PacketPew: A 2D Networked Shooter

CMP501: Network Game Development

Kushagra
Student ID: 2400020
MSc Computer Games Technology
18th December 2024

Introduction

PacketPew is a 2D networked shooter game in which 2 clients/players can connect to a centralised server and have a match to death.

Both players start with 100 health, and their goal is to kill the other player (`enemyHealth <= 0`) using bullets. The players can move around the environment, and shoot using the left mouse button.

However, the server is the authoritative identity and it controls the player's movement and bullets. This way, any unauthorised actions (cheats) can be controlled.

Architecture

The network architecture for this project is the “Client-Server” architecture where the clients connect to a centralised server using its IP address and port (Hall, 2023). The server is the governing body and handles key components of the game. The clients send what actions they want to do and wait for the server’s response on those inputs (Valve Corporation, 2024). Moreover, the server also dictates the scenarios which affect the win/lose states (bullet and collision logic).

Reasons for choosing Client-Server over others:

1. **Protection:** The server is the authoritative identity and it decides on key components of the game, this makes the game less vulnerable to client-sided cheats since the server will simply not verify those inputs. These attacks can include:
   a. Denial of Service: Since the clients do not know the IP address of other clients (unlike in peer-to-peer architecture), they cannot initiate DDoS attacks on them.
   b. Cheating: Since the server is authoritative, cheats that affect server-controlled variables are useless as the server can simply deny those changes.

2. **Performance:** By shifting network processing on the server, the client can solely focus on rendering and the physics of the game, this makes the game more enjoyable since it does not suffer from FPS drops caused by networking.

3. **Stability:** A client-server architecture is more stable than peer-to-peer since it depends on the server’s stability and network conditions. In peer-to-peer architecture, the network is dependent on the host player’s stability. The host can have poor network conditions which leads to a bad experience for all the players. Moreover, if the host player disconnects the whole network collapses. Hence it's easier to maintain stability by implementing a central server, only one machine’s network and stability are in question.

4. **Scalability:** Client-server architecture is more manageable and scalable than any other architecture since only the server-side code needs to be changed. In this project, the server can handle up to 2 clients, which can be changed by changing a `MAX_CLIENT` constant in the code. If it was peer-to-peer architecture the changes would be much more since every client will need to accommodate for

the new player. Also, as the number of players increases, the peer-to-peer architecture gets more difficult to manage whereas servers are easier to manage.

5. **Latency:** The latency can be constant in client-server since it solely depends on the client's proximity to the server. The constant latency makes it easier for the player to play and the developers to handle. In peer-to-peer, latency depends on how far away you are from the host of the network and what conditions the network is in.

Protocols

In networking, a protocol is a set of rules for formatting and processing data. The protocol used to transfer information between the client and the server (transport-layer protocol) is TCP (transmission control protocol).

In TCP, the data delivery is reliable and in order, so the client does not need to worry about dropped packets or out-of-order packets. It also provides flow control, and congestion control, and supports out-of-band data (Parziale et al., 2006).

The TCP is slower than unreliable UDP (user datagram protocol) (Glyph, 2014), however, the network load on this project is not nearly enough to make a difference. Hence, TCP is being used for all network transactions.

The format of the network packets (application layer protocol) is also crucial since it defines how the client or server interacts, it is as follows:

1. **Client-side packets:**
    a. The client-side packets start with "`1`" and on the server side, that denotes that the packet is successfully received. There are checks to ensure that.
    b. The packet then contains an identifier string, which denotes the type of the packet, e.g. `PLAYER_FIRE` means that the player has fired a bullet, now the server will recognise this type and do what's needed.
    c. Lastly, the information needed to complete the action on the server side. This can be anything from player position, and rotation, to delta time.

    Example from `Client.cpp` (this is a packet sent when the player presses WASD):

```
sf::Packet playerAct;
playerAct << 1 << "PLAYER_ACTION_MOVE" << curSequenceNo << playerMovDir <<
player.GetMoveRate() << player.GetPlayerSprite().getPosition()    <<
sf::Vector2f(player.GetMinX(), player.GetMinY()) << sf::Vector2f(player.GetMaxX(),
player.GetMaxY()) << player.GetPlayerSprite().getRotation();
```

2. **Server-side packets:**
    a. They start with the identifier string, which denotes the type of the packet on the client side and aids the client in determining the action. E.g. `ENEMY_FIRE` telling the player to fire an enemy bullet.

b. Lastly, the information needed to perform the action by the client.

