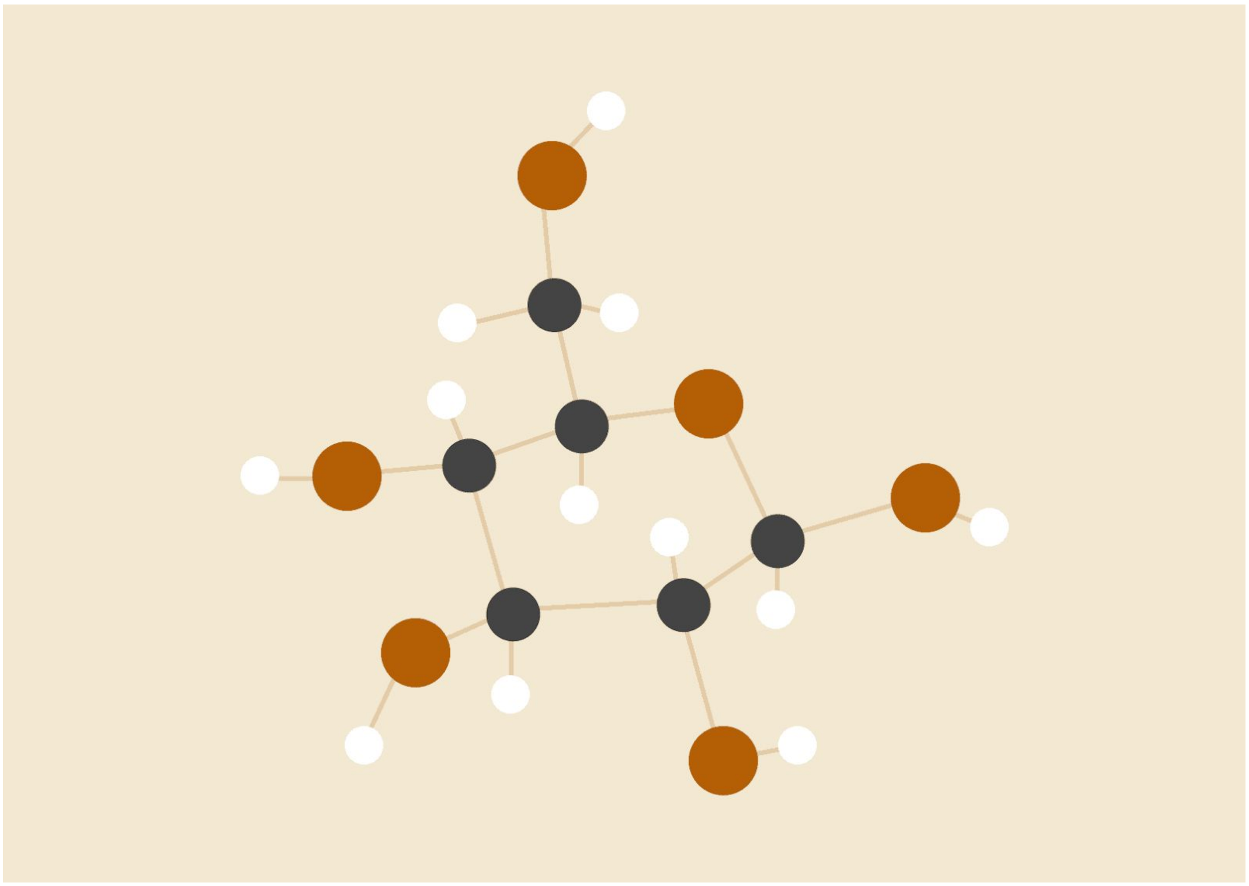
Using TCP for control detection and UDP for rendering data

Network Game Development



Rohan Menon

MSc Computer Games Technology

Architecture

The 2 usual options for setting up game multiplayer sessions are peer-to-peer and client-server.

For the game created, the main concerns were:

1) All game logic to be decided by a ***control entity***
2) All players should ***only communicate inputs*** to the entity
3) The control entity accepts input from everyone, applies game logic and ***sends out/streams updated objects*** and their positions

Clearly, a ***client-server*** architecture works best in this case.

The client will have a ***reliable*** connection to the server that will keep updating inputs(mouse position, keys pressed etc).

The server will have a ***fast*** but not necessarily reliable connection that all clients will listen to and will send out updated object positions that the clients will then predict positions from.

This ensures:

1) All participating clients will get the ***same instance of the game***
2) Clients ***cannot tamper*** with the running of the game, server decides the rules
3) ***Time syncing between clients*** will not be an issue as all clients only need to sync with 1 entity, the rendering server

Protocols

Testing was done to find the ideal protocol for communicating between the client and server. In the end,

A. the client is using TCP to communicate to the server and
B. the server is using UDP to communicate to the client.

Sending client data to the server

Since the client needs to have a *reliable* way of sending input to the server a **TCP**(Transmission Control Protocol) is used to send the data to the server. This ensures that every input done by the client will be received on the server and acknowledged and prevent

any packet loss from happening.
Although the game logic can handle loss of data input packets and it might have made more sense for the transmission to be faster rather than reliable but in most games, reliable input communication would be more important than faster communication especially since UDP packets are considerably more unreliable than TCP.

