This project is about understanding ***TCP/UDP sockets*** and also to learn ***C*** and ***GO languages***. We have to write a ***client − server*** program in which the client can contact the ***db server*** to access the database. I also need to write a ***service − map*** program which resides at a well-known ***UDP socket***. The socket is a 5-digit number 2XXXX where XXXX is last 4-digits of our CSU ID (in our case, **1749**).

Let's begin with the ***servicemap.go***.
The ***service − map*** server (written in ***GO***) ***accepts*** registration from the ***db server*** (written in ***C***) and advertises ***socket address*** to the prospective ***client*** (written in ***GO***). When a database starts, it first requests a ***local TCP Port***. Then, it ***broadcasts*** a registration message in ***UDP***.

For example: ***PUT BANK*620 *tcp − portnum***

The service-mapper will create a ***GO − routine*** (i.e. a thread) to handle the message. That is, it stores the service name ***BANK*620** along with the ***IP address*** and ***Port Number*** of the ***db server*** in its table, and replies a message "***OK***" to the ***db server***. When a client starts, it ***broadcasts*** a request message.

For example: ***GET BANK*620**

The service-mapper will create a ***GO − routine*** to return a ***datagram*** containing the information of the ***db server*** (i.e. dotted ***IP*** and ***Port Number*** in ***ASCII***).
Example of ***datagram***: **137.148.204.16 ∶ 8745**

So, to explain the code of ***servicemap.go***,

checkError(***err error***) function:

- **Purpose:** A utility function that checks if an error has occurred (***err*** is not ***nil***). If an error is detected, it prints the error message and exits the program.
- **Parameters:** ***err*** - the error to check.
- **Operation:** If ***err*** is not nil, it prints "***Error occured***: " followed by the error message, and then exits the program with an exit code of **0**.

HandleMsg(***ServerCLNS* ∗ *net.UDPConn*, *adr* ∗ *net.UDPAddr*, *buf string***) function:

- **Purpose:** This is a Go routine (a lightweight thread) designed to handle incoming UDP messages asynchronously. It parses the message, distinguishes between PUT and GET requests, and acts accordingly.
- **Parameters:**
  - ***ServerCLNS* ∗ *net.UDPConn***: A pointer to the UDP connection on which the server is listening. It's used to send replies back to the client.
  - ***adr* ∗ *net.UDPAddr***: The address of the client that sent the message. It's used to send the reply to the correct client.
  - ***buf string***: The message received from the client, converted to a string for processing.

- **Operation:** The function first prints the source of the message and its content. It then splits the message into tokens using spaces as delimiters. Depending on whether the first token is "$\boldsymbol{PUT}$" or "$\boldsymbol{GET}$", it performs different actions:
  - If the request is a "$\boldsymbol{PUT}$", it stores the sender's IP and the specified port (from the message) in a global variable $'\boldsymbol{serv\_table}'$ with the key "$\boldsymbol{server}$". It then sends an "$\boldsymbol{OK}$" reply to the sender.
  - If the request is a "$\boldsymbol{GET}$", it retrieves the stored service information from $'\boldsymbol{serv\_table}'$ and sends it back to the requester.
  - A global mutex $'\boldsymbol{mutex}'$ is declared alongside $'\boldsymbol{serv\_table}'$. This mutex is used to ensure that updates to $'\boldsymbol{serv\_table}'$ (in the "$\boldsymbol{PUT}$" request handling) are done with mutual exclusion, preventing race conditions.
  - In the $\boldsymbol{HandleMsg}$ function, before the $'\boldsymbol{serv\_table}'$ map is updated, $'\boldsymbol{mutex.Lock}()'$ is called to acquire the lock. After the update is completed, $'\boldsymbol{mutex.Lock}()'$ is called to release the lock, allowing other goroutines to acquire the lock and safely update the map.

$\boldsymbol{main}()$ function:

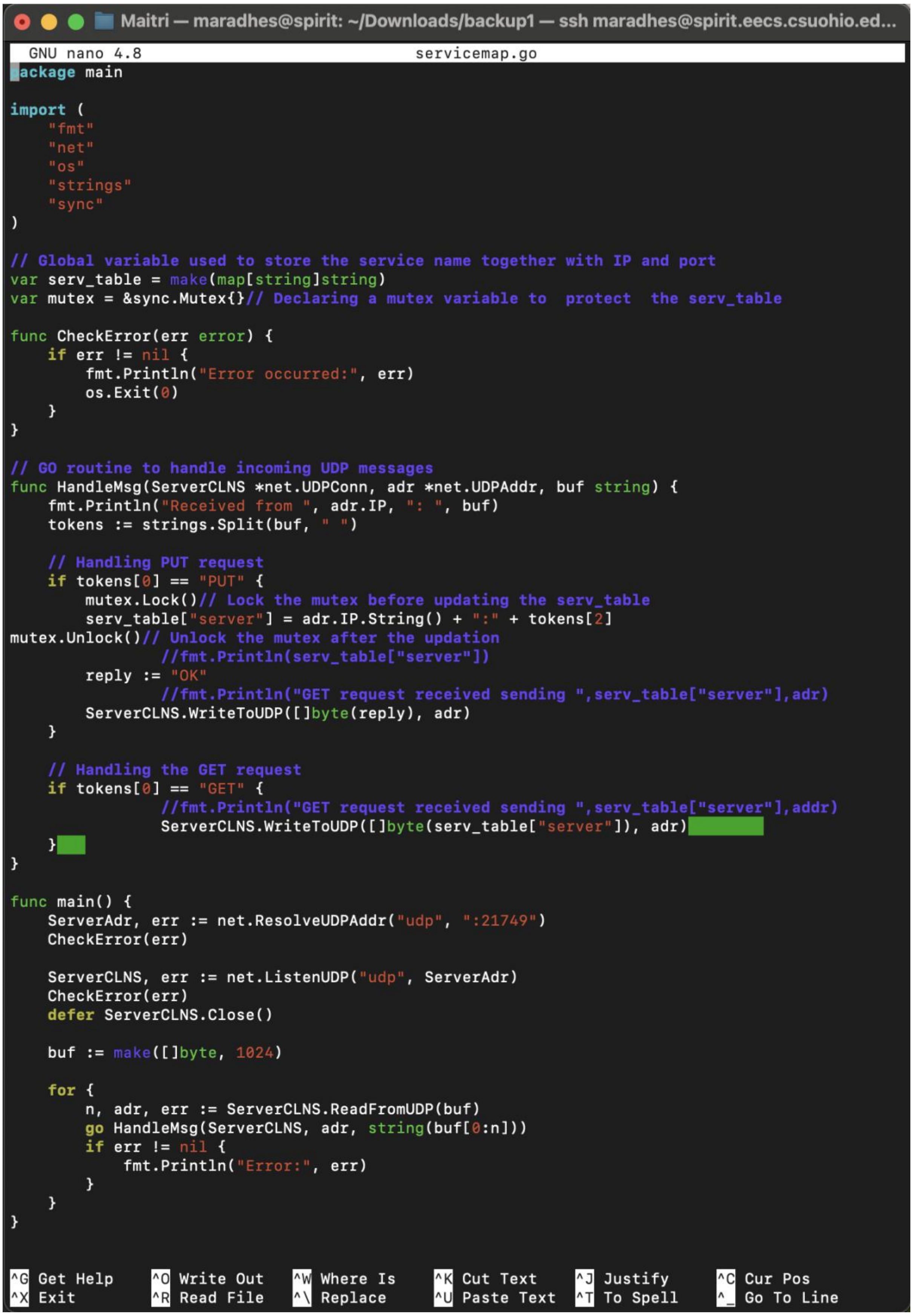
- **Purpose:** The main entry point of the program. It sets up the UDP server and enters a loop to continuously read messages from clients.
- **Operation:**
  - It first resolves the UDP address for the server to listen on port $\mathbf{21749}$ on all interfaces (":$\mathbf{21749}$").
  - Then, it creates a UDP socket for the server with $\boldsymbol{net.ListenUDP}$ and starts listening for incoming UDP packets on the resolved address.
  - A byte slice $'\boldsymbol{buf}'$ is created to read incoming packets.
  - The server enters an infinite loop where it reads from the UDP connection into $'\boldsymbol{buf}'$. For each read operation, it launches a new Go routine $'\boldsymbol{HandleMsg}'$ to handle the message asynchronously. This allows the server to handle multiple requests in parallel without blocking on a single request.
  - If an error occurs during the read operation, it prints the error.
- **Global Variables:**
  - $'\boldsymbol{serv\_table}: \boldsymbol{make}(\boldsymbol{map}[\boldsymbol{string}]\boldsymbol{string})'$: A global variable that acts as a simple in-memory key-value store. In this context, it's used to store the IP and port information of a service, with the key presumably being the service name (although the code currently hardcodes the key as "$\boldsymbol{server}$").
  - $'\boldsymbol{fmt}'$: The $'\boldsymbol{fmt}'$ package is used for formatting I/O operations, like reading input and printing output to the terminal. In this program, it's used to print error messages and logs to the console with functions like $'\boldsymbol{fmt.Println}'$.
  - $'\boldsymbol{net}'$: The $'\boldsymbol{net}'$ package provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets. In this UDP server example, it's used extensively for all network-related operations:
    - $'\boldsymbol{net.ListenUDP}'$ creates a UDP listener for the server, allowing it to receive UDP packets sent to the specified address.
    - $'\boldsymbol{net.UDPConn}'$ is a type representing a UDP network connection. This program uses it to read from and write to UDP connections.
    - $'\boldsymbol{net.UDPAddr}'$ represents the address of a UDP end point.

- **Imports:**
    - $'\boldsymbol{os}'$: The $'\boldsymbol{os}'$ package provides a platform-independent interface to operating system functionality. It's used for calling $'\boldsymbol{os.Exit(0)}'$ to exit the program with a status code of 0 when an error occurs.
    - $'\boldsymbol{strings}'$: The$'\boldsymbol{strings}'$ package implements simple functions to manipulate UTF-8 encoded strings. It's used in this program for $'\boldsymbol{strings.Split}'$ to split the received UDP message into tokens based on spaces. This parsing helps in identifying and processing the command ("$\boldsymbol{PUT}$" or "$\boldsymbol{GET}$") and its arguments from the message.
    - $'\boldsymbol{sync}'$: The $'\boldsymbol{sync}'$ package provides basic synchronization primitives such as mutual exclusion locks ($'\boldsymbol{sync.Mutex}'$). The program utilizes $'\boldsymbol{sync.Mutex}'$ to ensure that updates to the global $'\boldsymbol{serv\_table}'$ variable are done safely across multiple goroutines, preventing race conditions or data corruption.

In the program, $'\boldsymbol{strings.Split}'$ is used to split the UDP message received from clients based on spaces (" "). This is particularly useful for handling simple text-based commands where the command and its arguments are separated by spaces.

Hence, to summarize, the program demonstrates a basic UDP server that can register and retrieve service location data (IP and port) based on simple text commands. It leverages Go's concurrency model by using Go routines to handle incoming requests in parallel, allowing for efficient processing of UDP packets without blocking on I/O operations. This makes it well-suited for scenarios where quick responses to lightweight requests are required, such as service discovery or registration mechanisms in a distributed system.

