Michigan State University
Computer Science & Engineering Department
CSE422 Computer Networks, Spring 2017

Laboratory 1: Introduction to Socket Programming

Due: 23:59 Thursday, February 9, 2017

1 Goal

Gain experience with socket programming by implementing a simple game (Tic-Tac-Toe) using both UDP and TCP sockets. In this game, two players take turns marking spaces in a 3 x 3 grid. The player who first successfully marks all spaces in a row, column or diagonal wins the game.

2 Overview

In this lab, you will implement a distributed version of Tic-Tac-Toe, which will comprise two C++ programs, a client program and a server program. Two players each run a client program that interacts with the server program to simulate the gameplay. This lab will help you to gain experience with socket programming using the Berkeley socket interface. In order to focus on the details of socket programming, various aspects of the game and most of the command parsing are provided to you as skeleton code. [link] (Note: Using the skeleton code is not mandatory.)

This lab is worth 5% of the final grade and is composed of 50 points. This lab is due at 23:59 (11:59 PM) on Thursday, February 9, 2017. **No late submission will be accepted**. You will submit your lab using the CSE *handin utility* (http://secure.cse.msu.edu/handin/).

3 Specification

In this lab, you are required to implement a client program for the game Tic-Tac-Toe. The complete server program is provided to you in the skeleton code. The server program and client program will interact by exchanging messages to simulate the gameplay. The server supports a single pair of clients. A simple message/packet format and details of a simple protocol are provided in the file: `MyPacket.h`. The server and client programs will handshake through a TCP connection and the gaming interaction will be based on UDP messages. (Although UDP is an unreliable protocol, for this lab your program does not need to tolerate lost packets.)

