Networking Application Documentation

Shubh Ravishankar Gawhade

MSc Computer Games Technology, 2023-24

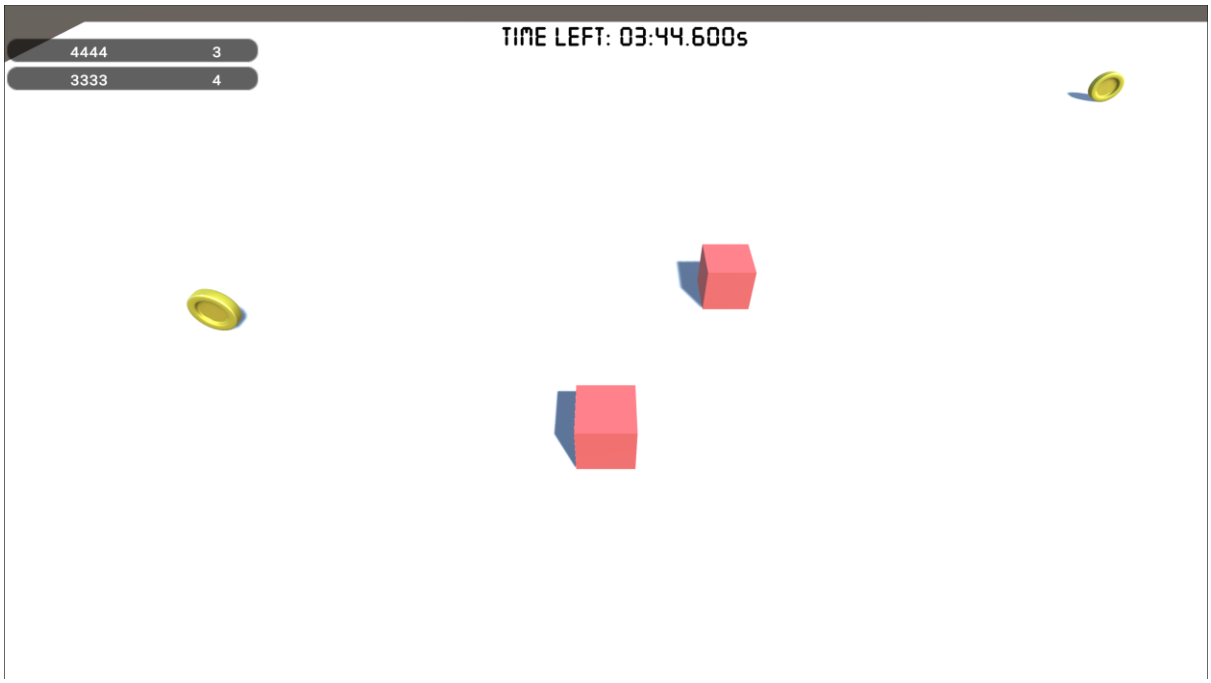

Figure 1: Networking Application Screenshot

School of Design and Informatics

Abertay University

Contents

List of Figures

Summary

The project uses a client-server architecture, which is a common model for networked applications. In this architecture, there is a server that hosts the game logic and the game state, and there are clients that connect to the server and interact with the game world and other players. The clients send their inputs (such as key presses and commands) to the server, and the server processes them and updates the game state accordingly. The server then sends the updated game state back to the clients, who render it on their screens *(Figure 2)*.

Cheating is a major issue in multiplayer games, as some players may try to manipulate the game state or the network communication to gain an unfair advantage over others. To prevent cheating, the project uses an authoritative server, which means that the server is the only entity that has the complete and correct game state, and the clients are just dumb terminals that display the game state and send inputs to the server. The server validates all the data and inputs from the clients and updates them with the information. The server also does not reveal any sensitive information to the clients, such as the server-side code or the hidden state of other players. This way, the server can ensure that all the players have an equal and consistent experience of the game and that no player can cheat or tamper with the game state. This architecture has several advantages and disadvantages for multiplayer games, especially in terms of cheating prevention and performance.

Some of the advantages of using an authoritative server are: Equal experience for all connected players, a Centralized system for all data and actions, a High level of scalability, organization, and efficiency, Reduction of data replication, and Server-side code is not available to players.

Some of the disadvantages of using an authoritative server are: Prone to denial-of-service (DoS) attacks, Data packets can be spoofed or modified during transmission, Expensive to start up and implement, if the server goes down all the players lose connection immediately, it is also prone to phishing and man in the middle attacks, the server is infected by a virus it can be transferred to the client.

