Multiplayer Network Synchronization Snake Game

Derek Ching
Computer Science
Northeastern University
Boston, United States
ching.de@husky.neu.edu

Ishan Patel
Computer Science
Northeastern University
Boston, United States
patel.ish@husky.neu.edu

Prateek Pisat
Computer Science
Northeastern University
Boston, United States
pisat.p@husky.neu.edu

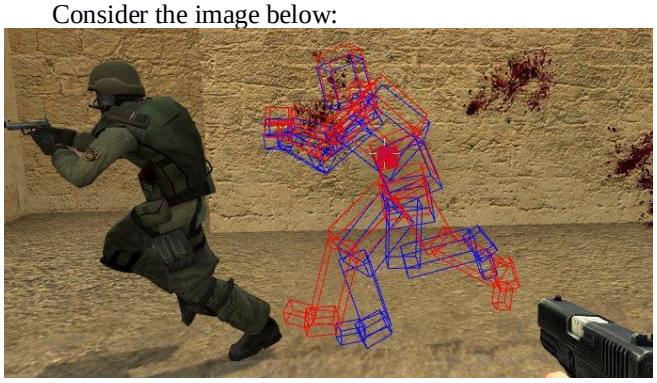
***Abstract* — We have built a snake game in python using pygame library. This python binary is controlled by an external C++ backend, which handles decisions about movements of the players on the screen of the multiplayer python snake game. The C++ backend oversees determining who will be the server, and who will be the clients. Only one player can be the server. When the server received all client's messages a message handler will distribute global coordinates / collision logic to each player's snake python game.**

Keywords — Thread Synchronization, Game Design, Named Pipes, Network Communication, Python, C++.

I. Problem Statement

Multiplayer games use a Client-Server networking architecture. Usually a server is a dedicated host that runs the game and is authoritative about world simulation, game rules, and player input processing. A client is a player's computer connected to a game server. The client and server communicate with each other by sending small data packets at a high frequency. A client receives the current world state from the server and generates video output based on these updates. In contrast with a single player game, a multiplayer game must deal with a variety of new problems caused by packet-based communication. Network bandwidth is limited, so the server can't send a new update packet to all clients for every single world change. Network packets take a certain amount of time to travel between the client and the server (i.e. half the ping time). This means that the client time is always a little bit behind the server time. Furthermore, client input packets are also delayed on their way back, so the server is processing temporally delayed user commands. In addition, each client has a different network delay which varies over time due to other background traffic and the client's framerate. These time differences between server and client causes logical problems, becoming worse with increasing network latencies. In fast-paced action games, even a delay of a few milliseconds can cause a "laggy" gameplay feeling and make it hard to hit other players or interact with moving objects. Besides bandwidth limitations and network latencies, information can get lost due to network packet loss.

Consider the image below:



Because of network latency, the figure in blue might be what one client perceives to be the enemy player position, but the figure in red is where the server is really placing the character

II. Our Solution

We have a Status Thread, that locks the server, using a mutex, while communicating with the client, so that, the packet order does not change.

We also have a synchronization system installed that ensures that all the threads are in sync with the server. The synchronization thread, constantly gets the most updated game state from the server and all the client threads are updated with this updated data. Hence all the client threads have a consistent copy of the game.

III. Introduction

We have implemented a network multiplayer snake game, that allows multiple players to connect to a server and play the game a bit more competitively. Since the current gaming scene tends to lean heavily towards games that multiple people can play together, we, as gamers, wanted to extend a classic game to a more modern multiplayer scenario.

Our project is primarily divided into 4 major components:

- The Game
- The Server
- The Clients.
- The Backend

IV. The Components

A. The Game

We started by building a snake game, purely in Python. Python's pygame library provides a quick and easy way to implement quality games in a short amount of time. Each player will have one separate instance of the game running. Each instance is a Python process which is synchronized with the rest of the instances. We current have a support for up to 4 players playing simultaneously.

