CMPUT 313 – Computer Networks **Winter 2025**

Assignment 1 – Socket Programming and Protocol Design

This assignment is due **February 11, 2025, at 5:00pm** Mountain Time.

Objective

This programming assignment provides hands-on experience in using the socket API and enhance your understanding of the TCP transport.

Playing Tic-Tac-Toe over the Internet

Tic-tac-toe is a two-player game played on a three-by-three grid. The players take turns marking the spaces in the grid with `X` or `O`, with `X` being used by the starting player. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. If the board is filled in and neither player has won, the game is a draw. The game ends when either one player wins or the game is drawn. Note that a player may choose to play on a position that has already been claimed by them or the other player. In that case the player's move is ignored and they are notified to choose a new position.

This assignment has two parts. In each part, you will design a protocol on top of TCP allowing people to play this game over the Internet. Additionally, in Part I, you will implement a server (also acting as Player 1) that plays the game with a client (Player 2) following this protocol. In Part II, you will implement a concurrent game server that allows multiple pairs of clients to play the game simultaneously in different rooms. Two clients playing the game against each other are assumed to be in the same room, identified by a small integer. The game server is an intermediary between the clients in this part, as they cannot communicate with each other directly.

Your protocols must define the syntax and semantics of the messages exchanged between a client and the server, and the actions that must be taken after receiving each message by the respective party.

Part I: Server acting as a player

The server, acting as Player 1, listens on a port to receive a connection request from a client who wishes to play the game with the server. Assuming that the server is not playing Tic-Tac-Toe with another client, the connection request from this client (Player 2) is accepted and the game starts. You are free to set Player 1 as the starter player or choose the starting player randomly. You may assume that the client knows the port number that the server uses to accept connections (the port number is passed as a command line argument to your program as described later). Once the game ends, the server closes the connection with the client, and both will exit subsequently.

The server is implemented in `server-p1.cpp` and the client is implemented in `client-p1.cpp`. See the Starter Code section for details.

Consider the following questions when designing the protocol for Part I:

- How does the server inform the client who established a connection that the game can begin?
- How does the server communicate initial game information, e.g., the starting player, empty game board, to the client?
- How does the client indicate the position they want to mark?
- How does the server communicate the current state of the game board to the client?

- How does the server handle invalid moves or actions (e.g., making a move out of turn, selecting an invalid or marked position)?
- How does the server notify the client when the game ends due to a win, draw, or client disconnection or inactivity? Suppose the server will close the connection if the client does not make a move for 30 seconds.

Each message/command in the designed protocol needs to be precisely specified in the README file.

Part II: Concurrent server acting as an intermediary

In this part, the server is an intermediary between the clients that play against each other. It is a concurrent server meaning that it can communicate with multiple clients concurrently using the same port. To this end, once a client connects, the server creates a thread to handle communications with that client and continues listening to the port. By specifying different `roomID`s, multiple pairs of clients can play Tic-Tac-Toe simultaneously.

Your task is to establish client-server connections using TCP sockets, handle message passing between clients and the server, and manage game state synchronization across sockets given the `roomID`. Assume that the `roomID` is a small integer between `1` and `5`, and that clients know the port number that the server uses to accept connections.

To join a room, a player has to specify the `roomID` of that room. Examples are provided in the following section: Game Mechanics. The first player that joins a room is called Player 1 and the player than joins this room later is called Player 2. As mentioned in Part I, you are free to set Player 1 as the starter player or choose the starting player randomly.

You can assume that once a player has joined a room (e.g., room 1), they cannot leave or join another room. When the game ends, the server closes the connection with both clients, and these clients will exit subsequently. However, the server continues running, as there might be other active rooms or future connection requests.

The server is implemented in `server-p2.cpp` and the implementation of clients is in `client-p2.cpp`. See the Starter Code section for details.

