Distributed Systems and Networking – INFR 3830U

Delightful Systems - Roboy in the Hood

Introduction

Our game is powered by PhyreEngine, an engine developed by Sony for use with the PlayStation line of consoles. PhyreEngine has it's own platform specific wrapper for operating system level networking libraries. For example, on Windows it has its own wrapper for WinSock, on Sony Consoles it has its own wrapper for the respective console's operating system's networking library. Each of these wrappers inherit from a common base interface so depending on the platform the game is being built for, the compiler is will link the correct library. So theoretically, using Phyre's builtin "send message" functions should just work across platforms. Sounds great, right? Unfortunately there is literally no documentation for the library, searching online helpless since the engine is proprietary to Sony and the engine's source is confidential.

We attempted to try and figure out the engine's networking component but after much effort and increased stress from a quickly approaching deadline we decided to abandon it and write our own WinSock wrapper. Since WinSock is a Windows' platform specific networking layer, the networking component of the game only works on Windows.

Networking Component Architecture

We implemented a Client - Server architecture. A client sends messages to the server who sends it to all servers. Both the client and server have two sockets, a TCP socket and a UDP socket. We two types of categories for messages: chat messages and gameplay messages.

Message Types

Chat messages use a TCP socket to send messages to the server. Chat messages have several headers which the server uses to handle the message correctly. In the message below you can see the different types of chat messages our game uses.

```
enum ChatMsg
{
    MSG_NO_HEADER = 0,
    MSG_HEARTBEAT,          //= "~1"
    MSG_DISCONNECT,         //= "~2"
    MSG_MY_NAME,            //= "~3"
    MSG_CHAT_MSG,           //= "~4"
    MSG_GAME_START          // = "~5"
};
```

MSG_NO_HEADER	Rarely used, demontes a message which has no identified.
MSG_HEARTBEAT	Since game updates are only sent when the player inputs, we need send a heartbeat every X frames to tell the server the client is still here. This allows us to disconnect a player even if the player unexpectedly leaves.
MSG_DISCONNECT	When a client exits the game, this is sent to the server. The server will remove the client from its std::vector of clients.
MSG_MY_NAME	When you first connect to a server, you are required to enter a name. This name is used to identify the client in chat messages in the lobby.
MSG_CHAT_MSG	Chat messages displayed in the lobby.
MSG_GAME_START	The message sent from the server when the server starts the game.

Gameplay messages are a little different. We use a UDP socket to send gameplay updates and we only have one type of gameplay message. Gameplay updates are only sent when a client's state changes.

```
enum GameplayMsg
{
    MSG_NO_HEADER = 0,
    MSG_GAMEPLAY,           // = "~1"

};
```

We only have one type of gameplay message because all the different "types" of messages are packed into one buffer. This was the best solution we could come up with, rather than having to deal with a "position" message and a "player hurt" message, we packed it all into one buffer and sent it. This is ideal because we are able to break down a message upon arrival and search for a particular header and update accordingly.

Client Architecture

When a client joins a game, two threads are created for the client. One thread listens for gameplay messages and the other listens for chat messages. Two threads are created because UDP sockets and TCP sockets use different functions to listen for messages. Furthermore, sockets in blocking mode block execution of the thread until a message is received. By having two separate threads listening, we are able to easily manage the handling of messages. The client sends messages in the main game loop. Each client creates a struct for themselves which describes critical information needed by WinSock, the thread and a buffer of the most recent gameplay message.

```
struct ClientThreadData //used by client
{
    DWORD   threadIDTCP;
    HANDLE  threadHandleTCP;
    DWORD   threadIDUDP;
    HANDLE  threadHandleUDP;

    std::string gameplayBuffer;
};
```

Server Architecture

When the server is created, a thread is created to handle new connections. When a new connection is detected, the new client's information is stored and a welcome message is sent. In the image below, you can see the struct created for each client when they join the session. We have a std::vector of this type and when messages are sent to all clients, we go through this vector. When a client disconnects, we remove them from this vector, this is how we handle connection and disconnection of a client.

```
struct ClientAndThreadData
{
    ClientAndThreadData() {m_name = "Player No Name";};
    DWORD   threadID;
    HANDLE  threadHandle;
    SOCKET  clientSocket;

    char ipAddr[11];
    int myID;   //id = index in hosts vector
    int timeSinceLastMessage;
    std::string m_name;
};
```

As you can see in the image above, we keep track of the player's name, thread data and socket descriptor.

<u>Gameplay Implementation</u>

As for the gameplay implementations of the networking requirement, simple linear velocity dead reckoning and easing is implemented into the PVP gameplay sessions for all other player instances. Since our game consists of hard to predict player movements, as it is a third person shooter, it is more difficult to implement movement prediction algorithms, as the player is not reliant on velocity and acceleration. The player is free to move in any direction with very minimal time between direction changes. This means that sudden direction changes, in addition to the player's ability to jump at any time, will not be possible to predict accurately. Linear velocity dead reckoning is used minimally in order to keep the players

actual position as correct as possible, since dodging the other players shots is an important aspect of the game.

The actual dead reckoning algorithm implemented in the game relies on the incoming data sent by the other clients (to the server). This data only encapsulates the player's world matrix, which includes the orientation, scale, and position. By default, this data is sent every other frame in order to reduce the load on the server when multiple players are in one session. It is up to each client to simulate the easing on the position data in order to make up for the mission frames and create a smooth and responsive gameplay environment.

Each client has a list of active players, and an ID assigned to them by the server. Every n frames, defined by the update interval, the client sends all information about that player's transformations and position to the server, in addition to their ID. The server distributes the message to all connected clients, as well as handling the data itself, and when the clients receive the message, they update the instance of the player in the list at the given ID. Each player instance also holds data about it's previous position, which is used for easing and dead reckoning prediction.

When new data is received about a player instance, the current position of that player is recorded and then overridden by the new position. Velocity is calculated between these two points, and set on the player instance. This means that the player will move in a straight line between the last two points in a prediction algorithm that works for simple movement.

$$V = (P_{current} - P_{last}) * \text{movement speed}$$

In addition to dead reckoning, easing is also implemented to help smooth the whole networked game environment. When the client receives an update about the player, if the new position is different to a threshold than the current position calculated with the velocity the player was moving at, then the following easing formula is applied to move the player instance to the new and correct updated position, to create the simulation of the other connected players moving slowly.

$$Pos = Pos - ((Pos - NewPos) / easeFactor)$$

Where ease factor is determined by the distance between the current point and where the player needs to be animated to, taking into consideration the maximum speed of the player.

To ensure that the bandwidth is as low as possible given the frequent updates, the player's position is only sent to the server if the player has moved. This is implemented by simply checking the magnitude of the players position between the players last position, and only updating if the frame timer mod the update interval is 0, and the magnitude is not 0. The game also implements a shooting mechanic, where a ray cast is sent from the player in the direction the player is aiming. On a successful ray cast, the player that is hit is found in a list, and then checked against for what player ID that player has. The client that calculated the hit then sends to the server which player ID was hit, along with the special character

representing the hit message, which then the server passes the hit message to the correct player ID.