# Project #3: TCP Socket Programming

# ECE 650 – Spring 2020

**See course site for due date**

## General Instructions

1. You will work individually on this 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.

Beej's Guide to Network Programming is an excellent reference resource for this assignment.

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. "`getsockname`" would help find the port on which your client program has bound to listen (See FAQ Q2).

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>, …
```

**Sample Ringmaster Output:**

```
Potato Ringmaster
Players = 3
Hops = 200
Player 1 is ready to play
Player 0 is ready to play
Player 2 is ready to play
Ready to start the game, sending potato to player 2
Trace of potato:
2,1,2,0,2,1,0,2,...
```

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
```

**Sample Player Output:**
**Player 0 (in this example not the last player):**

```
Connected as player 0 out of 3 total players
Sending potato to 2
Sending potato to 1
Sending potato to 1
Sending potato to 2
Sending potato to 2
Sending potato to 2
...
Sending potato to 1
```

**Similar for Player 2**

**Player 1 (in this example the last player, i.e. receives potato on last hop):**

```
Connected as player 0 out of 3 total players
Sending potato to 0
Sending potato to 2
Sending potato to 0
Sending potato to 0
Sending potato to 2
Sending potato to 2
Sending potato to 2
...
Sending potato to 0
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 "netid_proj3.zip" to your sakai dropbox location, e.g.:

```
zip netid_proj3.zip ringmaster.c player.c potato.h Makefile
```

## FAQ

1) Should the game work across the network on different machines?
    a) Yes. Network sockets must work on the network, therefore, running the ringmaster and player on different machines should still allow a functional game.
2) Will our code be checked against Valgrind for memory errors?
    a) Not directly. However, these issues may show up in other testing as practical problems. Also, it is always recommended to write leak free code.
    Some common issues and how to fix them:
        - Sending uninitialized bytes on the socket: Use `memset()` to initialize the data buffer.
        - Forgetting to free objects allocated on the heap: e.g. forgetting `freeaddrinfo()` after `getaddrinfo()`, `free()` after `malloc()`, etc.

    How to use valgrind to catch above issues:

    ```
    valgrind -v --leak-check=full --track-origins=yes
    ```
3) How to use `getaddrinfo()`?
    a) Look at the man page. Make sure to loop through the results of `getaddrinfo()` for the bind etc. as seen in the man page. This is to help overcome practical constraints of invalid hostnames, etc.
4) Which address and port should I bind to for accepting incoming connections?
    - The address you bind to will be the address that clients will connect to. For example, **if you bind to 127.0.0.1, others can only connect by connecting to 127.0.0.1, which limits the clients to be on the same machine**. (See FAQ Q1)
    - However, if you bind to `0.0.0.0` (aka `INADDR_ANY`), clients can connect to you using any address, including `127.0.0.1` and the public IP of the machine (if any). Therefore the most general solution is to bind to address `0.0.0.0`.
    - If you have a specific port number to bind to, use that. Otherwise, if you specify 0 in `sin_port` when calling bind, the OS will assign a port for you. After that, you can use `getsockname()` to see which port the socket is bound to.

5) How do I know my own IP address?
    - The command `ip addr` can show you the IP address of each of your network interface.
    - For the purpose of Project 3, neither the ringmaster nor the player has to do that. The player can know the ringmaster's address by resolving its hostname, which is given in the arguments. The ringmaster can know the player's address as well; see the next question.
6) If a client connects to me, how do I know its IP address?
    - The `accept` syscall will fill in the `addr` argument, which is a `sockaddr` struct containing the client's IP address.
7) My laptop cannot accept incoming connections.
    - Your laptop may not have a public IP address and/or a hostname that is recognized by others. Solving these problems is beyond the scope of Assignment 3. You might be interested in the following readings, though: Network address translation, Port forwarding, DMZ.
    - For the purpose of this assignment, it is a good idea to test your programs on machines that have public IPs and/or hostnames. VCM is a good choice, and I believe you should be able to get at least 2 VMs.
8) My system has a max limit on number of incoming socket connections I can accept. Is this okay?
    a) Yes. On the `accept` call of ringmaster, you may only be able to support 1020 incoming connections. This can also be checked using `ulimit -n`. You do not have to modify or worry about this.
9) Do I care about what `send()` and `recv()` return?
    a) Yes. See the man pages for both. Also, make sure to handle the error conditions and socket closure cases, e.g. -1 for error, 0 for closed socket on recv, etc. See the man pages for more details.
10) What is `select()`? Do I care about it?
    a) See `man select`. `select()` is a function that allows you to see which of your file descriptors is ready to send you content. Since this game involves each player being connected to two other players and you may get a potato from any side, you need to use this function to check where to recv data from.
    b) Also, make sure to reset the `readfds` structure used by this function every time since after calling select, it only contains the fds of sockets that are currently sending you data.
11) `send()` fails to send my complete message.
    a) This is not a bug. This is how send() is designed. In C/C++ without using additional libraries, the basic `send()` does not guarantee that all of the bytes specified in message buffer size will be sent. `On success, these calls return the number of bytes sent.` See man page for send.
    You may need to use some loops to make sure your entire message buffer is sent.

12) `recv()` fails to receive complete message.
    a) Same as above. However, you can use `MSG_WAITALL` if you're certain about the number of bytes you need to receive.
13) What data structure should I send in the socket buffer?
    a) The socket buffer takes a byte stream. It may be good enough to have a protocol that sends the size of a string followed by a string, i.e. char stream in your buffer. The size will let you know how much data to receive, since these communications may be of varied length. Feel free to play around and invent your own protocol. Sending structs on a socket buffer will need more work as opposed to a string.
14) Can number of hops be 0?
    a) Yes. "*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.*" - Described in Communication Mechanism
    No trace of hops is printed in this case.
15) What is the first hop?
    a) The first hop is from the ringmaster to the first player, selected at random.
16) What happens on the last hop? / When should the game end?
    a) When the potato is passed to a player on the last hop, the game ends. The full trace of the potato should then be sent to the ringmaster -- this does not count as a hop. The game ends and all processes must close/complete. The order in which processes exit does not matter.
17) When I try to run the ringmaster consecutively using the same port it gives me an error that the port is in use although I ended the program. What should I do?
    a) It takes a while for the port to be set as free again. You can let the OS know to set it to free immediately by using the `setsockopt()` with `SO_REUSEADDR` option. See ref.
18) Can I add some time delay in my code for synchronization?
    a) No, you may not. Synchronization must be achieved using a good logical series of steps and any loops as required to ensure success of connections. Timing based connections are arbitrary and not acceptable solutions. This will also cause the grading time for your code to get out of hand and may result in a penalty.
19) Do I have to deal with a malicious/arbitrary process connecting to my players/ringmaster and sending random messages?
    a) No. You can design your own communication protocol and adhere to it. We will not test how you handle arbitrary connections in your game instead of your own players, etc.
20) Is there a specification on how to handle issues such as player disconnecting in the middle of the game, loss of network, etc.?
    a) No, you may choose how you deal with these situations. However, silent failure is a bad software engineering practice.