Code of ***servicemap.go***:


Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio.ed...
GNU nano 4.8
servicemap.go

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
    "sync"
)

// Global variable used to store the service name together with IP and port
var serv_table = make(map[string]string)
var mutex = &sync.Mutex{}// Declaring a mutex variable to  protect  the serv_table

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error occurred:", err)
        os.Exit(0)
    }
}

// GO routine to handle incoming UDP messages
func HandleMsg(ServerCLNS *net.UDPConn, adr *net.UDPAddr, buf string) {
    fmt.Println("Received from ", adr.IP, ": ", buf)
    tokens := strings.Split(buf, " ")

    // Handling PUT request
    if tokens[0] == "PUT" {
        mutex.Lock()// Lock the mutex before updating the serv_table
        serv_table["server"] = adr.IP.String() + ":" + tokens[2]
mutex.Unlock()// Unlock the mutex after the updation
                //fmt.Println(serv_table["server"])
        reply := "OK"
                //fmt.Println("GET request received sending ",serv_table["server"],adr)
        ServerCLNS.WriteToUDP([]byte(reply), adr)
    }

    // Handling the GET request
    if tokens[0] == "GET" {
                //fmt.Println("GET request received sending ",serv_table["server"],addr)
                ServerCLNS.WriteToUDP([]byte(serv_table["server"]), adr)
    }
}

func main() {
    ServerAdr, err := net.ResolveUDPAddr("udp", ":21749")
    CheckError(err)

    ServerCLNS, err := net.ListenUDP("udp", ServerAdr)
    CheckError(err)
    defer ServerCLNS.Close()

    buf := make([]byte, 1024)

    for {
        n, adr, err := ServerCLNS.ReadFromUDP(buf)
        go HandleMsg(ServerCLNS, adr, string(buf[0:n]))
        if err != nil {
            fmt.Println("Error:", err)
        }
    }
}
```

We had the copy the database $\boldsymbol{db24}$ from the directory. Each record in the database has the following fixed format:

$\boldsymbol{struct\ record}$ {
    $\boldsymbol{int\ acctnum}$;
    $\boldsymbol{char\ name}[\mathbf{20}]$;
    $\boldsymbol{char\ phone}[\mathbf{16}]$;
    $\boldsymbol{int\ age}$;
}$\boldsymbol{accinfo}$;

The record is stored in binary (not ASCII format). Hence, to see the file content, we need to use the $\boldsymbol{od}$ command.

Let's continue with the $\boldsymbol{client.go}$.
The client gets a command from the standard input and then forwards the command with two newline characters appended at the end to the server.
For example: $\boldsymbol{qry\ acctnum\backslash n\backslash n}$
$\boldsymbol{upd\ acctnum\ amount\backslash n\backslash n}$
$\boldsymbol{quit\backslash n\backslash n}$
The $\boldsymbol{qry}$ command requests the server to get and reply the record for a person. The $\boldsymbol{upd}$ command asks the server to add an amount to a person's record in the database.

So, to explain the code of $\boldsymbol{client.go}$,
The Go program is designed to discover a service via UDP broadcast and then establish a TCP connection to interact with it. Here's a function-by-function explanation:

Network Communication and Discovery

UDP Broadcast for Service Discovery

- **Purpose:** The initial part of the program deals with discovering a service on the network. It sends a UDP broadcast message to a predefined broadcast address and port. Broadcasting is used here because the client does not know the exact address of the server it needs to communicate with. The message "GET BANK620 " is intended for services listening on the network that recognize this request format and are configured to respond to it.
- **Implementation:**
  - $'\boldsymbol{net.ResolveUDPAddr}'$ is used twice to resolve both the local address for binding the UDP socket ($'\mathbf{0.0.0.0:0}'$) and the broadcast address ($'\mathbf{137.148.205.255:21749}'$). The former allows the operating system to automatically choose an available port and interface, while the latter specifies the target address for the broadcast message.
  - $'\boldsymbol{net.ListenUDP}'$ creates a UDP socket bound to the resolved local address. This socket is used for sending the broadcast message and receiving the reply.
  - $'\boldsymbol{conn.WriteToUDP}'$ sends the broadcast message. The use of $'\boldsymbol{WriteToUDP}'$ allows specifying the destination address for each message sent, which is necessary for broadcasting.
  - The call to $'\boldsymbol{conn.SetReadDeadline}'$ prevents the program from waiting indefinitely for a response. This is particularly important in a broadcast scenario where a response may not be guaranteed.

TCP Connection for Reliable Communication

- **Purpose:** After receiving the service's address through UDP, the program establishes a TCP connection to the service. TCP is chosen for the subsequent communication because it provides a reliable, ordered, and error-checked delivery of a stream of bytes between the program and the service.
- **Implementation:**
  - The service's IP address and port number are parsed from the UDP reply and used to form an address string in the " $<\boldsymbol{ip}>:<\boldsymbol{port}>$ " format.
  - $'\boldsymbol{net.Dial}'$ establishes a TCP connection to the service using this address. The established connection ($'\boldsymbol{tcp\_conn}'$) is then used for two-way communication.

User Interaction and Protocol Handling

- **Purpose:** The program enters an interactive loop where it reads commands from the user, sends them to the server, and displays the server's responses. This part simulates a basic command-line client for the service.
- **Implementation:**
  - A $'\boldsymbol{bufio.NewReader}'$ is created to read from $'\boldsymbol{os.Stdio}'$, allowing the program to handle user input line-by-line.
  - Inside the loop, each command line read from the user is appended with two newline characters before being sent. This is crucial if the server expects each command to be terminated by two newlines, which might be part of the service's protocol for delineating commands.
  - The program listens for a response after each command is sent. It uses a byte slice $'\boldsymbol{res}'$ to read the response from the server. The response is then converted to a string and printed to the console, providing immediate feedback to the user.

Detailed Points:

- **Error Handling and Exit Strategy:** The program makes liberal use of $'\boldsymbol{fmt.Println}'$for printing errors before exiting. This approach ensures that any issues encountered during execution are immediately surfaced to the user. However, the program exits with a status code of 0 on errors, which is unconventional. Typically, a non-zero exit status indicates an error. Modifying $'\boldsymbol{os.Exit}(\boldsymbol{0})'$ to $'\boldsymbol{os.Exit}(\boldsymbol{1})'$ or another non-zero value for errors would be more aligned with common practices.
- **Network Timeouts and Delays:** Setting a read deadline on the UDP socket is a critical aspect of handling network timeouts effectively. It ensures that the program remains responsive even if no service is available to respond to the broadcast message. This pattern is essential in network programming, especially in discovery mechanisms where the presence of a responding service cannot be guaranteed.
- **Converting Reply to IP and Port:** The parsing logic assumes the reply is in a very specific format ($'\boldsymbol{IP}:\boldsymbol{port}'$). While this works under controlled conditions, real-world scenarios might require more robust parsing with error checking to handle unexpected or malformed responses gracefully.
- **Sending Commands with Newlines:** Appending two newline characters to each command is a simple yet effective way to meet protocol requirements with minimal changes to the codebase. It demonstrates how client-side adjustments can be made to accommodate server expectations.

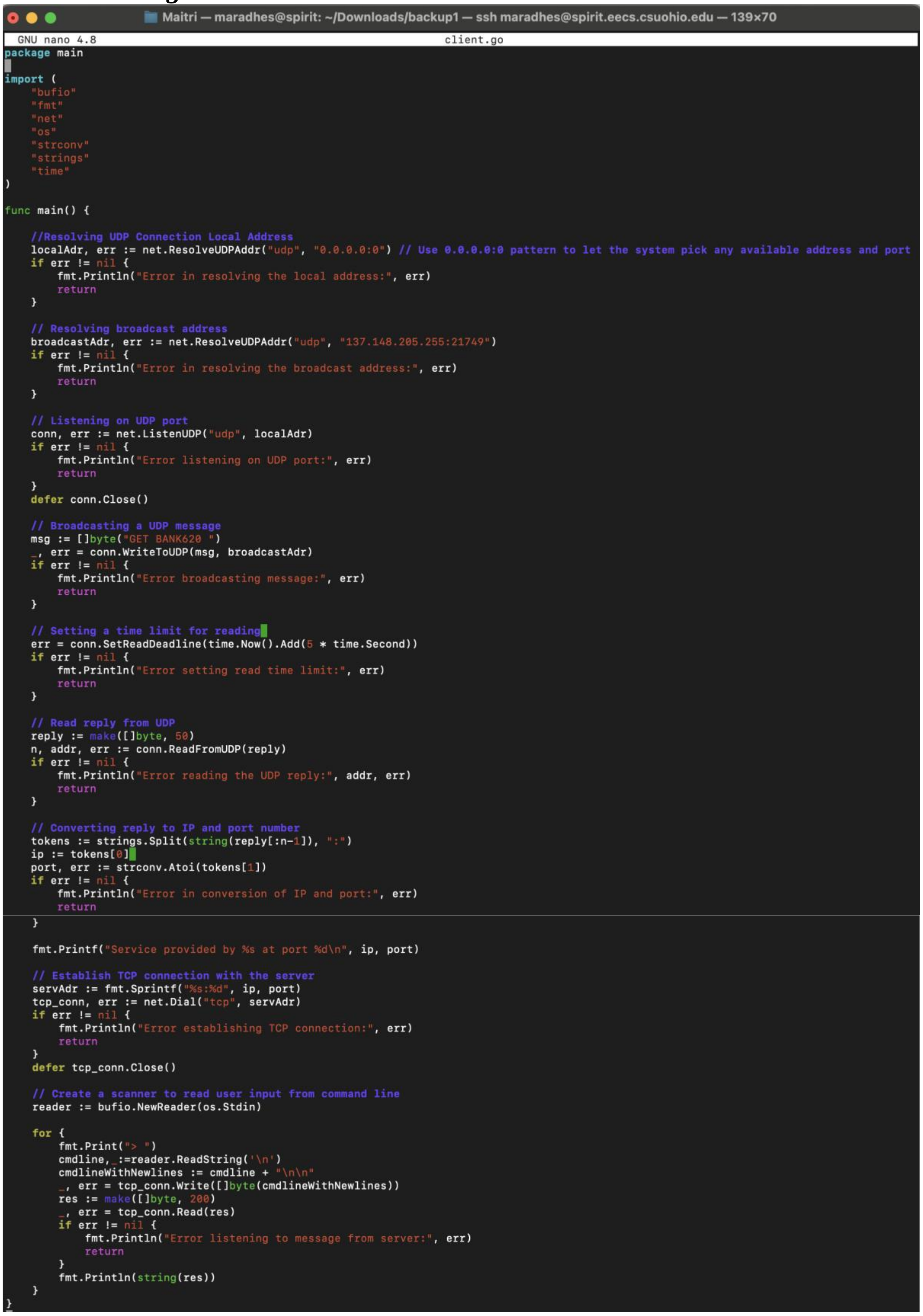
Imports:

- $'\boldsymbol{bufio}'$ **:** The $'\boldsymbol{bufio}'$ package provides buffered I/O. It wraps around an $'\boldsymbol{io.Reader}'$ or $'\boldsymbol{io.Writer}'$ interface to create buffered readers ($'\boldsymbol{bufio.Reader}'$) and writers ($\boldsymbol{bufio.Writer}'$), facilitating efficient reading and writing by minimizing disk or network I/O operations. In this context, it's likely used to read input from the user or possibly from a file or network connection in a buffered manner.
- $'\boldsymbol{fmt}'$**:** The $'\boldsymbol{fmt}'$ package implements formatted I/O with functions analogous to C's $'\boldsymbol{printf}'$ and $'\boldsymbol{scanf}'$. It's used for formatting strings, reading input, and printing output to the console. It's one of the most frequently used packages for handling text output for logs, debugging, or user interaction.
- $'\boldsymbol{net}'$**:** The $'\boldsymbol{net}'$ package provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets. This program uses it for network communication tasks like listening for connections, connecting to servers, and transmitting data over the network.
- $'\boldsymbol{os}'$**:** The $'\boldsymbol{os}'$ package provides a platform-independent interface to operating system functionality, including file and process management. It's often used to access command-line arguments, handle environment variables, and work with the filesystem. In networked applications, it might be used for reading environment variables or handling files (for configuration or logging).
- $'\boldsymbol{strconv}'$ **:** The $'\boldsymbol{strconv}'$ package implements conversions to and from string representations of basic data types. It is commonly used to convert strings to numerical types (like integers or floats) and vice versa, which is especially useful for parsing numeric data received as text from user input or over the network.
- $'\boldsymbol{strings}'$ **:** The $'\boldsymbol{strings}'$ package provides functions for manipulating UTF-8 encoded strings. It includes functionality for string searching, trimming, replacing, splitting, and joining, among others. This can be particularly useful for parsing and constructing text protocols, handling user input, or generating text output.
- $'\boldsymbol{time}'$**:** The $'\boldsymbol{time}'$ package supplies functionality for measuring and displaying time. It includes functions to retrieve the current time, manipulate time and date values, format and parse time representations, and handle durations and time zones. In network applications, it might be used for setting timeouts, measuring elapsed time, or scheduling future actions.

These imported packages collectively enable the program to perform a wide range of tasks: managing network connections, processing text, interacting with the operating system, and handling timing and scheduling tasks.

This Go program exemplifies how to combine UDP and TCP protocols for service discovery and communication, handling user input in real-time, and ensuring robust error management in network applications.

Code of $client.go$:



```
Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio.edu — 139×70
  GNU nano 4.8                                                              client.go
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func main() {