Consider the following questions when designing the protocol for Part II:

- How does a client tell the server that they want to join a room with the given `roomID`?
- How does the server confirm to a client that their request to join a room was accepted? How does the server notify a client if an attempt to join a room fails, i.e. the room is full already? Note that a room is considered full when it contains two players.
- How does the server inform a client that another player has joined and the game is ready to start?
- How does the server communicate initial game information (e.g., the starting player, empty game board) to both clients?
- How does the server communicate the current state of the game board to clients after each move?
- How does the server handle invalid moves or actions (e.g., making a move out of turn or selecting an invalid or marked position)?
- How does the server notify the clients when the game ends due to a win, draw, or client disconnection or inactivity? Suppose the server will close the connection if the client does not make a move for 30 seconds.

Each message/command in the designed protocol needs to be precisely specified in the README file.

Game Mechanics

Part I

Let us assume that the client (Player 2) is selected as the starting player. Once the game starts, Player 2 claims a board position between 1 and 9 by entering `MARK <int>` in the terminal. This move is then sent to the server, which updates the board by marking this position as being owned by Player 2 and claims an unmarked board position itself. It then sends the updated board to the client. This continues until the game ends. Once the game ends, the server closes the TCP connection with the client and exits.

Below is an example of what each player sees in their terminal window.

Player 2:

```
You go first
> MARK 2
-------------
| O | X | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
> MARK 1
Try again
> MARK 8
-------------
| O | X | 3 |
-------------
| O | 5 | 6 |
-------------
| 7 | X | 9 |
-------------
> MARK 5
You won
-------------
| O | X | 3 |
-------------
| O | X | 6 |
-------------
| 7 | X | 9 |
-------------
```

Player 1:

```
Game started
-------------
| 1 | X | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
> MARK 1
-------------
| O | X | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | X | 9 |
-------------
> MARK 4
You lost
-------------
```

```
| O | X | 3 |
-------------
| O | X | 6 |
-------------
| 7 | X | 9 |
-------------
```

Part II

The game mechanics are similar to Part I with only two differences. First, two clients must join a room by entering `JOIN <int>` in the terminal, before the game can begin. The server notifies the clients once this condition is met. Second, once the game ends in a room, the server closes the TCP connections with both clients and marks the respective room as available so that it can be used by other clients in the future. The server does not exit even if there is no active room.

Let us assume that Player 1 is selected as the starting player. Below is an example of what each player sees in their terminal window.

Player 1:

```
> JOIN 1
Joined room 1
Waiting for player 2
Player 2 joined
You go first
> MARK 3
-------------
| 1 | 2 | X |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | O |
-------------
> MARK 8
-------------
| 1 | 2 | X |
-------------
| 4 | O | 6 |
-------------
| 7 | X | O |
-------------
> MARK 2
You lost
-------------
| O | X | X |
-------------
| 4 | O | 6 |
-------------
| 7 | X | O |
-------------
```

Player 2:

```
> JOIN 1
Joined room 1
Game started
-------------
| 1 | 2 | X |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
```

```
> MARK 9
-------------
| 1 | 2 | X |
-------------
| 4 | 5 | 6 |
-------------
| 7 | X | O |
-------------
> MARK 5
-------------
| 1 | X | X |
-------------
| 4 | O | 6 |
-------------
| 7 | X | O |
-------------
> MARK 1
You won
-------------
| O | X | X |
-------------
| 4 | O | 6 |
-------------
| 7 | X | O |
-------------
```

Starter Code

You will find two directories in the cloned repository: one containing the starter code in C and the other one containing the starter code in C++. Depending on which language you choose, you should edit the files in the respective directory. For example, the following files are provided in the C++ directory: `ttt.cpp, ttt.h, client-p1.cpp, server-p1.cpp, client-p2.cpp, server-p2.cpp`.

Files `ttt.cpp` and `ttt.h` contain the source code of a simple, 2-player, terminal tic-tac-toe game. You should find the basic functionalities needed for the game here, including a function for printing the board that we have implemented for you. You are free to add additional `struct, class` or `function` as you see fit.

In files `client-p1.cpp, server-p1.cpp, client-p2.cpp, server-p2.cpp`, we have provided function headers for functionalities that we expect in your protocol and final implementation. You may use those as a guide when coding your implementation. We have also showed how you may call the `drawBoard()` function from `ttt.cpp`.

Lastly, we have implemented an `input_parser()` function that parses the user inputs used to join and play the game, i.e., `MARK <int>` for Part I and `MARK <int>, JOIN <int>` for Part II. These inputs will be used to test your submission. Your programs should always process user inputs with the `input_parser()` function.

