جامعة نيويورك أبوظبي

NYU ABU DHABI

CS-UH 3010
Operating Systems, Spring 2025

Project Phase 2

Hariharan Janardhanan:hj2342
Moujahid Moussa:mm12515

Architecture and Design

In this project, we designed a Remote Command Line Interface (CLI) Shell application by enhancing our initial local-shell implementation to support remote command execution using TCP sockets. The overall structure follows a clear **Client-Server** model:

Client:

- Captures user commands.
- Sends commands via a TCP socket to the server.
- Receives execution results and displays them to the user.

Server:

- Listens for client connections using sockets.
- Receives commands from the client.
- Parses commands, executes them, and returns the results through the socket.

Key Design Choices

Separation of Concerns:

- We separated key functionalities into distinct modules for better maintainability:
- **Command Parser (command_parser.c)**: Handles parsing user commands and splitting multiple commands connected by pipes (|).
- **Command Executor (executor.c)**: Contains logic for executing single commands (execute_command) and piped commands (execute_piped_commands).
- **Socket Handling (main.c)**: Manages client-server communication via sockets, including setup, accepting connections, sending, and receiving data.

Transition from Local to Remote Shell:

- Initially, our CLI shell directly accepted user input locally via standard input (getline()) and executed commands locally (commented-out old design).
- The revised design incorporates TCP sockets, enabling remote execution by replacing local user input reading with socket-based communication.

Command Output Handling:

- Implemented pipes (pipe()) to capture command execution output, ensuring results are sent back clearly to the client.
- Utilized process creation (fork()) to isolate command execution, allowing the parent process to handle socket communication and output redirection separately.

Important Data Structures and Algorithms

Arrays of strings (char *commands[MAX_CMDS]):

- Chosen for simplicity and efficiency in handling multiple piped commands. Each array element holds one individual command parsed from user input.
- This simplified the logic needed to manage multiple commands for piped execution.

Buffers (char buffer[BUFFER_SIZE]):

- Used fixed-size character arrays as command and output buffers to streamline communication and avoid dynamic memory allocation complexity.
- Ensured sufficient buffer size (BUFFER_SIZE = 4096) to accommodate typical command outputs and error messages.

Sockets (TCP):

- TCP sockets were explicitly chosen for reliable command execution and data integrity, ensuring commands and their outputs reach their destinations without loss or corruption.

File Structure and Code Organization

Our codebase follows a straightforward and modular file organization:

```
OS_Project/
├── command_parser.c      // Handles parsing logic
├── command_parser.h
├── executor.c            // Handles command execution logic
├── executor.h
├── main.c                // Manages socket communication and main logic
├── client.c              // Simple socket client for remote shell interaction
├── Makefile              // Compilation and cleanup rules
```

Implementation highlights

Core Functionalities

Our primary goal was to execute shell commands remotely on a server machine, sending results back to the client clearly and robustly.

Specifically, our implementation achieved:

- client-server communication through **TCP sockets**.
- handling of shell command execution, including support for piped commands.
- Detailed logging for clear tracing of execution flow and error detection.
- Dynamic and explicit error handling, ensuring meaningful feedback to clients.

Important Functions and Algorithms

Our codebase is modularized into three main components:

1. Command Execution (executor.c):

- **execute_command(char *cmd)**

Executes single shell commands. It involves parsing the command, handling redirections, executing using execvp, and explicitly returning meaningful exit codes to communicate success (exit(0)) or specific failure (exit(127)).

- **execute_piped_commands(char **commands, int cmd_count)**

Executes multiple commands connected by pipes (|). It carefully manages inter-process communication via pipes (pipe()), ensuring each subprocess communicates correctly. It returns the final command's exit status explicitly to the parent, clearly showing whether execution succeeded or failed.

2. Command Parsing (command_parser.c):

- **parse_command**

Parses user input into executable arguments, handling special cases such as file redirections (<, >, 2>). It explicitly trims whitespace and checks for empty or invalid commands, significantly enhancing command reliability and user clarity.

3. Socket Communication and Execution Management (main.c):

The core logic involves:

- Initializing and managing TCP sockets (socket(), bind(), listen(), accept()).
- Continuously receiving client commands through sockets, clearly logging every step.
- Explicitly capturing command execution output and error streams via pipes.
- Clearly checking child-process exit statuses, dynamically handling errors (e.g., "command not found" cases) with structured logs ([INFO], [RECEIVED], [EXECUTING], [ERROR], [OUTPUT]).

