CSE1205/2100 Introduction to Computer Science and Engineering Network Gaming (T3) Lab

Introduction

In this lab, we will borrow some code from the Computer Engineering Internet of Things / Networks course to develop a networked version of a tic-tac-toe (T3) game. In doing so, you will also become accustomed to a simple UDP network protocol and server-client network models.

You will also learn more about the concepts in this lab in CSE 3318 (Algorithms and Data Stuctures), CSE 3320 (Operating Systems), CSE 4352 (IoT/Networks), and CSE 4344 (Networks).

Network Quick Primer:

When data is sent on an Ethernet connection in many applications, the data is sent between IP address and port pairs (collectively called a socket addresses).

The IP addresses we will use in the lab use the IPv4 protocol. An IPv4 address is 4 octets (4 bytes) long. In the lab, all the addresses are numbered 192.168.1.x, where x is a number between 1 and 254. To configure your RPi to operate on the lab network, you should enable DHCP so it will automatically get a unique address on the lab network. This will allow you to send data to and receive data from other computers in the lab.

Often on a network, there is a server and a client. For instance, when you use a web browser to connect to a web site, the browser is a client and the web site is a server. A physical or virtual machine may provide many services to users.   Port numbers are used to expose services to external users using well known port numbers for each service. For instance, SSH and SFTP connect to a server using port 22 and web servers often use port 80. So when a web browser tries to get access a web site, it contacts the IP address at port 80. When port 80 is accessed, traffic is routed to the web server to process the get and post requests from the web browser. When port 22 is accessed, an SSH server handles the traffic.

There are two primary IP protocols that are used to send data on these ports – Transport Control Protocol (TCP) and User Defined Protocol (UDP). TCP is more complicated as it contains complexity to reliably send large files consisting of many packets of data. UDP on the other hand is very simple, but it just sends packets of data just once so there is a chance for a packet of data to get lost. So the trade-off is complexity vs simplicity.

Network Use for our Game:

For our lab, we will use the UDP protocol and implement a client and server system for game play. The server will monitor port 4096 and the client will monitor port 4097 for inbound UDP traffic. The choice of 4096 and 4097 are arbitrary, but are required for this project so game applications can inter-operate.

In our lab, you can send a string using the provided sendData function:

```
bool sendData(const char ipv4Address[],
               int port, const char str[])
```

You can receive a string using the provided receiveData function:

```
void receiveData(char str[], int str_length)
```

The string value sent and received with these message will be one of the following 10 case-sensitive strings:

invite	Invite a user to start a game
A1, A2, A3, B1, B2, B3, C1, C2, or C3	Game moves

If two users want to play, one of the users starts their game application as a server, which waits to accept an invitation from a client to start a game. The other user starts their game application as a client and sends an invitation to the server to start a game. In both applications, users alternate turns and send their game moves to each other. In this implementation, the client sending the invitation will request the server make the first game move.

Application Requirements – General:

You should write your lab solution in a file t3.c, that includes the udp.h header. When you compile the code, you will use the following command:

```
gcc -o t3 t3.c udp.c
```

Application Requirements – Command Line:

The application is named t3. Invoking the application requires 2 additional options – the IP address of the remote machine you want to reach and the operating role of the application. The role will indicate whether the application starts as a:

- Server and accepts invitations from a client

- A client and sends an invitation to a server

The command line syntax is:

```
./t3 REMOTE_IP ROLE
```

To start as a server accepting invitations from 192.168.1.x, the command line is:

```
./t3 192.168.1.x accept
```

To start as a client sending an invitation to 192.168.1.y, the command line is:

```
./t3 192.168.1.y invite
```

Your code solution should:

- Verify that there are 3 arguments (argc == 3)

- Once the arg count is verified:

    - the IP address should be stored

    - The last argument is verified as “accept” or “invite”

    - The role (server or client respectively) is stored

Application Requirements – Listener Port Startup:

On startup, the application should open a socket that listens for inbound messages from the remote IP addresses’ port (the opponent). A call to

```
bool openListenerPort(const char ipv4Address[], int port)
```

with the remote address from the command line and the correct port (4096 for client, 4097 for server).

If the function does not return a true condition indicating success, you should show the error and exit the application.

