UNIVERSITY of ABERTAY DUNDEE

School of Design & Informatics

Session 2019/20

CMP501: Network Game Development

Space Invaders with Network Streaming using SFML

Daniel Zammit Student ID 1905316

MSc Computer Games Technology

Table of Contents

Introduction

The objective of this paper is to highlight the techniques employed in network video games to address issues that arise from network latency. Latency is the delay of information over a connection from a system to another. It can be a result of congestion on the network or geographical distance between the connected parties (Mitchell, 2019).

A single player space invaders clone with streaming functionality was developed using Simple and Fast Multimedia Library (SFML) to demonstrate client prediction and interpolation techniques. SFML was the library of choice as it allowed for intuitive media management during the game development while not preventing demonstration of the application layer protocol implemented in the game. For the purposes of this project, an open-source tool called clumsy was used to simulate poor network conditions on the client side. The tool enabled testing to be carried out for the proposed techniques when the game was being affected by latency.

Architecture

The network architecture that was chosen for the project is client-server. The generated executable allows the user to run as the server or client, which allowed for flexibility during testing. Furthermore, having a dedicated server ensured that connected clients would receive information from a well-known dedicated machine on the network, and all calculations related with the gameplay were monitored by the server (Posey, 2000) . In order to achieve a streaming application, the server was able to establish multiple connections with clients, that received information on the current state of the game.

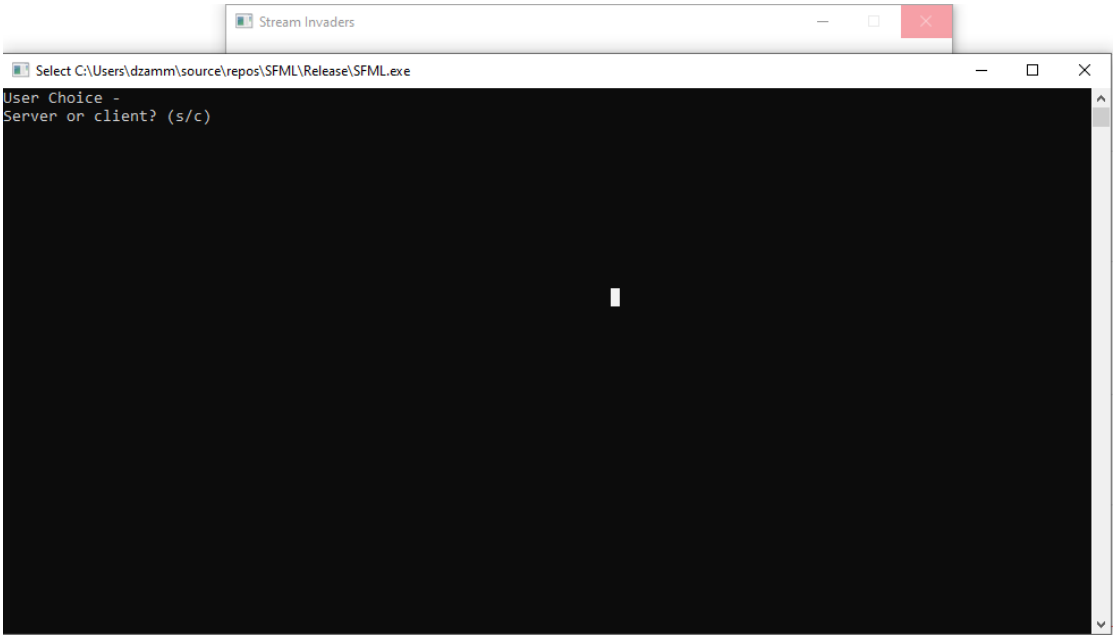

Figure 1 - Space Invaders Menu

Protocols

For a streaming application, it is vital to continuously receive a stream of data sent from the server, with minimal delays. Action shooting games demand that the latest information from the server is received and interpreted with priority (Chen-Chi *et al.,* 2009). To achieve this, UDP was used to send packets with a tick rate of 30. Therefore, the transmission of server updates was at most 30 times per second.

With less overhead in UDP, the game state with client-server implementation was effortlessly preserved. However, in order to overcome some disadvantages of this protocol when latency was introduced, an application-layer protocol was also designed to handle game state changes and order of events. The SFML library included the *sf::Packet* class that provided means for data serialisation over the network from both the client and the server. The following table contains the ordered data that was sent in each packet originating from the server.