Another challenge of using a client-server architecture is network latency, which is the time it takes for the data packets to travel between the server and the clients. Network latency can vary depending on the distance, the bandwidth, the congestion, or the quality of the network. Network latency can introduce a delay in the gameplay, which can affect the responsiveness, accuracy, and fairness of the game. For example, a player may press a button to perform an action, but the action may not happen immediately on their screen, because the input must be sent to the server, processed by the server, and sent back to the client. This delay can make the game feel sluggish, unresponsive, or out of sync. This delay can also cause the players to see different versions of the game state, which can lead to conflicts, errors, or inconsistencies. For example, a player may shoot at another player, but the other player may have already moved away from that position because the shooter's

client has not received the latest game state from the server. This can make the game feel unfair, inaccurate, or unpredictable.

To mitigate the effects of network latency, the project uses several techniques, such as prediction, reconciliation, and interpolation. These techniques aim to reduce the perceived delay and improve the smoothness and consistency of the gameplay. These techniques are explained in more detail in the following sections.

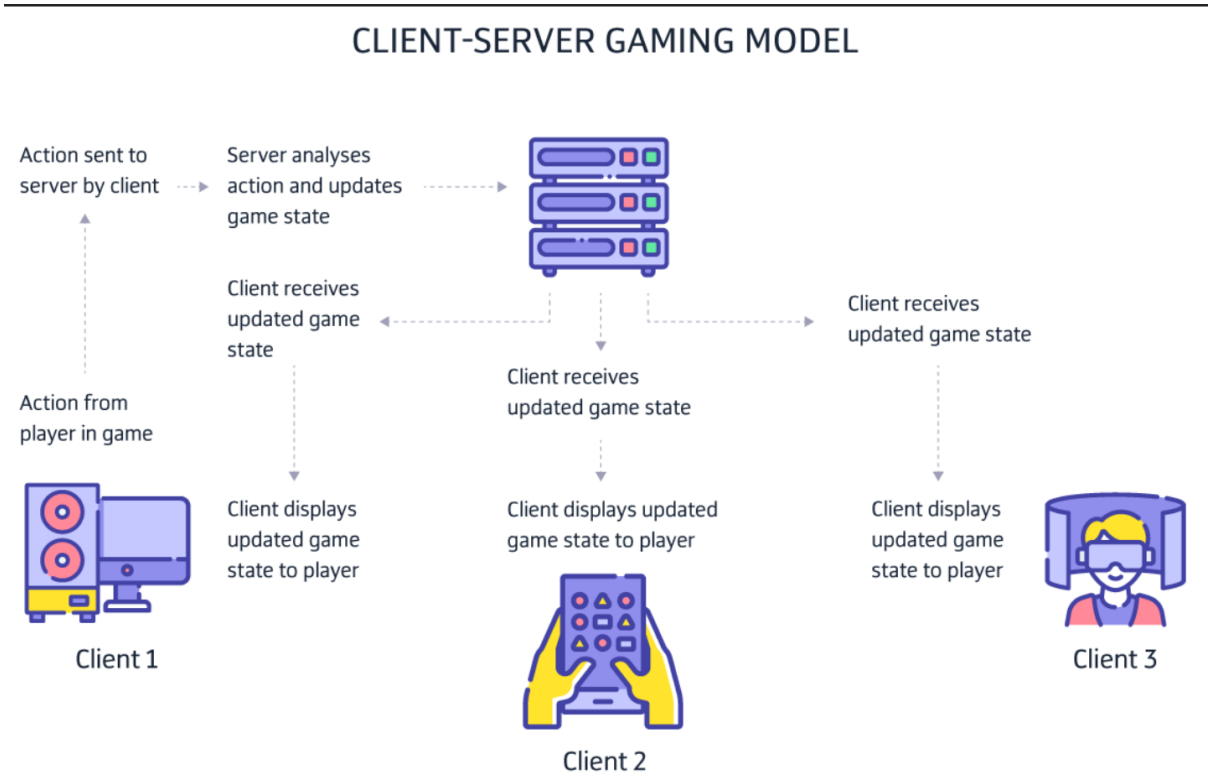

Figure 2: Client - Server Model

Protocols and API

The game uses the reliable TCP protocol [1] to send all pieces of information such as positions, inputs, and other connection data. TCP is a reliable and connection-oriented protocol that guarantees message delivery and order. Since the game isn't a fast-paced game, and can manually limit the number of packets transmitted, TCP works best for it. The game also uses a custom application layer protocol developed in Unity, which defines the format and meaning of the messages exchanged between the server and the clients. The protocol uses different types of data structures to indicate the content of the messages. The protocol also provides methods for converting objects to byte arrays and vice versa, using serialization and deserialization techniques.