Critical Code Snippets

Server-side Child-Process Execution (dynamic error handling):

```
    // Parent process: capture output and error dynamically
    close(pipefd[1]); // Close the write end of the pipe

    int status; // Declare a variable to hold the status of the child process
    waitpid(pid, &status, 0); // Wait for the child process to finish

    char output[BUFFER_SIZE] = {0}; // Declare a buffer to hold the output
    ssize_t output_len = read(pipefd[0], output, BUFFER_SIZE - 1); // Attempt to read the output from the pipe
    close(pipefd[0]); // Close the read end of the pipe

    // Dynamically handle errors based on the child's exit status
    if (WIFEXITED(status) && WEXITSTATUS(status) == 127) {
        printf("[ERROR] %s\n", output); // Print the error message if the child process exited with status 127
        write(client_fd, output, strlen(output)); // Send the error message to the client
        printf("[OUTPUT] Sending error message to client: \"%s\"\n", output); // Print message indicating an error mes
    } else {
        write(client_fd, output, output_len); // Send the output to the client
        printf("[OUTPUT] Sending output to client:\n%s\n", output); // Print message indicating output was sent to the
    }
}
```

This explicit handling ensures dynamic and clear feedback to the client about command execution status.

Error Handling and Edge Cases

Throughout development, we explicitly handled various edge cases and errors clearly:

- **Missing commands before or after pipes (| ls, ls |):**

Explicitly detected and handled these cases, sending clear error messages back to clients without terminating the connection:

```
// Handle errors detected during command parsing
if (error_detected || cmd_count == 0 || buffer[0] == '|') {
    char *error_message; // Declare a pointer to hold the error message

    // Determine the appropriate error message based on the error condition
    if (buffer[0] == '|') {
        error_message = "Error: Missing command before pipe.\n";
    } else if (cmd_count > 0 && strlen(commands[cmd_count - 1]) == 0) {
        error_message = "Error: Missing command after pipe.\n";
    } else {
        error_message = "Error: Empty command between pipes.\n";
    }

    fprintf(stderr, "[ERROR] %s", error_message); // Print the error message to the standard error
    write(client_fd, error_message, strlen(error_message)); // Send the error message to the client
    printf("[OUTPUT] Sending error message to client: \"%s\"\n", error_message);
    continue; // Skip to the next iteration of the loop
}
```

- **Empty commands between pipes (ls || grep):**

Explicitly detected empty commands after splitting by pipe and trimming, providing clear feedback.

- **Invalid commands:**

Explicitly returned “command not found” errors via a dedicated exit status (127), clearly logged and returned to the client.