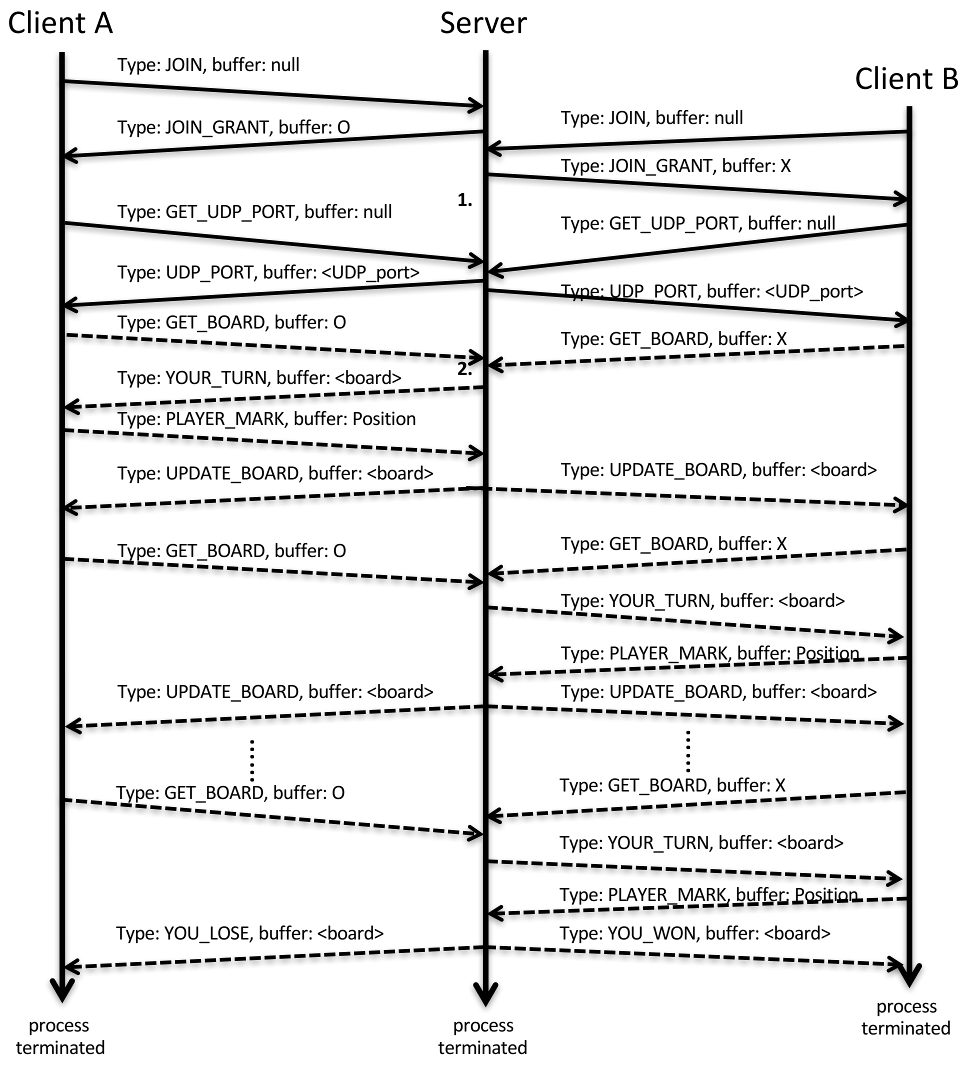
Client A
Server
Client B
Type: JOIN, buffer: null
Type: JOIN_GRANT, buffer: O
Type: JOIN, buffer: null
Type: JOIN_GRANT, buffer: X
1.
Type: GET_UDP_PORT, buffer: null
Type: GET_UDP_PORT, buffer: null
Type: UDP_PORT, buffer: <UDP_port>
Type: UDP_PORT, buffer: <UDP_port>
Type: GET_BOARD, buffer: O
Type: GET_BOARD, buffer: X
2.
Type: YOUR_TURN, buffer: <board>
Type: PLAYER_MARK, buffer: Position
Type: UPDATE_BOARD, buffer: <board>
Type: UPDATE_BOARD, buffer: <board>
Type: GET_BOARD, buffer: O
Type: GET_BOARD, buffer: X
Type: YOUR_TURN, buffer: <board>
Type: PLAYER_MARK, buffer: Position
Type: UPDATE_BOARD, buffer: <board>
Type: UPDATE_BOARD, buffer: <board>
Type: GET_BOARD, buffer: O
Type: GET_BOARD, buffer: X
Type: YOUR_TURN, buffer: <board>
Type: PLAYER_MARK, buffer: Position
Type: YOU_LOSE, buffer: <board>
Type: YOU_WON, buffer: <board>
process terminated
process terminated
process terminated

Figure 1: The protocol for lab 1. The thick vertical lines denote processes. The solid/dashed lines denote communication through a TCP/UDP socket. 1. The server program creates a UDP socket. 2. The server determines which client takes the first turn.

3.1 Tic-Tac-Toe Gameplay

The server program hosts a Tic-Tac-Toe game for one pair of client programs (players). The server program hosts a "board" consisting of a 3 x 3 grid of spaces. Initially all the spaces are empty. One player is assigned 'O' and the other is assigned 'X'. The server program sends the board to the client programs and also informs one of the client programs that it is its turn. That client program marks a space by sending a message to the server program indicating which space it wants to mark. Upon receiving the message from the client program, the server updates the grid and sends the updated board to both players. The server program informs the other client program that it is its turn when responding to the next `GET_BOARD` message. This board marking continues until one player wins the game by marking all spaces in a row, column or diagonal, or until the game ends in a draw, where spaces are marked without a winner.

A simple implementation of the non-networked aspect of Tic-Tac-Toe game is provided in `TicTacToe.*`.

3.2 File `MyPacket.h`: packet format and protocol

The packet format for this lab is defined as having only two fields: a) message type: `unsigned int` type and b) message buffer: `char buffer[16]`. The protocol uses of several message types as defined in `MyPacket.h`. The operation of the protocol is visualized in Figure. 1.

3.3 The server program

Example invocation: `./lab1Server`

The server program takes no argument. This program is responsible for listening for incoming client requests and handling the Tic-Tac-Toe gameplay for the clients.

