt"
	"net"
	"os"
	"strings"
	"encoding/json"
)

const BUFFERSIZE int = 1024
var AllClient_conns = make(map[net.Conn]string) // global
var newclient = make(chan net.Conn)
var lostclient = make(chan net.Conn)
var allLoggedIn_conns = make(map[net.Conn]interface{})
type User struct{
	Username string
	Login bool
	Key string //to store the key
}

func main() {
	if len(os.Args) != 2 {
		fmt.Printf("Usage: %s <port>\n", os.Args[0])
		os.Exit(0)
	}
	port := os.Args[1]
	if len(port) > 5 {
		fmt.Println("Invalid port value. Try again!")
		os.Exit(1)
	}
	server, err := net.Listen("tcp", ":"+port)
	if err != nil {
		fmt.Printf("Cannot listen on port '" + port + "'!\n")
		os.Exit(2)
	}
	fmt.Println("ChatServer in GoLang developed by Phu Phung, SecAD,
revised by Christopher Bussen")
	fmt.Printf("ChatServer is listening on port '%s' ...\n", port)
	go func(){
		for{
			client_conn, _ := server.Accept()
```

```
                  newclient <- client_conn
            }
      }()
      for{
            select{
                  case client_conn := <- newclient:
                        AllClient_conns[client_conn] =
client_conn.RemoteAddr().String()
                        welcomemessage :=fmt.Sprintf("\nA new client is
connected from '%s'. Waiting for login!\n",
                              client_conn.RemoteAddr().String())
                        fmt.Println(welcomemessage)
                        // if(login(client_conn)){
                        go client_goroutine(client_conn)
                  case client_conn := <- lostclient:
                        delete(AllClient_conns,client_conn)
                        byemessage := fmt.Sprintf("A client '%s'
DISCONNECTED!", client_conn.RemoteAddr().String())
                        fmt.Println(byemessage)
            }
      }
}

func client_goroutine(client_conn net.Conn){
      var buffer [BUFFERSIZE]byte
      // loggedin := login(client_conn)
      // fmt.Printf("loggedin: %s", loggedin)
      login(client_conn)
      user := allLoggedIn_conns[client_conn].(User)

      for {
            byte_received, read_err := client_conn.Read(buffer[0:])
            if read_err != nil {
                  lostclient <- client_conn
                  fmt.Println("Error in receiving...")
                  return
            }

            //userlist functionality
            bufferString := string(buffer[0:byte_received])
            userInput := strings.Split(bufferString, ":")
            if(strings.Compare(bufferString,".userlist")==0){
                  userlist := userList(client_conn)
```

```
                users := strings.Join(userlist, " ")
                fmt.Printf("\nSent data: user list to '%s'",
user.Username)
                sendto(client_conn, []byte(users))
          }else if(strings.Compare(userInput[0], "To")==0){  //private
chat functionality
                      for client_conn, _ := range allLoggedIn_conns{
                            currentuser :=
allLoggedIn_conns[client_conn].(User)
                            if(strings.Compare(userInput[1],
currentuser.Username)==0){
                                  fmt.Printf("\nSent data: Private
message to '%s': %s\n", userInput[1],userInput[2])
                                  sendto(client_conn, []byte("Private
message from " + user.Username + ":" + userInput[2]))
                            }
                      }
                      // sendto(client_conn, []byte("Invalid username"))
          }else{ //public chat to all logged in users
                for client_conn, _ := range allLoggedIn_conns{
                      sendto(client_conn, []byte("Public message from " +
user.Username + ":" + bufferString))
                }
                fmt.Printf("\nSent data: Public message from '%s': %s\n",
user.Username,buffer)
          }
    }
}

func checklogin(data []byte) (bool, string, string){
    //expecting format of {"username":"..","password":".."}
    type Account struct{
          Username string
          Password string
    }
    var account Account
    err := json.Unmarshal(data, &account)
    // fmt.Printf("Received login data: %s", data)
    // fmt.Printf(account.Username)
    if err!=nil || account.Username == "" || account.Password == "" {
          fmt.Printf("JSON parsing error: %s\n", err)
          return false, "", `[BAD LOGIN] Expected:
{"Username":"..","Password":".."}`
```

```
	}
	fmt.Printf("\nDEBUG << Got: account=%s\n", account)
	fmt.Printf("DEBUG << Got: username=%s, password=%s\n",
account.Username,account.Password)

	if checkaccount(account.Username,account.Password) {
		return true, account.Username, "logged"
	}

	return false, "" , "Invalid username or password\n"
}

func checkaccount(username string, password string) bool {
	if username == "bussenc" && password == "password"{
		return true
	}
	if username == "bussenc1" && password == "password"{
		return true
	}
	if username == "testuser" && password == "password"{
		return true
	}
	return false
}

func login(client_conn net.Conn) bool{
	var buffer [BUFFERSIZE]byte
	byte_received, read_err := client_conn.Read(buffer[0:])
	if read_err != nil{
		fmt.Println("Error in receiving...")
		if _, client_authenticated := AllClient_conns[client_conn];
client_authenticated{
			lostclient <- client_conn
		}else{
			fmt.Println("An unauthenticated client is DISCONNECTED!")
		}
		return false
	}
	logindata := buffer[0:byte_received]
	fmt.Printf("Received data: %s, len=%d\n", logindata, len(logindata))
	authenticated, username, loginmessage := checklogin(logindata)
	if authenticated {
		fmt.Println("DEBUG << Username and password found! User '" +
```