Application Requirements – Server Startup:

If the application starts as a server, you should wait for an invitation with the following call:

```
void receiveData(char str[], int str_length)
```

This function will not return until a string has been received from the client. The code should verify the string received is “invite” or exit the program is this is not the case. Once the invitation has been received or if an error occurs, indicate this on the console.

Application Requirements – Client Startup:

If the application starts as a client, you should send an invitation using the following call:

```
bool sendData(const char ipv4Address[],
               int port, const char str[])
```

with the remote address from the command line and the remote port you want to receive the message (your opponent), and the string to send (“invite”). The string is case sensitive.

The application should also indicate that an invitation has been sent.

Application Requirements – Shutdown:

When the application ends, release the socket used in the listener port using the following call:

```
void closeListenerPort()
```

Application Requirements – Game Play Initialization:

The game board layout is defined by a letter-number pair as follows:

```
A1  A2  A3
B1  B2  B3
C1  C2  C3
```

At the beginning, the game board can be initialized with periods to make the game field visible using a function like this:

```
void clearBoard(char board[3][3])
{
    for (int r = 0; r < 3; r++)
        for (int c = 0; c < 3; c++)
            board[r][c] = '.';
}
```

By definition please note the following:

- the user running the server application makes the first move.

- the server will be ‘x’ and the client will be ‘o’.

Application Requirements – Game Play Loop:

In a while loop, you will alternate between making a move and getting a move from your opponent until a winner or draw occurs.

Here are the steps make a move:

1. The user will enter a 2 character letter-number pair string.

2. The move is recorded in the board by calling addMove(), a function that you write, that verifies:
    - string length must be 2 characters
    - the first character is A, B, or C
    - the second character is 1, 2, or 3
    - must be legal (the position must not have already been played)
    - if not a legal move, go back to step 1

The function prototype for this function is:

```
bool addMove(char board[3][3], char move[], char xo)
```

which returns true is successful and false otherwise.

3. A function, showBoard(), that you write is called to display the board:

```
void showBoard(char board[3][3])
```

4. The letter-number pair string is sent to the other player using the provided sendData() function.

5. A call is made to a function, isWinner(), that you write is called to determine if the user (xo) where xo is ‘x’ or ‘o’:

```
bool isWinner(char board[3][3], char xo)
```

6. A check is made to see if the game is a draw (no winner and 9 moves). Obviously, a draw can be determined in less than 9 moves, but checking for 9 moves is a simpler implementation.

Here are the steps to process a move from your opponent:

1. The user call the provided receiveData() function to get the string.

2. The move is recorded in the board by calling addMove(), a function that you write, that verifies:
    - string length must be 2 characters
    - the first character is A, B, or C
    - the second character is 1, 2, or 3
    - must be legal (the position must not have already been played)
    - if not a legal move, exit game

The function prototype for this function is:

```
bool addMove(char board[3][3], char move[], char xo)
```

which returns true is successful and false otherwise.

3. A function you write is called to display the board:

```
void showBoard(char board[3][3])
```

4. A call is made to a function, isWinner(), that you write is called to determine if the user (xo) where xo is ‘x’ or ‘o’:

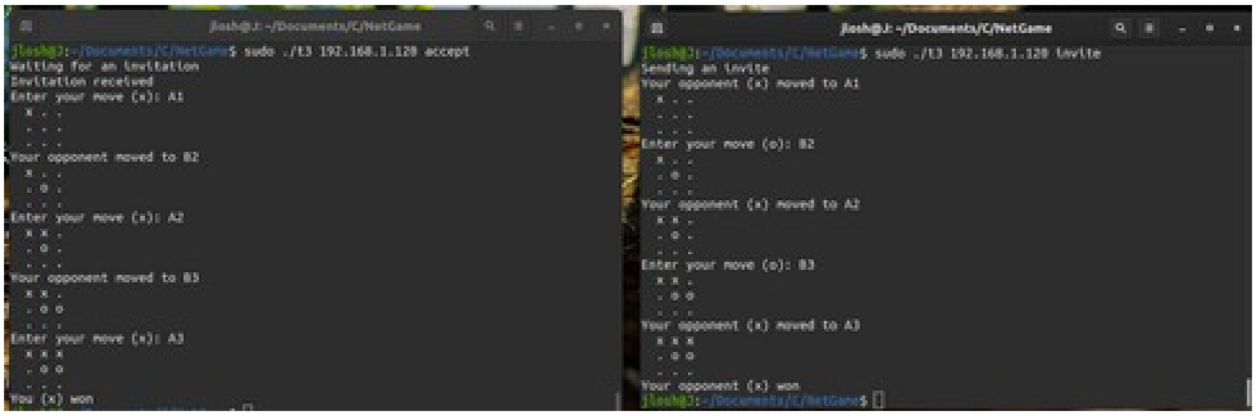
```
bool isWinner(char board[3][3], char xo)
```

