School of Computer Science

CS 4133/5133: Data Networks Fall 2023

Project 1

Name: Ujwala Vasireddy

INTRODUCTION:

The TCP Client-Server Programming project represents a significant milestone in the journey of mastering network communication within the C programming language. Its primary goal is to construct a robust client-server duo proficient in executing uncomplicated password verification to access account details. This undertaking lays a crucial foundation for comprehending the intricate realm of networking applications.

The client program takes the initiative by establishing a connection with the server, transmitting user credentials for validation. Upon a successful verification process, the server promptly imparts pertinent account information to the client. It is crucial to underscore that only meticulously specified combinations of user IDs and passwords are accorded legitimacy. This documentation strives to furnish a comprehensive overview of the project's specifications, complemented by an in-depth exploration of the problem-solving methodology and the intricacies of implementation. Having embarked on this initial programming project, I've acquired firsthand knowledge of TCP iterative client-server interaction in the realm of C programming. Through the practical application of the socket interface, I've honed my skills and gained a solid grasp of the pivotal stages involved in crafting a networking application. This experience has been instrumental in deepening my understanding and knowledge of this crucial aspect of software development.

OBJECTIVE:

This project's central objective is to foster a profound grasp of TCP iterative client-server interactions, with a specific emphasis on leveraging the socket interface within the realm of C programming.

SPECIFICATIONS:

a. **Server Program:**
   The server program plays a pivotal role in this client-server communication. It is responsible for listening to incoming connections, authenticating user credentials, and providing appropriate responses.
   Requirements:
   - Argument Handling: The server program should accept an argument specifying the port it will be listening on. This ensures that multiple students do not request the same port.
   - Socket Initialization: The server initializes a socket for communication using the specified port. It also sets socket options for reusability.
   - Listening for Connections: The server starts listening for incoming connections, akin to waiting for a call.

- Accepting Connections: When a client attempts to connect, the server accepts this incoming connection, creating a dedicated socket for communication with that client.
- User Authentication: The server receives and verifies user credentials sent by the client. It checks whether the user ID and password combination is valid.
- Sending Responses: If authentication is successful, the server sends a success message along with the associated account information. In case of failure, it sends an appropriate failure message.
- Graceful Disconnection: The server handles client requests to disconnect and also manages unexpected client disconnections.
- Timeout Handling: The server implements a timeout mechanism to close the connection if no input is received within a specified time frame.

b. Client Program:

The client program serves as the initiating point of contact in the client-server interaction. It's responsible for establishing a connection with the server, sending user credentials, and processing the server's responses.

Requirements:

- Argument Handling: The client program must accept two arguments: the server's name and the port it intends to connect to. These arguments are crucial for establishing the connection.
- User Input and Interaction: After connecting to the server, the client prompts the user to enter a user ID. This user ID is then transmitted to the server for authentication.
- Sending User Credentials: Once the user ID is obtained, the client forwards it to the server. Upon receiving an acknowledgment from the server, the client prompts the user for the corresponding password.
- Transmitting Password: The client sends the entered password to the server for validation.
- Handling Server Responses: The client processes the server's response, which could be a success message along with account information or a failure message in case of invalid credentials.
- Displaying Account Information: If authentication is successful, the client displays the received account number and balance.
- Closing the Connection: After completing the interaction, the client sends a "QUIT" message to the server to request a graceful disconnection.

PROBLEM-SOLVING APPROACH:

Server Code (vasireddyP1Server.c):

1. Server Initialization:
   - createServerSocket: This function creates a socket, which serves as the communication endpoint. It's akin to setting up a phone line.
   - setSocketOptions: Configures the socket options, allowing the server to reuse the address and port. This enables the server to quickly restart if it's closed and reopened.
   - bindSocket: Associates the socket with an IP address and port number, much like registering a phone number to a physical address.
2. Listening for Connections:
   - listenForConnections: Initiates the server to listen for incoming connections, akin to turning on a phone and waiting for calls.
3. Accepting a Connection:
   - When a client wants to connect, it establishes a connection by creating its own socket.
   - acceptConnection: This function is used to accept the incoming connection, creating a new dedicated socket for communicating with this specific client.
4. Client Interaction:
   - With the connection established, the server and client now have dedicated sockets for communication. They can send and receive messages.
5. Initial Message:
   - The server sends an initial message to the client, prompting them to provide valid credentials. This introduction is like starting a conversation during a phone call.
6. User Authentication:
   - The client provides a username and password. The server reads this information using recv.
   - authenticateUser: This function checks if the provided credentials match any of the predefined users. It's similar to verifying the caller's identity.
   - If authentication is successful, the server sends back a success message along with the user's account information.
7. Continued Communication:
   - If authentication is successful, the client and server can continue to communicate. This could involve performing further tasks or operations.
8. Client Disconnection:
   - If the client is done, it can request to disconnect by sending a "QUIT" message. The server handles this message and terminates the communication.
   - If the client disconnects unexpectedly, the server detects this through recv and closes the connection.
9. Error Handling and Cleanup:
   - The server implements error handling to address unexpected issues. For example, if a function call fails, an error message is printed, and the program exits with an error code.
   - Once the server is done with a client, it cleans up resources by closing the sockets.

