Block Party: An Online Platforming Game

Joshua Bridges, 2203030@abertay.ac.uk

Architecture

This project began with the idea of a small lobby containing a select number of clients. The application was then designed to be a turn-based platforming game, where the players take turns adding to the level and making a gradually more complex experience. With these goals in mind, the decision for the architecture of this platform was between a single server that multiple clients would connect to, or a peer-to-peer setup. The memory of older multiplayer games that used clients as "hosts" for each lobby initially made it seem like peer-to-peer architecture was the right decision. However, shortly after starting to plan the connections, it became apparent that this was the wrong course. Shown in Figure 1 and Figure 2, the memory overhead for peer-to-peer architecture would be much larger than the client-server system. The switch to the client-server architecture also cut down the necessary number of TCP channels for communication. The drop from six to four channels may not seem very impressive but it is a 33% reduction in the required TCP communications. This was important to the project because the TCP communication carries a much larger overhead than UDP and with four clients may prove to be a bottleneck in the performance.

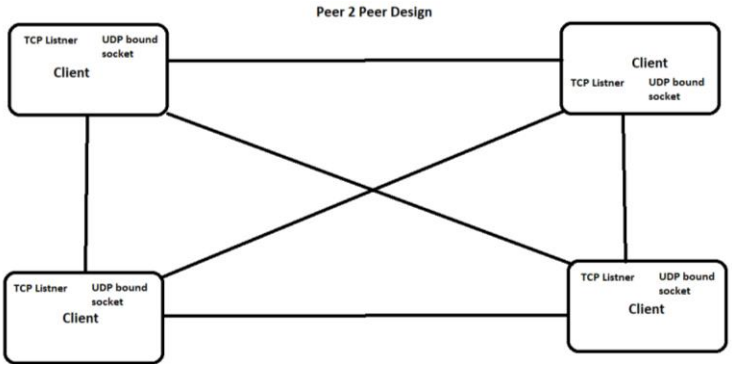

Figure 1: Design of a Peer-To-Peer approach to the game architecture

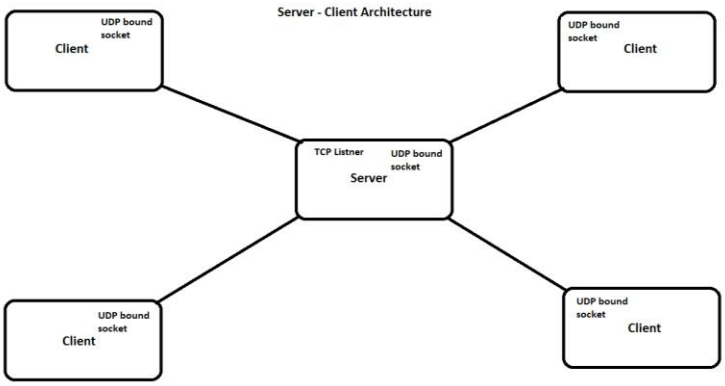

Figure 2: Design of a Client-Server architecture implementation.

Protocols

The transport layer protocols were not a decision between TCP or UDP, but rather how and where to use each protocol at the application layer. TCP was chosen for its consistency and guarantee that the data will arrive intact and in-order. This was preferable for situations where there would be infrequent communication as well as necessary information for clients to have accurately. In the application layer, TCP protocols were used to send updates about the game state. Score, new blocks, and new player connections are all updated infrequently and only via one communication. Therefore, these updates

made great use of the TCP protocols to not generate a substantial overhead while also guaranteeing communications. On the contrary, UDP protocols were used where there was a constant stream of communications. In this application, UDP was utilized for position information. Each client and the server contain a bound UDP socket which is used to send delta snapshots of the client's position to the server. The server then relays these updates to the other clients. As these updates can come very frequently, in almost every frame, TCP would have created slow communications and bottlenecks. However, since UDP does not guarantee order, the frame number for each position update is used in the application to discard old updates that have arrived out of order.

API

SFML was used as an external library to write the network code in this application [1]. This API demonstrated a remarkable improvement over the base WinSock library. First, readability was of paramount importance for an introductory project in networking. SFML provided simple functionality to create and connect TCP and UDP sockets that worked as well as you could request without needing the many different setup functionalities that WinSock needed. In addition, every function call returns an SFML status code that provides more detailed information on the status of the socket when necessary. This detail was important for debugging much of the program and changing behavior based on the status of a socket. Finally, and most importantly, SFML provides a very useful Packet class for wrapping information. The simplicity and versatility that this change brought to the project cannot be understated. Many initial errors with the networking code were due to mismanagement of the information to be sent. SFML packets cleared up every issue in that department, providing a clean way to order the information in each packet and access it in that order.