    //Resolving UDP Connection Local Address
    localAdr, err := net.ResolveUDPAddr("udp", "0.0.0.0:0") // Use 0.0.0.0:0 pattern to let the system pick any available address and port
    if err != nil {
        fmt.Println("Error in resolving the local address:", err)
        return
    }

    // Resolving broadcast address
    broadcastAdr, err := net.ResolveUDPAddr("udp", "137.148.205.255:21749")
    if err != nil {
        fmt.Println("Error in resolving the broadcast address:", err)
        return
    }

    // Listening on UDP port
    conn, err := net.ListenUDP("udp", localAdr)
    if err != nil {
        fmt.Println("Error listening on UDP port:", err)
        return
    }
    defer conn.Close()

    // Broadcasting a UDP message
    msg := []byte("GET BANK620 ")
    _, err = conn.WriteToUDP(msg, broadcastAdr)
    if err != nil {
        fmt.Println("Error broadcasting message:", err)
        return
    }

    // Setting a time limit for reading
    err = conn.SetReadDeadline(time.Now().Add(5 * time.Second))
    if err != nil {
        fmt.Println("Error setting read time limit:", err)
        return
    }

    // Read reply from UDP
    reply := make([]byte, 50)
    n, addr, err := conn.ReadFromUDP(reply)
    if err != nil {
        fmt.Println("Error reading the UDP reply:", addr, err)
        return
    }

    // Converting reply to IP and port number
    tokens := strings.Split(string(reply[:n-1]), ":")
    ip := tokens[0]
    port, err := strconv.Atoi(tokens[1])
    if err != nil {
        fmt.Println("Error in conversion of IP and port:", err)
        return
    }

    fmt.Printf("Service provided by %s at port %d\n", ip, port)

    // Establish TCP connection with the server
    servAdr := fmt.Sprintf("%s:%d", ip, port)
    tcp_conn, err := net.Dial("tcp", servAdr)
    if err != nil {
        fmt.Println("Error establishing TCP connection:", err)
        return
    }
    defer tcp_conn.Close()

    // Create a scanner to read user input from command line
    reader := bufio.NewReader(os.Stdin)