The server program creates a TCP socket and waits for incoming TCP connections. This TCP socket's port will be assigned by the operating system and printed to the console. We assume that the clients know this port number, because the clients are started after the server. When a client connects to the server, it sends a `JOIN` message to the server, and the server assigns the mark (either 'O' or 'X') to the client via a `JOIN_GRANT` message.

After two clients have joined the game, the server program creates a UDP socket, whose port number is also assigned by the operating system. The server program waits for a `GET_UDP_PORT` message from each client program. The server program discards **ANY** message that is not of type `GET_UDP_PORT`, and in this case, terminates the program. If the incoming packet's type is `GET_UDP_PORT`, the server program responds a message of type `UDP_PORT` via the TCP socket. The `UDP_PORT` packet contains the UDP port number in the buffer.

The game (marking the grid, sending the board and exiting) is played using the UDP socket. After both client programs get the UDP port, they start playing the game by sending a `GET_BOARD` message. The `GET_BOARD` message's buffer comes with the mark assigned to the client. The server distinguishes the clients by their marks (either 'O' or 'X'). The server program determines whose turn it is and responds with a `YOUR_TURN` message to that client.

Suppose the two clients are denoted as client A and client B, and client A takes the first turn. The server does not send any message to client B before client A has finished her turn. When client A receives a `YOUR_TURN` message, the player will be prompted and allowed to mark a space. The player can specify which position she wants to mark. For example, if the player wants to mark on position e by entering the command `MARK e`. The client program A sends a `PLAYER_MARK` message containing the position to be marked to the server program. The server program checks if client A has won the game and responds according to the game result. If client A has won the game, the server sends a `YOU_WIN` message to client A and sends a `YOU_LOSE` message to client B. If the game ties after client A's move, the server sends `TIE` message to both client A and B. If client A has not won the game, the server sends an `UPDATE_BOARD` message including the updated board to **both** client A and client B. When client A and client B receive the `UPDATE_BOARD` message, both send a `GET_BOARD` message for next turn. The server sends `YOUR_TURN` to client B this time. These steps repeat until one of the clients wins or the game ties when all spaces are marked. Moreover, when a client sends an `EXIT` message, the server grants this exit, returns an `EXIT_GRANT` to `both` clients and terminates the server program.

The complete server program is provided in the skeleton code.

3.4 The client program

Example invocation: `./lab1Client -p 48192 -s homer.cse.msu.edu`

The client program is required to accept the following arguments.

- `-s` is the server address (domain name or IP address).
- `-p` is the TCP port number that the server listen for incoming connection.

The client resolves the server address using `gethostbyname()` and connects to this server over TCP. The client then sends a new game request `JOIN` to the server, in order to inform the server that she wants to play the game. The server responds with a `JOIN_GRANT` message to the client along with the mark ('O' or 'X') assigned to the client. As mentioned in previous section, when two clients have connected to the server, the server program creates a UDP socket. Both client programs send a `GET_UDP_PORT` message to get the UDP port number.

After the clients obtain the UDP port number, a game is created at the server side. The client programs send `GET_BOARD` messages to start a turn. The server determines who gets the first turn and responds to that client program with a `YOUR_TURN` message. On receiving a `YOUR_TURN` message, the client program prompts the player and asks which position she wants to mark. When the player enters the position she wants to mark, the client checks the validity of the move. If it is valid, the client program sends a `PLAYER_MARK` message to the server. The server responds with different kinds of messages according to the game status, as mentioned in the previous section.

Please note that the `JOIN` message is sent **by the client program automatically** right after the client's TCP connection to the server is established. The player does **NOT** need to issue a `JOIN` command and neither does the player need to issue `GET_UDP_PORT`. The player can issue only two types of commands: `MARK` and `EXIT`. Please note that the player can only issue commands when it is her turn. The parsing function `get_command()` accepts only those two commands and is provided in `lab1Client.h`.

A skeleton client code is provided in `lab1Client.cc`. Several utility functions, including command parsing and argument parsing, are provided in `lab1Client.h`.