B. The Server

The server acts as the host for a game. Any player can be the server, by using the '-s' option. The server creates a socket that is bound to the server's IP Address, and is listening to the Post 8080 by default.

The server itself uses multiple threads to divide the functionality to different pieces.

- The Observer Thread:
  Responsible for checking if the server is running, every 5ms. In case the server fails, the game should end, disconnecting all the players, to avoid any socket connection exceptions.

- The Main Thread:
  This is the thread that is responsible for communicating with the client threads, maintaining socket descriptors for each client and handling error conditions. The main thread is also responsible for storing the global state of the game. This game state is provided to all the clients, to ensure a consistent gaming environment.
- The Message Thread:
  Responsible for handing global messages to and from the clients, for example, to start the game or to end the game.
- The Key Thread:
  This thread is responsible for communicating with the Python frontend and the C++ backend, for example, reading messages from the FIFO pipe, sending various messages back to the client, informing them that the server has left the game, or that the food/ apple has repositioned itself to a new location, or if a snake is dead or alive, etc.
- The Time Thread:
  Responsible for synchronizing the timing events/ screen refreshes for all the individual instances of the python frontend.

C. The Client

Each non-server entity in the game is a client. A client connects to a server, and then waits for a game to start. Like the server, the client uses multiple threads to divide it functionality into smaller, manageable pieces.

- The Observe Thread:
  This thread checks if the client is up and running every 5ms. If, at any point in time, the client fails/ crashes, that instance will be destroyed, and the said player will be disconnected from the game.
- The Main Thread:
  This is the main communication thread, that is in charge of connecting to the server via a socket. It's other main role is to read important messages from the server pertaining to state information about the other players, time synchronization, the state of the apple, start and end messages from the server, and message that other players are disconnected.
- The Message Thread:
  This thread is responsible for sending a message to the server, as long as it is running.
- The Client Key Thread:
  This is the thread that will control the messages between the python frontend and the backend C++, which will open and read from a FIFO pipe the messages from the python frontend and send the various messages back to the server. These messages include that the client exited from his game, that the apple has been eaten and relocated, the state of the snake (alive/dead), their id, and each snake's location, and whether the client has won or not.

D. The Backend

The Backend is responsible for determining which player acts as the server, initializing the server, connecting clients to the server based on the arguments passed to the executable.