```
username + "' is successfully logged in!")

            currentLoggedUser := User{ Username : username, Login: true}
            allLoggedIn_conns[client_conn] = currentLoggedUser

            fmt.Printf("Online users: ")
            userlist := userList(client_conn)
            users := strings.Join(userlist, " ")

            newLoginMessage := fmt.Sprintf("New user '%s' logged in to Chat
System from %s. Online users: %s",
                  username, client_conn.RemoteAddr().String(), users)

            for client_conn, _ := range allLoggedIn_conns{
                  sendto(client_conn, []byte(newLoginMessage))
            }

            chatWelcomeMessage := fmt.Sprintf("\nYou have logged in
successfully with username %s.\n Welcome to the Chat System. Type anything
to send to public chat.", username)
            sendto(client_conn, []byte(chatWelcomeMessage))
            optionsMessage := fmt.Sprintf("\nType 'To:Receiver:Message' to
send to a specific user.\nType .userlist to request online users.\nType
.exit to logout and close the connection.\n")
            sendto(client_conn, []byte(optionsMessage))

            return true
      }else{
            fmt.Println("DEBUG << Non-login data. Error message: " +
loginmessage)
            sendto(client_conn,[]byte(loginmessage))
            login(client_conn)
      }
      return false
}

func sendto(client_conn net.Conn, data []byte){
      _, write_err := client_conn.Write(data)
            if write_err != nil {
                  fmt.Println("Error in sending... to" +
client_conn.RemoteAddr().String())
                  return
            }
```

```
}

func sendtoAll(data []byte){
	for client_conn, _ := range AllClient_conns{
		_, write_err := client_conn.Write(data)
		if write_err != nil {
			fmt.Println("Error in sending...")
			continue //move on next iteration
		}
	}
	fmt.Printf("Received data: %s Sent to all connected clients!\n",
data)
}

func userList(client_conn net.Conn) []string{
	var userlist []string
	for client_conn, _ := range AllClient_conns{
		user := allLoggedIn_conns[client_conn].(User)
		fmt.Printf("%s ", user.Username)
		userlist=append(userlist, user.Username)
	}
	return userlist
}
```

chatclient.js code:

```
var net = require('net');

if(process.argv.length != 4){
	console.log("Usage: node %s <host> <port>", process.argv[1]);
	process.exit(1);
}

var host=process.argv[2];
var port=process.argv[3];

if(host.length >253 || port.length >5 ){
	console.log("Invalid host or port. Try again!\nUsage: node %s
<port>", process.argv[1]);
	process.exit(1);
}
```

```
var client = new net.Socket();
console.log("Simple chatclient.js developed by Christopher Bussen and Phu
Phung, SecAD");
console.log("Connecting to: %s:%s", host, port);

client.connect(port,host, connected);

function connected(){
      loginsync();
      key();
}

var readlineSync = require('readline-sync');
var username;
var password;
function loginsync(){
      console.log("Connected to: %s:%s", client.remoteAddress,
client.remotePort);
      console.log("You need to login before sending/receiving messages.")
      //wait for user's response
      username = readlineSync.question('Username:');
      if(!inputValidated(username)) {
            console.log("Username must have at least 5 characters. Please
try again!");
            loginsync();
            return;
      }
      //Handle the secret text (e.g. password)
      password = readlineSync.question('Password:', {
            //typed text on screen is hidden by `*`
            hideEchoBack: true
      });
      if(!inputValidated(password)) {
            console.log("Password must have at least 5 characters. Please
try again!");
            loginsync();
            return;
      }
      var login = `{"Username":"${username}","Password":"${password}"}`;
      client.write(login);
}
```

```
function inputValidated(data){
	if(data.length > 4){
		return true;
	}
	else{
		return false;
	}

}

client.on("data", data =>{
	console.log("Received data:" + data);
});
client.on("error", function(err){
	console.log("Error");
	process.exit(2);
});
client.on("close", function(data){
	console.log("Connection has been disconnected");
	process.exit(3);
});

function key(){
	const keyboard = require('readline').createInterface({
		input: process.stdin,
		output: process.stdout
	});
	keyboard.prompt();
	keyboard.on('line', (input) => {
		console.log(`You typed: ${input}`);
		if(input === ".exit"){
			client.destroy();
			console.log("Disconnected!");
			process.exit();
		}
		else
			client.write(input);
	});
}
```