Example from `Server.cpp` (this packet is sent 20 times a second, it is the position packet for the player):

```
sf::Packet playerPosPacket;
sf::TcpSocket& client = *clientInputsList.back().socket;
playerPosPacket << "PLAYER_POS" << clientInputsList.back().sequenceNo <<
clientInputsList.back().newPos;
```

APIs

API or application programming interface in this project can be defined as a set of rules and protocols that allow to program and configure network properties.

The API chosen for this project is SFML (Simple Fast Multimedia Library) (Gomila, 2007), SFML provides a simple interface to the various components of the PC, to ease the development of games and multimedia applications. It is composed of five modules: system, window, graphics, audio and network (Gomila, 2007). It also has a collection of nicely laid out and easy-to-understand tutorials, which were of great help during development (Gomila, 2023).

it suited the project well because of its ease of use, tutorials, and vast helping community.

Integration

Integrating the network logic and gameplay logic is a crucial step and it determines the game performance as well as network processing latency. This project incorporates multithreading hence making networking quicker and decreasing the load on the main game render thread.

1. **Networking Thread:**
   a. The networking thread is responsible for handling networking logic, including sending packets, receiving packets, and checking server live status, it's code can be seen in `NetworkingThread.h` and `NetworkingThread.cpp`.
   b. It is also responsible for creating a plethora of variables that are shared with the main thread, a few of which are:

```
atomic<bool> isRunning(true);
atomic<bool> appClose(false);
mutex networkMutex;
sf::TcpSocket socket;
```

   c. The locking of these variables is also a concern since both the threads can access them at the same time and that can lead to improper read/write, and deadlock problem. To ensure that doesn't happen, whenever a variable is accessed a lock is created using the mutex. This can be done as follows:

```
{
```

```
            lock_guard<mutex> lock(networkMutex);
        }
```

Now the variables can safely be accessed in the `lock_guard` scope.

2. **Main Thread:**

    a. The main/game thread is responsible for all the other tasks including connecting to the server, game loop, play loop, and processing the packets based on type. Its code can be found in `Client.cpp` with some dependencies on other files such as `Player.h`.

    b. The processing of packets is done here instead of the networking thread because it makes the networking thread light and fast which in turn means quicker sending and receiving of the packets.

    c. The thread adds the packets that need to be sent in a shared list using the `lock_guard` mechanism. It accesses the received packet list for processing using the same logic.

    d. it is also responsible for controlling the execution of the network thread, using a boolean named `isRunning.` Once the network thread is no longer needed (in cases of win, lose, or quit), it's paused by setting the bool to false and calling `networkThread.join().`

Prediction/Interpolation

The prediction mechanism plays a critical role in lag compensation for both the player and the enemy positions.

For the enemy position, the prediction can be implemented using the positions of the current and previous packet, which can calculate the velocity which in turn can be applied to give the new position. Alternatively, the server can send the direction the enemy was moving in, and if the speed is known, the new position can easily be calculated (Gambetta, n.d.).

This project implements the latter approach. The server sends the enemy position, and move direction every 50ms to the player, and the player then predicts the position for the next 50ms using its speed value, since both the enemy and player are moving at the same speed. To stop the prediction, the move direction is reset once the predicted position is equal to the packet position. To correct the overshoot smoothly, created by the last packet, the predicted position is linearly interpolated (lerped) between the exact position and predicted position based on the delta time (time elapsed since the last frame). The code for this implementation is as follows:

```
//retrieving information from the packet
if (type == "ENEMY_POS_ROT") {
    curPacket >> enemyMoveDir >> enemyPos >> enemyRot >> serverTime;
    if (lastEnemyPos != enemyPos) {
        lastEnemyPos = enemyPos;
    }
    else {
        enemyMoveDir = sf::Vector2f(); //if the position is same, clear the move direction,
don’t need to move more
    }
}
//drawing the enemy
```

```
predictedPos += enemy.MovePredicted(enemyMoveDir); //predicting the new position based on the
move direction
predictedPos += (lastEnemyPos - predictedPos) * (deltaTime * 4); //lerping based on delta
time, multiplied by 4 to increase the lerp speed
enemy.draw(window, deltaTime, predictedPos, lastEnemyRot); //drawing with the new predicted
and lerped position
```

The player position prediction is simpler since it only requires the input, the sequence number, and a list of the inputs.