Sending server data to client

This was not the case for server to client communication. Since all game logic is being decided by the server, there is only 1 instance of the game that can be broadcast/streamed to the client. This means all clients can simply listen in on a **UDP**(User Datagram Protocol) socket and receive the positions of the objects in the current game being run on it.

This ensures less load for the server as well because it doesn't have to explicitly keep a track of all client TCP connections(even though it already does) and individually loop through them and send out what is essentially the same data to all the client TCP sockets.

The Communication

The messages being exchanged to form this linkage are:

1. **TCP:**
    a. CONNECT (Server To Client) has values: { Vector2f **spawnPos**, int **playerNumber**, float **serverTime** }
    b. RETURN_CONNECT (Client To Server) has values: { string **username**, short **udpPort**, float **returnServerTime** }
    c. CONTROL (Client To Server, struct used: **Messages.ControlMessage**) has values { Vector2f **mousePos**, bool **isForwardPressed**, bool **isAttackPressed**, bool **isBlockPressed**, float **atTime** }
    d. DISCONNECT (Client To Server) has values { string **username** }

2. **UDP:**
    a. RENDER_PLAYER (Server To Client, struct used: **Messages.PlayerMessage**) has values { int **playerNumber**, string **playerID**, Vector2f **position**, Vector2f **aimAt**, int **health**, bool **isAttacking**, bool **isBlocking**, float **time**, bool **isDead** }
    b. RENDER_BULLET (Server To Client struct used: **Messages.BulletMessage**) has values { int **playerNumber**, string **bulletID**, Vector2f **position**, float **time**, bool **isDead** }

API

I used the **SFML(Simple and Fast Multimedia Library)** because of its socket architecture and render capabilities.

The TCP Socket

A **tcpListener** on the server to **listen()** for clients trying to connect to it.

When a client's tcpSocket makes a **connect()** call, the server **accepts()** the connection, stores the ipAddress, and the client's socket reference in a custom struct called **ClientRef**, and sends back a confirmation to the client. Now, both sockets can use **send()** and **receive()** to communicate with each other.

The UDP Socket

The **udpSocket** for the server is bound to a predefined port on the server's machine using the **bind()** call. The server will keep sending all RENDER_BULLET and RENDER_PLAYER packets on a loop using **send()**.

All clients have a reference to the Server's IP address and will **receive()** from the server's IP address and the predefined port.

Integration

Starting the application

The **Server** class starts a **RenderWindow** instance and receives and handles all server socket connections on *a seperate thread*. This is so that *RenderWindow* can do changes with negligible performance issues and better **RENDER** packets are sent back to the client.

The **Client** runs a **Game** window that renders data received. It has socket connections and receives updates from all objects on the server's **RenderWindow** instance. It also immediately tries to **connect()** to the server when created.

How communication happens

A **CONNECT** packet is sent by the **server to client** when a TCP Connect from client happens and the server does **listen()** on the listener.

When server gets TCPSocket.Connect(), it spawns a player at a **spawnPos** and assigns a **playerNumber** to it based on the current playerCount. It sends this back to the client with the current **serverTime** as a **CONNECT** packet. The server also initializes and stores a reference for the clients details in a struct called **ClientRef** in a list **mClients**.

When client sends a TCP connection, it blocks the socket until it receives **CONNECT** from server. When it does, it immediately sends a RETURN_CONNECT with the **username** that the user selected, the **udpPort** that has been selected for this client(in case the client wants to send data via udp, it doesn't for now) and the **serverTime** that was sent by Server in **CONNECT**. After this, the TCP Socket is set to nonBlocking to not interfere with the client game loop.

The server calculates the **latency for client** by subtracting the returnServerTime from currentServerTime. And uses this latency when sending out UDP updates in **RENDER_BULLET** and **RENDER_PLAYER**.

Dead Objects

Both Client and Server now start exchanging information. The server sends message updates and also sends if the player or bullet is a dead object in the bool **isDead**. Dead objects are still sent from the server and client for a few packets, shown by **Player.isDeadSentCount** and **Bullet.isDeadSentCount**. This is so that if packet loss happens, the player and bullet information will still be updated on the Client for a few packets.

After a certain number of packets with the death information have been sent(**GameConstants.onKillSendTimeout**), the server stops sending the RENDER_BULLET and RENDER_PLAYER packets for these objects.

Client Loop:

This is the clients keeps sending out the **CONTROL** packets to each clientReference on the server which modifies that clientRef's player. It sends the current **mousePos** the client is aiming, is the **player attacking**, if he is **blocking** and if he's **moving forward**. It also sends the time at which this control was pressed in case this can be used to check player input history(not done right now).

Clients also move objects in the game according to the input from RENDER_PLAYER and RENDER_BULLET packets from the server. It stores the last two positions of the objects and then interpolates them based on the current time being seen by the client. It keeps a track of the last udp packet sent by the server and only accepts the packet data if it gets a newer packet than the last packet. It receives **PlayerMessage** and **BulletMessage**.