4 Guidelines

Working alone or in pairs. You may work on this assignment individually or in pairs (not in groups of 3 or more, however). If you prefer to work in a pair, **both** students must submit a copy of the solution and identify their MSU NetID in a README file. If you prefer to work individually, please clearly state that you are working individually and include your MSU NetID in the README file.

Programming Language. You must implement this project using `C++`. The skeleton code is written in `C++`. Please clearly state the command to compile your project submission in your README file.

Testing your code. Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge`, `mcclure`, `apu`, `krusty`, `rod` or `skinner`. A statement must be provided in the README file's header. You will **not** be awarded any credit if your lab submission compiles on none of those machines.

5 Deliverables

You will submit your lab using the CSE *handin utility* (http://secure.cse.msu.edu/handin/) . Please submit all files in your project directory. If you start your lab with the skeleton code, submit all files, even if the file is not modified.

This lab is due no later than 23:59 (11:59 PM) on Thursday, February 9, 2017. **No late submission will be accepted.**

The compilation must be done using a makefile, which is also provided in the skeleton code.

The code should compile and link on CSE machines in 3353 EB. You will not be awarded any points if your submission does not compile using makefile. Please test your programs before handing them in.

A README file is required. The README file should include a header, sample output and any relevant comments. Please provide the following items in header of the README: whether you are working individually or in a pair, the MSU NetIDs of the submitting students, a list of machines on which you have compiled your code, the command used to compile your code. Two sample README file headers are as follows:

```
Student NetID: alice999, I am working with bob99999.
Compilation tested on: ned, skinner, marge ...
Command for compile: gcc proj1.c -o proj1
```

```
Student NetID: doejohnQ, I work on this project individually.
Compilation tested on: ned
Command for compile: make
```

6 Example

Follows is an example of output from the client and server. Your output may differ.

1. Invoke the server
```
>./lab1Server
[TCP] Tic-Tac-Toe server started...
[TCP] Port: 35250:
```

2. Invoke the first client. The client connects to the server. The server responds with a `JOIN_GRANT` message and assigns the mark `O` to first client. The first client sends a `GET_UDP_PORT` right after it gets `JOIN_GRANT`.
First client:
```
>./lab1Client -s carl.cse.msu.edu -p 35250
[TCP] Tic Tac Toe client started...
[TCP] Connecting to server: carl.cse.msu.edu:35250
[TCP] Sent: JOIN
[TCP] Rcvd: JOIN_GRANT O
[TCP] Sent: GET_UDP_PORT
```

Server:

```
>./lab1Server
[TCP] Tic-Tac-Toe server started...
[TCP] Port: 35250
[TCP] Recv: JOIN
[TCP] Sent: JOIN_GRANT O
```

3. Invoke the second client. The client connects to the server. The server responds with a `JOIN_GRANT` message and assigns the mark `X` to second client.
   Second client:
   ```
   >./lab1Client -s carl.cse.msu.edu -p 35250
   [TCP] Tic Tac Toe client started...
   [TCP] Connecting to server: carl.cse.msu.edu:35250
   [TCP] Sent: JOIN
   [TCP] Rcvd: JOIN_GRANT X
   [TCP] Sent: GET_UDP_PORT
   ```

   First client:
   ```
   >./lab1Client -s carl.cse.msu.edu -p 35250
   [TCP] Tic Tac Toe client started...
   [TCP] Connecting to server: carl.cse.msu.edu:35250
   [TCP] Sent: JOIN
   [TCP] Rcvd: JOIN_GRANT O
   [TCP] Sent: GET_UDP_PORT
   ```

   Server:
   ```
   >./lab1Server
   [TCP] Tic-Tac-Toe server started...
   [TCP] Port: 57775
   [TCP] Recv: JOIN
   [TCP] Sent: JOIN_GRANT O
   [TCP] Recv: JOIN
   [TCP] Sent: JOIN_GRANT X
   ```

4. Right after the second client connects, the server program creates a UDP socket. It receives the two `GET_UDP_PORT` messages and respond with the UDP port to the clients.
   Server:
   ```
   ......
   [TCP] Sent: JOIN_GRANT X
   [UDP:45481] Gameplay server started.
   [TCP] Recv: GET_UDP_PORT
   [TCP] Sent: UDP_PORT 45481
   [TCP] Recv: GET_UDP_PORT
   [TCP] Sent: UDP_PORT 45481
   ```

5. Both clients receive the `UDP_PORT` message and both clients send `GET_BOARD` message. The first client gets the first turn. The second client does not get any response and it is blocked.