When the user presses WASD, the system creates a direction vector based on the input, adds it to an input list with a sequence number, and adds it to the send packet list with the proper type string, for server reconciliation. Then it applies the direction and speed to the player sprite, assuming the inputs are valid. When the server responds, the player position is set to the position received, and the remaining actions, based on the sequence number, from the input list, are applied again. This mechanism ensures that the player motion is in accordance to the server and smooth and instant on the player side. The code for this is as follows:

```
//when the user presses WASD
sf::Vector2f playerMovDir = player.CheckMove(); //retrieving the direction from the inputs
if (playerMovDir.x != 0 || playerMovDir.y != 0) {
    sf::Packet playerAct;
    playerAct << 1 << "PLAYER_ACTION_MOVE" << curSequenceNo << playerMovDir…..
    //position packet code, irrelevant to prediction

    actions.push_back({ curSequenceNo++, playerMovDir }); //storing the action in a list
    player.MovePredicted(playerMovDir); // Optimistic movement
}
//server reconciliation
if (type == "PLAYER_POS") {
    int serverSequenceNo;
    curPacket >> serverSequenceNo;
    sf::Vector2f newPos;
    curPacket >> newPos;
    if (player.GetPlayerSprite().getPosition() != newPos) {
        player.GetPlayerSprite().setPosition(newPos); //applying the position received from
the server
        player.GetGunSprite().setPosition(newPos);
        for (const auto& action : actions) {
            if (action.sequenceNo > serverSequenceNo) { //checking for the actions
                player.MovePredicted(action.moveDir); //applying the remaining actions
           }
        }
        actions.erase( //deleting the action for which the position has been received
            std::remove_if(actions.begin(), actions.end(),
                [serverSequenceNo](const PlayerActions& action) {
                    return action.sequenceNo == serverSequenceNo;
                }),
            actions.end());
    }
}
```

Testing

Testing the application on various network conditions is also a crucial step as it shows the conditions where the program breaks and the game is unplayable, which in turn helps in improving the network logic and implementation.

The application was tested using Clumsy (Tao, 2023), an application which stops living network packets and captures them, lags/drops/tampers/.. the packets on demand, and then sends them away. It is widely used in evaluating applications for poor network conditions.

The following cases were tested:

1. **RTT Lag (50ms, 100ms, 150ms, 200ms)**
2. **Packet Drop (5%, 10%, 15%, 20%)**
3. **Bandwidth**

The results are as follows:

1. **RTT Lag:**
    a. **50ms:**
        i) Motion: The player and enemy motion are smooth and the lag in motion is hardly noticeable, the prediction is working correctly and providing a pleasant experience.
        ii) Shooting: Shooting is working as expected and the collisions are being recognised by the clients properly, the lag is unnoticeable.
    b. **100ms:**
        i) Motion: The player and enemy motion are a bit less smooth and the lag is slightly noticeable, however, the prediction system is preventing it from being unplayable.
        ii) Shooting: Shooting is working as expected but with a slight delay, collisions are also being recognised by the clients. The game is still enjoyable.
    c. **150ms:**
        i) Motion: The player and enemy motion is noticeably lagged now, but the prediction is helping to keep the game smooth.
        ii) Shooting: The shooting system is still working as expected, however, the collisions are slightly delayed since the collision packet is taking 75ms to reach the player. The game is still playable.
    d. **200ms:**
        i) Motion: The player's motion is smooth because of the optimistic movement, the enemy's motion is laggy however prediction helps in smoothing it out. The case of “I shot you on my screen” seems to be within reach.
        ii) Shooting: The shooting is delayed by 100ms which is unpleasant and can conflict with position, and the game is borderline playable because of it.