PlayerMessage.position and **BulletMessage.position** is the current server position. The client uses these values and extrapolates positions based on the last two serverTick **PlayerMessage.time** and **BulletMessage.time** received**.**

If a client is closed, it sends out it's **username** to the server in a **DISCONNECT** and the server removes the client reference and kills the player that was connected to the client.

Server Loop:

The server loops through all players and based on the CONTROL packets it receives and sets the values for the players in **RenderGame**. **RenderGame**'s loop simulates the player actions and updates positions for all players and bullets. The **Server** then sends out **Messages.PlayerMessage** and **Messages.BulletMessage** as packets for every player and bullet respectively in RenderGame, every server tick.

The server loops through all players and sends a packet update for each one. **PlayerMessage.AimAt, PlayerMessage.isAttacking, PlayerMessage.isBlocking** are all values received by the CONTROL packet. **PlayerMessage.Health** is the health based on bullets hitting the player. **PlayerMessage.Time** is the current server time and **PlayerMessage.playerNumber** and **PlayerMessage.playerID** are used to assign a colour and identify the player on both client and server. It also loops through all **bullets** and sends a packet update for each one.

Prediction

Both the players and bullets on the client positions are calculated using **Dead Reckoning**. This means that every bullet and client on the server stores the last 2 positions it received from the server and the Server times the positions were reported at. It then extrapolates where the current position of the object should be based on the Client's tick time. This is done in **Player.PredictPosition** and **Bullet.PredictPosition**. The storing of the positions happens in **Player.AddMessage** and **Bullet.AddMessage** every UDP receive on the client.

The Client also interpolates the positions of the bullet to the new positions predicted by the PredictPosition function using the function in **Game.Interpolate2f**. This is done every **Update** in the **Game** class that is created by the client.

Bouncing the bullet on a shielded player

An interesting point for prediction comes when the game needs to bounce the bullet that hits a player that is shielded. In this case the interpolation might make it look like the bullet did not hit the player and started to move away on its own. Especially if there is packet loss. The logic was changed so that instead of the bullet bouncing, the game kills the bullet that hits a player and then creates a new bullet heading in the direction of the bounce. This can be seen in the RenderGame.Update at line **110 in RenderGame.cpp**.

Testing

Testing for the application was done using clumsy a tool that allows you to introduce inefficiency in the network packets being sent. When **packet loss** was introduced, it noticed how the RENDER_PLAYER and RENDER_BULLET packets being sent from the server will still need to be sent for a few loops because sometimes the packet might not receive information to say that a particular **Bullet** or **Player** should not be rendered. As mentioned in the Dead Objects section of integration.

But due to **dead reckoning**, latency and packet loss definitely isn't as much of an issue it would have been if I was only sending position updates to the client and significantly smoothes out the motion.

Although, since the signal needs to make a round trip to the client to the server and then back to the client again(Input signal, processing and then render signal to the client), **latency does cause significant issues for running the game**. But to a certain extent this can't be helped and in my opinion is an okay trade off for ensuring that the same instance of the game will be rendered to every player from the server.

Due to the time stamping of the render packets, **out of order packets, dropped packets** and **duplicated packets** also did not affect the gameplay in any significant way although the rendering of the game did break, game rules were still followed. This will be shown in the Demonstration video as well.

Demonstration Video and Game link

Link to the game exe:

https://drive.google.com/open?id=1gxh43sDkc6nYhSMjiNHIK4NQtaVfmBuL

Link to code:

https://bitbucket.org/rohanm92/network-programming-test/src/master/

https://drive.google.com/open?id=1P8WQY4OqhjKstpHT6WTn0wSRgIluewNn

Link to video:

https://youtu.be/0DpAjEppBFA

References

Pupius, R. (2015) *SFML Game Development By Example.* Packt Publishing Ltd.

Simple and Fast Multimedia Library(SFML) (2019) *API Documentation*. Available at: https://www.sfml-dev.org/documentation/2.5.1/ (Accessed: 14/11/2019)

Clumsy (2019) *Clumsy v0.2*.
Available at: https://jagt.github.io/clumsy/ (Accessed: 28/11/2019)
Moreira, A. and Hansson, H.V. and Haller, J. (2013) *SFML Game Development.* Packt Publishing Ltd.

A new method for path prediction in network games
Theoretical and Practical Computer Applications in Entertainment, [Vol. 5, No. 4]
https://cie.acm.org/articles/a-new-method-for-path-prediction-in-network-games/
(Accessed: 10/11/2019)
By Shaolong Li, Changja Chen, Lei Li

CHALLENGES FOR NETWORK COMPUTER GAMES
https://pdfs.semanticscholar.org/eed3/be14ef8cd7488c8bd246ab8d5f4fd9169ffe.pdf
(Accessed: 18/10/2019)
By Yusuf Pisan

Dead Reckoning Using Play Patterns in a Simple 2D Multiplayer Online Game
International Journal of Computer Games Technology
https://www.hindawi.com/journals/ijcgt/2014/138596/
(Accessed: 9/11/2019)
By Wei Shi, Jean-Pierre Corriveau and Jacob Agar