This API came with slight limitations to the networking code. Quick searches revealed that SFML did not have the capacity for asynchronous IO built in so the code would need to rely on non-blocking IO which is functional but not ideal due to wasted CPU cycles.

Integration

The network communications in this project were built into a custom platforming game created in C++ using several online tutorials as inspiration [2]. Instead of having multiple players on one screen controlled by multiple users, this project uses TCP and UDP to facilitate multiplayer gameplay via a server. This connection is achieved through each client automatically connecting via TCP to the specified server IP address. This method requires the server to be running prior to the client launch and the client to have total knowledge of the server IP and port for TCP and UDP beforehand. A configuration file can be a flexible solution to this situation. However, currently due to file structure, the IP and ports for the server are hardcoded as global definitions.

The networking code in this project is divided into two key sections: client application and server application. The client application has a set of functionalities that facilitate communication with an external server. On the other hand, the server application has code set up to facilitate up to four clients

connecting to the “lobby” at any given time. Both applications require the same standard setup for communication as shown in Figure 2. The clients utilize a single TCP and UDP socket each while the server has a UDP and a TCP listener to receive the client connections.

The client code is structured around the standard game loop, occurring once every frame, at 60 frames per second. However, before that can happen, the client connects to the server by connecting to a TCP listener on the server. The client is then assigned one of four available players and initialized by receiving the game state from the server. Once this information is received via TCP, the UDP socket is set up on the client and the sockets are then set to non-blocking for the remainder of the game. In the main game loop, the client checks the header of any TCP packets it receives and calls respective server methods to handle the outcomes. There are six different types of TCP messages the client can receive during gameplay: next turn, new block, score update, new player, disconnect, end of game, and reset. UDP is then used for the remaining task, position updates. When the UDP socket receives a packet, the application reads the frame number to identify if the packet is in order. The application then saves the position information in the frame history for each player to use with prediction.

The server functionality is built around the server itself rather than any game application. More specifically, the primary feature is the main server loop. The server handles initialization for itself, so the sockets never need to block, and are set to non-blocking IO from the start. The server handles a dynamic list of the client connections that it iterates through during the main server loop and checks for any transmissions. The use of an SFML socket selector was explored to remove the loop for this check, however, the selector requires a blocking call to a wait function. This would therefore freeze the server and stop UDP communications, which is detrimental to performance. The clients are stored with their TCP sockets and their index for UDP communication. The index is used to create a unique UDP port for each client since testing was occurring on the same IP address.

The result of this non-blocking setup on the client and server applications is a seamless flow of information back and forth. The application doesn’t need to worry about the readiness of the sockets because once they are ready the loop will reach them and parse all the necessary information. All the applications simply need to be ready to handle each type of packet as they arise.

This implementation does have a cheat in it that would be a major drawback to official applications, that needs to be recognized. The server is not authoritative, meaning that while the server relays information to every client it does not check the information a client sends and trusts the client to be correct about itself. This creates a large vulnerability for potential cheaters and exploitations, but also massively decreases the overhead during runtime and removes many sources for runtime issues due to latency. In future work on this project the server will be made into an authoritative server to remove these vulnerabilities.

Prediction

Prediction proved to be the most interesting part of the application to resolve. This occurred because of some key insights that indicated prediction was either harmful to the application or simply not beneficial to the performance from the client side. Nevertheless, the code to run prediction is implemented in the code base and readily available to view or activate at will.

The decision of how to implement prediction came down to what was desired from the application. The physics engine of the application indicated that quadratic prediction would be the preferred method. However, during implementation, and after checking the math, it became apparent that quadratic prediction would never predict the player to come to a standstill. Additionally, a large aspect of the network position updates is that the server updates every *other* player but not the turn player themselves. That is because the client is the ultimate source for their own position in this game. As a result, prediction only ends up being important for the spectating players during any given turn. The final deciding factor was the turn-based nature of the game. The players are not acting simultaneously, and subsequently, the exact position of each player on a good or bad connection does impact the gameplay. The main importance of implementing prediction and interpolation is to blend the position of players with a prediction in situations of a poor connection. However, in this application, latency does not break the experience. Instead, latency simply delays the game while it's not your turn. Then during your turn, the quality of the network connection does not come into play at all.

