CS3205 Computer Networks

(Jan – May 2021)
Prof. Siva Ram Murthy
Dept. of CSE, IIT Madras
Assignment 1: Simple Mail Client/Server Application Implementation
Due date: March 5, 2021, 11:59 PM, On Moodle
Extension: 15% penalty for each 24hr period;
Max. of 48hrs past the original deadline

Problem Statement

The objective of this assignment is to implement a simple email client, and a mail server using the Socket Interface.
We will be implementing a minimal client and server in this assignment. The objective of the assignment is to get familiarized with writing a network server and a client.

User Input Interface

The *func*() function in *emailclient.c* contains the code for providing and dealing with the user input. It provides the interface for both the main-prompt and sub-prompt. It takes the input and checks if it has the valid commands and passes the input to the client-side network interface.

// when is the interface called from after socket setup?

We maintain a character array *buffer*[*MAX_BUFFER*] to store each input string in the prompt. There are two types of prompts supported for the client. Each prompt supports four different instructions mentioned below. Each instruction is then called using the *network_interface*() function with an instruction number. The invalid input is not transferred to the network interface and thus the prompt asks for the input again. The valid inputs are as follows:

Main prompt

Main-prompt>

1. **Listusers**
   This command is used to list all the users whose spool files are available on the server. This command corresponds to instruction number 0.

2. **Adduser <userid>**

This command adds the user with name *userid* in the spool file list if there does not exist one. Here *userid* is the name of the user whose spool file is to be made. This command corresponds to instruction number 1.

3. **Quit**
   This command quits the main prompt and exits the client side communication with the server. This command corresponds to instruction number 7.

4. **SetUser <userid>**
   This command sets the current prompt to a user prompt only if that user exists in the server. Further all the instructions are executed in the user prompt until it gets out of it using Done instruction. And again the main prompt gets activated. All the instructions in the user prompt will be corresponding to the particular user which is set through this command. This command corresponds to instruction number 2.

User prompt

Sub-prompt-userid>

1. **Read**
   This command reads the mail inside the spool file at a pointer set for the file. And increments the pointer so that the next Read instruction will read the next mail. This command corresponds to instruction number 4.

2. **Delete**
   This command deletes a mail in the spool file at the pointer set for the file. And then the pointer points to the next mail in the file (if there exists one). This command corresponds to instruction number 5.

3. **Send <userid>**
   This command sends a mail to the provided *userid* (if it exists) from the current file which is set. This command corresponds to instruction number 5.

4. **Done**
   This command is used to get out of the user prompt. This command corresponds to instruction number 6.

Node

We made a Node structure in node.h file with the following properties:

```
char name[MAX_BUFFER];   // store the name of file
```

```
int messages;           // store the total number of mails
int current;            // store the current pointer mail in the file
```

We have helper methods and functions in node.c file which includes with their signatures in node.h as follows:

struct Node* Node(char[]);
To create a structure of type Node with name given in the argument.

char* getName(struct Node* node);
To get the name of the node structure. Not really needed as we can directly get the name using node->name structure access.

int getMessages(struct Node* node);
To get the messages counter of the node structure. Not really needed as we can directly get the name using node->messages structure access.

int getCurrent(struct Node* node);
To get the current counter of the node structure. Not really needed as we can directly get the name using node->current structure access.

void incrementMessage(struct Node* node);
This function is used in SEND instruction and called from the server. It increments the message field of the node structure.

void incrementCurrent(struct Node* node);
This function is used in READM instruction and called from the server. It increments the current field of the node structure. If the current field gets more or equal to the messages field then it equates to 0.

void updateDeleteCounter(struct Node* node);
This method decrements the messages field of the node structure. Now if the current gets equal to or greater than the messages field then it is set back to 0.

int checkName(char* x, char* y);
This function checks whether the string x and y are the same or not.

Network Interface

This interface is provided by the *network_interface*() function in *emailclient.c*. It identifies the command using the instruction number and processes the input from the buffer string. Then it communicates with the server using the *w_and_r*() function. This function writes the message to the server and consecutively reads the response from the server by updating the buffer string.

The server side does the execution based on the packets sent by the client using *read*() function. It maintains a buffer[MAX_BUFFER] to store the input and output from and to the client It stores the userid[MAX_BUFFER] to store the name of the file sent by the client. It uses a FILE* pointer fp to access and modify any files. Then it creates an array of Node structures with MAX_FILES elements and sets them all to NULL. It then does the required computation after identifying the instruction type and then sends a corresponding packet to the client using *write*() function.