V. DIAGRAMS

A. Class Diagram

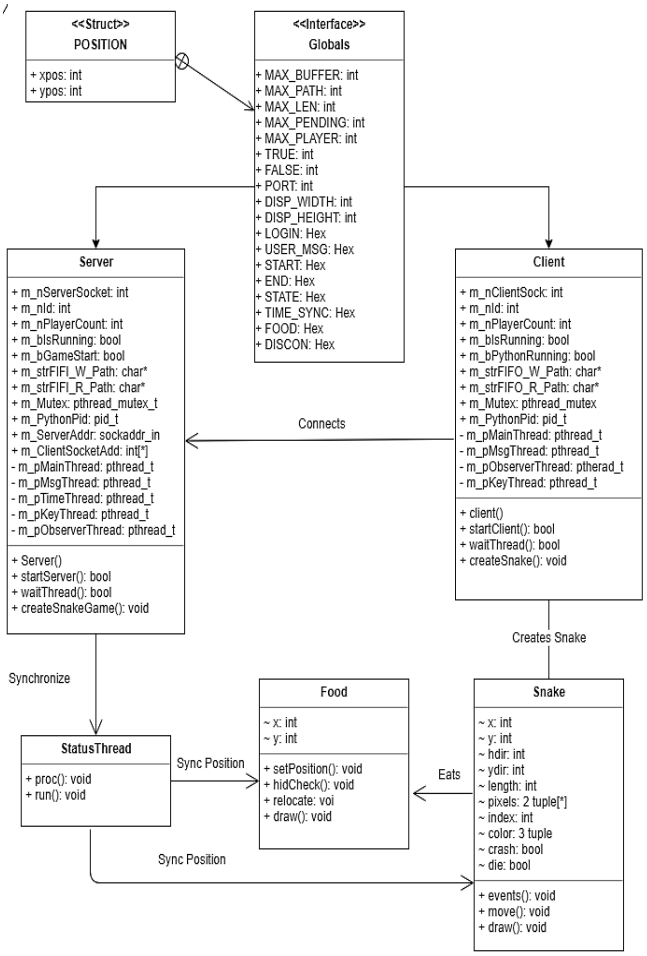


Fig.1. Class Diagram

a. Globals: This a header file that simply describes all the static attributes for our game, for example, the maximum number of players, the size of the game window, certain Boolean constants, etc. These values are stored in the back-end to reduce/ avoid using magic constants and updating certain game-wide global variables becomes easier.

b. Position: this is a struct, that is defined in the "globals.h" header file. This struct is used to define the position of an item, in the game, be it, a snake or food for the snake.

c. Server: This class encapsulates the server for the game. Any player can choose to be a server by using the "-s" option while running the game. The function of this class is to start a TCP server bound to the host's IP address, listening on the port that is specified in the globals header file. The server then waits for all the players to connect (specified by the MAX_PLAYER in globals.h). Once all the players have connected to the server, the server initializes the game and sends the required game data to all the connected clients.

d. Client: Each client represents a (non-server) player in the game. The client is responsible for

instantiating a snake object based on the parameters provided to it by the server, for example, the player index.

e. Snake: The snake class encapsulates an in-game snake, that is controlled by one player. Each snake object is constantly listening for key-events. On the appropriate key event (left, right, up or down) the events function will update the parameters of the snake respectively.

f. Food: The food class represents the food, once consumed, will increase the length of the snake.

g. StatusThread: This class is responsible for thread synchronization amongst all the players and ensures that all the players have consistent copy of the game.

B. Activity Diagram

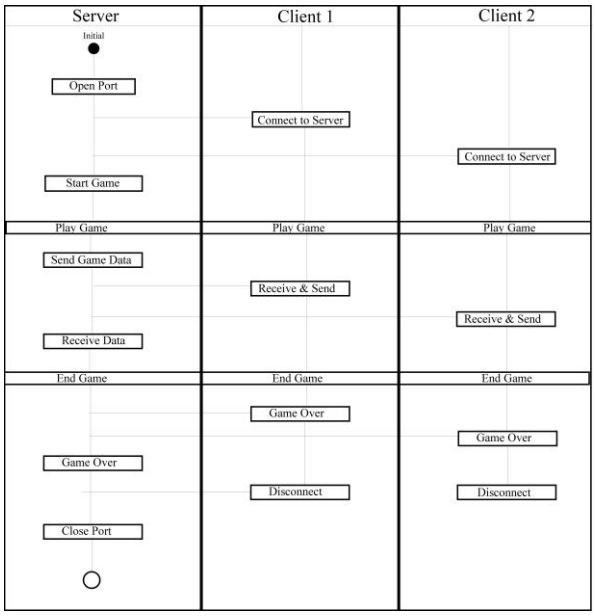


Fig. 2. Activity Diagram

a. Initially server opens the connection ports for the clients to connect to.

b. Clients connect the server through the ip of the host network. After the connection is established successfully, the clients wait for the user to start the game.

c. When the server starts the game, the data is transferred from the server to client and client to server. Client sends the updates to the server and then server sends that data to all the connected clients.

d. When a client loses the game, it is not disconnected immediately, but it keeps on receiving the data of the game from the host network.

e. When one of the players wins the game, the server sends the signal to end the game. All the clients are disconnected from the host network and the connection sockets are closed by the server.

VI. TUTORIAL

Following are the steps to compile and run the game, both as a client and a server.

A. Compilation

We have provided a compile script that compiles the required files. The script is a bash script and may require execute privileges to run.