Due to the nature of this project, it was still necessary to create a strategy and rudimentary implementation for prediction. Prediction used a quadratic formula to account for accelerations on the client. As a result, contrary to linear prediction's two frames, this application stores the past three frames of position data for a client in a queue seen in Figure 3. During any frame where position data is not received from the server, prediction runs to generate a new coordinate for the player. Then, whenever the new position is acquired from the server, the game blends the last predicted position with the new true values. The blending is performed through simple linear interpolation once these new communications finally succeed

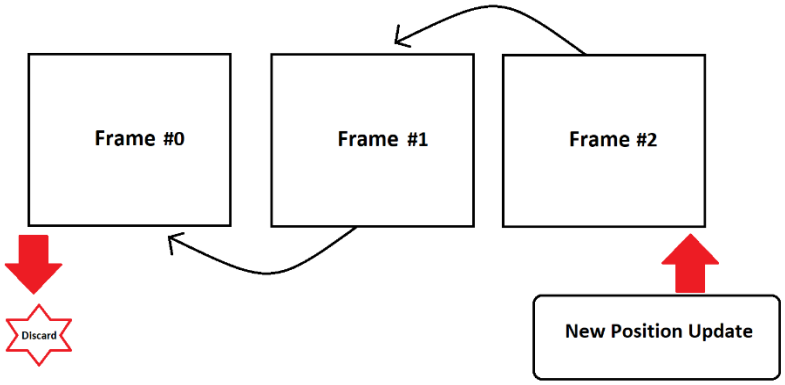

Figure 3: Layout of frame storage for use with quadratic prediction.

The culmination of all these ideas was the realization that prediction was not beneficial to any client during the experience. However, it was important to the project to attempt this predictive behavior. Therefore, prediction is built into the application and, for better or worse, adjusts the player position when there are no incoming updates.

Testing

Practical analysis of the project was performed using the Clumsy tool to simulate dropped, slow, and out of order packets on the local network. Four different types of scenarios were investigated: no-prediction-perfect-network, prediction-perfect-network, no-prediction-poor-network, and prediction-poor-network.

The results for no-prediction were impressive even with the poor network conditions. Significant issues only began to arise when the dropped packet percentage became 30% or higher. However, the worst situations to arise were a slightly different waiting position, which is unimportant, and delays in turn changes/score updates. The turn change delay would be the most impactful issue to work towards solving in the future. When the drop percentage is below 50%, there was consistency in the signals being delayed, but arriving nonetheless. Above 50% drop percentage, the server messages were observed to not arrive even after significant waits. This caused a detrimental impact on the game. A possible solution to this may be to make the messages use UDP protocol instead. However, that would require more robust checking and further testing to confirm.

Prediction testing displayed gameplay bugs that were visual, and did not impact the direct gameplay, but were still readily apparent. First, the prediction test displayed the same high chance fail case at a greater than 50% drop percentage that no-prediction did. Second, flickering was observed for the predicted player's movements. Third, at some points the players that are being predicted can disappear from the game window altogether. The last two issues mentioned are almost certainly correlated and are probably the result of issues with the three-frame queue.

In conclusion, the current network code is not without its flaws in poor network conditions and with many dropped packets. However, even at 300ms of lag the gameplay experience is not heavily impacted, prediction or no prediction.

Appendix

1. Gomila, L. (2007) *Simple and fast multimedia library*, *SFML*. Available at: https://www.sfml-dev.org/index.php (Accessed: October 20, 2022).

2. Zaidan, D. (2019) *Pong Learn Programming*, *GitHub*. Available at: https://github.com/DanZaidan/pong_learn_programming (Accessed: October 10, 2022).

Addendum

Thank you for reading my report on the Block Party game that I developed. I learned so much about network code and building applications with network compatibility. However, I remain slightly disappointed in how a few elements of this project ended. While I ran out of time to fix everything I wanted, there are some issues that I want to highlight for future improvement.

As mentioned in the paper, I would like to make the server authoritative to remove the availability of exploits coming from the client application. Similarly, poor network conditions occasionally delay TCP communications. While UDP is built to ignore this, the application does not handle the situation after a TCP signal is sent to the server and before the server sends a response. This would provide a significantly less “jank-y” experience for any user with bad connection. The largest area to revisit would be prediction and interpolation. The implementation in this application is barebones, and can be seen in the demonstration video to cause flickering rather than improve the situation. Despite how it seems, this issue likely does not arise from the network code or the prediction calculations. Rather, the problem comes from the way the game calls requisite functions and stores the frame data used in prediction.

These three features: the authoritative server, TCP delay control, and proper prediction would be the most important changes to improve the application performance in poor network conditions.