CMP501 Network Game Development

Multiplayer Maze Runner

Charlie Joshi 2104598

Introduction

The project is a real time server client multiplayer Maze Runner.The game comprises a player (the runner) and an opponent (the chaser). Both of the players are present within a randomly generated maze and are spawned at two different ends of the same layout. The objective of the game is for the runners to traverse through the maze as quickly as possible to the exit door, and for the chasers to attempt to stop or slow down the runners. Game ends if the runners reach the exit and in that case they win; else if the opponents intercept the runner before they make it to the exit then the chasers win.

Architecture

The Architecture for this game is selected as client-server, as there could be multiple players (i.e clients) playing the game, and a centralized controller (i.e server) can better control or determine the game state, as well as communicate various events that occur when players join, during the game play, and when the game ends.

Although peer-to-peer architecture has a slight advantage, where actions of players can be broadcast to all players directly, it requires clients to make calculations of positions for all players (duplicate calculations) that may go out of sync which makes debugging the application very difficult in the event that something goes wrong. Hence a client-server approach was adopted.

Protocols

This game uses three transport-layer protocols, broadcast (UDP), multicast (UDP), and TCP. It was decided that no Application-layer protocols would be used, as they may introduce data and time overheads. The outline of the above mentioned protocols is as follows :

1. Broadcast

- Broadcast is used by client(s) **only once** at startup, to know the location of the server
- When client(s) start, not knowing where the server is, send a broadcast message requesting to join the Maze Game.
- **The server responds to the request, giving details of TCP and multicast addresses and ports of the server.**
- With this information, the client joins the multicast group, and starts listening for messages on the given IP address and port. It also connects to the server on the TCP on the given IP address and port for sending moves
- This enables **"automated server discovery"** for the client, and no hard-coding of addresses/ports is needed

2. TCP/IP (Unicast)

- TCP protocol is used by client(s) mainly to send initial and changed locations to the server.
- The TCP protocol is used here, as once connected, the socket is always connected to send data with no overhead or delay in sending.

3. Multicast

- Multicast is used by the server to send information **simultaneously to all clients** to communicate game state, client states, and client moves. If the messages were sent to clients sequentially, clients will receive the messages with a slight difference in time.
- Multicast is used in this scenario since **only the clients who have joined the multicast group** receive the message.
- Multicast is similar to broadcast in a sense, but as mentioned above, it delivers messages only to the intended recipients, and does not "pollute" the entire LAN with said messages.

Network API

Since this project was intended to run on systems with a windows operating system, WinSock APIs that are built-in inside the Windows Operating system have been used considering that they have proven stability and optimal performance.

Rendering for graphics was done using SFML and while SFML provides support for networking protocols it did not support multicasting and all aspects of networking needed for this project and hence as mentioned in the prior paragraph winsock was chosen as the preferred networking medium.

Integration

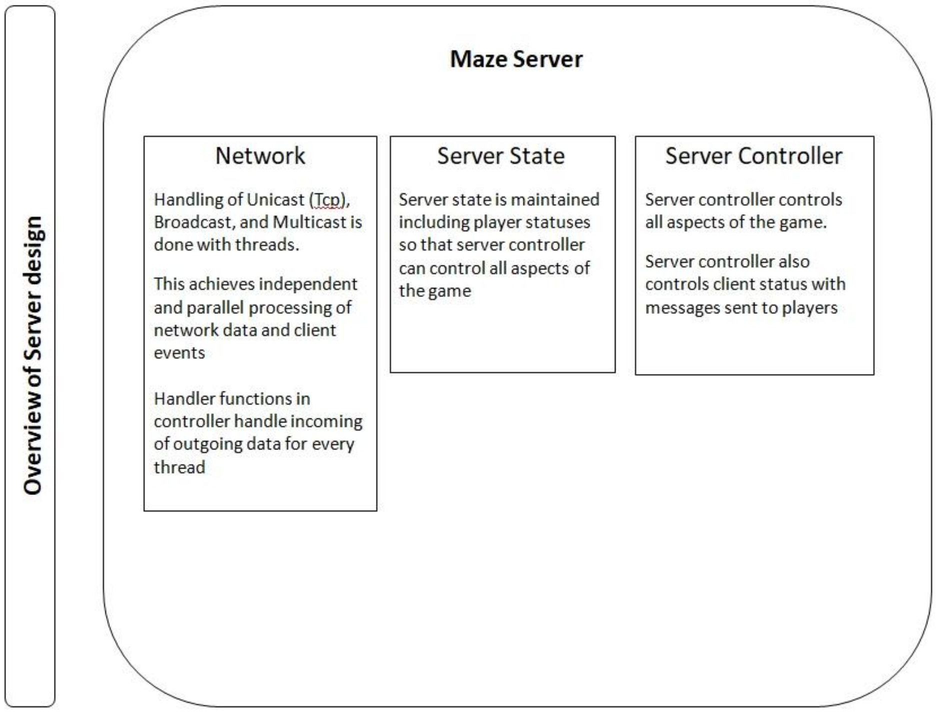
The network code for each of TCP, Broadcast, and Multicast, is modular in both server and client, and is logically separate from the rest of the game code (controller, state etc.), which is also similarly modular. This code for each of the TCP, Broadcast, and Multicast runs in separate threads, to achieve asynchronous operation.