Client Code (vasireddyP1Client.c):

1. Client Initialization:
   - The client program creates a socket using the createServerSocket function, similar to how the server does.
   - setSocketOptions allows reusing the address and port.
2. Connecting to the Server:
   - The client uses connect to establish a connection with the server, much like dialing a phone number to initiate a call.
3. User Input and Authentication:
   - The client prompts the user for a username and password.
   - It then sends this information to the server using send.
   - The client code also handles timeout scenarios where no input is received within a specified time.
4. Receiving Server Responses:
   - The client waits to receive messages from the server using recv. This is similar to waiting for the other party to respond during a phone call.
   - It handles different types of messages, such as authentication success, account information, or failure messages.
5. Continued Communication:
   - If authentication is successful, the client and server can continue to communicate. This could involve performing further tasks or operations.
6. User Interaction and Disconnecting:
   - The client can send messages to the server, which the server can receive and respond to.
   - If the client wants to disconnect, it sends a "QUIT" message to the server.
7. Error Handling and Cleanup:
   - Similar to the server, the client implements error handling to address unexpected issues.
   - The client also cleans up resources by closing the socket when it's done.

DATA STRUCTURES AND ALGORITHMS:

In this project, I've strategically chosen data structures and algorithms to ensure efficient handling of user information and secure authentication.

Server Code:

Data Structures:

- struct UserRecord: This structure holds user information, including userId, password, accountNumber, and balance.

Error Handling:

- Consider implementing error handling mechanisms, such as logging or sending error messages to clients in case of failures.

Algorithms:

Authentication Algorithm (authenticateUser function):
Input: username, password, users array, length
Output: Pointer to struct UserRecord or NULL
Algorithm:

- Iterates through the users array using a loop.
- Compares the provided username and password with the stored values.
- If a match is found, returns a pointer to that user's record.
- If no match is found after iterating through all users, returns NULL.

Handling Client Communication (handleClientCommunication function):
Input: newSocket (socket file descriptor for the connected client)
Output: None
Algorithm:

- Initializes variables and data structures for communication.
- Sends an initial message to the client.
- Waits for a specified time period to receive the user id (uses select function for timeout).
- Handles user id input and sends an acknowledgement back to the client.
- Waits for a specified time period to receive the password.
- Handles password input and validates the user.
- If validation is successful, sends user account information to the client.
- Continuously waits for user input (QUIT or other messages) and responds accordingly.

User-Defined Functions:

- struct UserRecord *authenticateUser(const char *username, const char *password, struct UserRecord users[], int length):
  Responsible for authenticating a user based on provided credentials.
  Returns a pointer to the matched user record or NULL if no match is found.
- int createServerSocket():
  Creates and initializes a server socket.
  Returns the socket file descriptor.
- void setSocketOptions(int serverSocket):
  Sets socket options for the server socket, specifically SO_REUSEADDR, allowing the reuse of local addresses.
- void bindSocket(int serverSocket, int port):
  Binds the server socket to a specified port.
- int listenForConnections(int serverSocket):
  Puts the server socket in listening mode, allowing it to accept incoming connections.
- int acceptConnection(int serverSocket):
  Accepts a client connection, returning a new socket file descriptor for communication with the connected client.
- void handleClientCommunication(int newSocket):
  Handles communication with a connected client.

Includes sending and receiving messages, authenticating the user, and handling the quit prompt.

Additional Considerations:
Timeout Handling:
The server uses select to implement a timeout mechanism for user input. This ensures that the server doesn't wait indefinitely for input.

Client Code:

User-Defined Functions:

- int createClientSocket():
  Creates and initializes a client socket.
  Returns the socket file descriptor.
- int connectToServer(int clientSocket, const char *serverIP, int port):
  Establishes a connection to the server with a specified IP address and port.
  Returns the client socket.
- void communicateWithServer(int clientSocket):
  Handles the communication with the server.
  Includes receiving initial messages, sending user id and password, handling server responses, and handling the quit prompt.