    for {
        fmt.Print("> ")
        cmdline,_:=reader.ReadString('\n')
        cmdlineWithNewlines := cmdline + "\n\n"
        _, err = tcp_conn.Write([]byte(cmdlineWithNewlines))
        res := make([]byte, 200)
        _, err = tcp_conn.Read(res)
        if err != nil {
            fmt.Println("Error listening to message from server:", err)
            return
        }
        fmt.Println(string(res))
    }
}
```

Next, the $\boldsymbol{server.c}$.
The server returns the result via an ASCII message with the characters $\boldsymbol{\backslash n \backslash n}$ appended at the end and the TCP connection should be kept alive until getting the quit command.
When the $\boldsymbol{server}$ establishes a connection with a client, it forks a child process to handle the requests. The child process has to use the operations $\boldsymbol{lseek}$ and $\boldsymbol{lockf}$ in UNIX. The $\boldsymbol{lseek}$ function can locate the position to be rewritten in a file, while the $\boldsymbol{lockf}$ can lock a file (or even a record) to prevent multiple clients from updating the same record simultaneously. The parent process should use a $\boldsymbol{signal - changing}$ routine to clean up any child process which has terminated.

So, to explain the code of $\boldsymbol{server.c}$,
This C program serves as a server application that manages a simple database (a file named "db24") of accounts. It allows clients to update and query account records through TCP connections, while also broadcasting its presence via UDP. The server uses a multi-process approach to handle client requests concurrently. Here's a breakdown of its key functions:

$\boldsymbol{signal\_catcher(int\ the\_sig)}$ function:

- **Purpose:** Catches $\boldsymbol{'SIGCHLD'}$ signals to clean up zombie processes left by terminated child processes. This is essential in a server that forks a new process for each client connection.
- **Parameters:** $\boldsymbol{the\_sig}$ - the signal number (not used in the function but necessary for the signature).
- **Operation:** Calls $\boldsymbol{'wait(0)'}$ to clean up zombie processes.
- **More Information:**
  - **Zombie Processes:** When a child process terminates, it still remains in the system with an exit status for the parent to read. Until the parent reads this status (using $\boldsymbol{'wait'}$ or similar functions), the child remains in a "$\boldsymbol{zombie}$" state, consuming resources.
  - **Preventing Resource Leaks:** By catching $\boldsymbol{'SIGCHLD'}$ and calling $\boldsymbol{'wait(0)'}$ the server ensures that terminated child processes do not linger and consume system resources.

$\boldsymbol{query\_database(int\ acctnum, char * buffer)}$ function:

- **Purpose:** Queries the "$\boldsymbol{db24}$" database file for a record matching $\boldsymbol{'acctnum'}$ and formats a reply message with the account details.
- **Parameters:**
  - $\boldsymbol{'acctnum'}$ - the account number to query.
  - $\boldsymbol{'buffer'}$ - a string buffer to store the reply message.
- **Operation:** Opens the database file, reads records sequentially, and checks for a matching account number. If found, it formats a message with the account details into $\boldsymbol{'buffer'}$.
- **More Information:**
  - **Database Access:** The database is a simple file ($\boldsymbol{'db24'}$), where each record is a $\boldsymbol{'struct\ record'}$. instance. The function sequentially reads these records using $\boldsymbol{'read()'}$.
  - **Record Matching:** If a record matching $\boldsymbol{acctnum'}$ is found, it formats a response string including the account's name, account number, and value into $\boldsymbol{'buffer'}$. If no matching record is found, an error message is formatted into $\boldsymbol{'buffer'}$ instead.

    - **Resource Management:** The database file is opened at the start and closed before returning, ensuring that file descriptors are not leaked.
    -

$\boldsymbol{update\_database}(\boldsymbol{int\ acctnum}, \boldsymbol{double\ value})$ function:

- **Purpose:** Updates the value associated with $'\boldsymbol{acctnum}'$ in the database by adding $'\boldsymbol{value}'$ to it.
- **Parameters:**
    - $'\boldsymbol{acctnum}'$ - the account number to update.
    - $'\boldsymbol{value}'$ - the amount to add to the account's value.
- **Operation:** Similar to $\boldsymbol{query\_database}$, but it locks the record for writing, updates the account's value, and writes the updated record back to the file.
- **More Information:**
    - **Locking for Write Access:** It uses $'\boldsymbol{lockf}(\boldsymbol{fd}, \boldsymbol{F\_LOCK}, \boldsymbol{sizeof}(\boldsymbol{struct\ record}))'$ to lock the record for exclusive access. This is crucial for preventing race conditions where two or more server processes might attempt to update the record simultaneously.
    - **Record Updating:** Upon finding the matching record, the function updates the account's value and writes the modified record back to the database file using $'\boldsymbol{write}()'$.
    - **Unlocking:** After updating, the record is unlocked with $'\boldsymbol{lockf}(\boldsymbol{fd}, \boldsymbol{F\_ULOCK}, \boldsymbol{sizeof}(\boldsymbol{struct\ record}))'$, allowing other processes to access it.

$\boldsymbol{get\_ip\_address}(\boldsymbol{char} * \boldsymbol{buffer}, \boldsymbol{size\_t\ buflen})$ function:

- **Purpose:** Retrieves the server's first non-loopback IPv4 address and stores it in $'\boldsymbol{buffer}'$.
- **Parameters:**
    - $'\boldsymbol{buffer}'$ - a buffer to store the IP address.
    - $'\boldsymbol{beflen}'$ - the size of $'\boldsymbol{buffer}'$.
- **Operation:** Iterates over the server's network interfaces to find the first non-loopback IPv4 address. Uses $'\boldsymbol{getifaddrs}'$ to get a list of interfaces and $'\boldsymbol{getnameinfo}'$ to convert the first suitable address to a string.
- **More Information:**
    - **Interface Iteration:** It iterates over all network interfaces obtained via $'\boldsymbol{getifaddrs}(\&\boldsymbol{ifaddr})'$. For each interface, it checks whether it is not a loopback ($'!\,(\boldsymbol{ifa}{-}{>}\ \boldsymbol{ifa\_flags}\ \&\ \boldsymbol{IFF\_LOOPBACK})'$) and is of family $'\boldsymbol{AF\_INET}'$.
    - **Address Retrieval:** For the first suitable interface, it uses $'\boldsymbol{getnameinfo}()'$ to convert the interface's address to a string format, storing it in $'\boldsymbol{buffer}'$.

$\boldsymbol{main}()$ function:

The $\boldsymbol{main}$ function in this C program orchestrates the setup and operation of a server that communicates with clients using TCP for request handling and UDP for service announcement. It demonstrates a combination of network programming techniques, process management, and simple file-based database operations. Here's a detailed breakdown of its operation:

Signal Handling for Zombie Process Cleanup

- The program registers a signal handler for $'\boldsymbol{SIGCHLD}'$ using $'\boldsymbol{signal}(\boldsymbol{SIGCHLD}, \boldsymbol{signal\_catcher})'$. $'\boldsymbol{SIGCHLD}'$ is sent to a parent process whenever one of its child processes terminates. The signal handler, $'\boldsymbol{signal\_catcher}'$, uses $'\boldsymbol{wait}(\mathbf{0})'$ to wait for all child processes that have terminated, thus cleaning up ("reaping") any zombie processes. Zombie processes are child processes that have terminated but still occupy an entry in the process table.

TCP Socket Creation and Binding

- A TCP socket is created with $'\boldsymbol{socket}(\boldsymbol{AF\_INET}, \boldsymbol{SOCK\_STREAM}, \mathbf{0})'$. This socket listens for incoming client connections.
- The server's address structure $'(\boldsymbol{serv\_adr})'$ is configured to listen on any interface ($'\boldsymbol{INADDR\_ANY}'$) and to let the system dynamically assign a port $('\boldsymbol{serv\_adr}.\boldsymbol{sin\_port} = \mathbf{0}')$.
- The socket is bound to this address with $'\boldsymbol{bind}()'$. Binding assigns the specified address to the socket, making it a named socket that can receive connections.
- The program uses $'\boldsymbol{getsockname}'$ to retrieve the assigned port number after binding, since it was set to $'\mathbf{0}'$ to allow dynamic assignment. This port number is later broadcasted over UDP for service discovery.

UDP Broadcasting for Service Announcement

- The server announces its presence on the network by sending a broadcast message via UDP. This message contains the service name "$\boldsymbol{BANK620}$" and the dynamically assigned TCP port. Clients can listen for this broadcast to discover the server's IP address and port.
- A UDP socket is configured with $'\boldsymbol{setsockopt}(\boldsymbol{sk}, \boldsymbol{SOL\_SOCKET}, \boldsymbol{SO\_BROADCAST}, \ldots)'$ to enable broadcasting.
- The broadcast message is sent using $'\boldsymbol{sendto}()'$, targeting a predetermined broadcast address and UDP port.

Listening for TCP Connections and Forking Child Process

- The server listens for incoming TCP connections with $'\boldsymbol{listen}(\boldsymbol{orig\_sock},\ \mathbf{5})'$, specifying a backlog of 5, which is the number of pending connections that can be queued.
- For each incoming connection, $'\boldsymbol{accept}()'$ creates a new socket $('\boldsymbol{new\_sock}')$ for communication with the connecting client. The server then forks a new process to handle this client.
- The child process handles client requests in a loop, reading messages with $'\boldsymbol{recv}()'$, processing them, and sending responses with $'\boldsymbol{send}()'$. It supports two types of messages: queries $('\boldsymbol{qry}')$ for account information and updates $('\boldsymbol{upd}')$ to account values.
- The parent process continues to accept new connections, and the cycle repeats.

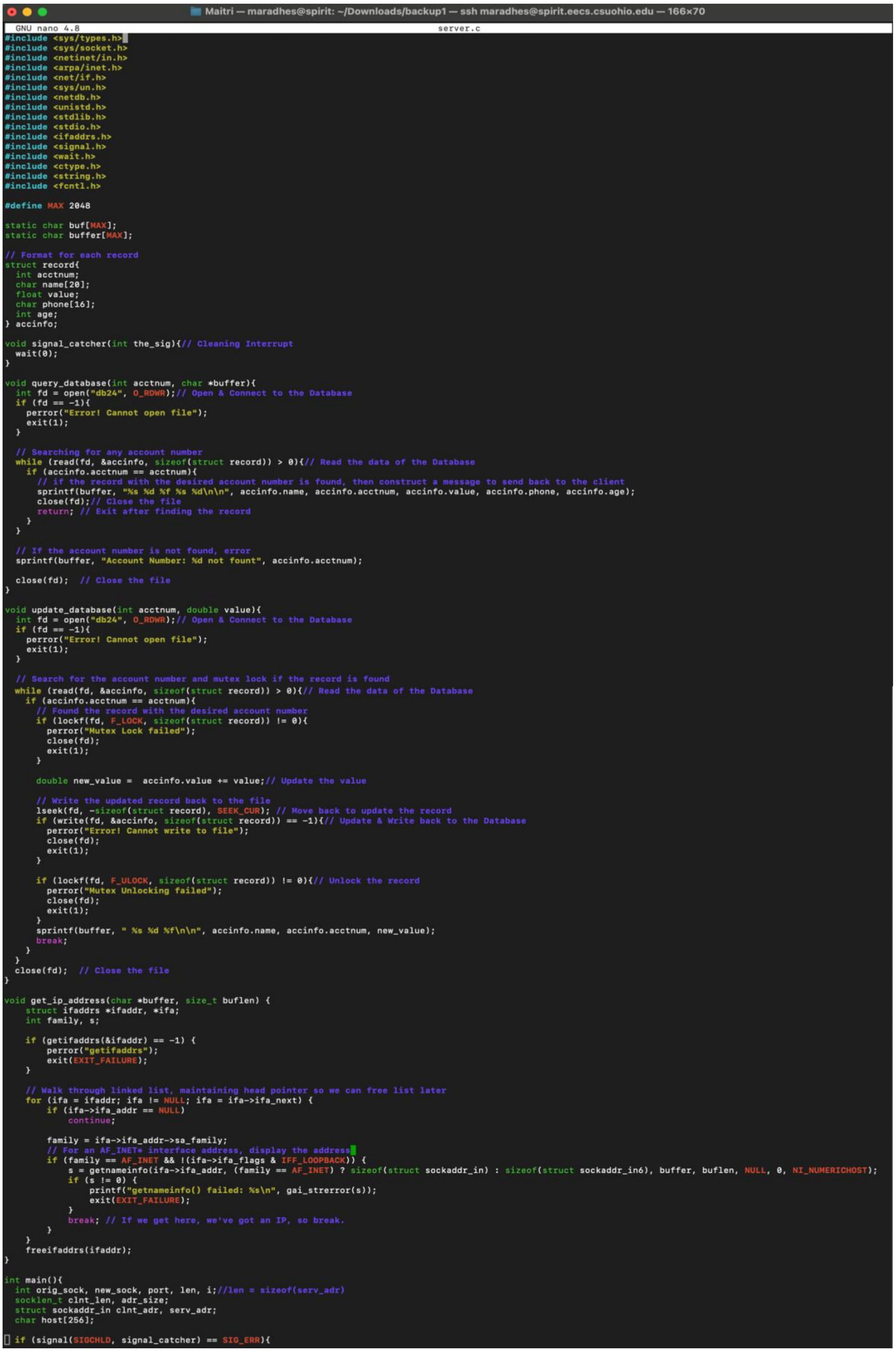
Error Handling and Resource Management

- Throughout its operation, the program checks for errors (e.g., failure to create a socket, bind, listen, accept connections, read from or write to the database). On encountering an error, it prints an error message and exits or continues as appropriate.
- Child processes are responsible for closing their dedicated client sockets $('\boldsymbol{new\_sock}')$ before terminating to ensure resources are released properly.

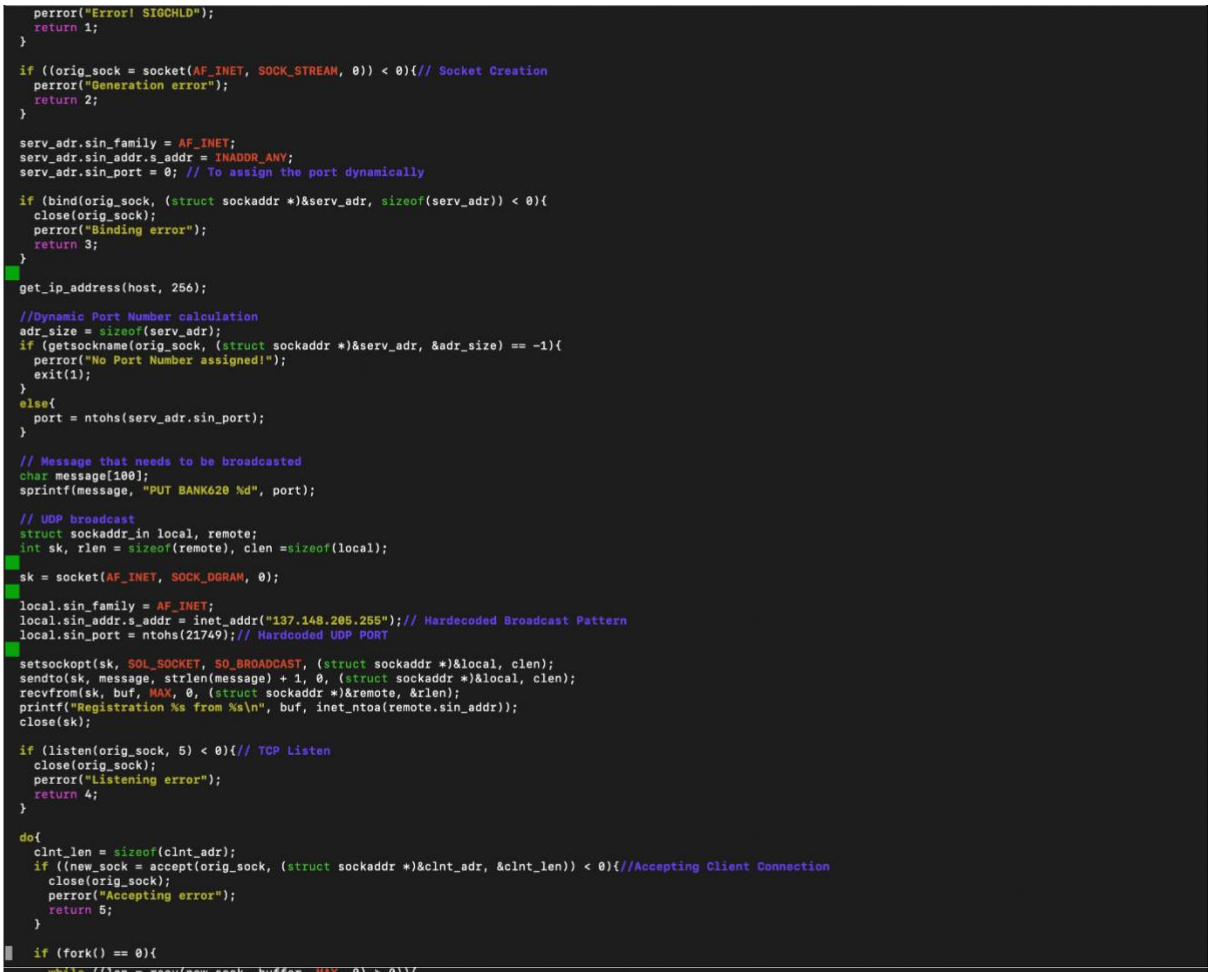
Imports:

- $' < sys/types.h > '$: Provides definitions of data types like $'pid\_t'$ (used for process IDs) and $'size\_t'$ (used for sizes of objects). It's often required when using other system call-related header files.
- $' < sys/socket.h > '$: Defines functions and data structures needed for socket programming, enabling the program to perform network communication over TCP or UDP.
- $' < netinet/in.h > '$: Contains constants and structures needed for internet domain addresses. For instance, it defines $'struct\ sockaddr\_in'$ which is used for IPv4 addresses.
- $' < arpa/inet.h > '$: Provides declarations for functions related to Internet operations, including functions for converting between host and network byte order, and for manipulating numerical IP addresses ($'inet\_addr,'inet\_ntoa'$).
- $' < net/if.h > '$: Used for the configuration and management of network interfaces; provides data structures and constants used with network interface control operations. For example, it defines $'IFF\_LOOPBACK'$ used to check if an interface is a loopback interface.
- $' < sys/un.h > '$: Defines structures and constants needed for UNIX domain sockets, which are sockets used for inter-process communication (IPC) on the same host.
- $' < netdb.h > '$: Defines functions and structures for network database operations, such as translating a hostname to an IP address, which is part of the Domain Name System (DNS) resolution process.
- $' < unistd.h > '$: Provides access to POSIX operating system API, including numerous essential functions like $'read'$, $'write'$, $'close'$, and $'fork'$.
- $' < stdlib.h > '$: Standard General Utilities Library. It includes functions involving memory allocation, process control, conversions, and others. Functions like $'exit'$ and $'atoi'$ are defined here.
- $' < stdio.h > '$: Standard Input and Output Library. It provides functionalities for file and data stream input and output operations. This is where functions like $'printf'$ and $'sprintf'$ come from.
- $' < ifaddrs.h > '$: Defines the $'getifaddrs'$ function and related structures. This function retrieves the network interfaces of the local system, which can be used to find IP addresses assigned to the server.
- $' < signal.h > '$: Provides facilities for setting handlers for different types of signals. Signals are used for inter-process communication, with $'signal'$ and $'sigaction'$ being common functions for setting signal handling functions.
- $' < wait.h > '$: Includes declarations for waiting for process termination. It defines the $'wait'$ and $'waitpid'$ functions which are used for synchronizing with terminated child processes, thus preventing zombies.
- $' < ctype.h > '$: Contains functions to test and map characters. This includes character classification functions like $'isdigit'$ or $'isalpha'$, and functions to convert between uppercase and lowercase.
- $' < string.h > '$: Contains functions for handling strings and arrays. It's used for operations like comparing, copying, and concatenating strings, with functions like $'strlen'$, $'strcmp'$, $'strcpy'.$
- $' < fcntl.h > '$: Defines operations on file descriptors like opening, locking files. It provides the $'open'$, $'fcntl'$, and $'lockf'$ functions, allowing control over file descriptors beyond simple reading and writing.

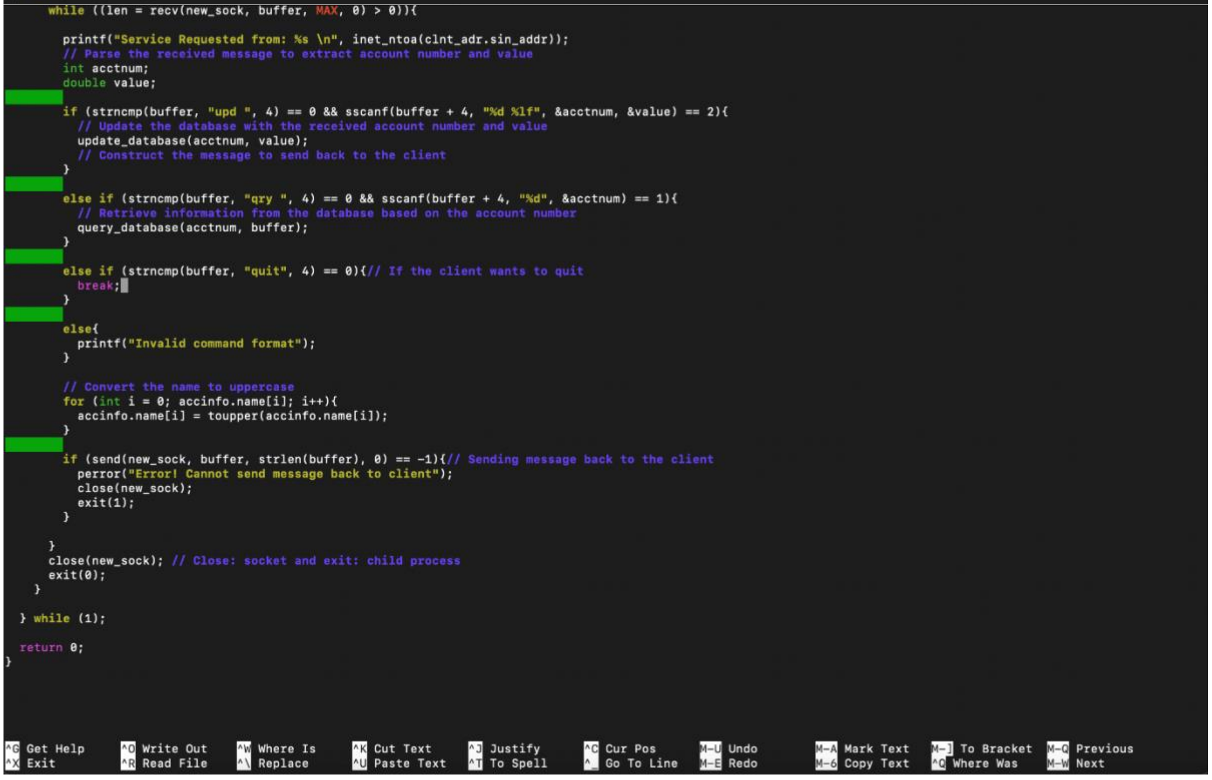
Code of ***server.c***:



```
Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio.edu — 166×70
  GNU nano 4.8                                                                    server.c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <sys/un.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <ifaddrs.h>
#include <signal.h>
#include <wait.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>

#define MAX 2048

static char buf[MAX];
static char buffer[MAX];

// Format for each record
struct record{
  int acctnum;
  char name[20];
  float value;
  char phone[16];
  int age;
} accinfo;

void signal_catcher(int the_sig){// Cleaning Interrupt
  wait(0);
}

void query_database(int acctnum, char *buffer){
  int fd = open("db24", O_RDWR);// Open & Connect to the Database
  if (fd == -1){
    perror("Error! Cannot open file");
    exit(1);
  }

  // Searching for any account number
  while (read(fd, &accinfo, sizeof(struct record)) > 0){// Read the data of the Database
    if (accinfo.acctnum == acctnum){
      // if the record with the desired account number is found, then construct a message to send back to the client
      sprintf(buffer, "%s %d %f %s %d\n\n", accinfo.name, accinfo.acctnum, accinfo.value, accinfo.phone, accinfo.age);
      close(fd);// Close the file
      return; // Exit after finding the record
    }
  }

  // If the account number is not found, error
  sprintf(buffer, "Account Number: %d not fount", accinfo.acctnum);

  close(fd);  // Close the file
}

void update_database(int acctnum, double value){
  int fd = open("db24", O_RDWR);// Open & Connect to the Database
  if (fd == -1){
    perror("Error! Cannot open file");
    exit(1);
  }

  // Search for the account number and mutex lock if the record is found
  while (read(fd, &accinfo, sizeof(struct record)) > 0){// Read the data of the Database
    if (accinfo.acctnum == acctnum){
      // Found the record with the desired account number
      if (lockf(fd, F_LOCK, sizeof(struct record)) != 0){
        perror("Mutex Lock failed");
        close(fd);
        exit(1);
      }

      double new_value =  accinfo.value += value;// Update the value

      // Write the updated record back to the file
      lseek(fd, -sizeof(struct record), SEEK_CUR); // Move back to update the record
      if (write(fd, &accinfo, sizeof(struct record)) == -1){// Update & Write back to the Database
        perror("Error! Cannot write to file");
        close(fd);
        exit(1);
      }

      if (lockf(fd, F_ULOCK, sizeof(struct record)) != 0){// Unlock the record
        perror("Mutex Unlocking failed");
        close(fd);
        exit(1);
      }
      sprintf(buffer, " %s %d %f\n\n", accinfo.name, accinfo.acctnum, new_value);
      break;
    }
  }
  close(fd);  // Close the file
}

void get_ip_address(char *buffer, size_t buflen) {
    struct ifaddrs *ifaddr, *ifa;
    int family, s;

    if (getifaddrs(&ifaddr) == -1) {
        perror("getifaddrs");
        exit(EXIT_FAILURE);
    }

    // Walk through linked list, maintaining head pointer so we can free list later
    for (ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next) {
        if (ifa->ifa_addr == NULL)
            continue;

        family = ifa->ifa_addr->sa_family;
        // For an AF_INET* interface address, display the address
        if (family == AF_INET && !(ifa->ifa_flags & IFF_LOOPBACK)) {
            s = getnameinfo(ifa->ifa_addr, (family == AF_INET) ? sizeof(struct sockaddr_in) : sizeof(struct sockaddr_in6), buffer, buflen, NULL, 0, NI_NUMERICHOST);
            if (s != 0) {
                printf("getnameinfo() failed: %s\n", gai_strerror(s));
                exit(EXIT_FAILURE);
            }
            break; // If we get here, we've got an IP, so break.
        }
    }
    freeifaddrs(ifaddr);
}

int main(){
  int orig_sock, new_sock, port, len, i;//len = sizeof(serv_adr)
  socklen_t clnt_len, adr_size;
  struct sockaddr_in clnt_adr, serv_adr;
  char host[256];

  if (signal(SIGCHLD, signal_catcher) == SIG_ERR){
```



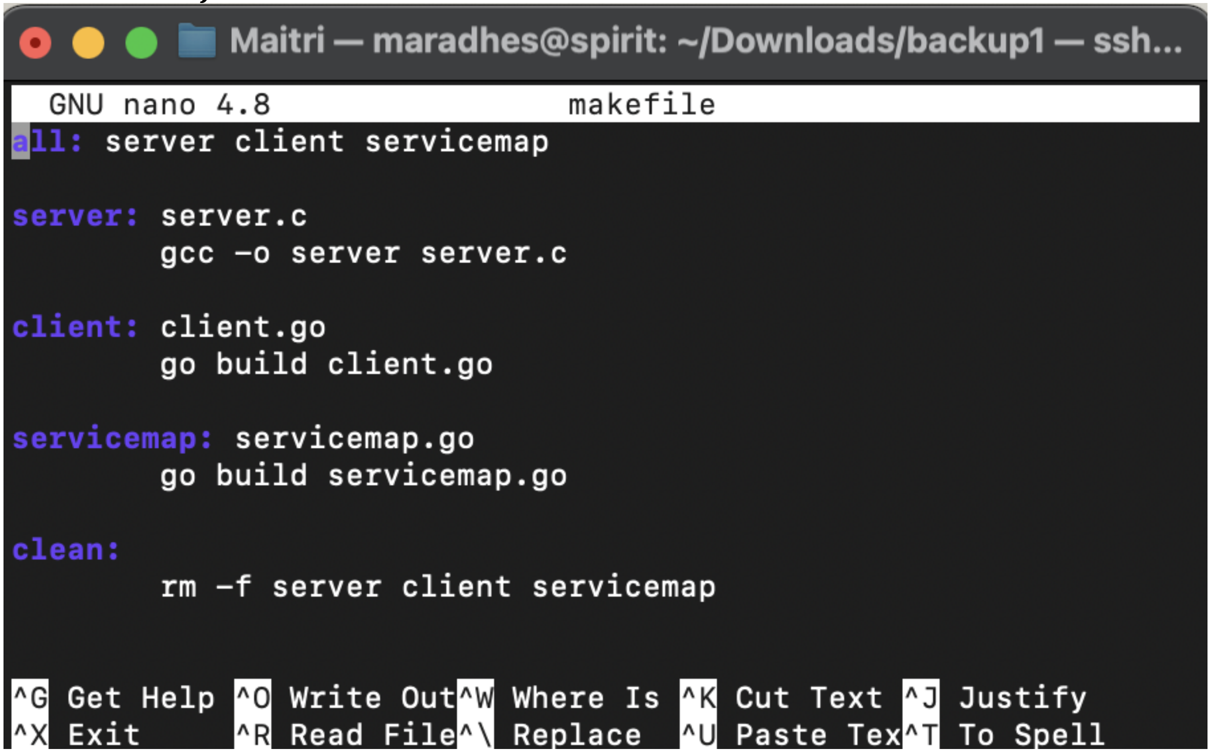
```c
    perror("Error! SIGCHLD");
    return 1;
  }

  if ((orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0){// Socket Creation
    perror("Generation error");
    return 2;
  }

  serv_adr.sin_family = AF_INET;
  serv_adr.sin_addr.s_addr = INADDR_ANY;
  serv_adr.sin_port = 0; // To assign the port dynamically

  if (bind(orig_sock, (struct sockaddr *)&serv_adr, sizeof(serv_adr)) < 0){
    close(orig_sock);
    perror("Binding error");
    return 3;
  }

  get_ip_address(host, 256);

  //Dynamic Port Number calculation
  adr_size = sizeof(serv_adr);
  if (getsockname(orig_sock, (struct sockaddr *)&serv_adr, &adr_size) == -1){
    perror("No Port Number assigned!");
    exit(1);
  }
  else{
    port = ntohs(serv_adr.sin_port);
  }

  // Message that needs to be broadcasted
  char message[100];
  sprintf(message, "PUT BANK620 %d", port);

  // UDP broadcast
  struct sockaddr_in local, remote;
  int sk, rlen = sizeof(remote), clen =sizeof(local);

  sk = socket(AF_INET, SOCK_DGRAM, 0);

  local.sin_family = AF_INET;
  local.sin_addr.s_addr = inet_addr("137.148.205.255");// Hardecoded Broadcast Pattern
  local.sin_port = ntohs(21749);// Hardcoded UDP PORT

  setsockopt(sk, SOL_SOCKET, SO_BROADCAST, (struct sockaddr *)&local, clen);
  sendto(sk, message, strlen(message) + 1, 0, (struct sockaddr *)&local, clen);
  recvfrom(sk, buf, MAX, 0, (struct sockaddr *)&remote, &rlen);
  printf("Registration %s from %s\n", buf, inet_ntoa(remote.sin_addr));
  close(sk);

  if (listen(orig_sock, 5) < 0){// TCP Listen
    close(orig_sock);
    perror("Listening error");
    return 4;
  }

  do{
    clnt_len = sizeof(clnt_adr);
    if ((new_sock = accept(orig_sock, (struct sockaddr *)&clnt_adr, &clnt_len)) < 0){//Accepting Client Connection
      close(orig_sock);
      perror("Accepting error");
      return 5;
    }

    if (fork() == 0){
```



```c
      while ((len = recv(new_sock, buffer, MAX, 0) > 0)){

        printf("Service Requested from: %s \n", inet_ntoa(clnt_adr.sin_addr));
        // Parse the received message to extract account number and value
        int acctnum;
        double value;

        if (strncmp(buffer, "upd ", 4) == 0 && sscanf(buffer + 4, "%d %lf", &acctnum, &value) == 2){
          // Update the database with the received account number and value
          update_database(acctnum, value);
          // Construct the message to send back to the client
        }

        else if (strncmp(buffer, "qry ", 4) == 0 && sscanf(buffer + 4, "%d", &acctnum) == 1){
          // Retrieve information from the database based on the account number
          query_database(acctnum, buffer);
        }

        else if (strncmp(buffer, "quit", 4) == 0){// If the client wants to quit
          break;
        }

        else{
          printf("Invalid command format");
        }

        // Convert the name to uppercase
        for (int i = 0; accinfo.name[i]; i++){
          accinfo.name[i] = toupper(accinfo.name[i]);
        }

        if (send(new_sock, buffer, strlen(buffer), 0) == -1){// Sending message back to the client
          perror("Error! Cannot send message back to client");
          close(new_sock);
          exit(1);
        }

      }
      close(new_sock); // Close: socket and exit: child process
      exit(0);
    }

  } while (1);

  return 0;
}
```

Lastly, ***makefile***.
All that we had to do here was to put together the compiling and executing commands. Along with that, we also need the all and clean commands.

Code of ***makefile***:


Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh...
GNU nano 4.8
makefile
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Paste Tex ^T To Spell