Run the following command to assign required privileges:

$ chmod +x build.sh

To run the script, enter the following command:

$ ./build.sh

This will create an executable file named, “snake”, and a copy of the snake.py into the /bin/ folder.

```
bash$ chmod +x build.sh
bash$ ./build.sh
bash$
```

Fig. 3. Compilation and build

B. Running the Game

As mentioned previously the game can be run with two options, viz. server and client.

If you wish to be the server, you need to run the game with the “-s” option indicating to the backend that you are the server. Note that, for other players to connect to your game, they will need to know your IP Address. To fetch your IP Address, you can use the ‘ifconfig’ command. Once you know your IP Address, you can start you server with the following command:

$ ./snake -s

This will start the server and will wait for players to connect. You need to wait for at least one player to join your game. The game doesn’t run as a single player game.

Once you have at least one player connect to your game, you can start the game using the following command:

-c start

This will open a new window, one for each player connected to the game, and will continue till just one snake remains.

```
prteek@Prateek: ~/finalproject-ching_patel_pisat_fina
bash$ ./snake -s
Listen on port 8080
Waiting for incoming connections ...
-c start
```

Fig.4. Server Run

If you wish to be a client, you need to start the game with the ‘-c’ option indicating to the backend that you wish to connect to an already existing server. As mentioned earlier, you need to know the IP Address of the server. Once you are aware of the IP Address of the server, you can connect to a server using the following command:

$ ./server -c <ip_address>

If the IP Address was correct, then the server will accept the connection and appropriate message will be displayed to the client. Then the client needs to wait for the server to start the game.

```
bash$./snake -c 192.168.150.1
Login success. Please wait until start game
```

Fig.5. Connecting to the Server

```
bash$./snake -s
Listen on port 8080
Waiting for incoming connections ...
New connection , socket fd is 4 , ip is : 192.168.150.1 , port : 62864
Player 2 has joined.
New player created. Current player count is 2
```

Fig.6. Server Accepting a Connection

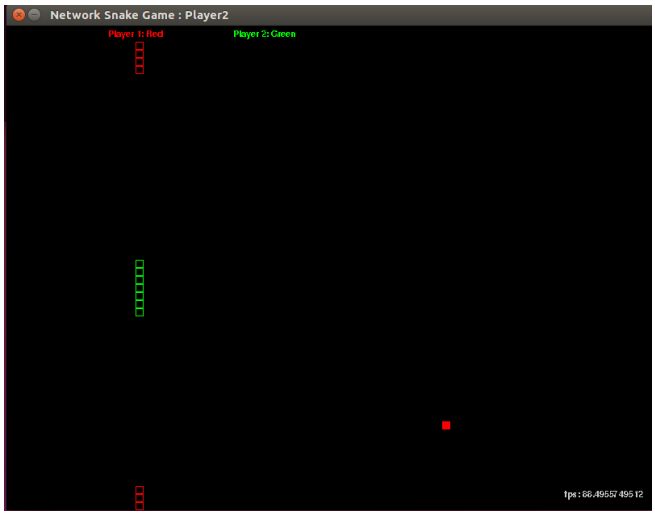


Fig. 7. Player-2 Screenshot

CONCLUSION

Using Mutexes and Thread Synchronization across multiple clients, our game can run at a smooth 60 Frames per Second, with minimal network latency. Furthermore, a local python thread ensures a consistent playing environment, eliminating the effects of the network latency, if any. Using a stable and well documented game library make the job of developing the game much easier and much faster, allowing us to focus on the more taxing parts of the project. Using named pipes/ FIFO files, inter-process communication is simplified, which are used to sync the local game data with the global game state. Also, dividing the functionality into distinct components/ classes, provides modularity, makes the code easier to manage and adding new features much easier.

REFERENCES

These are the research papers and websites that we used as a reference.

[1] https://www.dreamincode.net/forums/topic/402599-computer-networking-two-player-interactive-game/
[2] https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
[3] https://www.geeksforgeeks.org/socket-programming-cc/
[4] https://www.pygame.org/docs/