    Increasing RTT lag beyond 200ms (to 300ms) made the game unplayable with severe position and collision delays leading to conflicted shooting.

2. **Packet Loss:**
    a. **5%:**
        i) Motion: The player and enemy motion is smooth since only 5% of packets aren't received, there are a few hiccups in enemy motion but those are only noticeable if searched for. The game is playable in terms of motion.
        ii) Shooting: Shooting is essentially unaffected, the enemy and player are receiving and simulating their bullets correctly.
    b. **10%:**
        i) Motion: The player's motion is smooth on the client's end because of the optimistic movement but on the server side it can be laggy and jumpy. The enemy's motion has more and bigger hiccups but the prediction is helping in smoothing it out, also, the prediction overshoot is more noticeable now, since the packet is lost sometimes.
        ii) Shooting: The shooting logic is working fine, however, there are a few dropped bullets but that does not affect severely. Also, the rotation is sometimes off which makes the bullet take a different path.
    c. **15%:**
        i) Motion: The enemy's motion is more inaccurate, and has a lot of hiccups. The prediction is preventing it from being catastrophic. The prediction can now be seen working, as there are moments when the enemy keeps on moving in a direction. However, the game is still playable.
        ii) Shooting: The shooting logic is working as expected with more dropped and mis-timed bullets. The collision is being detected with rare misses. The game's shooting is playable.
    d. **20%:**
        i) Motion: The enemy is a lot more teleporting but the lerping and prediction are making it visually better by sliding it.
        ii) Shooting: The bullets are more frequently dropped and are much more inaccurate in terms of time of fire. The collision system is working but with an increased rate of misses. The game is borderline playable.

    Increasing packet loss beyond 20% (to 35%) made the game severely unplayable with cases of clients being disconnected, because of server live check failure.

3. **Bandwidth:** The application performs perfectly even on low bandwidth, >80kbps, but going below that leads to failure in position packets and server live check, which in turn closes the network thread and crashes the app.

    Ways to Improve: Lag can be improved by incorporating a global game clock and more rigorous lag compensation techniques. Implementing UDP can help as well since it's faster than TCP. Packet loss results can be improved by incorporating a method of callback for crucial packets. Bandwidth issues can be improved by critically analysing the current packet's size and choosing what to send.

Future Improvements

The game can be further improved to utilise the chosen architecture and protocols with full potential, with changes in both networking and gameplay. These changes can be:

1. Increase in client size: The server is currently limited to handling 2 clients, which can easily be changed, making the game utilise the full potential of the client-server architecture.
2. Gameplay sophistication: The gameplay can be further developed with abilities and different weapons, this will make the game more enjoyable and will introduce more challenges in terms of networking and gameplay smoothness (prediction) which will be interesting to solve.
3. Architecture: The architecture can be further developed, with UDP being used for positions, while TCP being used to maintain a connection and sending critical information, such as bullet collision.

References

Gambetta, G. (n.d.). *Fast-Paced Multiplayer* [Online]. Available at: https://www.gabrielgambetta.com/entity-interpolation.html (Accessed: 2 December 2024).

Glyph. (2014). *Your game doesn't need UDP yet* [Online]. Available at: https://thoughtstreams.io/glyph/your-game-doesnt-need-udp-yet/ (Accessed: 27 November 2024).

Gomila, L. (2007). *Simple and Fast Multimedia Library (SFML)* [Framework]. Available at: https://github.com/SFML/SFML (Accessed: 23 October 2024).

Hall, B. (2023). *Beej's Guide to Network Concepts* [Online]. Available at: https://beej.us/guide/bgnet0/html/#clientserver-architecture (Accessed: 22 November 2024).

Parziale, L., Britt, D.T., Davis, C., Forrester, J., Liu, W., Matthews, C. & Rosselot, N. (2006). *TCP/IP Tutorial and Technical Overview* [Online]. Available at: https://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf (Accessed: 15 November 2024).

Tao, C. (2023). *Clumsy* [Framework]. Available at: https://github.com/jagt/clumsy (Accessed: 15 December 2024).

Valve Corporation. (2024). *Source Multiplayer Networking* [Online]. Available at: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (Accessed: 23 November 2024).