Assignment #3: TCP Socket Programming

ECE 650 – Spring 2018

See course site for due date

General Instructions

1. You will work individually on this the project.
2. The code for this assignment should be developed and tested in a UNIX-based environment, specifically the Duke Linux environment available at login.oit.duke.edu.
3. You must follow this assignment spec carefully, and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient. For this assignment, testing will be automated. If you do not follow exact instructions for your submission materials (file names, program output, etc.) points will be deducted.

Note: This assignment includes a programming portion (this document) and a written portion (separate document). Be sure to submit both!

Overview

In this assignment you will develop a pair of programs that will interact to model a game, which is described below. The game is simple, but the assignment will give you hands-on practice with creating a multi-process application, processing command line arguments, setting up and monitoring network communication channels between the processes (using TCP sockets), and reading / writing information between processes across sockets.

The game that will be modeled is called *hot potato*, in which there are some number of players who quickly toss a potato from one player to another until, at a random point, the game ends and the player holding the potato is "it". The object is to not be the one holding the potato at the end of the game. In this assignment, you will create a ring of "player" processes that will pass the potato around. Thus, each player process has a left neighbor and a right neighbor. Also, there will be a "ringmaster" process that will start each game, report the results, and shut down the game.

To begin, the ringmaster creates a "potato" object, initialized with some number of hops and sends the potato to a randomly selected player. Each time a player receives the potato, it will decrement the number of hops and append the player's ID to the potato. If the remaining number of hops is greater than zero, the player will randomly select a neighbor and send the potato to that neighbor. The game ends when the hop counter reaches zero. The player holding the potato sends it to the ringmaster, indicating the end of the game. The ringmaster prints a trace of the game to the screen (using the player identities that are appended to the potato), and shuts the game down (by sending a message to each player to indicate they may shut down as the game is over).

Each player will establish three network socket connections for communication with the player to the left, the player to the right, the ringmaster. The potato can arrive on any of these three channels. Commands and important information may also be received from the ringmaster. The ringmaster will have *N* network socket connections. At the end of the game, the ringmaster will receive the potato from the player who is "it".

The assignment is to create one ringmaster process and some number of player processes, then play a game and terminate all the processes gracefully. You may explicitly create each process from an interactive shell; however, the player processes must exit cleanly at the end of the game in response to commands from the ringmaster.

Communication Mechanism

In this assignment, you will use TCP sockets as the mechanism for communication between the ringmaster and player processes. Your programs must use exactly the command line arguments described here. The ringmaster program is invoked as shown below, where `num_players` must be greater than 1 and `num_hops` must be greater than or equal to zero and less than or equal to 512 (make sure to validate your command line arguments!).

```
ringmaster <port_num> <num_players> <num_hops>
```

The player program is invoked as:

```
player <machine_name> <port_num>
```

where `machine_name` is the machine name (e.g. login-teer-03.oit.duke.edu) where the ringmaster process is running and `port_num` is the port number given to the ringmaster process which it uses to open a socket for player connections. If there are *N* players, each player will have an ID of *0*, *1*, *2*, to *N-1*. A player's ID and other information that each player will need to connect to their left and right neighbor can be provided by the ringmaster as part of setting up the game. The players are connected in the ring such that the left neighbor of player *i* is player *i-1* and the right neighbor is player *i+1*. Player *0* is the right neighbor of player *N-1*, and Player *N-1* is the left neighbor of player 0.

Zero is a valid number of hops. In this case, the game must create the ring of processes. After the ring is created, the ringmaster immediately shuts down the game.

Resources:

Refer to our lecture notes and example code on TCP sockets for establishing communication between the ringmaster and players.

You will also find that you will need to use the "select" call over a set of file descriptors from both the ringmaster and player processes in order to know when some information has been written to one of a set of the socket connections. You will likely also find the functions "gethostname" and "gethostbyname" helpful in establishing the connections between neighboring players in the ring and between players and the ringmaster.

Finally, you will need to create random numbers (e.g. between 0 to N). To do so, you may use the rand() call. Your code should first seed the random number generator:

```
srand( (unsigned int) time(NULL) + player_id );
```

Then you may generate a random number between 0 and N-1 using:

```
int random = rand() % N;
```

Output:

The programs you create must follow the description below precisely. If you deviate from what is expected, it will impact your grade.

The following describes **all** the output of the `ringmaster` program. Do not have any other output.

Initially:

```
Potato Ringmaster
Players = <number>
Hops = <number>
```

Upon connection with a player (i.e. each player should send some initial message to the ringmaster to indicate that it is ready and possibly provide other information about that player):

```
Player <number> is ready to play
```

When launching the potato to the first randomly chosen player:

```
Ready to start the game, sending potato to player <number>
```

When it gets the potato back (at the end of the game). The trace is a comma separated list of player numbers. No spaces or newlines in the list.

```
Trace of potato:
<n>,<n>, …
```

The following describes **all** the output of the `player` program. Do not have any other output.

After receiving an initial message from the ringmaster to tell the player the total number of players in the game, and possibly other information (e.g. info about that player's neighbors):

```
Connected as player <number> out of <number> total players
```

When forwarding the potato to another player:

```
Sending potato to <number>
```

When number of hops is reached:

```
I'm it
```

Detailed Submission Instructions

Your submission will include source code files and a Makefile that you create. Your source code files should contain at least the following:

1. **ringmaster.c** – The source code for your ringmaster program as described above.
2. **player.c** – The source code for your player program as described above.
3. **Makefile** – A makefile that will compile ringmaster.c and player.c into executable programs named ringmaster and player, respectively
4. **potato.h** – A source code file containing a potato structure that can be sent across TCP sockets between players and between players and the ringmaster.

You will submit a single zip file named "hw3.zip" to your sakai dropbox location, e.g.:

```
zip hw3.zip ringmaster.c player.c potato.h Makefile
```