Trabalho Prático 1
Batalha de Naves – StarFleet Protocol

Disciplina de Redes de Computadores
Semestre 2025/2

Name – Seungbin Han

ID – 2025550850

Class - REDES DE COMPUTADORES - TN

1. Introduction

1-1. General Objective

The main objective of this project is to implement *Starfleet Protocol* (brief TCP-based game in terminal) using sockets on the Linux environment.

To satisfy the functional requirements, the client must transmit the user's input to the server, and the server must calculate the result based on the battle rules and send the feedback of the turn to the user. The final game message changes according to the ending situation, and the final inventory results also must be displayed.

To meet the non-functional requirements, it is necessary to program the game using POSIX C socket and the executables must be able to be run in Linux environment.

1-2. Source Code Configuration

The source code is organized into three subdirectories: *src_server*, *src_client*, *src_common* (for shared functions, structures and variables). The root *Makefile* compiles all source files into object files, reusing common objects to link and creates the final *bin/client* and *bin/server* executables.

2. Body

2-1. System Architecture

The server accepts client connections, randomly generates enemy actions (by *rand* and *srand(time(NULL))*, calculates the battle results, and sends the result to the client. The server is also responsible for all game state management, which includes tracking the inventory data required for the final report (by updating structure *Inventory*). The client is responsible for receiving user input, forwarding it to the server, and displaying the result received from the server.

2-2. Program Flow

After the connection between the client and server is established, the server sends an *MSG_INIT* to the client, delivering a welcome message and signaling the start of communication. Following that, the server sends an *MSG_ACTION_REQ* to the client, prompting it to receive user input and send back and *MSG_ACTION_RES.*

The server then calculates the battle result based on the input received from the client, updates the inventory, and sends feedback to the client. If at least one side's input is *HYPER_JUMP* or if either ship's HP drops to 0, the server sends *MSG_GAME_OVER* to signal that the game has ended. Finally, it sends the final inventory result with *MSG_INVENTORY* to client.

2-3 Communication Protocol Specification

The entire communication protocol is built by shared data structures and enumerations define in *src_common/util_common.h.* Below are the newly added structure and enumeration.

Inventory Structure

The server tracks game statistics using the *Inventory* structure. This data is sent to the client at the end of the game.

```
typedef struct
{
    int client_hp;
    int server_hp;
    int client_torpedoes;
    int client_shields;
    int total_turns;
} Inventory;
```

Action Enumeration

To improve code readability and avoid magic numbers, an *Action* enum defines the symbolic names for all possible choices:

```
typedef enum
{
    LASER_ATTACK,
    PHOTON_TORPEDO,
    SHIELDS_UP,
    CLOAKING,
    HYPER_JUMP,
    ACTION_CNT
} Action;
```

2-3. Game Logic Implementation

The server determines the outcome of each turn based on a rule based on the requirements. In the current implementation, this logic is handled by nested switch statements that compare the *client_action* against the *server_action.* While this approach is straightforward for the actions, it was recognized that it lacks scalability, maintainability and readability, which led to the suggestion of a lookup table approach in the “Possible Improvements” section below. *HYPER_JUMP* actions are checked before the *get_battle_result* function to ensure they are processed first, as required by the rules.

2-4. User Input Validation and Processing

When receiving user input, first, the program is designed to read the input as a c-string using *fgets* and then convert it to a number using *atoi* function. In this process, whitespaces in back and forth are removed using the *get_str_start_point* function (which strips spaces in front). To distinguish between cases where the *atoi* function's result is 0 the *stnrcmp* function is used to verify the original input. As a result, the program interprets inputs like "1 ", " 1 ", " 1 a " and "1a" all as the integer 1. To allow for future scalability, the client only performs a numeric check on the input, leaving the range validation to the server.

2-5. Implementing Reliable Communication

Although TCP provides a reliable byte-stream service, depending on network conditions, it may not send or receive the requested amount of data in a single operation. To address this, this program implements *send_reliable* and *recv_reliable* functions to ensure that data is consistently transmitted and received to match the exact size of the *BattleMessage* structure.

2-6. Exception Handling

The client is designed to close its file descriptor and exit the program when an exception occurs. The server, however, is designed to close the connection with the current client and wait for a new one on an exception, unless the error occurs during the initial socket setup. The *error_handling* function prints the error message, the location where the error occurred, and the *errno* value, if available. If either the server or client receives *SIGINT*, a signal handler closes the sockets to release resources before terminating the program. Additionally, the *SIGPIPE* signal, which occurs when sending to a broken pipe, was handled with *SIG_IGN* to prevent the program from terminating due to this interrupt.

3. Conclusion

3-1. Difficulties Encountered

Memory Management Issue

A segmentation fault occurred in the program, even though no dynamic memory allocation was used. After debugging with GDB, the cause was identified: the return address on the stack was being corrupted. This happened because the length parameter for *strncpy* was incorrectly passed as *sizeof(battleMessage)* instead of the correct *sizeof(battleMessage.message)*.

To reduct the possibility of this confusion, the *battleMessage* variable was renamed to *bm,* which significantly improved the code's readability.

Infinite Wait Issue

An infinite wait occurred because the return values of the *send* and *recv* functions were not

being checked. This caused the program to hang when either the server or client connection was terminated unexpectedly.

To fix this, the code was modified to *properly* handle the return values of those functions. Additionally, to prevent the server from waiting indefinitely on a dead connection, the *setsockopt* function with the *SO_KEEPALIVE* option was implemented. This allows the server to periodically check the validity of the TCP connection. Now, if the server detects that a client has disconnected, it will close that connection and prepare to accept a new client. The client will terminate the program on connection loss.

3-2. Possible Improvements

Although the current code calculates battle results using multiple switch statements, this approach is poor in terms of scalability, readability, and maintainability. By storing each outcome in a two-dimensional array, the result could be accessed simply with an expression like *result[client_action][server_action]*. If there are changes to the rules or new actions are added, the array could be modified by expanding it or changing its data. It would be much simpler than changing nested switch statements.

Additionally, while the *error_handling* function currently prints to the terminal, it could be modified to export error log files instead.

3-3. Project Summary

This project successfully implemented a functional, TCP-based client-server game, the *Starfleet Protocol*.' Main achievements include establishing reliable data transmission over a stream-based protocol, handling user input robustly, and managing game state on the server.