.net sockets using asynchronous sending and receiving. The program also uses a queue to send and receive data so that the data isn't corrupted, or the machines aren't bombarded with too many requests. The average packet size noted during transmission is around 200 bytes with the largest packet being 274 bytes for the reply from the server when a player joins as it sends the list of players and their status. The smallest packet is 143 bytes containing the player along with the ready status.

Since Unity runs on a single thread, the server loop needs to run asynchronously on another thread so that Unity can run the game code at the same time. The send-receive loop starts with the server listening on a port at an IP address. When it receives a connection request, it accepts it and creates a new player to store the player's socket.

The player class holds many different values and functions related to the player. It holds the player's name, ID, score, owned objects, and byte arrays to send and receive data, "DataUpdateType" struct to know what kind of data is being received. Other data structures included are the different types of data. The class also includes functions to return a data structure to be populated, convert an object to a byte array, and convert a byte array to an object based on the data update type.

Once the player connects, it creates a new player like the server and stores the socket and player name. It then creates a struct and converts it to bytes to send this information to the server. There are 2 send classes, one for the player and one for the server with the player having limited functionality of the functions in the server send class. They are asynchronous and can be awaited with the "await" function. It takes in 3 parameters, the player class, data to send in bytes, and the sent type which is an enum that allows 3 ways to send data. They are: "ReplyOne", "ReplyAllButSender" and "ReplyAll" which are self-explanatory. The player can only send data to the server. On calling this method, it gets the size of the message, appends the data update type as an integer, and calls the internal send method which first sends the size of the message and then the data itself. The server and the players receive functions work in the same way by calling the "CheckForDataLength" method which receives the size of data which is always 4 bytes. It then reads the size of the data from the first 3 bytes and the data update type from the last one by casting it back to its type. The "ReceiveData" method is then called which receives the data of size read from before. It

then deserializes the data based on update type and creates a struct to store this data, its update type, and the player which is added to a queue to be processed by “HandleDataServer” or “HandleDataClient” respectively. Finally, the “ReceiveData” method resets the buffers and calls the “CheckForDataLength” method again and this loop continues for every client which is connected.

Since the game is server authoritative, the server processes all the data before it is confirmed on the client ranging from joining, the client's ready status, registering game objects in the scene to collecting coins, and updating the score. The joining data from the players is validated, and an id is assigned as I didn’t want the players name to be the unique id for each player since people can have the same names. The player is then added to a list which will be used to send and keep the information about all players as well as provide each of them with a copy of the necessary fields.

Inputs and other sent data must be processed at the same time and intervals for it to produce the same output. But time is relative and varies by a bit for all devices. Thus, each device needs to be able to measure time independently and at the same intervals. To overcome this, I created a tick-based system [2] that starts a timer on the server increasing the tick by 1 for every $1/30^{th}$ of a second. The clients when connecting, must sync themselves to the server tick for them to be able to have a time reference for every action. I achieved this by calculating the time it takes to connect to the server and the server to send its tick and sub-tick values in response. This gives us the round-trip time (RTT) or ping. Adding half of its value to the server tick value syncs both clocks.

The game must also start at the same time for everyone. The time left for the game should also be synced across players. To achieve this, when the last player loads the scene, the server sends everyone a tick value which is equal to the current server tick plus a delay value of the tick from the data sent by the last player to load and the current tick. Once the players receive this value, all players wait for the tick after which the players can control their characters.

Prediction and Interpolation

When the player tries to move their character, the game stores it on the player and sends them with a tick timestamp for when it was performed. Inputs are sent only when there's a change in the input values rather than every few times a second so as to not clog the server with traffic. The inputs are then simulated and displayed to the user. Using this method is possible with only deterministic calculations. A deterministic calculation always returns the same output for any given input(s) therefore, making it easy to predict the future. [3] Meanwhile, the server takes these inputs with their timestamps of when they occurred and moves the player back in time from its player location buffer which stores values for the player for every tick including position and input data. Taking this and previous values, the server re-simulates the player to the current tick. It continues to simulate the last sent values till it receives new ones and repeats the process several times every tick *(Figure 3).* If there's a change in player position, the server sends that data every 10 ticks (0.33s) with the current timestamp. Back on the player's side, the ticks are already simulated and displayed along with the values being stored. When the player receives this data, it cross-checks if the data is right along its calculations, and if not, it goes back to that point in time and from the input data stored on the player, it re-simulates the player till the current timestamp *(Figure 4)*. Giving the predicted output an error range, if there's a position error, it must be implemented on the player. This is called reconciliation and since the server always is the source of truth, the player will have to be moved to this position. [4] Moving the player suddenly causes there to be a loss of immersion and can look bad for the player. Therefore, the positions need to be interpolated between the predicted values for the current tick and the current values by a Lerp (linear interpolation) function which outputs values in between two values. The time function can be used to smooth out the interpolation and can be seen as smaller steps to provide a smooth transition. When there are multiple players connected, the server sends the state information to all the players. If the player is not in control of the object, the player cannot predict for another player. To solve this, much like the server, the player will try to predict the future position of any other player based on previous data. If there are any discrepancies in positions beyond an error value, the positions are once again interpolated. This is how the game keeps the view consistent and playable for everyone even though they have different latencies.

