

## Homework 3: TCP Chat Server/Client

Submitted by:

Name: Amir Amber | ID: 207477753

Name: Yarin Edri | ID: 208076919

### 1. Overview

In this assignment, we implemented a text-based chat application using the TCP protocol in the Linux environment. The system consists of two executables: `hw3server` and `hw3client`. The server acts as a central hub that multiplexes connections from multiple clients, managing communication routing (broadcasting and private messaging) using non-blocking I/O.

### 2. Design & Architecture

The solution relies on the BSD Socket API and the `select()` system call to handle concurrency without using multi-threading.

- I/O Multiplexing (`select()`): Both the server and the client run in a single-threaded loop blocked by `select()`.
  - Server: Monitors the main listening socket (for new connections) and all active client sockets (for incoming messages) simultaneously.
  - Client: Monitors STDIN (keyboard input) and the server socket (incoming network data) simultaneously. This ensures the client can receive messages while the user is typing, without blocking.

### 3. Data Structures

The primary data structure used in the server is a global mapping array:

```
C  
#define MAX_FD 1024  
char *client_names[MAX_FD];
```

- Purpose: To map a socket's File Descriptor (integer) directly to the client's username (string).
- Efficiency: This allows for fast access time. When a message arrives on socket *i*, we immediately know the sender is `client_names[i]`.
- Memory Management: Names are allocated using `strdup` upon connection and freed using `free` upon disconnection to prevent memory leaks.

### 4. Implementation Details

#### 4.1. Connection & Handshake

1. Startup: The server binds to the specified port and listens.
2. Connection: When `select()` detects activity on the listener socket, `accept()` is called to create a new client socket.
3. Handshake: The client immediately sends its username upon connection. The server reads this initial message and stores it in the `client_names` array, printing the required log: `client <name> connected from <IP>`.

#### 4.2. Message Routing Protocol

The server processes incoming data based on the message content:

- Normal Messages:
  - If the message is standard text, the server formats it as `SenderName : Message`.
  - It iterates through all active file descriptors in the `readfds` set and uses `write()` to send the data to all connected clients (including the sender).
- Whisper Messages (@name):
  - The server parses the buffer to check if it starts with @.
  - It extracts the target name (string between @ and the first space).
  - It searches the `client_names` array for a match. If found, the message is sent only to that specific file descriptor.
- Exit Command (!exit):
  - The client sends the `!exit` string to the server before closing its socket locally.

- The server detects the disconnection (via `read()` returning 0), cleans up the resources, and prints `client <name> disconnected`.

## 5. Challenges & Solutions

- Non-Blocking Input: The main challenge was ensuring the client user interface did not freeze while waiting for server messages. This was solved by adding `STDIN_FILENO` to the client's `select()` set.
- Resource Cleanup: To avoid "zombie" connections or memory leaks, we ensured that `close()` and `FD_CLR()` are called immediately upon detection of a closed socket (`read` returning  $\leq 0$ ), along with freeing the allocated username string.

## 6. Compilation & Usage

The project includes a Makefile for automated building.

Compilation:

```
Bash  
make
```

Running the Server:

```
Bash  
.hw3server <port>  
# Example: ./hw3server 12345
```

Running the Client:

```
Bash  
.hw3client <server_ip> <port> <name>  
# Example: ./hw3client 127.0.0.1 12345 Amir
```