Variable Name	Description
timeStamp	An integer type containing the server time
playerXPosition	A float type containing the x vector component of the player
playerYPosition	A float type containing the y vector component of the player
bulletStatus	A boolean type determining whether the bullet is alive
enemyShotID	An integer type indicating the last enemy shot
enemyXPosition	A float type containing the x vector component of the enemy
enemyYPosition	A float type containing the y vector component of the enemy
enemyStatus	A string that contains identifiers for enemies which have already been destroyed
mPlayerIsAlive	A boolean type determining whether the player is alive

Table 1 - Application Layer Protocol

The *enemyStatus* variable is used to update information on the client side in the case where the client connects after the player started to interact with the game and had already destroyed some spaceships. The rest of the received variables updated the game state on the client side, the view of objects was emulated based on the player's instance which was the acting server.

API

Simple Format Media Library (SFML) was selected for the development of the project as it enabled low level development of the networking techniques that govern the scope of this paper. Furthermore, the library presented few difficulties when developing the core game, as certain logic and syntax comparable to other libraries and game engines. SFML also provides seamless interaction and support between Windows and C++ development cycle. The documentation provided and the community helped in the process of understanding how to use the library and integrate the networking component to the game (Haller, 2013).

Integration

The network code was designed with asynchronous I/O in mind. In order to prevent the starting connections from blocking, SFML has a boolean function to set the socket to non-blocking. However, in the demo application, the *SocketSelector* class was utilised to determine when the socket was ready to receive data. The same class allowed to read from multiple sockets as well. If there was data to be read, this would be read and used to update the client game state, otherwise a small timeout of 10ms had to pass before checking the state of the socket again. The small timeout enabled asynchronous I/O for the space invaders clone.

```
sf::UdpSocket socket;


sf::IpAddress recipient = sf::IpAddress::getLocalAddress();
char data[100] = "Connection with client established!";

if (socket.send(data, 100, server, port) != sf::Socket::Done && !sendInitialToServer) // if send once dont't send again
{
    // added up top send this only once
    // error...
}
else{
    sendInitialToServer = true;
}


sf::SocketSelector selector;
selector.add(socket);

if (selector.wait(sf::milliseconds(10.f))) { // not enough time for server to be created with 0.1f // previously was 100

    // received something
    if (selector.isReady(socket)) {

        // Wait for a message
        sf::IpAddress sender;
        sf::Packet playerData;
        float playerXPosition;
        float playerYPosition;
        float clientXPosition;
        float clientYPosition;
        float clientEnemyXPosition;
        float clientEnemyYPosition;
        float yEnemyOffset = 50.f;
        float xEnemyOffset = 60.f;
        bool bulletShot;
        bool playerAlive;
        int enemyShotIDReceived;
        std::string enemyStatusReceived;
        int packetTime;
        // predicted position coordinates
        float x_;
        float y_;
        float x_one;
        float y_one;
        float x_final; // interpolated value
        float y_final; // interpolated value


        socket.receive(playerData, sender, port);
```

Figure 2- Asynchronous I/O on UDP Socket using SFML

The application executable had send and receive functions in order to switch the socket state respectively. The network logic was placed inside the main window loop. The same port was used to start sending packets using UDP at both ends. The game uses UDP to send constant player updates and relies on the same transport protocol to start the connection. A limitation on the current implementation was the reliability of UDP packets that were sent to the server from the client. Every packet that was sent from the server side required acknowledgement from a blocking send on the client.

```
// Network Logic
if (userChoice == 's') {

    sf::Time deltaTick = tickClock.restart();
    timeSinceLastTick += deltaTick;
    while (timeSinceLastTick > tickRate) { // i.e tick > 30hz

        timeSinceLastTick = sf::Time::Zero;
        createUdpServer(SERVER_PORT);

    }

}
else {
    runUdpClient(SERVER_PORT);
}
```

Figure 3 - Server sends 30 updates every second

Prediction

In order to emulate a sensible game state to the connected client, prediction was implemented based on a linear model. The model required the last two known positions of the player's spaceship in order to calculate the velocity between the two positions received. This was achieved by storing the contents of the packets received for the player data sent from the server and pushing the information as a struct into a vector holding player messages.

```
// Handles prediction
PlayerMessage msg;
msg.time = packetTime;  msg.x = clientXPosition; msg.y = clientYPosition;
messageHolder.push_back(msg);
int messageHolderSize = messageHolder.size();
std::cout << "Message Holder Size - " << messageHolderSize << std::endl;
if (messageHolder.size() >= 3) {
    const PlayerMessage& msg0 = messageHolder[messageHolderSize - 1];
    const PlayerMessage& msg1 = messageHolder[messageHolderSize - 2];
    const PlayerMessage& msg2 = messageHolder[messageHolderSize - 3];

    /* Linear Model */
    int latency = gameTimer - msg1.time;
    float velocity_x = (msg0.x - msg1.x) / (msg0.time - msg1.time);
    float velocity_y = (msg0.y - msg1.y) / (msg0.time - msg1.time);
    x_ = msg0.x + velocity_x * latency;
    y_ = msg0.y + velocity_y * latency;
    x_one = msg1.x + velocity_x * latency;
    y_one = msg1.y + velocity_y * latency;
    // Interpolation
    x_final = x_ * (0.5f) + x_one * (0.5f);
    y_final = y_ * (0.5f) + y_one * (0.5f);
    std::cout << "gameTimer " << gameTimer << " - msg1.time " << msg1.time << " = " << latency << "ms" << std::endl;
    player->setPosition(x_final, y_final); // the interpolated value between two points
}
else {
    //player->setPosition(clientXPosition, clientYPosition);
}
// end of prediction handling
```

