

# **Operating Systems**

## **Phase-3 Report**

**Teammates:**

- 1. Maisha Mahrin**
- 2. Bhavicka Mohta**

## **Operating Systems Project Report**

### **Basic CLI Implementation in Server with Multiple Clients (phase 3):**

In this phase, we built upon phase 1 and phase 2 and created a socket communication model where the CLI shell was usable from the client end, and multiple clients could access the server at the same time.

#### **Overview of Phase 1:**

In phase 1, we built a CLI shell that supported single, double or triple piped commands, general command line Linux commands etc.

#### **Overview of Phase 2:**

In this phase, we had built upon phase 1 and created a socket communication model where the shell was usable from the client end.

#### **Code Supports:**

1. Ctrl+C to exit
2. “exit” command to exit
3. Single, double or triple piped commands like:  
`cat file3.txt | grep “yasin” | tee file4.txt | wc -l`
4. Supports error checking from input

\*handled using signal.h header file

5. Inputting invalid commands wait a bit and direct you to inputting the correct command again
6. Handles empty command

**List of Unsupported Commands:**

1. cd
2. ping
3. man

**Multiple Client Code Testing:** With and without error in input:

The image displays three terminal windows side-by-side, showing the process of compiling and running a C++ program.

**Terminal 1 (Left):** Shows the compilation of `MaishaBhavickaClientPhase3New.cpp` using `g++`. The command is `g++ -o c1 MaishaBhavickaClientPhase3New.c -l pthread`. The output shows the compiler flags and the resulting executable `c1`. A warning is displayed: `warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]`. The program is then executed with `./c1`, and the output shows the program's behavior when run without arguments.

**Terminal 2 (Middle):** Shows the execution of the program `c1` with the `--help` flag. The output shows the program's behavior when run with the `--help` flag, displaying the accepted new socket and the received message from the client.

**Terminal 3 (Right):** Shows the execution of the program `c1` with the `--help` flag. The output shows the program's behavior when run with the `--help` flag, displaying the accepted new socket and the received message from the client.

The image shows a Kali Linux desktop environment with three terminal windows open. The top-left window shows the server's output, including the start of the program, listening on port 3001, and receiving connections from a client. The top-right window shows the client's commands, including the start of the program, connecting to the server, and sending a message. The bottom window shows the server's output, including the start of the program, listening on port 3001, and receiving connections from a client. The desktop background is a dark blue and black pattern. The taskbar at the bottom shows various application icons, including a file manager, a terminal, and a web browser. The system clock in the top right corner shows the date and time as Friday, April 29, 2024, at 9:25 AM.

\*handled using signal.h header file

The image displays three terminal windows side-by-side, illustrating a network communication between a C# server and a Java client. The leftmost window is the C# server's console, showing it listening on port 3001, accepting connections, and receiving messages from the client. The middle window is the Java client's console, showing it connecting to the server, sending a 'hello' message, and receiving a response. The rightmost window is a second terminal, likely for the Java client, showing it receiving the response from the server. The terminals are running on a Linux system, as indicated by the window title 'Terminal' and the file manager icon in the top bar. The date and time 'Fri 29 Apr 2:25 PM' are visible in the top right corner.