The following instruction numbers and their execution in the network interface are as follows:

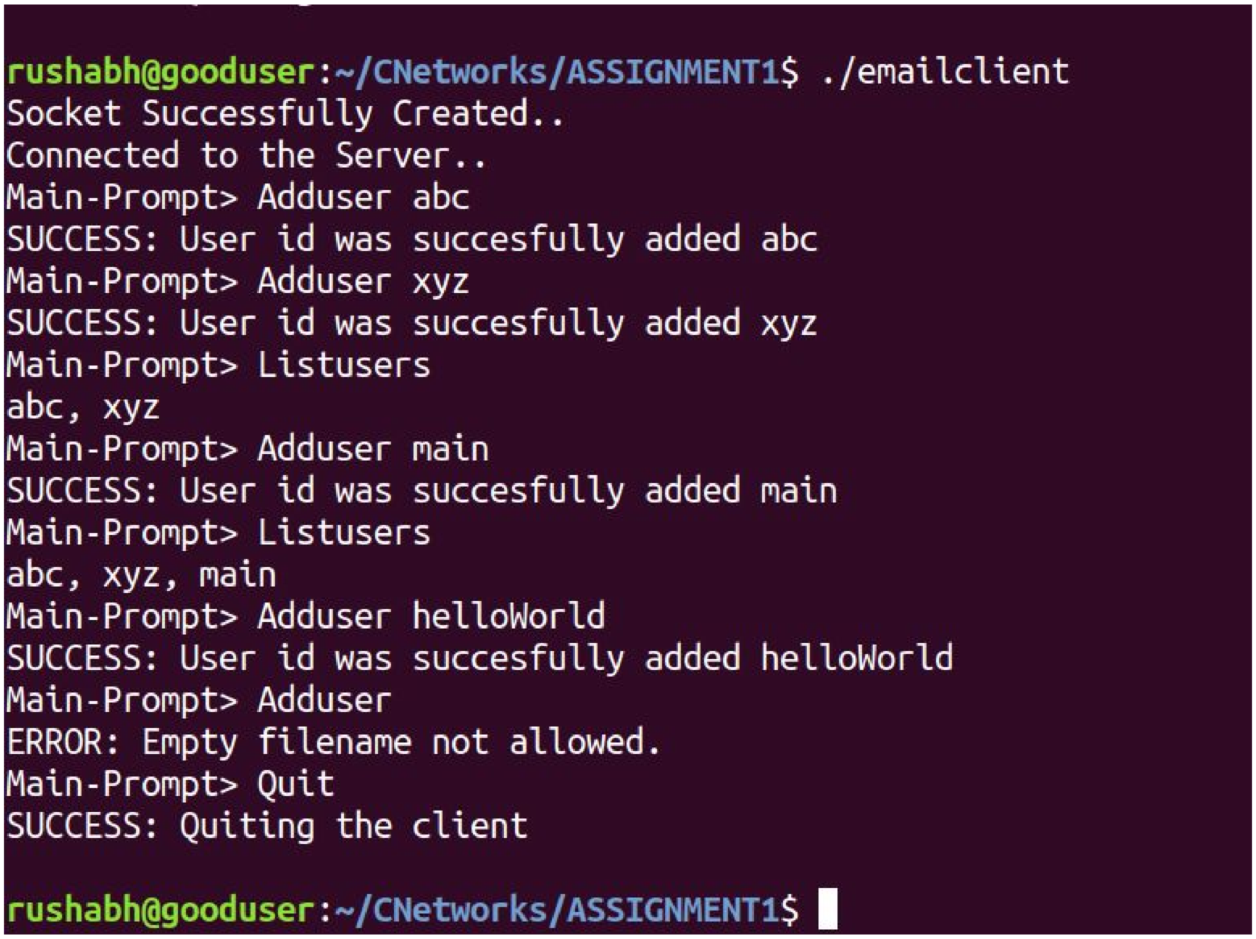
Instruction 0: LSTU
This instruction "LSTU" is simply sent to the server. And we get the list of all file names (comma separated manner) which we print as the server response.
At the server, we send back the list of all the file spools names present in the folder by simply iterating through our files array. All the file names are written in the buffer in a comma separated fashion.

Instruction 1: ADDU <userid>
We first get the name of user id by ignoring the white spaces in between and only taking the first non-zero length string. If the string is empty we return an error saying "Empty files not allowed". In case of non-zero length we set the instruction "ADDU <user id>" with a single space in between to the server. And the server output is simply printed out.

At the server, we read this non-empty user id. First we iterate through the files to check if this user id file is already present in the files. In case it is, we send back an error saying "Userid already present" . Now we check if we have extra space to create a new file by again iterating through all the files. If not, we send back the error "MAX files reached. Cannot add more files". If we have space, we create a new file using our file creation fopen() by setting the name of the file as "<user id>.nxt". We also update our files array by creating an object using the constructor *Node(char* name)* and thus send back "SUCCESS: User id was successfully added".