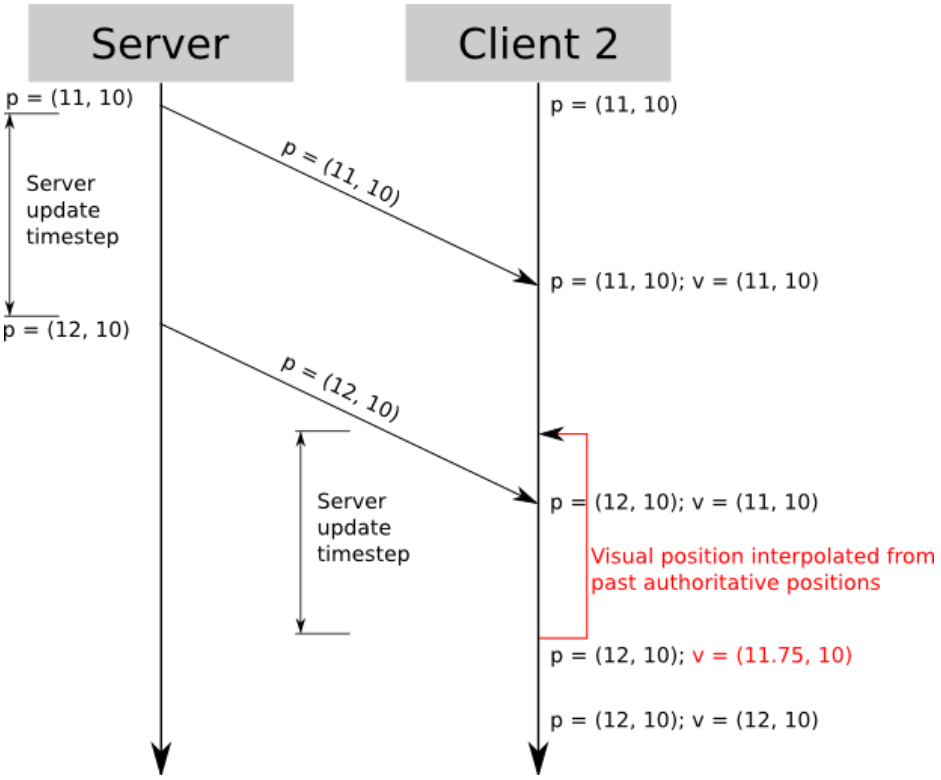

Figure 3: Client 2 renders Client 1 "in the past", interpolating the last known positions