The image displays three terminal windows from a Kali Linux desktop, illustrating a reverse shell attack. The top window (left) shows the victim's system with a netcat listener on port 135. The middle window shows the attacker's system, which successfully connects to the victim and obtains a root shell. The bottom window (right) shows the victim's system again, where a second netcat listener on port 135 receives a connection from the attacker's IP address.

```
Terminal 1 (Victim):
ls|more
ls|sort
ps|more
ps|sort
ps|sort|wc
ps|more|sort
ls|sort|wc
ps|sort|wc
ps|more|wc
ls|more|wc
ls|more|sort|wc
ls|sort|more|wc
ps|sort|more|wc
ps|more|sort|wc
Dir : /home/bm3001 : @bm3001 ls
SENT
Client Received message from server on socket 3, here is the message :
MaishaBhavickaClientPhaseNew.c
MaishaBhavickaServerPhaseNew.c
Makefile
ManyCommand.c
ODNewestProj.c
PhaseMaishaBhavicka.c
Pipe.c
c1
client
hello
server
workClient.c
workServer.c

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ps
SENT
Client Received message from server on socket 3, here is the message :
PID TTY      TIME CMD
4259 pts/1    00:00:00 server
7558 pts/1    00:00:00 bash
25985 pts/1   00:00:00 server
25986 pts/1   00:00:00 ps

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ls |wc
SENT
Client Received message from server on socket 3, here is the message :
13      13      181

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ls -lh|wc
SENT
Client Received message from server on socket 3, here is the message :
14      119     777

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ^C
Exiting client.
bm3001@DCLAP-V1156-CSD: ~$
```

```
Terminal 2 (Attacker):
Server Received message from client on socket 5, here is the message :
rmidr hello
handling new client in a thread using socket: 4
Listening to client....

Accepted New Socket, Waiting for New Client

Server Received message from client on socket 4, here is the message :
rmidr hello
handling new client in a thread using socket: 5
Listening to client....

Accepted New Socket, Waiting for New Client

Server Received message from client on socket 5, here is the message :
ps
handling new client in a thread using socket: 4
Listening to client....

Accepted New Socket, Waiting for New Client

Server Received message from client on socket 4, here is the message :
rmidr hello
^C
Exiting server.
bm3001@DCLAP-V1156-CSD: ~$
```

```
Terminal 3 (Victim):
SENT
Client Received message from server on socket 3, here is the message :
/home/bm3001

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ps
SENT
Client Received message from server on socket 3, here is the message :
PID TTY      TIME CMD
4259 pts/1    00:00:00 server
7558 pts/1    00:00:00 bash
24338 pts/1   00:00:00 server
24339 pts/1   00:00:00 ps

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 pwd2
SENT
Client Received message from server on socket 3, here is the message :
/home/bm3001

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 pwd
SENT
Client Received message from server on socket 3, here is the message :
/home/bm3001

Before closing sockAfter closing sock
Dir : /home/bm3001 : @bm3001 ^C
Exiting client.
bm3001@DCLAP-V1156-CSD: ~$
```

\*handled using signal.h header file

## How to Run

We've provided a Makefile with our two C files. Once you have downloaded them all in the same place, open two (or more) terminal windows. On one (or more), write `make client`, on the other, run `make server`. Make sure they are all in the same directory. After this, you can enter `./server` and `./client` to run the server and client respectively. Make sure to run the server first, otherwise the connection will fail. You can run `make clean` to get rid of the object files afterwards. Alternatively, you can also just run `make` at the beginning which will make all files and then run `./server` and `./client` on different terminals.

Running without a Makefile: You can run without a Makefile, you just need to have two or more terminals open, and then run both the `client.c` and the `server.c` in separate terminals. Make sure to run the server first. Give your CLI command inputs on the client terminal, which is processed in server and you see the results in client terminal. Then on the client terminal, you can enter the commands you want and you will get the outputs there. If you get the error "bind failed: Address already in use" enter "kill -9 \$(lsof -t -i:5564)" and rerun the commands.

### How to exit Client:

To get out of the terminal, use `exit` command. This will exit the current client and the server will wait for the connection of a new client.

Also, using `Ctrl+C` will exit the client or server\*

### Adding a New Client after Exiting from an old one:

Take out a separate terminal and run the `.c` file using `gcc -o c VWorkingClient.c`. Start using the new client without an issue. The same terminal as the former exited client might not work in the remote server. So, nevertheless, take out a new terminal everytime you add a new client.

\*handled using `signal.h` header file

The user defined functions used in the code:

```
void init_shell()
```

It prints the username of the PC running the shell for which we use the `getenv()` function.

```
void printDir()
```

Everytime to when the user is prompted to type a shell command, the directory of the shell file where the shell script is running gets printed.

```
commandList()
```

Prints all the available commands

```
void parseSpace(char* inputString, char** parsedArgs )
```

Parses the spaces in single line commands

```
void execArgs(char** parsedArgs, int newsocket)
```

This function executes the user commands for single line commands using `execvp` in a child process.

