Aleksander Nowicki and Samuel Young
Final Project Writeup
CS60 Fall 2025
11 / 21 / 2025

Rollback Netcode for a 2-Player Fighting Game

Summary:

We used python to implement a simple, proof of concept version of a 2-player arcade-style fighting game (think “Street Fighter”). We then implemented two “netcodes” (system that the game uses to communicate between players over the network) which allow the game to be played between players on two devices over the network. The first, “delay” based netcode simply waits to receive inputs from both players before advancing the game, which leads to delays, high input latency and lost inputs in poor network conditions. The second “rollback” netocode runs the game at a constant speed on each device, making educated assumptions about the remote players’ inputs when network communication is delayed and re-simulating once remote inputs do arrive.

Usage Instructions:

Dependencies:
You will need to install pygame-ce (pip install pygame-ce)

To Run the game in Delay Based Netcode Mode:
As Server:
python3 driver.py <port>

Type 1 and press enter

As Client:
python3 driver.py <ip> <port>

To Run the game in Rollback Based Netcode Mode:
As Server:
python3 driver.py <port>

Type 2 and press enter

As Client:

```
python3 driver.py <ip> <port>
```

Driver:

File: driver.py

Summary:

Driver parses the command line arguments and gets user to choose from a menu to determine whether the program will run as server or client during the initial connection phase (after game start, connection is peer-to-peer with no distinction between client and server), which netcode to use and what port (and potentially IP address) to use for the connection. It then establishes a connection between the two devices and passes off relevant information to delay or rollback netcode functions.

Functions:

parse_args(command_arguments)

Parse sys.argv to return player role (server or client) port and ip

get_mode()

Let the user pick which netcode to run by entering a number (delay or rollback)

Game Logic:

File: game_logic.py

Summary:

The game is implemented using the pygame module (pygame-ce). Each player controls a colored square, which can move left (a key) and right (d key), as well as attack (s key). Once an attack is initiated, it takes 60 frames to complete, during which time an animation is displayed. Once the 60 frames are complete, if the other player is in range their health is reduced by 5 points. There is no win logic (not relevant for proof of concept)

The game code is designed specifically to work well with rollback netcode. Namely, game simulation and rendering are entirely separate. This allows multiple frames of gameplay to be re-simulated in a rollback scenario without interrupting the player’s view. Game simulation must also be very fast to allow this. Since only controls are passed between the two players, the same control inputs must always produce the same outcome. Therefore, we avoided using floating point math, which can tend to vary between devices.

Data Structures:

GameState
Stores entirety of game state in a given frame, including player positions etc. Easily stored to and loaded from a list.

ControlState
Stores a single player’s control inputs for a given frame. Easily stored to and loaded from a list.

Functions:

render_frame(game_state, window)

Displays the given state of the game (game_state), which is a GameState object, to the given pygame window.

update_state(current_game_state, p1_control_state, p2_control_state)

Returns a new GameState object which contains the game one frame after the given current_game_state calculated using the passed control states (p1_control_state and p2_control_state) to determine player actions

Testing:

The **game_local.py** file utilizes the functions from **game_logic.py** to implement the game locally (where player 2’s controls are jkl). This allowed testing the basic game functionality separate from issues potentially introduced by the network.

Delay Netcode:

File: delay_netcode.py

Summary:

This module implements online multiplayer for the game using a delay-based netcode system. Effectively, the code samples the local player's input and sends that to the remote player. The program then blocks, waiting to receive input from the remote player before running the game simulation for that frame with both the local and remote input.

Functions:

encode_control_message(frame_number, control_state)

Returns a byte string with a message of the format: FS|frame_number|control_state|\n
Where control_state is stored as a list and then formatted in json.

decode_buffer_first_message(buffer)

Takes the input buffer (which is a byte string), extracts the first message stored there, parses the message and then returns received frame_number, control_state and the remainder of the buffer that was not yet processed

run_game(player_number, remote_socket)

Takes player number (1 or 2, 1 is left, 2 is right) and a socket on which to communicate with the remote player.

Loops infinitely until one or the other player disconnects. Set up pygame, then loop getting local input, sending it, waiting for remote input, and then advancing the game by one frame.

Testing:

Testing started with a connection within the local machine before testing between two separate devices. The delay_netcode module contains a flag which when set simulates a poor network connection by introducing a randomized delay before sending each control message. Our network connection was generally very good, so this simulated delay allowed us to discover an issue where multiple messages would arrive simultaneously, causing issues we otherwise would have missed. A large number of print statements, now removed, were utilized in testing.

To debug issues with connections and message format, wireshark was used to observe incoming and outgoing TCP messages.

Rollback Netcode:

File: delay_netcode.py

Summary:

This module implements online multiplayer for our game using a rollback netcode system. When bad network connection causes the remote player's inputs not to arrive in time to simulate the next frame, local simulation can continue using assumptions about what the remote player's input is likely to be. When the remote player's input eventually arrives, the game is "rolled back", returning to the state it was in when that input occurred. The simulation is then re-run back to the current frame, filling in guesses for any input that has not yet been received. This all takes place without changing the view being displayed to the player, with the goal of being un-noticeable.