ASSUMPTIONS:

In crafting the implementation for this project, I've operated under certain key assumptions to maintain focus and coherence with the project specifications. These assumptions have guided the development process:

- User Interaction Modality:
  I've assumed that user interaction will be conducted through a command-line interface. Both the client and server programs will utilize this interface for input prompts and message displays.
- Defined User Credential Format:
  The assumption is that user IDs and passwords will strictly follow a specified format. In this context, both user IDs and passwords are expected to comprise alphanumeric characters exclusively.
- Authentication Criteria Rigidity:
  It's presupposed that the only valid combinations of user ID-password pairs are those explicitly provided in the project specifications. Any other pairing will be treated as invalid.
- Single Client Connection at a Time:
  The architecture presumes that the server will exclusively handle one client at a time. Once a client establishes a connection, the server will engage in authentication and communication with that specific client until the interaction concludes.
- Synchronous Interaction Dynamics:

The interaction flow between the client and server is envisioned as synchronous. This signifies that each step in the process (e.g., sending user ID, receiving acknowledgment, transmitting password, etc.) is finalized before progressing to the subsequent phase.

- Absence of Concurrent Client Requests:
  I've assumed that multiple clients will not concurrently attempt to connect to the server. The server's design caters to the sequential handling of clients, one at a time.
- Limited Scope of Error Handling:
  While due consideration has been given to error handling, the main emphasis is directed towards the core functionalities of authentication and communication. A comprehensive treatment of all conceivable error scenarios is beyond the current scope.
- Password Security Focus:
  For this project, the primary emphasis is placed on networking functionalities rather than intricate security measures. Therefore, elements such as password encryption and advanced security protocols have not been integrated.

These assumptions have served as guiding principles to ensure a focused and viable implementation within the defined project scope. They remain aligned with the provided project specifications while balancing complexity against achievability.

STEPS AND SCREENSHOTS:

I used the following command for copying the client and server files from my local system to the remote server at the specified destination path.

```
E:\OU Assignment\DN\Project1>scp vasireddyP1Client.c vasi0011@gpel9.cs.nor.ou.edu:/home/vasi0011/programs
vasi0011@gpel9.cs.nor.ou.edu's password:
vasireddyP1Client.c                                           100% 7716   192.5KB/s   00:00
```

```
E:\OU Assignment\DN\Project1>scp vasireddyP1Server.c vasi0011@gpel9.cs.nor.ou.edu:/home/vasi0011/programs
vasi0011@gpel9.cs.nor.ou.edu's password:
vasireddyP1Server.c                                           100%   10KB 192.8KB/s   00:00
```

I used the following command to compile the source code file vasireddyP1Server.c and produced an executable file named vasireddyP1Server that you I run to execute the program.

```
vasi0011@gpel9:~/programs$ gcc vasireddyP1Server.c -o vasireddyP1Server
```

I used the following command to compile the source code file vasireddyP1Client.c and produced an executable file named vasireddyP1Client that I can run to execute the program.

```
vasi0011@gpel9:~/programs$ gcc vasireddyP1Client.c -o vasireddyP1Client
```

This command is starting the server program vasireddyP1Server and configuring it to listen on port 9633 for client connections.

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Server 9633
```

When I run this command, I'm instructing the client program to establish a connection with the server located at the IP address 10.194.8.51, specifically on port 9633. This command sets up a

communication link between my client and the designated server, allowing them to exchange information.

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
```

This indicates that the server is now up and running, eagerly anticipating a client to establish a connection.

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Server 9633
[ + ] Socket successfully created.
[ + ] Socket successfully binded.
[ + ] Server listening...

**********************************
[ + ] Waiting for a client to connect.
```

The client successfully established a connection with the server. Upon connection, the server sent a message instructing the client to join with valid credentials in order to retrieve account details.

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id:
```

The server has accepted the connection request from the client. It acknowledges that a client has successfully connected to it. This means the server and client can now communicate with each other over the established connection.

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
```

Now, let's consider different scenarios:

The interaction between the client and server when the client provides correct user id and password.

Client side: (Output: User Validation Successful)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: davi0027
[ + ] Acknowledgement Received
[ + ] Password: Crc51RqV
[ + ] Server : User Validation Successful

Type `QUIT` to close the connection :
```

Server side: (Output: Received)

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: davi0027
[ + ] Waiting for 30 seconds to receive Password...
[ + ] Received Password: Crc51RqV
```

The interaction when the client provides incorrect user id.

Client side: (Output: User Validation Unsuccessful)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: BHAT0092
[ + ] Acknowledgement Received
[ + ] Password: G6M7p8az
[ + ] Server : User Validation Unsuccessful
Type `QUIT` to close the connection :
```

Server side: (Output: Credentials mismatch)