So, we pass a vector as well as the name of the command to `execvp` to run the command.

```
int parsePipe(char* inputString, char** parsedArgsPiped)
```

Parses the pipes in piped commands : We then pass our input string into a function which counts the number of pipes (“|”) in the input. If there are 0 pipes, which means it’s a normal separate command, it is passed into a functions which parses the input according to space and then these parsed arguments are passed into a method called `execArgs` which forks a child and uses `execvp` inside it to execute the commands.

In the case that there are pipes, we create pipe mechanisms to write the output of the first child to the read end of the next pipe/parent, whatever the case might be.

```
void execArgsPiped1(char** parsedArgsPiped, int newsocket)
```

\*handled using `signal.h` header file

This function executes the user commands for one piped commands using `execvp` in 2 separate child processes. So, we pass a vector as well as the name of the command to `execvp` to run the command. We redirect the output to the client using `dup2()`;

```
void execArgsPiped2(char** parsedpipe, int newsocket)
```

This function executes the user commands for two piped commands using `execvp` in 3 separate child processes. So, we pass a vector as well as the name of the command to `execvp` to run the command. We redirect the output to the client using `dup2()`;

```
void execArgsPiped3(char** parsedpipe, int newsocket)
```

This function executes the user commands for two piped commands using `execvp` in 4 separate child processes. So, we pass a vector as well as the name of the command to `execvp` to run the command. We redirect the output to the client using `dup2()`;

### **Server-Client Communication:**

The goal of this phase was to create a server which could provide remote accessibility to our shell. The client could send a request to the server and it would get the output from there. This is how we went about implementing that:

#### **Socket creation:**

```
int server_fd = socket(domain, type, protocol)
```

server\_fd: socket descriptor, an integer (like a file-handle)

domain: integer, specifies communication domain. Since we are communicating between processes on different hosts connected by IPV4, we use `AF_INET`

type: communication type `SOCK_STREAM`: TCP(reliable, connection oriented)

`SOCK_DGRAM`: UDP(unreliable, connectionless), we build a tcp socket, so use

\*handled using `signal.h` header file



SOCK\_STREAM.

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet

### **Bind:**

```
int bind(int server_fd, const struct sockaddr *addr, socklen_t addrlen);
```

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure).

### **Listen:**

```
int listen(int server_fd, int backlog);
```

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. In our code, we have used 10 because if a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

### **Accept:**

```
int new_socket= accept(int server_fd, struct sockaddr *addr, socklen_t *addrlen);
```

It extracts the first connection request on the queue of pending connections for the listening socket, server\_fd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

### **Reading the message:**

\*handled using signal.h header file

Here `int read(new_socket, message, 1024);` Reads the message from the client using socket `new_socket` and stores the string in the message.

### Stages for Client

**Socket connection:** Exactly same as that of server's socket creation

#### Connect:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The `connect()` system call connects the socket referred to by the file descriptor `sockfd` to the address specified by `addr`. Server's address and port is specified in `addr`.

#### Problems Faced and Solution:

1. *How to exit from Parent in the main function of the server.c, when the child in the main is handling every operation [here, we actually are terminating the functions with this specific client.c]?*

We added a global variable `flag` that becomes 1 in the child when `exit` has been typed, as it's a global variable, it's value in parent also gets updated and in parent we also close socket if `flag==1`, after which at the beginning of the loop which keeps server alive till infinity we add `flag=0` to initiate the start of a new client.

2. *Changes that allowed us to stop the break in client / server when we input wrong command:*

\*handled using `signal.h` header file

We forked a child in the main and the whole program of phase 2 is handled in the child, and we don't have any wait functions in the execArgs functions, aside from the parent in the main function

3. *Buffer Overflow and Buffer Not being emptied:*

We used the buffer polling technique of using the recv function to force read all the characters in the buffer to empty out the buffer before a new request is sent.

4. *Exiting with Ctrl+C:*

Using signal.h header file and signal(SIGINT, serverExitHandler) function handled this signal operation