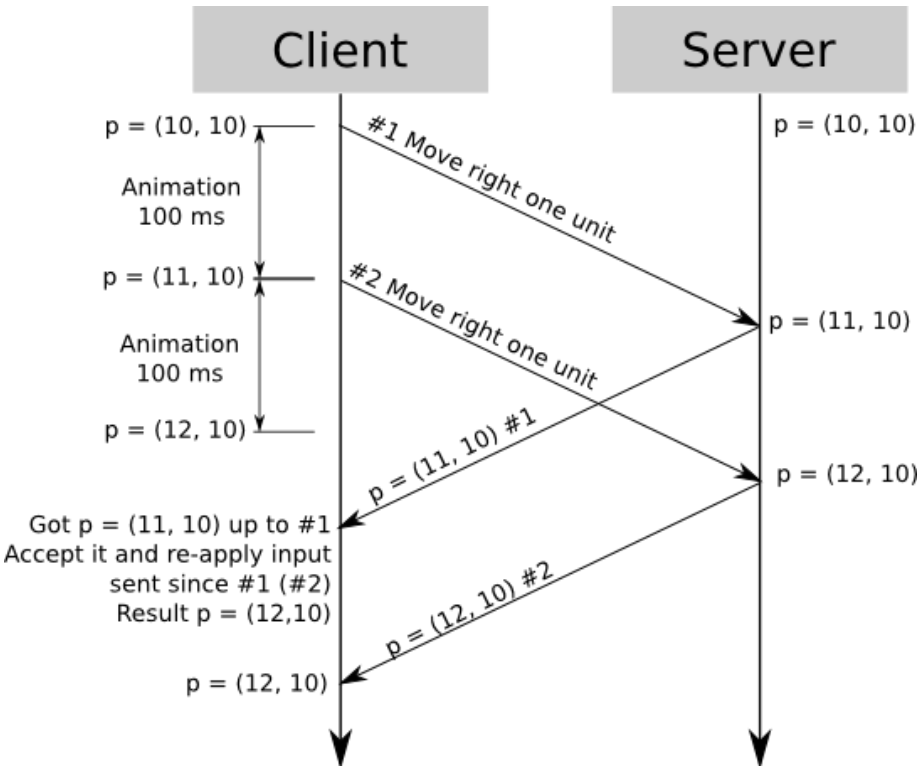

Figure 4: Client-side Prediction + server reconciliation

Testing

Note: Since the performance of the game heavily depends on the clocks being synced, all clients were first connected and synced before testing under bad network conditions to prevent errors due to unforeseen de-syncs. A future mechanism that regularly checks for de-syncs and corrects them would be ideal.

After testing the game under various network conditions using a tool called Clumsy [5]. Here's how it behaved under:

Lag (100ms) – The game is playable with noticeable reconciliation which feels like an unknown force pulling the player in a direction. On further testing, you can get used to it after getting a feel for the movement and the player ends up where it is expected with the controls as the reconciliation is quite accurate. For the other players, movement looks very smooth with minimal delay.

Lag(200ms) – The game is less playable with choppy movement for the player but still very accurate. It's harder to predict the player's location as the player keeps snapping to older states before its actual position making it harder to control and predict. For other players, the movement is still very smooth with a noticeable delay.

When testing over different devices, the game ran perfectly with variable latency like in the real world.

Drop (50%) – The game cannot handle dropped packets and can break the game if certain packets aren't received.

Throttle (50%) – The player experiences some choppy movements but can still control the player pretty well. For other players, the movement is smooth without delay.

Duplicate (50%) – The game is playable by the player and has rare, unnoticeable to very little reconciliation causing the player to stop/ move faster, other players see a consistent view.

Tamper – The game breaks by failing to decode packets and use the data.

TCP RST Packet – The game breaks due to this packet signalling that it was the last packet sent by a device and causes disconnection. This causes socket errors as the data isn't allowed to go through.

Bandwidth limit(10kb/s) – The player experiences little choppy movement noticeable on the server screen too. The other players do not experience it and see smooth movement between predicted points.

Bandwidth limit(5kb/s) – The movement is very choppy and in some instances, it causes the game to break by the positions not matching on the server and player. The other players see the server state which is not the players' actual position.

Most of the above problems and Desync issues can be solved by adding more error management for the server or client to quickly reset and pick up from where it left off.

Common deserialization issues occur if a lot of packets are spammed in a short duration. However, this normally doesn't happen due to the implementation of a queue system for handling messages. It can be prevented by checking for duplicate packets and processing only the latest packet with relevant information.

The networking program handles many of the various inconsistencies found in a normal network and can perform even better with some modifications to the prediction and better optimization of the packets to reduce traffic.

References

[1] IEvangelist (n.d.). Sockets in .NET - .NET. [online] learn.microsoft.com. Available at: https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/sockets-overview [Accessed: 12 October 2023]

[2] www.youtube.com. (n.d.). Networking for Virtual Worlds - YouTube. [online] Available at: https://www.youtube.com/playlist?list=PLoRL6aS9crowO6h2SL7k9lUV5eeC6uqnx [Accessed 5 Dec. 2023]

[3] Gambetta, G. (2019). Client-Server Game Architecture - Gabriel Gambetta. [online] Gabrielgambetta.com. Available at: https://www.gabrielgambetta.com/client-server-game-architecture.html [Accessed: 15 November 2023]

[4] www.youtube.com. (n.d.). Ajackster - YouTube. [online] Available at: https://www.youtube.com/@Ajackster [Accessed: 10 December 2023]

[5] jagt.github.io. (n.d.). clumsy, an utility for simulating broken network for Windows Vista / Windows 7 and above. [online] Available at: https://jagt.github.io/clumsy/ [Accessed: 25 November 2023]