Instruction 2: USER <userid>
This is the SetUser command. Here again we check for non-empty user id and send the corresponding error. And now we set the instruction "USER <user id> with a single space in between and send it to the server. Now we check the server output by comparing the first 7 characters with "ERROR: " and thus for the correct match we return -1 to the user interface. This implies we cannot set the prompt to user mode.

At the server, we read this non-empty user id. First we iterate through the files and check if this file is present or not using the *checkName*() function which checks if two strings are the same or not. In case we don't find any matching file, we send back the error message "No such file found with name = <user id>". Otherwise we set the currentFile node pointer to this file we found and send a message "SUCCESS: <user id>" to the client.



```
rushabh@gooduser:~/CNetworks/ASSIGNMENT1$ ./emailclient
Socket Successfully Created..
Connected to the Server..
Main-Prompt> Adduser abc
SUCCESS: User id was succesfully added abc
Main-Prompt> Adduser xyz
SUCCESS: User id was succesfully added xyz
Main-Prompt> Listusers
abc, xyz
Main-Prompt> Adduser main
SUCCESS: User id was succesfully added main
Main-Prompt> Listusers
abc, xyz, main
Main-Prompt> Adduser helloWorld
SUCCESS: User id was succesfully added helloWorld
Main-Prompt> Adduser
ERROR: Empty filename not allowed.
Main-Prompt> Quit
SUCCESS: Quiting the client

rushabh@gooduser:~/CNetworks/ASSIGNMENT1$
```

Instruction 3: READM
We simply send the instruction "READM" to the server and print the response given by it.

At the server, as this is the user-set command, we first check if our currentFile node pointer is set to a valid file or is pointing to NULL. If it is pointing to NULL, we send the error message "Current file is NULL". Otherwise we get the name, number of messages and the current pointer number of this file. We make our fp (FILE pointer) to point this file by using *fopen*() and "r" read mode. To reach the current pointer in file message (mail), we skip the previous mails by just reading them using *fgetc*() function. And then the current mail while reading is written to the buffer using *strncat*() function. We then increment the current counter using the *incrementCurrent*() method. And we send back the buffer to the client.

Instruction 4: DELM

We simply send the instruction “READM” to the server and print the response given by it.

At the server, as this is the user-set command, we first check if our currentFile node pointer is set to a valid file or is pointing to NULL. If it is pointing to NULL, we send the error message “Current file is NULL”. Otherwise we get the name, number of messages and the current pointer number of this file. We make our fp (FILE pointer) to point this file by using *fopen*() and “r” read mode. Our idea is simple as we cannot delete a part of the text portion in the file in a simple manner, we will copy all the contents of the file except the delete portion to a temporary file named “tempfile”. We then delete the original file using the *remove*() function and rename this temporary file to the original file name using the *rename*() function. To copy only the required mails we iterate till the current pointer and write it back to the temporary file using *fputc*() function. We then ignore the next mail and then again write back the remaining mails if they exist. Then we safely modify the current and messages field using *updateDeleteCounter*() method.

Instruction 5: SEND <user id>
Here again we check for non-empty user id string and send the corresponding error if it is empty. On non-empty user id, we prompt a “Type message: “ and ask for a mail message which the user inputs. We use the getThreeHash() function to read the input only until three consecutive hashes. And now we set the instruction “SEND <user id> <message>“ with a single space in between and send it to the server. And the server output is simply printed.