Mutual Exclusion: In `server-p2.cpp`, you will need to use a lock, e.g. a `pthread_mutex_t`, to ensure mutual exclusion when accessing and modifying variables shared among threads, such as `player1Socket, player1Socket,` and `numActivePlayers`. The lock needs to be acquired before accessing the shared variables and released afterwards so other threads are not blocked indefinitely.

Makefile: You should use the Makefile provided.

Running Your Code

To test your submission for Part I, run the two commands below in separate terminal windows:

```
$ ./p1server 10000
$ ./p1client 127.0.0.1 10000
```

In the above example `10000` is the server port number and `127.0.0.1` is its IP address. Thus, the IP address and port number of the server will also be passed as command line arguments to the main function in `client-p1.cpp`. The server port number will be passed as a command line argument to the main function in `server-p1.cpp`. Notice that the client port number is not provided, as it should be chosen by the operating system.

Similarly, to test your submission for Part II, run the following commands in separate terminal windows:

```
$ ./p2server 10000
$ ./p2client 127.0.0.1 10000
$ ./p2client 127.0.0.1 10000
```

Submission

Submit all the required files via GitHub Classroom. Do not use compressed files or archives.

The following files must be included in your repository.

1. The provided Makefile. You might add more targets to this Makefile, if needed.
2. A plain text document, called `README.md` (use this exact name) in the root directory of the cloned repository, that contains a header (see below), specifies the language used for coding (C or C++), explains the design of your protocol for each part (i.e., messages that can be sent and actions taken upon receiving these messages) and elaborates on how you tested your implementation. Make sure you cite all sources that you used in this file. You may use a markup language, such as Markdown, to format this plain text file.

   At the very top of the README file before any other lines, include **a header** with the following information: **your name, student ID (SID), and CCID**. For example:

   ```
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   # Name : Jane Doe
   # SID : 1234567
   # CCID : jdoe
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   ```

3. All files required to build your project including the header file and your C/C++ files for both parts.

Grading

Part 1: 50% Correctness: 75%, Protocol design and description: 25%

Part 2: 50% Correctness: 75%, Protocol design and description: 25%

In both parts, we expect your protocol description to satisfy the following requirements:

- Describes syntax and semantics of the messages exchanged between client and server, and the actions that must be taken upon receiving each message by the respective party.
  - Do not over-complicate your protocol. Marks may be deducted if it is overly complicated.
- Provides justification for your design choices

For each part, correctness marks are given if your client-server application satisfies basic functionality, and handles errors and timeouts as described earlier.

Miscellaneous Notes

- This assignment must be completed individually without consultation with anyone. Questions can be asked from TAs in the labs.
- Your code must be written in C (C11 standard) or C++ (C++11 standard), and compile cleanly using `gcc` or `g++` with `-Wall` flag, i.e. producing no warning or error. Marks will be deducted if there are warnings or errors.
- You must use POSIX sockets API in your implementation. For concurrency support in Part II, use POSIX threads and mutex lock from `<pthread.h>`. If you write code in C++, it is also possible to use the thread class in C++ from `<thread>`. However, we only show examples for the POSIX threads library in the lab.
- A POSIX mutex lock can be acquired and released by a thread using `pthread_mutex_lock` and `pthread_mutex_unlock` respectively.
- If you are running the client and the server on the same machine, use `localhost` or `127.0.0.1` as the host name/IP address, and a high port number, e.g. between `10000` and `60000`, for the server. Do not use port numbers in the range of `0-1023`. These are reserved ports. Test your client/server code by running as non-privileged user. This will allow you to capture reserved port restrictions from the kernel.
- We suggest that you draw a time diagram of the game when designing the protocol. This diagram has three vertical lines corresponding to two clients and the server. A horizontal line from a client to the server or the server to a client represents a message exchange. In this time diagram, show the messages that need to be exchanged to play Tic Tac Toe.
- We will compile and test your submission in the Ubuntu VM that we provided. So make sure that your code compiles and runs correctly in the VM.
- You are required to use git to track the progress of your work. You will not receive any marks if the git history shows less than 3 commits after the initial commits by GitHub Classroom.