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: BHAT0092
[ + ] Waiting for 30 seconds to receive Password...
[ + ] Received Password: G6M7p8az
[ - ] Credentials mismatch
```

The interaction when the client provides incorrect password.

Client side: (Output: User Validation Unsuccessful)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: yogi0067
[ + ] Acknowledgement Received
[ + ] Password: fgtyhjuj
[ + ] Server : User Validation Unsuccessful
Type `QUIT` to close the connection : S
```

Server side: (Output: Credentials mismatch)

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: yogi0067
[ + ] Waiting for 30 seconds to receive Password...
[ + ] Received Password: fgtyhjuj
[ - ] Credentials mismatch
```

The situation when there's a timeout while the client is entering the user id.

Client side: (Output: Client disconnected. Username was not provided within 30 seconds)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: davi0067
[ - ] Client disconnected.
[ - ] Username was not provided within 30 seconds
```

Server side: (Output: Waiting for 30 seconds to receive User Id. No connection occurred within 30 seconds. Closing the connection to the client.)

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] No connection occurred within 30 seconds.
[ - ] Closing the connection to the client.

**********************************
[ + ] Waiting for a client to connect.
```

The scenario where a timeout occurs while the client is entering the password.
Client side: (Output: Client disconnected. Password was not provided within 30 seconds)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: jack0046
[ + ] Acknowledgement Received
[ + ] Password: Cfw61RqV
[ - ] Client disconnected.
[ - ] Password was not provided within 30 seconds
```

Server side: (Output: Waiting for 30 seconds to receive Password. No connection occurred within 30 seconds. Closing the connection to the client)

```
**********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: jack0046
[ + ] Waiting for 30 seconds to receive Password...
[ - ] No connection occurred within 30 seconds.
[ - ] Closing the connection to the client.

**********************************
[ + ] Waiting for a client to connect.
```

The process of the client sending a "QUIT" message and the subsequent termination of the connection.
Client side: (Output: Server requested to close the connection. Client disconnected)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: davi0027
[ + ] Acknowledgement Received
[ + ] Password: Crc51RqV
[ + ] Server : User Validation Successful
[ INFORMATION ] Account Number: 14632873
[ INFORMATION ] Balance: 70,000
Type `QUIT` to close the connection : QUIT
[ - ] Server requested to close the connection.
[ - ] Client disconnected.
```

Server side: (Output: Received: QUIT. Closing the connection to the client)

```
*********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: davi0027
[ + ] Waiting for 30 seconds to receive Password...
[ + ] Received Password: Crc51RqV
[ + ] Received : QUIT
[ - ] Closing the connection to the client.
```

The user input is expected to strictly adhere to a specified format, which includes uppercase characters for the "QUIT" command. In this context, if a user inputs "quit" (in lowercase) or any other variant not matching the exact format, it will be considered as invalid input.

Client side: (Output: Type ‘QUIT’ to close the connection :)

```
vasi0011@gpel9:~/programs$ ./vasireddyP1Client 10.194.8.51 9633
Socket successfully created.
[ + ] Connected to the server.
[ + ] Received: Please join the server with valid credentials to retrieve the details
[ + ] User Id: davi0027
[ + ] Acknowledgement Received
[ + ] Password: Crc51RqV
[ + ] Server : User Validation Successful
[ INFORMATION ] Account Number: 14632873
[ INFORMATION ] Balance: 70,000
Type `QUIT` to close the connection : quit
Type `QUIT` to close the connection : Quit
Type `QUIT` to close the connection : QUIT
[ - ] Server requested to close the connection.
[ - ] Client disconnected.
```

Server side: (Output: Received : quit)

```
*********************************
[ + ] Waiting for a client to connect.
[ + ] Server accepted the client...
[ + ] A client got connected.
[ + ] Waiting for 30 seconds to receive User Id...
[ + ] Received User Id: davi0027
[ + ] Waiting for 30 seconds to receive Password...
[ + ] Received Password: Crc51RqV
[ - ] Received : quit
[ - ] Received : Quit
[ + ] Received : QUIT
[ - ] Closing the connection to the client.
```

CONCLUSION:

This project marks a significant milestone in my journey towards mastering network communication in the C programming language. The primary goal of creating a functional client-server pair for password verification and account information retrieval has been met.

Throughout the development process, I encountered and successfully addressed various challenges. One noteworthy hurdle was ensuring seamless communication between the client and server while handling potential timeouts and unexpected input scenarios. Additionally, crafting a robust authentication mechanism within the defined project scope demanded careful consideration.

The project has provided invaluable insights into socket programming, client-server interactions, and error handling. It has emphasized the importance of clear communication protocols and effective problem-solving approaches in network application development.

In summary, this endeavor has furnished me with a solid foundation in TCP iterative client-server interactions and equipped me with essential skills for future endeavors in network programming. It stands as a testament to the successful realization of project objectives and the acquisition of valuable knowledge in the field of data networks.