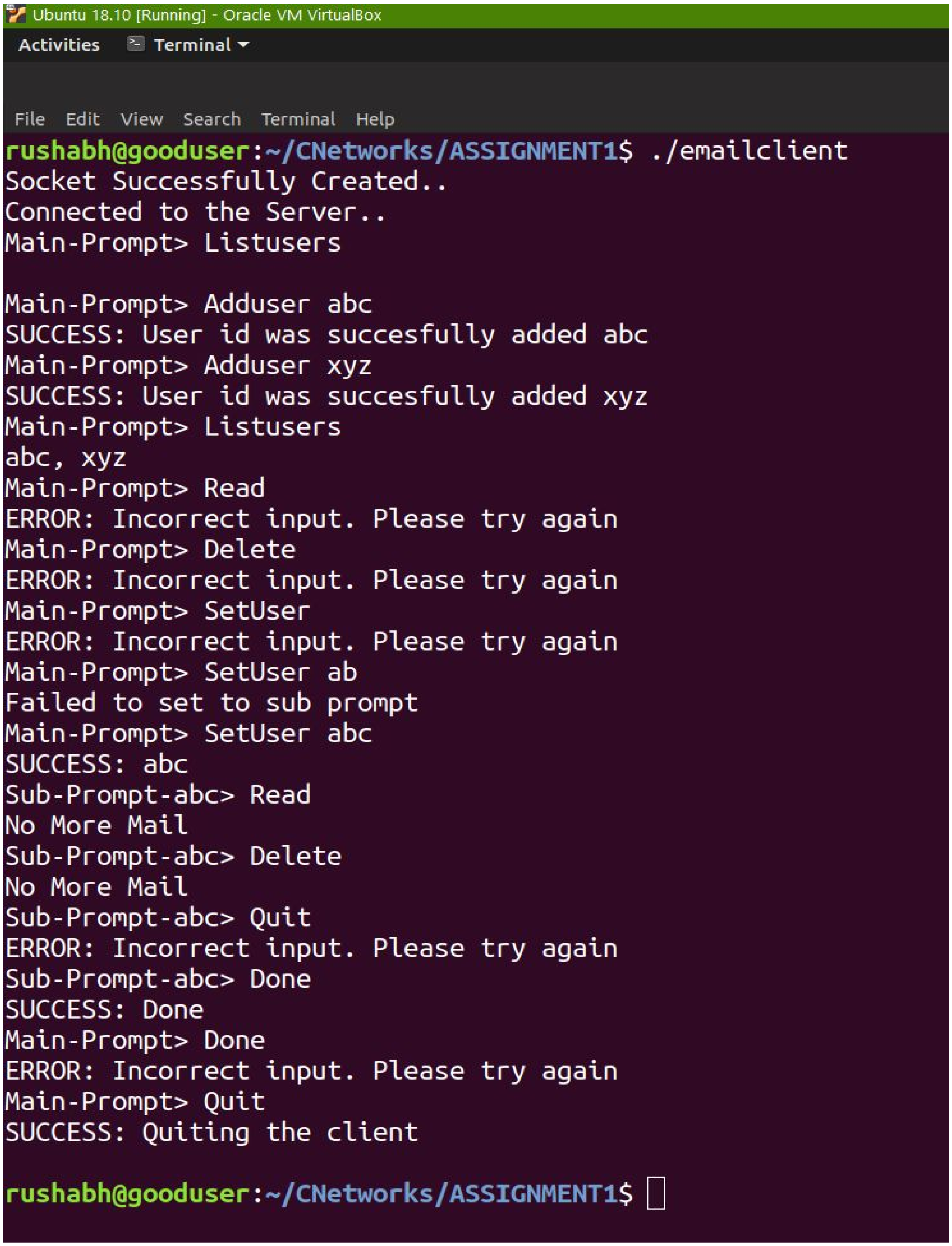
At the server, as this is the user-set command, we first check if our currentFile node pointer is set to a valid file or is pointing to NULL. If it is pointing to NULL, we send the error message “Current file is NULL”. Otherwise we get the names of both the files. We make a complete message by writing the From and To information consisting of the names of the spool files, appending the Date and then the message. The currentFIle pointer is the From file and the user id sent to the instruction corresponds to the To file. We open it using *fopen*() and “a” append mode. We append the message using the fprintf() function. On successful append, we send back “SUCCESS: Send done from the receiver id” message to the client.

Instruction 6: DONEU
We simply send the instruction “DONEU” to the server and print the response given by it.

At the server, we make the currentFile file pointer to point to NULL and send back the message “SUCCESS: Done” to the client.

Instruction 7: QUIT
We simply send the instruction “QUIT” to the server and print the response given by it.

At the server, we delete all the files with names from the files array. We delete all these file pointers using the *free*() function. We then send back the “SUCCESS: Quitting the client” message to the client.



```
rushabh@gooduser:~/CNetworks/ASSIGNMENT1$ ./emailclient
Socket Successfully Created..
Connected to the Server..
Main-Prompt> Listusers

Main-Prompt> Adduser abc
SUCCESS: User id was succesfully added abc
Main-Prompt> Adduser xyz
SUCCESS: User id was succesfully added xyz
Main-Prompt> Listusers
abc, xyz
Main-Prompt> Read
ERROR: Incorrect input. Please try again
Main-Prompt> Delete
ERROR: Incorrect input. Please try again
Main-Prompt> SetUser
ERROR: Incorrect input. Please try again
Main-Prompt> SetUser ab
Failed to set to sub prompt
Main-Prompt> SetUser abc
SUCCESS: abc
Sub-Prompt-abc> Read
No More Mail
Sub-Prompt-abc> Delete
No More Mail
Sub-Prompt-abc> Quit
ERROR: Incorrect input. Please try again
Sub-Prompt-abc> Done
SUCCESS: Done
Main-Prompt> Done
ERROR: Incorrect input. Please try again
Main-Prompt> Quit
SUCCESS: Quiting the client

rushabh@gooduser:~/CNetworks/ASSIGNMENT1$
```

What did I learn?

I learnt basic socket programming - regarding how to establish the connection between the server and client - functions like connect(), listen() etc, how are sockets passed using read() and write() function.
I learned C file handling with better illustration in this assignment - functions like fprintf(), rename() and remove().