```
all: server client servicemap

server: server.c
	gcc -o server server.c

client: client.go
	go build client.go

servicemap: servicemap.go
	go build servicemap.go

clean:
	rm -f server client servicemap
```

Output:

In order to run the program, we need 3 different OS (machines). The machines that I choose were:

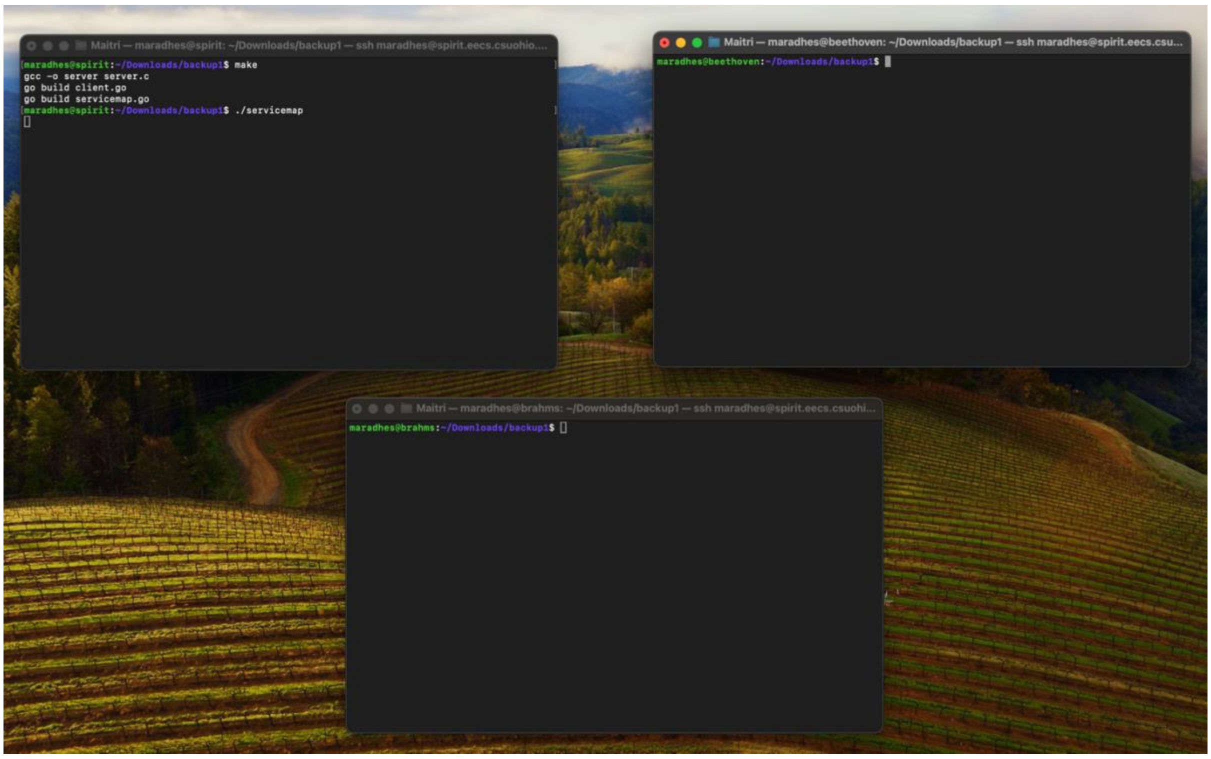
- $\boldsymbol{spirit}$ to run the $\boldsymbol{servicemap}$.
- $\boldsymbol{beethoven}$ to run the $\boldsymbol{server}$.
- $\boldsymbol{brahms}$ to run the $\boldsymbol{client}$.

To do that, I used remote login. I logged into my $\boldsymbol{spirit}$ account and I used $\boldsymbol{ssh}$ command in order to switch to other machines, $\boldsymbol{beethoven}$ and $\boldsymbol{brahms}$.

While we begin to run the program, there is a particular order that should be followed so that the execution goes correct. If the order is not followed, then the code will not run. Initially, while running, we first ran the $\boldsymbol{server.c}$ program. As the $\boldsymbol{servicemap.go}$ file had not yet been executed, the message was not getting broadcasted as required and hence, the $\boldsymbol{client.go}$ program would throw an error.

Hence, to make the order:

- Firstly, we need to $\boldsymbol{make}$ the files. This will create the necessary executable files which are required to run the files.
- Next, run the $\boldsymbol{servicemap}$.
  We need to run the $\boldsymbol{servicemap.go}$ file first as this is the file which stores the IP : Port data in its dictionary. As the data needs to be stored, the file that stores the data should be run first.



Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
```

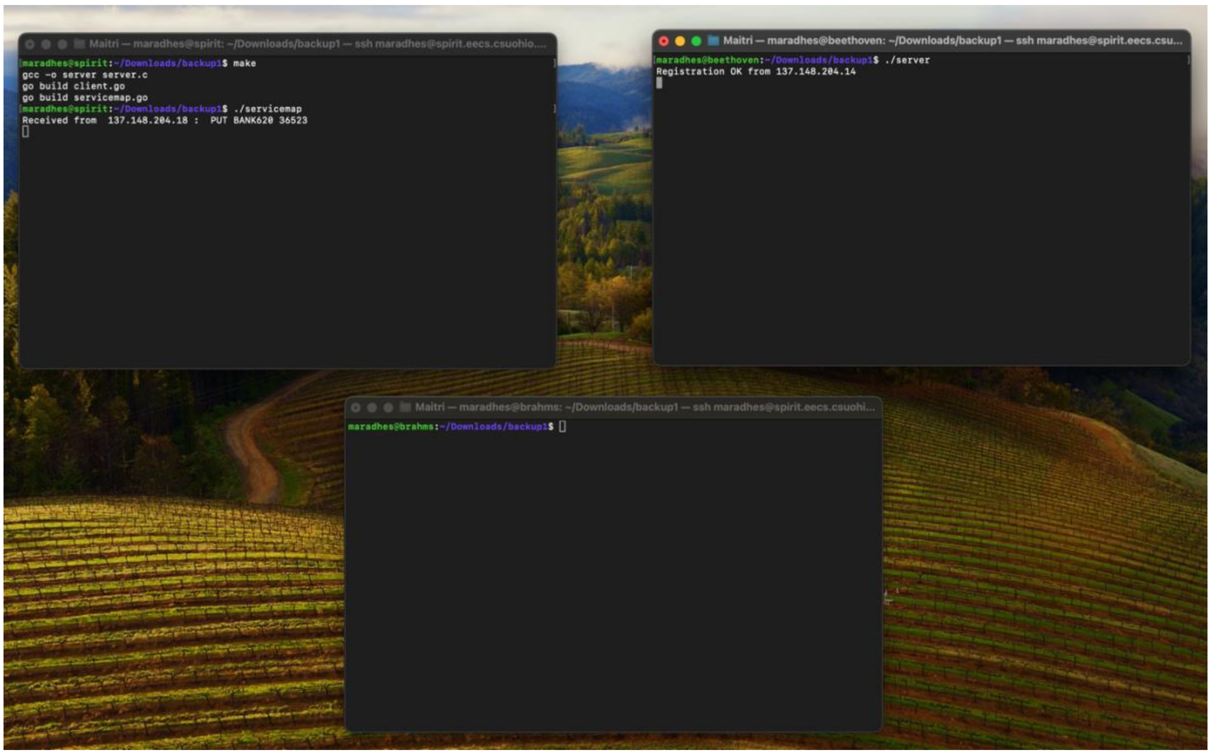
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...

```
maradhes@beethoven:~/Downloads/backup1$
```

Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@brahms:~/Downloads/backup1$
```

- Next, we need to run the ***server***.
    - Once we run the ***server***, it will try to contact the ***servicemap*** to register and will receive a registration confirmation.
      The message printed on the terminal of the ***server*** is: ***Registration OK from IP address***
      (IP address will be that of the ***servicemap***)
      In this case, the IP address is: **137. 148. 205. 14**
    - Similarly, the ***servicemap*** will also receive the broadcasted message form the ***server***.
      The message printed on the terminal of the ***servicemap*** is: ***Received from IP address* : *PUT BANK*620 *Port Number***
      (IP address and the Port Number(TCP Port) will be that of the ***server***).
      The Port Number is generated dynamically.
      In this case
        - The IP address is: **137. 148. 205. 18**
        - The Port Number is: **36523**
    - After running the ***servicemap. go***, we run the ***server. c*** file because the ***server*** is the one who broadcasts the message to ***PUT*** the data into the dictionary in the ***servicemap***. The data here is the Port Number of TCP Port which is on the ***server*** side. This data gets stored into the dictionary created on the ***servicemap*** side along with the Ip address of the ***server***.

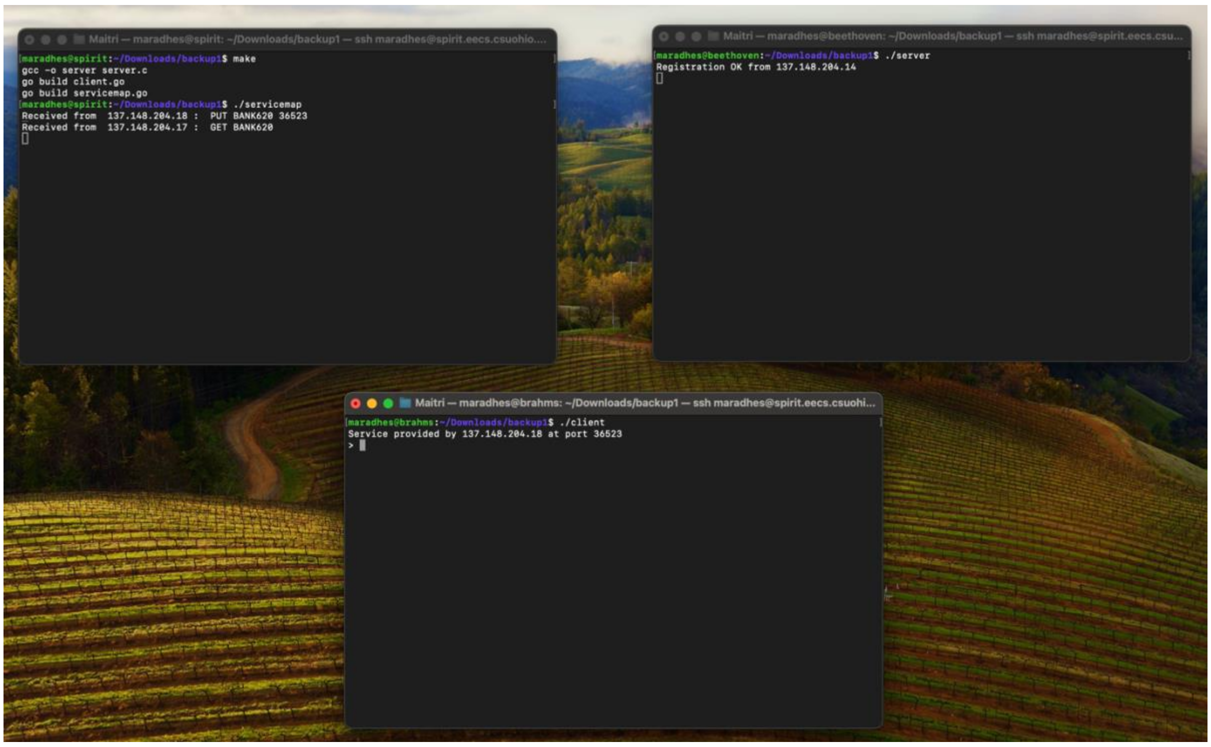

Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...
Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
```