5. A check is made to see if the game is a draw (no winner and 9 moves).

A typical game session (time proceeds top to bottom) looks like this:

Server monitoring Port 4097	**Client monitoring Port 4096**
Server opens and waits for an invitation from a pre-known client	
	Client opens and sends an invitation to the server
Server accepts the invitation	
The following pattern continues until a winner or draw is declared:	
a. User enters move for ‘x’ b. Move is verified as valid, if invalid, user is asked to try again c. Board is updated to show the move d. The move is sent to the client. e. If this is a winning move, then the winner is announced and the game ends f. If 9th move and no winner, a draw is declared and the game ends.	
	a. Client receives move b. Move is verified… if illegal, game ends c. Board is updated to show the move d. If this is a winning move, then the winner is announced and the game ends e. If 9th move and no winner, a draw is declared and the game ends.
	a. User enters move for ‘o’ b. Move is verified as valid, if invalid, user is asked to try again c. Board is updated to show the move d. The move is sent to the server. e. If this is a winning move, then the winner is announced and the game ends f. If 9th move and no winner, a draw is declared and the game ends.
a. Server receives move b. Move is verified… if illegal, game ends c. Board is updated to show the move d. If this is a winning move, then the winner is announced and the game ends e. If 9th move and no winner, a draw is declared and the game ends.	

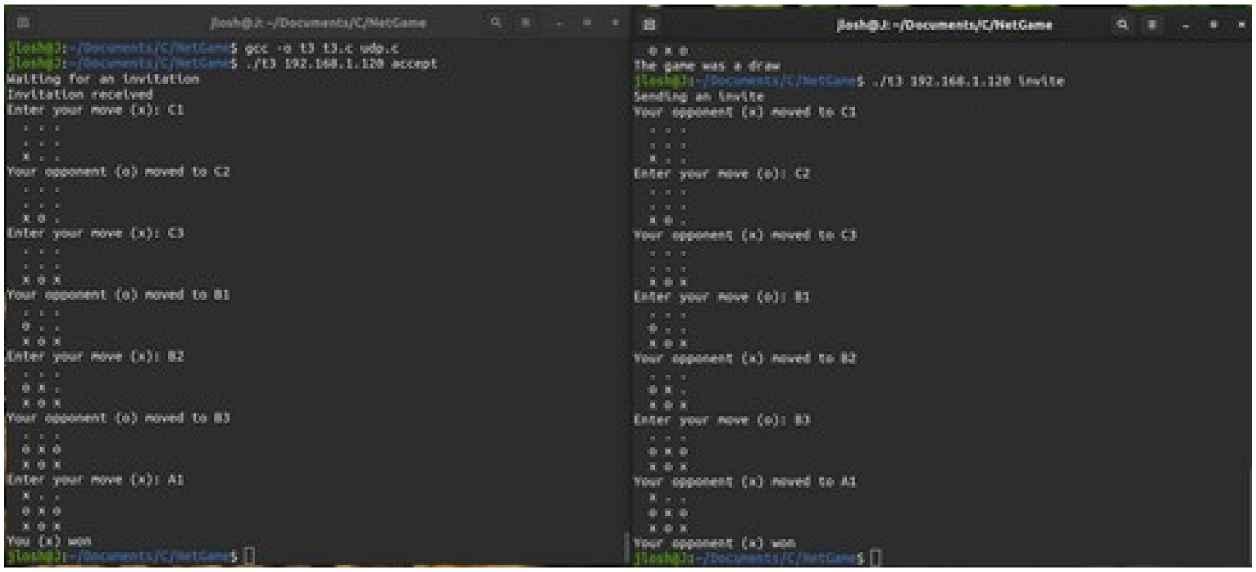
Here is a screen capture of game play to a win:



```
jlosh@J: ~/Documents/C/NetGame
jlosh@J:~/Documents/C/NetGame$ sudo ./t3 192.168.1.120 accept
Waiting for an invitation
Invitation received
Enter your move (x): A1
  x . .
  . . .
  . . .
Your opponent moved to B2
  x . .
  . o .
  . . .
Enter your move (x): A2
  x x .
  . o .
  . . .
Your opponent moved to B3
  x x .
  . o o
  . . .
Enter your move (x): A3
  x x x
  . o o
  . . .
You (x) won
```

```
jlosh@J: ~/Documents/C/NetGame
jlosh@J:~/Documents/C/NetGame$ sudo ./t3 192.168.1.120 invite
Sending an invite
Your opponent (x) moved to A1
  x . .
  . . .
  . . .
Enter your move (o): B2
  x . .
  . o .
  . . .
Your opponent (x) moved to A2
  x x .
  . o .
  . . .
Enter your move (o): B3
  x x .
  . o o
  . . .
Your opponent (x) moved to A3
  x x x
  . o o
  . . .
Your opponent (x) won
jlosh@J:~/Documents/C/NetGame$ 
```