   First client:

   ```
   ......
   [TCP] Sent: GET_UDP_PORT
   [TCP] Rcvd: UDP_PORT 45481
   [UDP] Sent: GET_BOARD O
   [SYS] Waiting for response ...
   [UDP] Rcvd: YOUR_TURN _________
   |---|---|---|
   | a | b | c |
   |---+---+---|
   | d | e | f |
   |---+---+---|
   | g | h | i |
   |---|---|---|
   [SYS] Your turn.
   [CMD]
   ```

   Second client:

   ```
   ......
   [TCP] Rcvd: JOIN_GRANT X
   [TCP] Sent: GET_UDP_PORT
   [TCP] Rcvd: UDP_PORT 45481
   [UDP] Sent: GET_BOARD X
   [SYS] Waiting for response ...
   ```

   Server:

   ```
   ......
   [UDP:45481] Gameplay server started.
   [TCP] Recv: GET_UDP_PORT
   [TCP] Sent: UDP_PORT 45481
   [TCP] Recv: GET_UDP_PORT
   [TCP] Sent: UDP_PORT 45481
   [UDP:45481] Rcvd: GET_BOARD O
   [UDP:45481] Sent: YOUR_TURN _________
   [UDP:45481] Rcvd: GET_BOARD X
   ```

6. The first client marks position e.
   First client:

```
......
[SYS] Waiting for response ...
[UDP] Rcvd: YOUR_TURN _________
|---|---|---|
| a | b | c |
|---+---+---|
| d | e | f |
|---+---+---|
| g | h | i |
|---|---|---|
[SYS] Your turn.
[CMD] MARK e
|---|---|---|
| a | b | c |
|---+---+---|
| d | O | f |
|---+---+---|
| g | h | i |
|---|---|---|
[UDP] Sent: PLAYER_MARK e
[UDP] Rcvd: UPDATE_BOARD ____O____
```

Second client:

```
......
[SYS] Waiting for response ...
[UDP] Rcvd: UPDATE_BOARD ____O____
[UDP] Sent: GET_BOARD X
[SYS] Waiting for response ...
[UDP] Rcvd: YOUR_TURN ____O____
|---|---|---|
| a | b | c |
|---+---+---|
| d | O | f |
|---+---+---|
| g | h | i |
|---|---|---|
[SYS] Your turn.
[CMD]
```

Server:

```
......
[TCP] Sent: UDP_PORT 45481
```

```
[UDP:45481] Rcvd: GET_BOARD O
[UDP:45481] Sent: YOUR_TURN _________
[UDP:45481] Rcvd: GET_BOARD X
[UDP:45481] Rcvd: PLAYER_MARK e
[UDP:45481] Sent: UPDATE_BOARD ____O____
[UDP:45481] Sent: UPDATE_BOARD ____O____
[UDP:45481] Rcvd: GET_BOARD O
[UDP:45481] Rcvd: GET_BOARD X
[UDP:45481] Sent: YOUR_TURN ____O____
```

7. The second client marks position a.
   Second client:

```
......
[SYS] Your turn.
[CMD] MARK a
|---|---|---|
| X | b | c |
|---+---+---|
| d | O | f |
|---+---+---|
| g | h | i |
|---|---|---|
[UDP] Sent: PLAYER_MARK a
[UDP] Rcvd: UPDATE_BOARD X___O____
[UDP] Sent: GET_BOARD X
[SYS] Waiting for response ...
```