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
```

```
maradhes@brahms:~/Downloads/backup1$ 
```

- Next, we run the ***client***.
    - Once we runt the ***client***, it will try to contact the ***servicemap*** to register and will receive the details of the service provider.
      The message printed on the terminal of the ***client*** is: ***Service provided by IP address at port Port Number***
      (IP address and the port number (TCP Port) will be that of the ***server***).
      The Port Number is generated dynamically.
      In this case
        - The IP address is: **137.148.205.18**
        - The Port Number is: **36523**
    - Similarly, the ***servicemap*** will also receive the broadcasted message form the ***server***.
      The message printed on the terminal of the ***servicemap*** is: ***Received from IP address : GET BANK620***
      (IP address and the will be that of the ***client***).
      In this case, the IP address is: **137.148.205.17**
    - After running the ***servicemap.go***, and the ***server.c***, we next run the ***client.go*** file because once the server has already made the broadcast to ***PUT*** the data into the dictionary in the ***servicemap***, the ***client*** can now retrieve the data from the ***servicemap*** file using ***GET***. This will fetch the IP address and the port number (TCP Port) of the ***server*** from which the further database queries can be performed.


Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...

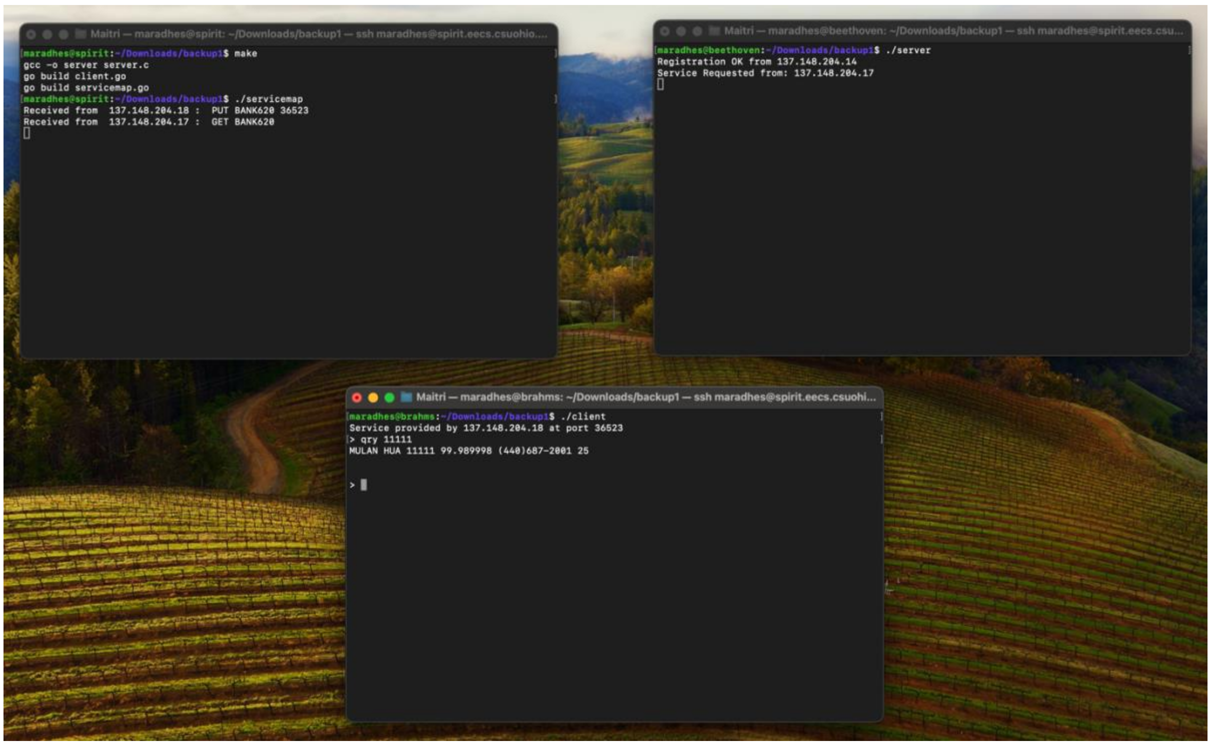
```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
```

Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
>
```

- Next, we run the test cases of the queries.

  - The 1st query is just a normal query where we give: $\boldsymbol{qry\ acctnum}$
    For example: $\boldsymbol{qry\ 11111}$
    This is given on the $\boldsymbol{client}$ terminal.
  - On this query, a service is requested to the $\boldsymbol{server}$ and the $\boldsymbol{server}$ has a message printed.
    The message being printed is:
    $\boldsymbol{Service\ Requested\ from\ IP\ address}$
    (IP address and the will be that of the $\boldsymbol{client}$).
    In this case, the IP address is: $\mathbf{137.148.205.17}$
  - This in return prints the Account holder's Name, Account Number, Value(Balance) in their account, Phone Number and the Age of the account holder.
    Hence, the output for the above query will be:
    $\boldsymbol{MULAN\ HUA}\ \mathbf{11111\ 99.989998\ (440)687-2001\ 25}$
    All of this is on the $\boldsymbol{client}$ terminal.
    After the message is printed, we have 2 newlines as told in the handout.



Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

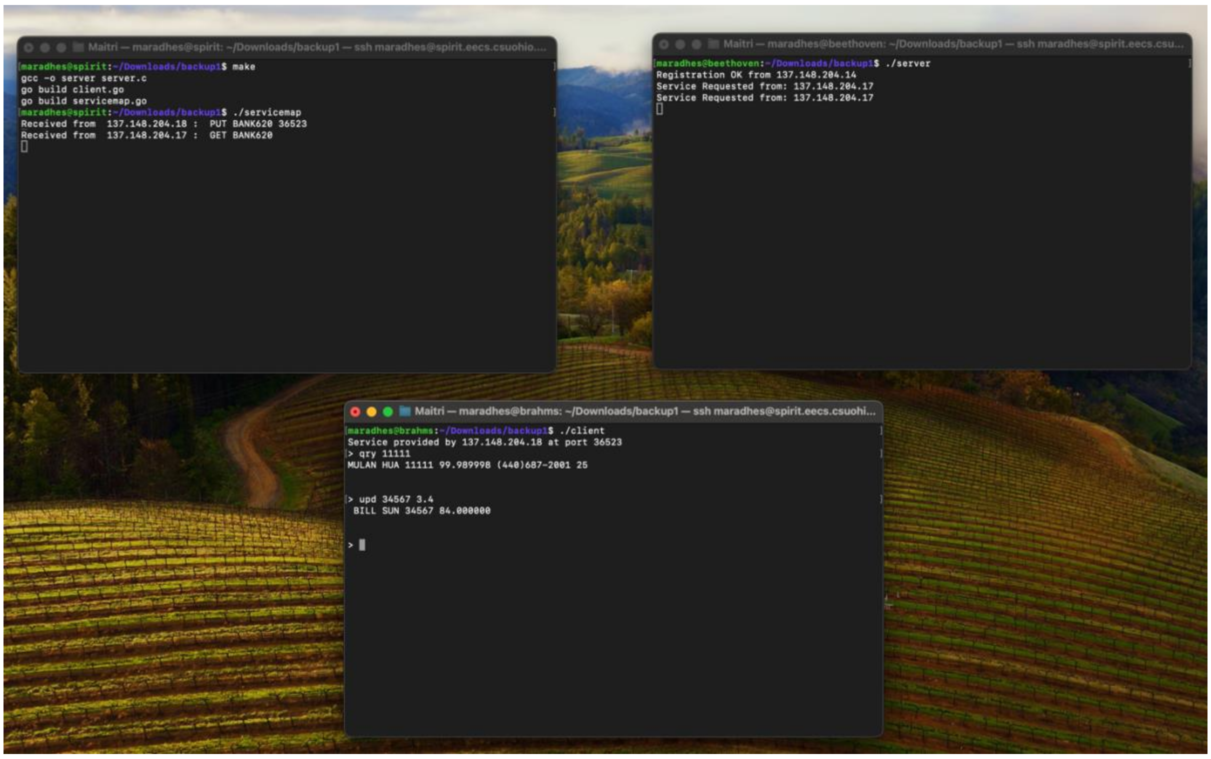
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
```

Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> 
```

- The 2nd query is an update query where we give: $\boldsymbol{upd\ acctnum\ upd\_val}$
  For example: $\boldsymbol{upd\ 34567\ 3.4}$
  This is given on the $\boldsymbol{client}$ terminal.
- On this query, a service is requested to the $\boldsymbol{server}$ and the $\boldsymbol{server}$ has a message printed.
  The message being printed is:
  $\boldsymbol{Service\ Requested\ from\ IP\ address}$
  (IP address and the will be that of the $\boldsymbol{client}$).
  In this case, the IP address is: $\mathbf{137.148.205.17}$
- This in return prints the Account holder's Name, Account Number, and Value(Balance) in their account.
  (It is the updated Value)
  Hence, the output for the above query will be:
  $\boldsymbol{BILL\ SUN\ 34567\ 84.000000}$
  All of this is on the $\boldsymbol{client}$ terminal.
  After the message is printed, we have 2 newlines as told in the handout.



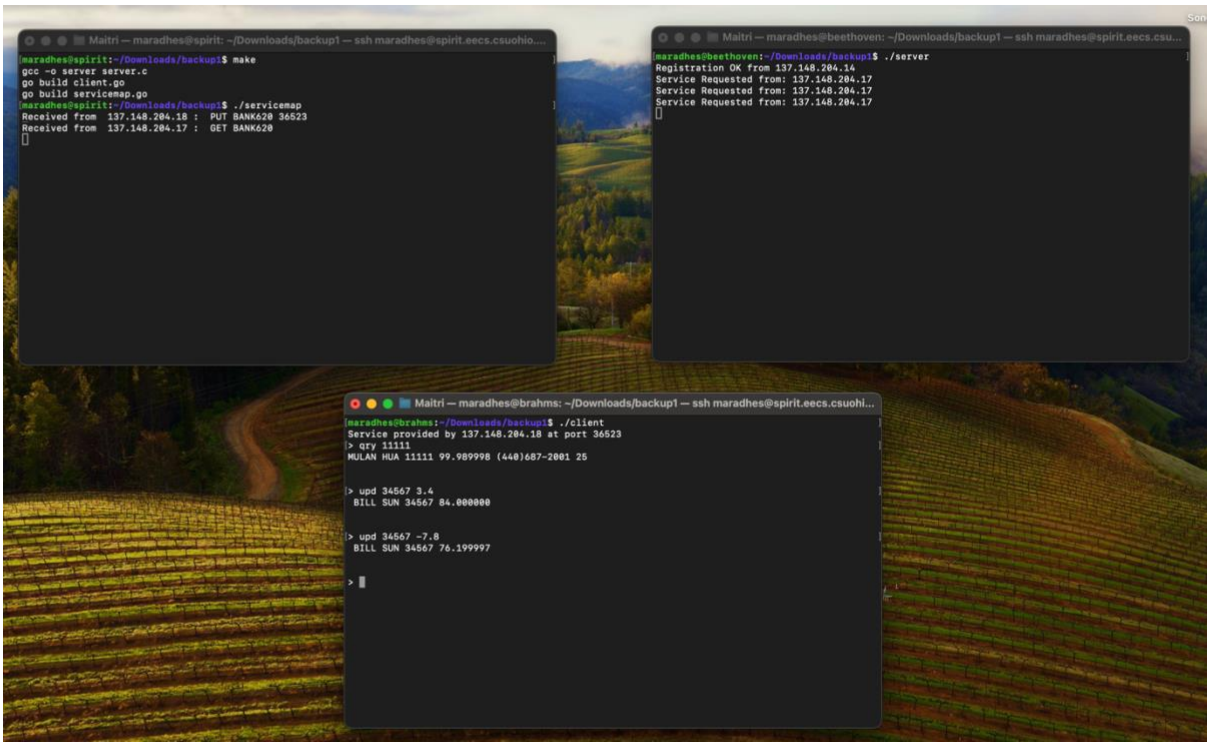
```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
```

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> upd 34567 3.4
 BILL SUN 34567 84.000000


> 
```

- The above updation happens with the use of mutex locks to perform **atomic transaction**. This method locks an account when a user is trying to update is so that no one else can make any updations. After the updation is complete, the account is unlocked. This helps us from **race conditions**.
- In this case, we are adding money into the account.

- The 3rd query is an update query where we give: $\boldsymbol{upd\ acctnum\ upd\_val}$
  For example: $\boldsymbol{upd\ 34567 - 7.8}$
  This is given on the $\boldsymbol{client}$ terminal.
- On this query, a service is requested to the $\boldsymbol{server}$ and the $\boldsymbol{server}$ has a message printed.
  The message being printed is:
  $\boldsymbol{Service\ Requested\ from\ IP\ address}$
  (IP address and the will be that of the $\boldsymbol{client}$).
  In this case, the IP address is: $\mathbf{137.148.205.17}$
- This in return prints the Account holder's Name, Account Number, Value(Balance) in their account, Phone Number and the Age of the account holder.
  Hence, the output for the above query will be:
  $\boldsymbol{BILL\ SUN}\ \mathbf{34567\ 76.199997}$
  All of this is on the $\boldsymbol{client}$ terminal.
  After the message is printed, we have 2 newlines as told in the handout.



Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
```

