Project Report

Program 1: Hot Potato

General Description:

The code tries to simulate a game of hot potato, wherein the 'listen' program acts as the ring master and lets players, simulated by the 'speak' program, into the game/ring. The player then passes a data structure (potato) to begin the game and at the end of the game, prints out a trace of the potato's path.

Design:

A 'simple but sturdy' approach has been used to design the software. All points (hopefully) in the flow where a race condition or a dead lock may occur have been accounted for by exchanging confirmation signals. This also eliminates any requirement for random 'sleep()' or other wait calls.

Steps followed by the 'listen' program:
1. Initiate dynamic data structures to store players' (speak program) data and the potato based on command line arguments.
2. Create a socket to bind to port given on command line.
3. Execute a 'listen' call and wait for players to connect.
Steps 4.1 and 4.2 below are executed in a loop for the number of players.
4.1 Get the hosts and addresses of the connecting players and store them in the 'players' data structure. Provide each player a player identification (pid) based on the order of connection. Obtain the port at which each player shall listen for a neighbor to connect. Store this too in a data structure.
4.2 Provide each player the port the previous players has opened for listening
The first player obviously gets the listening port of the last player in the end.
5. Wait for confirmation signal (a predefined negative number) from all the players that they have the required data to form the ring.
Steps 6.1 and 6.2 below are executed in a loop for the number of players.
6.1 Send a player a signal (represented by byte 0) to activate accept on its listening port
6.2 Send the next player a signal (represented by byte 1) to connect to the previous player.
Here again, the first player gets its connect signal only in the end. At this point the ring is formed.
7. Send the total size of the potato to expect to all players and then wait till a confirmation from all players that they have been able to allocate the space required for the potato.
8. Randomly select a player (random function seeded with time) and send it the potato.
9. Execute a 'select' for the file descriptors of all the players and wait for the potato to be sent back from any one of them. Print out the trace in the potato.
10. Send each player a signal to close the connect it created with the previous player, receive a confirmation from that player and repeat the same for the next player.
11. Exit

Steps followed by the 'speak' program:
1. Connect to the master ('listen' program) using the host address and port given in command line.
2.1. Receive from master its process id (pid). Creates a socket and starting from a predefined base port number, find a port to which it can successfully bind to. Send this port number to the master advertising it as its listening port. (see listening program step 4.1)
2.2. Receive from master the previous player's listening port (listening step 4.2)
3. Send confirmation signal to master (listening step 5)
4. Execute two 'receive's from the master. If it is the 'accept' signal, execute an accept on the socket given to the next player. If it is the 'connect' signal, execute (repeatedly for a predefined number of tries- after which an error message and shutdown is executed) a connect to the port

number and address of the previous player received from the master. (listening steps 6.1 and 6.2). Do not proceed before both receives have been completed.
5. Receive the total size of the potato from the master.
6. Execute a 'select' for the master's socket and for the two neighboring sockets.
Now send a confirmation to the master (listen step 7).
7. Receive the potato from one of the three sockets. The potato size is known. The header of the potato contains information regarding number of hops left. If the hops left = 0, send the potato back to master, print "I'm it" and wait for close signal from master. (listen step 10).
8. If hops left > 0, the process calculates its position in the potato and adds its 'process id'. It also updates the 'hops left' field.
9. The process picks a neighbor randomly based on a random function that uses the product of time and 'hops left' as seed and sends the potato to it.
10. The process goes back to the wait on select step- back to step 6.
11. The close signal from the master is similar to the potato structure, but has '-69' as the header. Upon receiving this signal, the process first closes the socket to the previous player, sends a confirmation to the master closes socket to the master and exits. (listen step 10)

Data Structures:
1. batata_vada: (The potato) is a dynamically allocated array of 'hops+1' integers, where hops is the total number of hops. The integer at the head is used to store the number of hops left. This data structure is maintained at the listen process and each of the speak processes.
2. my_data: This structure is maintained by each of the speak processes and contains the addresses of each of the neighbors and the process id of itself.
3. players: This data structure maintained by the master is a dynamically allocated array of a structure called player. Each player structure contains the players' address data and socket file descriptor.

Testing:

The following different scenarios have been cleared with the software:
1. The program has been tested 100 players with 50,000 hops, with the master on a separate host and the hundred players distributed randomly over two other hosts. I do not know till what upper limit these numbers can be stretched.
2. Invalid command line arguments: Invalid ports (only ports between 65420 and 65500 are allowed for the master. I spent some time determining this range), negative hop counts, less than two players etc. are reported and the program exits.
3. If free ports are unavailable, this is reported and the program exits.
4. Once the ring is formed, if any process, master or players, goes down due to any unseen reason, all the processes are informed. They report this and exit.
5. A very large number of hops will lead to memory shortage issues. The process will exit under this condition.
Most other general conditions have been found and accounted for.

Evaluation:

As mentioned before, a 'simple but sturdy' approach has been followed. Although this had positive features like ease of coding, ease of debugging, no sleep() or other wasteful calls, there may be some features that might have been further improved.
By careful planning there might have been a few confirmation signals that might be done away with. Also, given more time, I would have liked to improve the data structure of the potato- instead of dynamically allocating the total space for the full potato at each player and master at the beginning, I would have had only the player that is about to receive the potato allocate memory (I could have a protocol to communicate the present size of the potato before actually transferring it) just enough to receive the current size of the potato and append its own identity and then de-allocate this memory after sending it out again. Although, this would slow down the program, it

would greatly increase the upper bounds on the hop count and number of players (if all are on the same machine).
Most possible failures in the program have been checked for at multiple points of execution and the processes terminate with an error message on finding illegal values of flags or variables. The code has been sufficiently modularized and commented.