Data Structures:

rollback_list

This is a list of dictionaries, each with keys: local_input, remote_input, game_state_str. Each index in the list stores the inputs for the local and remote player as well as the game state for each frame of gameplay. By getting the length of the list, we can determine the current frame through which we have local data/ have simulated gameplay. Access to rollback list by multiple threads simultaneously is carefully avoided using lock.

Given more time for optimization, this list would be better replaced with a rolling window, which deletes game / control states more than a few seconds in the past, which will never be used again and are a waste of resources to store. As it is, the length of the list becomes unmaintainable after several minutes of runtime.

Functions:

encode_control_message(frame_number, control_state)

Returns a byte string with a message of the format: FS1|frame_number|control_state|\n
Where control_state is stored as a list and then formatted in json and 1 could be 2 for player 2.

decode_buffer_first_message(buffer)

Takes the input buffer (which is a byte string), extracts the first message stored there, parses the message and then returns received frame_number, control_state and the remainder of the buffer that was not yet processed

listen_thread(remote_socket)

Waits to receive a valid message from the remote player. Then, based on where this message is in relation to the local frame number, it takes one of several actions. If this happens to be the input for the same frame as the current local frame, we create a new entry in the rollback list with this remote input. If this input is for an earlier frame, move back to that index in the rollback list, input the received input, and simulate the game back up to the current frame.

run_game(player_number, remote_socket)

Initializes pygame. Launches a thread running listen_thread to listen for remote inputs. Sends a message to the remote player to start the game and then waits to receive one back to begin. Then, starts an infinite loop to collect local input, send that input data to the remote player. Appends input data to rollback list, guesses at remote inputs if needed and advances the local game state, storing that to rollback_list

Testing:

Testing was initially done by hosting the game locally on two separate terminals, the effects of the netcode could be observed. The connection over two separate terminals on the same machine is very fast, so minimal delays were experienced and so the effects of the netcode were not extremely noticeable. We then implemented another version of the file which introduced artificial delay before sending packets, resulting in the rollback being more noticeable.

Major Deviations from Plan:

Driver: Since we decided not to utilize a built in delay for the rollback netcode, we do not get that information from the user. Further, delay-based netcode turned out to be easy enough to implement that we did not both implementing a test message-only mode, which would have served limited use. We originally intended separate threads to be launched by the driver, but since threading was only required for rollback and not delay netcode, this did not make sense.

Game Logic: Game logic was mostly complete before the specs were written, so few, if any changes were made to that code.

Delay Netcode: Conceptually this remains unchanged from our specs, although we considerably underestimated the difficulty of formatting outgoing messages and processing incoming messages.

Rollback Netcode: Rather than a list of tuples as mentioned in our requirements spec, we used a list of dictionaries to store the game states. We also did not find it necessary to maintain pointers to the last known remote input since we would only receive inputs from before the current frame. Finally, the message format ended up being somewhat more complicated, although the information conveyed was largely the same. See section on rollback netcode for message format.

Currently the case where one machine is drastically ahead of another (i.e. in the case where one player's hardware is significantly weaker than the other's) the program does not function properly, as it just prints out the difference between the two machines in order to error test. We did not observe this error when testing and so did not feel that it was urgent to fix it. The implementation also drastically cuts down on the number of threads used, which may hurt performance but is more simple to understand.

Reflection / Future Improvements:

This simple implementation served to prove the feasibility of the rollback netcode system in the context of a simple python game and to cement our understanding of rollback netcode and general networking concepts from class. For example, a solid understanding of TCP and its functionality as a byte-stream, rather than individual messages like UDP was critical when we encountered an error where multiple messages were being read simultaneously. We also utilized strategies studied in class and for lab assignments in order to simultaneously handle operations while awaiting network messages.

Our rollback netcode implementation is functional, but has several limitations that we would address given greater time and scope. First, because each frame is simply appended to the rollback_list, without any entries ever being deleted, the list becomes longer and longer as the game progressive. After several minutes of gameplay, dealing with this list becomes problematically slow, leading to poor performance. A rolling window approach, where only the past few seconds of game states were stored, would fix this. In our implementation spec, we mentioned adding a timer, which could have also used to solve the problem if we had more time, but did not feel was necessary for a proof of concept.

Also, while our game implementation allowed us to demonstrate consistent movement and hiding network delays in attack start-up animations, the game has no win state. This is an important potential point of contention in online multiplayer games, and how disagreements between devices are handled here deserves further investigation. Finally, it would be valuable to implement some kind of inter-device communication regarding the current game state, which would allow for occasional checks and corrections incase the two players somehow became out of sync.

Sources:

The following articles were what inspired this project and were very helpful in developing our fundamental understanding of rollback netcode.

https://words.infil.net/w02-netcode-p4.html

https://drive.google.com/file/d/1cV0fY8e_SC1hIFF5E1rT8XRVRzPjU8W9/view