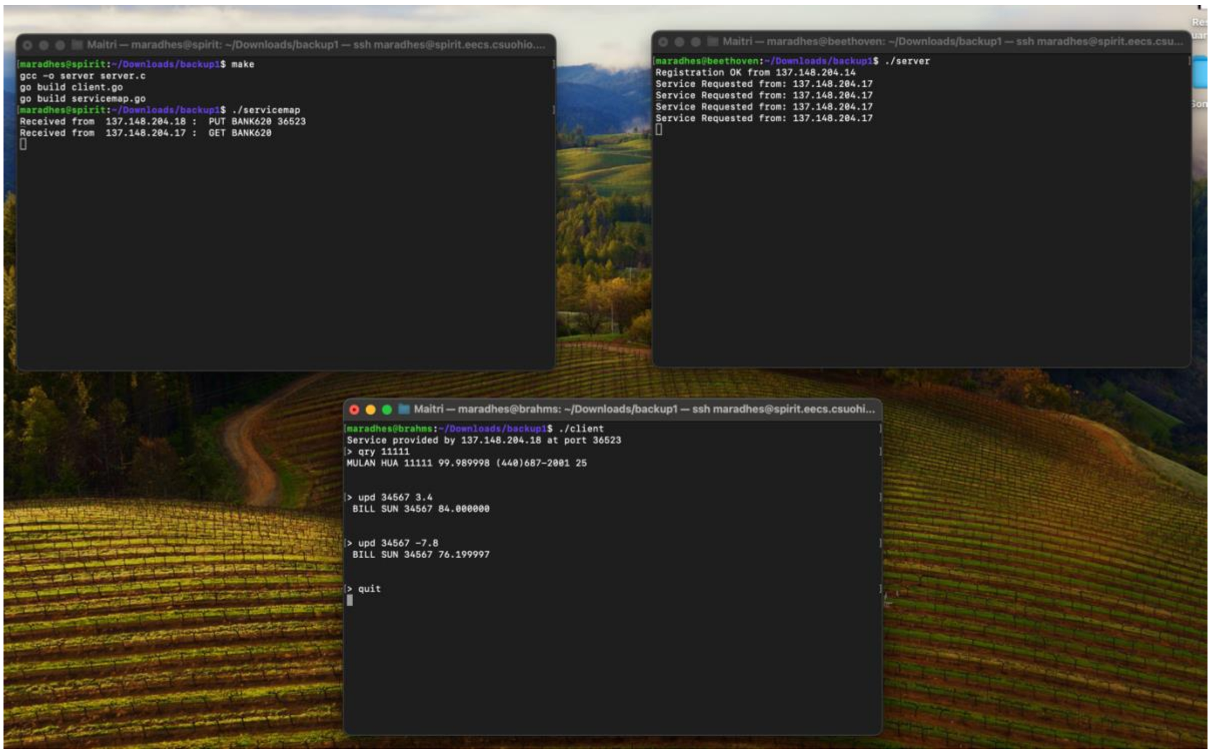
Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> upd 34567 3.4
 BILL SUN 34567 84.000000


> upd 34567 -7.8
 BILL SUN 34567 76.199997


> 
```

- The above updation happens with the use of mutex locks to perform **atomic transaction**. This method locks an account when a user is trying to update is so that no one else can make any updations. After the updation is complete, the account is unlocked. This helps us from **race conditions**.
- In this case, we are removing money from the account.

- The 4th query is a query where we give: ***quit***
  For example: ***quit***
  This is given on the ***client*** terminal.
- On this query, a service is requested to the ***server*** and the ***server*** has a message printed.
  The message being printed is:
  Service Requested from IP address
  (IP address and the will be that of the ***client***).
  In this case, the IP address is: **137.148.205.17**
- The output for the above query will be:

  It is a blank line with just a cursor at the extreme left of that line without the ability to give any more inputs. All of this is on the ***client*** terminal.

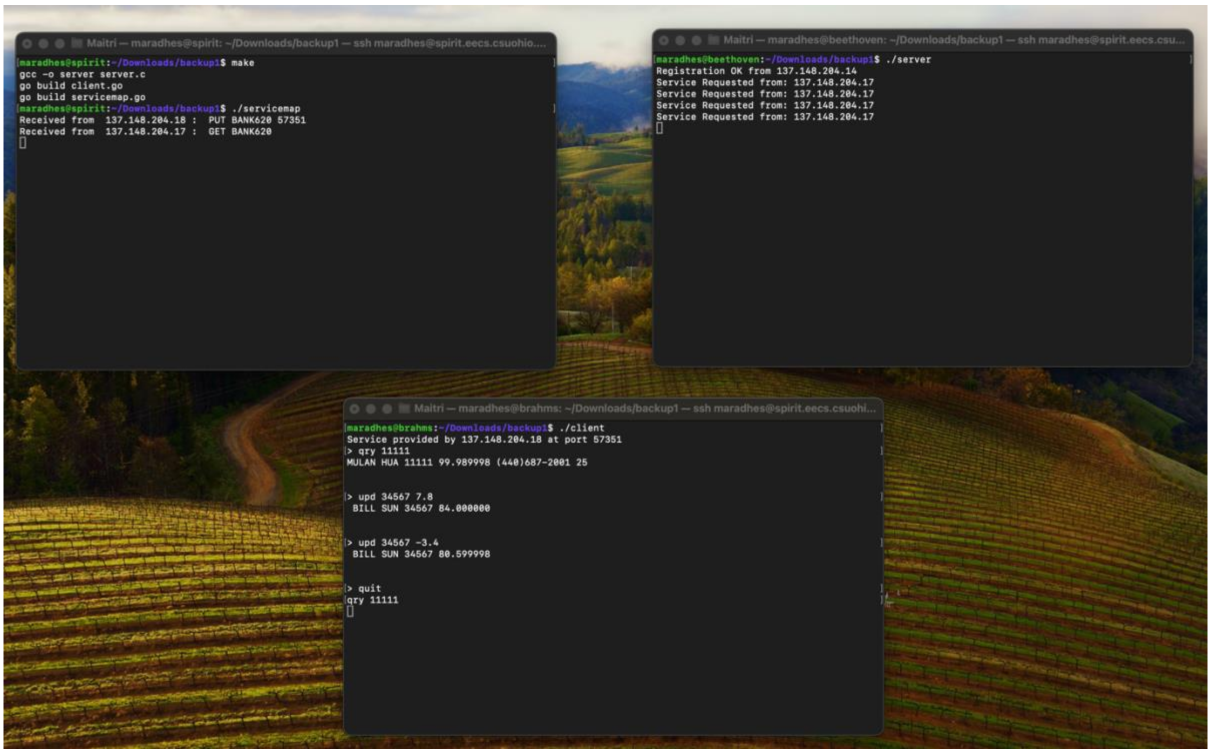


```
Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

```
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
```

```
Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> upd 34567 3.4
 BILL SUN 34567 84.000000


> upd 34567 -7.8
 BILL SUN 34567 76.199997


> quit
```

- Even if we try to give further command line inputs, the query will not run. We were able to give the command line input but the service was not requested to the ***server***.
- So far, every time we used to give an input on the command line, a service would be requested to the ***server*** but once we have ***quit***, no further requests will be made for any service to the ***server***.
- This indirectly tells us that the TCP connection is terminated and the ***client*** cannot request for any more service requests from the ***server***.

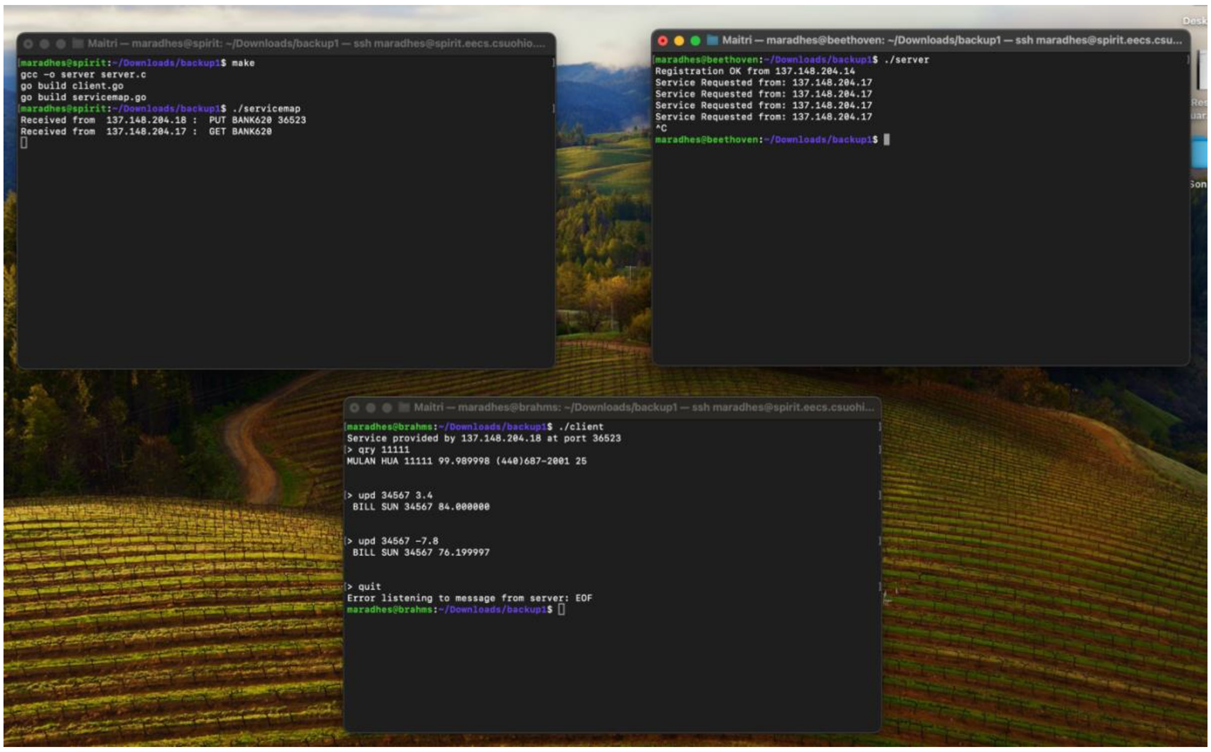

Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...
Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 57351
Received from  137.148.204.17 :  GET BANK620
```

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
```

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 57351
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> upd 34567 7.8
 BILL SUN 34567 84.000000


> upd 34567 -3.4
 BILL SUN 34567 80.599998


> quit
qry 11111
```

- Once we terminate the ***server***, the ***client*** reaches EOF automatically.


Maitri — maradhes@spirit: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohio....
Maitri — maradhes@beethoven: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csu...
Maitri — maradhes@brahms: ~/Downloads/backup1 — ssh maradhes@spirit.eecs.csuohi...

```
maradhes@spirit:~/Downloads/backup1$ make
gcc -o server server.c
go build client.go
go build servicemap.go
maradhes@spirit:~/Downloads/backup1$ ./servicemap
Received from  137.148.204.18 :  PUT BANK620 36523
Received from  137.148.204.17 :  GET BANK620
```

```
maradhes@beethoven:~/Downloads/backup1$ ./server
Registration OK from 137.148.204.14
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
Service Requested from: 137.148.204.17
^C
maradhes@beethoven:~/Downloads/backup1$
```

```
maradhes@brahms:~/Downloads/backup1$ ./client
Service provided by 137.148.204.18 at port 36523
> qry 11111
MULAN HUA 11111 99.989998 (440)687-2001 25


> upd 34567 3.4
 BILL SUN 34567 84.000000


> upd 34567 -7.8
 BILL SUN 34567 76.199997


> quit
Error listening to message from server: EOF
maradhes@brahms:~/Downloads/backup1$
```

Status of the project: ***Working***

Debugging:

There were multiple instances where we had located bugs. The primary ones which were found and fixed were the ones mentioned below:

1. In $\boldsymbol{client.go}$, during the address resolution to find the UDP connection's local address, we use the $\mathbf{0.0.0.0}$ pattern. Initially, this was not being done, which lead to an erroneous calculation of the address to be used.

2. Initially, we used the $\boldsymbol{gethostname}()$ function to identify the IP address of the machine running, but later, we modified the code to find the IP address using an alternative approach.

3. We also faced an issue in the database update, where we were to query the database and then modify the contents and the display the same, we noticed that the output of both the cases were different and cannot use the same output blocks to display the values after modification.

4. We also faced an issue in identifying the working of the $\boldsymbol{lseek}()$ function, it did not update the data in the required manner, but later, it did perform the update operation once we understood how to move the pointer across the records in the database.

5. While running the setup, ideally, the $\boldsymbol{servicemap}$ was to be up and running on a machine on the LAN first and then the server had to be run for the registration to be successful. Initially, we ran the server first and then the $\boldsymbol{servicemap}$ which led to no output on the said screens and the client threw an error as well. We rectified our mistake and managed to get the correct output.