   First client:

```
......
[SYS] Waiting for response ...
[UDP] Rcvd: UPDATE_BOARD X___O____
[UDP] Sent: GET_BOARD O
[SYS] Waiting for response ...
[UDP] Rcvd: YOUR_TURN X___O____
|---|---|---|
| X | b | c |
|---+---+---|
| d | O | f |
|---+---+---|
| g | h | i |
|---|---|---|
[SYS] Your turn.
[CMD]
```

8. The client keeps playing until a winner is generated or the game is a tie.

   First client:

```
......
[UDP] Rcvd: YOUR_TURN ____OX_OX
|---|---|---|
| a | b | c |
|---+---+---|
| d | O | X |
|---+---+---|
| g | O | X |
|---|---|---|
[SYS] Your turn.
[CMD] MARK b
|---|---|---|
| a | O | c |
|---+---+---|
| d | O | X |
|---+---+---|
| g | O | X |
|---|---|---|
[UDP] Sent: PLAYER_MARK b
[UDP] Rcvd: YOU_WON _O__OX_OX
Congratulations! You won!
```

   Second client:

```
......
[CMD] MARK i
|---|---|---|
| a | b | c |
|---+---+---|
| d | O | X |
|---+---+---|
| g | O | X |
|---|---|---|
[UDP] Sent: PLAYER_MARK i
[UDP] Rcvd: UPDATE_BOARD ____OX_OX
[UDP] Sent: GET_BOARD X
[SYS] Waiting for response ...
[UDP] Rcvd: YOU_LOSE _O__OX_OX
Too bad! You lose!
```

9. A player might exit the game in her turn.

   First client:

```
......
>./lab1Client -s carl.cse.msu.edu -p 59748
[TCP] Tic Tac Toe client started...
[TCP] Connecting to server: carl.cse.msu.edu:59748
[TCP] Sent: JOIN
[TCP] Rcvd: JOIN_GRANT O
[TCP] Sent: GET_UDP_PORT
[TCP] Rcvd: UDP_PORT 37854
[UDP] Sent: GET_BOARD O
[SYS] Waiting for response ...
[UDP] Rcvd: YOUR_TURN _________
|---|---|---|
| a | b | c |
|---+---+---|
| d | e | f |
|---+---+---|
| g | h | i |
|---|---|---|
[SYS] Your turn.
[CMD] EXIT
[UDP] Sent: EXIT
[UDP] Rcvd: EXIT_GRANT
You have left the game. You lose!
```

Second client:

```
......
>./lab1Client -s carl.cse.msu.edu -p 59748
[TCP] Tic Tac Toe client started...
[TCP] Connecting to server: carl.cse.msu.edu:59748
[TCP] Sent: JOIN
[TCP] Rcvd: JOIN_GRANT X
[TCP] Sent: GET_UDP_PORT
[TCP] Rcvd: UDP_PORT 37854
[UDP] Sent: GET_BOARD X
[SYS] Waiting for response ...
[UDP] Rcvd: EXIT_GRANT
Opponent has left the game. You win!
```

7 Grading

This lab is worth 5% of the final grade and is composed of 50 points. You will not be awarded any points if your submission does not compile.

```
General requirements: 5 points
_____  2 points: Coding standard, comments ... etc
_____  1 points: README file
_____  2 points: Descriptive messages (joining, winning ... etc)

Client: 45 points
_____ 15 points: Error checking (Handle return values < 0)
                 (-3 pt for each function call that is not checked)
_____  5 points: Resolves hostname (gethostbyname(())
_____  5 points: Handshaking over TCP (JOIN/GET_UDP_PORT)
_____  5 points: Playing the game through UDP (MARK/EXIT)
_____  5 points: The client issues JOIN/GET_UDP_PORT, not the player
_____  5 points: Use htons() to get the correct port
_____  5 points: Clean up, close sockets when done using them
```

8 Notes

Please feel free to mail the instructor Chin-Jung Liu liuchinj AT cse.msu.edu for questions or clarifications. Additional notes and FAQ will be posted on the website as well.