Here is a screen capture of game play to a draw:



```
jlosh@J: ~/Documents/C/NetGame
jlosh@J:~/Documents/C/NetGame$ gcc -o t3 t3.c udp.c
jlosh@J:~/Documents/C/NetGame$ ./t3 192.168.1.120 accept
Waiting for an invitation
Invitation received
Enter your move (x): C1
  . . .
  . . .
  x . .
Your opponent (o) moved to C2
  . . .
  . . .
  x o .
Enter your move (x): C3
  . . .
  . . .
  x o x
Your opponent (o) moved to B1
  . . .
  o . .
  x o x
Enter your move (x): B2
  . . .
  o x .
  x o x
Your opponent (o) moved to B3
  . . .
  o x o
  x o x
Enter your move (x): A1
  x . .
  o x o
  x o x
You (x) won
jlosh@J:~/Documents/C/NetGame$ 
```

```
jlosh@J: ~/Documents/C/NetGame
  o x o
The game was a draw
jlosh@J:~/Documents/C/NetGame$ ./t3 192.168.1.120 invite
Sending an invite
Your opponent (x) moved to C1
  . . .
  . . .
  x . .
Enter your move (o): C2
  . . .
  . . .
  x o .
Your opponent (x) moved to C3
  . . .
  . . .
  x o x
Enter your move (o): B1
  . . .
  o . .
  x o x
Your opponent (x) moved to B2
  . . .
  o x .
  x o x
Enter your move (o): B3
  . . .
  o x o
  x o x
Your opponent (x) moved to A1
  x . .
  o x o
  x o x
Your opponent (x) won
jlosh@J:~/Documents/C/NetGame$ 
```

Networking Game Lab Worksheet **Name** ________________________

Course/Section ______________

Please complete the following steps to complete the network game lab:

1. Configure your RPi to use DHCP and get an address automatically. Use the ifconfig command to get the address that is given, The IPv4 address should be 192.168.1.x. Please note the value x each time to start up your hardware in the lab. Others will need your full IPv4 address for game play to contact your RPi.

2. Create your application in a file called t3.c, following the requirements in the General Section. Build the project with the udp.c to make sure you are able to proceed.

NOTE: You can actually open up two copies of the application in two bash shells and play a game between the two application instances by using the 192.168.1.x address from step 1.

3. Add command line argument support to your application, following the requirements in the Command Line Section. You must determine whether you will be a server or client and what the remote address of your opponent's RPi is.

4. Add code to open a listener port so you can receive messages from other users, following the requirements in the Listener Port Startup Section.

5. Add code to close out the listener port at the end of the application, following the requirements in the Shutdown Section.

6. Depending on whether the application is started as a server or client (determined in step 3), invite your opponent to play (if you are a client) or accept an invite to play from your opponent to play (if you are a server), following the requirements in the Client Startup or Server Startup Sections.

7. Initialize the game following the requirements in the Game Play Initialization Section.

8. Code the game play part of the project, following the requirements in the Game Play Loop Section.

9. Lab checkout steps:

a. Invite the grader to play a game, connecting to the grader's RPi. Play the game to a draw and again until a winner is found.

b. Create a file, lab7.zip, that includes the following files;

- A JPEG image of a game session in the shell

- The source code your program

c. Upload the zip file to Canvas.

Thank you for attending the lab. We hope some of this material was new and interesting to you.

Drs. Losh and Eary