Figure 4 - Player prediction using linear model

Furthermore, linear interpolation was also implemented to provide smooth movement to the player spaceship between the two predicted positions (Mudford, 2018). The variables *x_final* and *y_final* represent the interpolated point from the diagram below.

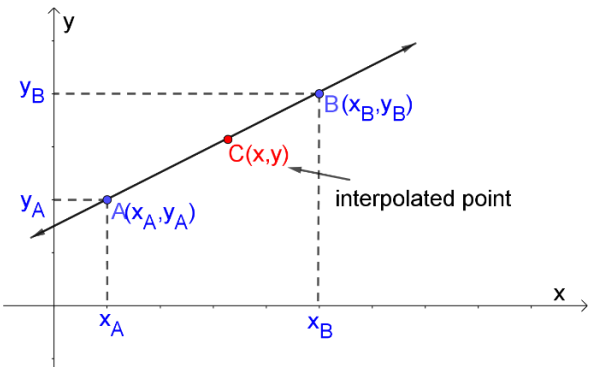

Figure 5 - Linear Interpolation

Testing

Space Invaders was tested on the localhost environment using clumsy as well as on a local area network (LAN) between two laptops, with the latter shown in the demonstration video submitted. Two tests were carried out; lag and 10% packet drop on inbound packets from the server to the client. The latency was calculated using the game time difference between the two most recent packets received on the client. The following results were obtained from running clumsy on the client-side when testing over LAN.

Packet Drop

Value (%)	Highest Latency (ms)
10	116
20	133
30	166

Outbound packets were filtered instead of inbound packets when testing was carried out on localhost. This was performed following the explanation of the inability to capture loopback packets as a technical limitation, explained in clumsy's manual (Tao, 2016). During testing, the application from the client-side showed a significant increase in delay and higher values for packet drop in the table rendered an unviewable game state.

Conclusion

To conclude, the developed game prototype is a functioning single player game with the ability to stream to online viewers over local area network. The techniques implemented for client prediction and interpolation have shown that for short distance connections the client was able to observe the game with minimal impact, even when network was congested. It is suggested that future work on this project includes a dedicated network thread which handles networking logic and client connections should be used for separation of concerns. TCP can be implemented to establish a reliable connection between client-server on top of the current UDP implementation for player data transmission. In addition, a connection class should be created to manage multiple client connections in order to provide feedback data on the game to each client respectively. With these two enhancements, it would be possible to develop a dedicated stream server that manages connections between the main player and their viewers. (The demonstration video can be found at https://www.youtube.com/watch?v=rBdwFHtc36A&t=)

References

Chen-Chi, W., Kuan-Ta, C., Chih-Ming, C., Huang, P. and Chin-Laung, L. (2009) 'On the Challenge and Design of Transport
Protocols for MMORPGs', *Multimedia Tools and Applications (special issue on Massively Multiuser Online Gaming Systems and Applications,* 45(1).

Haller, J. (2013) *SFML game development learn how to use SFML 2.0 to develop your own feature-packed game.* Birmingham: Packt Pub.

Mitchell, B. (2019) *Causes of Lag on Computer Networks, and What to Do About It.* Available at: https://www.lifewire.com/lag-on-computer-networks-and-online-817370 (Accessed: Dec 7, 2019).

Mudford, T. (2018) *Linear Interpolation.* Available at: https://www.trysmudford.com/blog/linear-interpolation/ (Accessed: Dec 8, 2019).

Posey, B. (2000) *Understanding the differences between client/server and peer-to-peer networks.* Available at: https://www.techrepublic.com/article/understanding-the-differences-between-client-server-and-peer-to-peer-networks/ (Accessed: Dec 7, 2019).

Tao, C. (2016) *Clumsy, a utility for simulating broken network;* Available at: http://jagt.github.io/clumsy/manual.html (Accessed: 08/12/2019).