Socket Communication

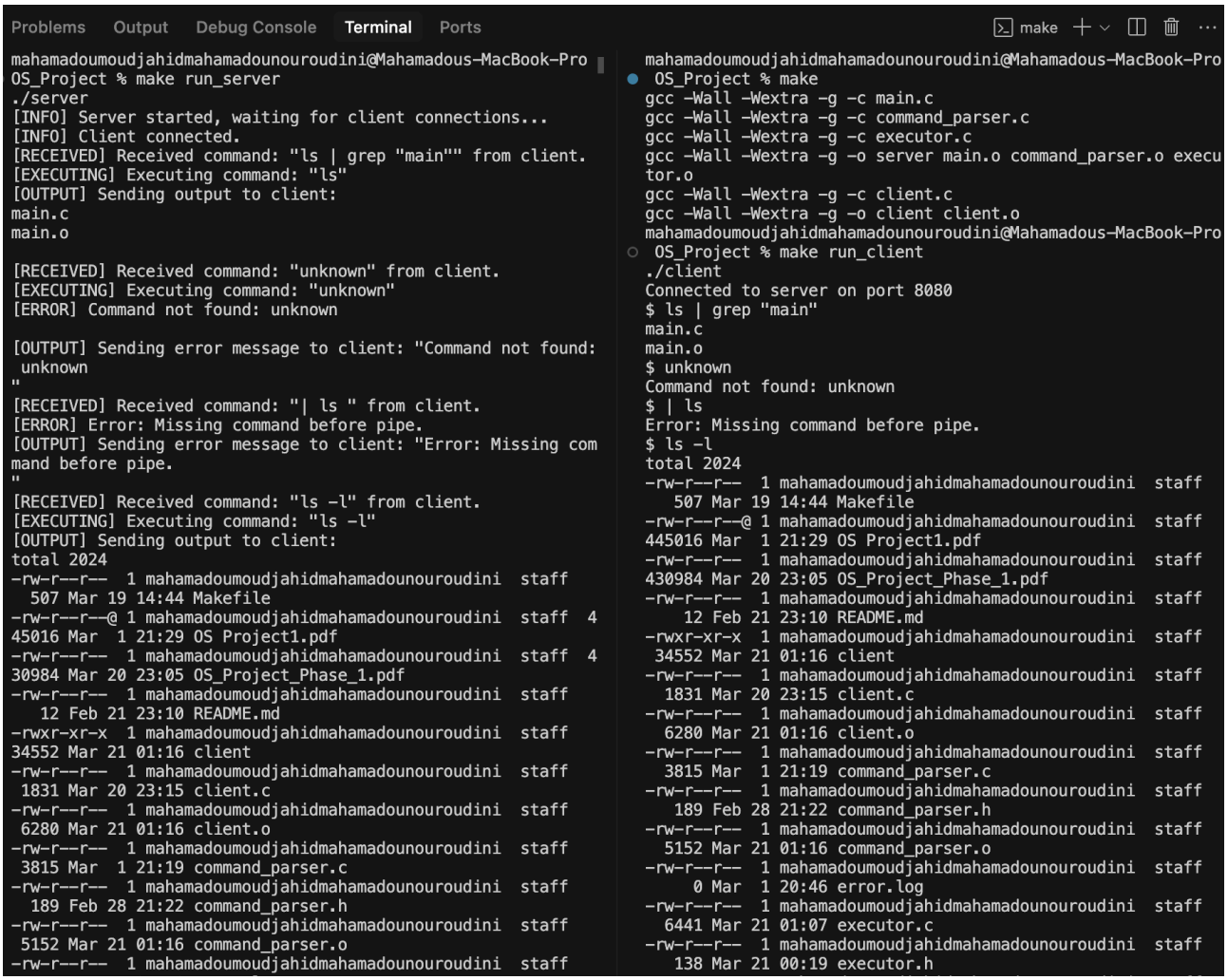
Our error-handling improved robustness:

- Each socket operation (socket(), bind(), listen(), accept(), read(), write()) explicitly checks for errors (perror()), clearly alerting us to any communication issues promptly.
- The server explicitly continues to serve new commands from the client until an “exit” command is explicitly received or a socket error occurs, ensuring clear and continuous operation.

Execution Instructions

1. **Compile the program**
   Open a terminal in the project directory and run: **make**
2. **Start the server**
   In the first terminal, start the server by running: **make run_server**
   The server should now be listening for incoming client connections.
3. **Start the client**
   Open another terminal and start the client by running: **make run_client**
   The client will establish a connection with the server.
4. **Interact with the client**
   - You will see a shell-like prompt where you can enter commands.
   - Type a valid shell command (e.g., ls, pwd) and press Enter.
   - The server will execute the command and return the output to the client.
   - If an invalid command is entered, an error message will be displayed.

Testing



Test 1: Piped command (ls | grep "main"):

- **Result: Correctly returned filtered filenames :**

main.c

Main.o

Test 2: Invalid command (unknown):

- **Result: Clearly returned explicit error:**

Command not found: unknown

Test 3: Edge case (| ls, missing command before pipe):

- **Result: Explicit error handled gracefully:**

Error: Missing command before pipe.

Test 4: Standard command (ls -l):

- **Result: Successfully displayed detailed directory listing as expected.**

On the server side we can see the logs for each operation

Challenges Faced

Handling Both Success and Failure ScenariosInitially, the server code worked well for success scenarios but did not handle command execution failures properly. Conversely, the error-handling version correctly detected failures but mistakenly flagged all commands as errors. Merging both functionalities while ensuring correct execution flow was a significant challenge.

Correctly Capturing Command OutputCapturing standard output and standard error from executed commands required careful redirection using pipes. Ensuring that output was properly read and forwarded to the client without truncation or corruption was a key technical hurdle.

Division of Tasks

- **Moujahid**
  - Developed command parsing and execution logic.
  - Implemented socket communication between client and server.
  - Handled inter-process communication using pipes.
  - Wrote the initial documentation and structured the report.
- **Hariharan Janardhanan**
  - Managed process creation and execution using fork() and exec().
  - Designed error handling mechanisms for both success and failure cases.
  - Debugged and tested edge cases to ensure robustness.
  - Implemented structured logging and debugging messages.

References

1. **Stack Overflow Discussions** – Various posts on handling pipes, command execution, and socket programming in C
2. **Official POSIX Documentation** – Used for ensuring compliance with system programming best practices