In addition to running code in threads, the code also uses **"select"** APIs for receiving data to know whether there data available, to achieve non-blocking operation, that continues to run the thread loop. This allows thread to allow processing of any other events (like quit), to be processed without any delay.

Server Client Network overview

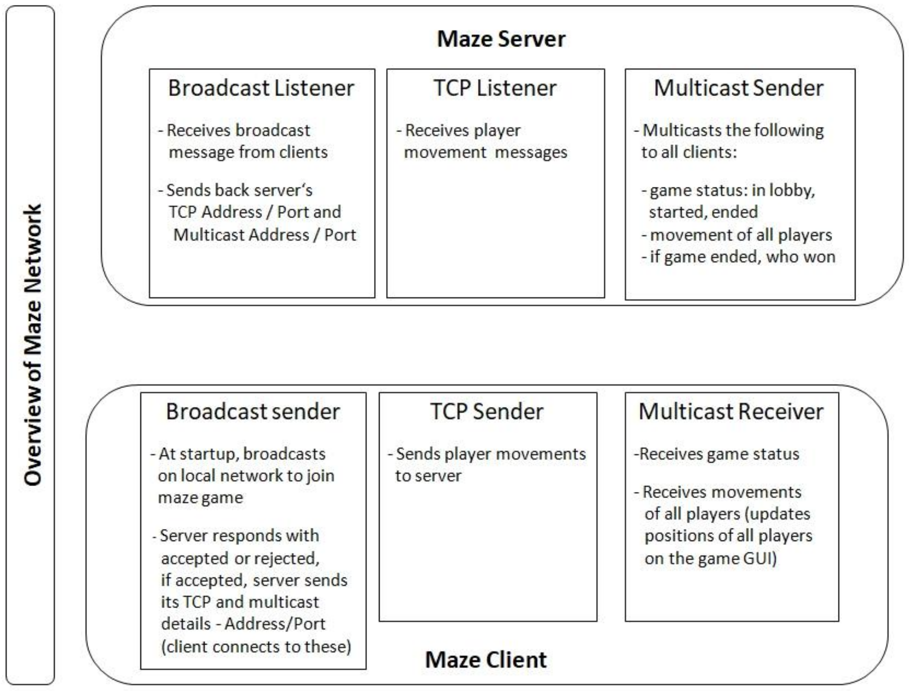
To better understand the layout of the server and client with the networking protocols that have been introduced above a visual layout has been prepared as follows:

1. Server overview - This is a self explanatory design of how the server has been set up with all of its underlying components
2. Network overview - This outlines how the server and client would interact over broadcast, tcp and multicast as sender and receivers of messages.



Overview of Server design
Maze Server
Network
Handling of Unicast (Tcp), Broadcast, and Multicast is done with threads.
This achieves independent and parallel processing of network data and client events
Handler functions in controller handle incoming of outgoing data for every thread
Server State
Server state is maintained including player statuses so that server controller can control all aspects of the game
Server Controller
Server controller controls all aspects of the game.
Server controller also controls client status with messages sent to players

Fig 1. Server Overview Design



Overview of Maze Network
Maze Server
Broadcast Listener
- Receives broadcast message from clients
- Sends back server's TCP Address / Port and Multicast Address / Port
TCP Listener
- Receives player movement messages
Multicast Sender
- Multicasts the following to all clients:
- game status: in lobby, started, ended
- movement of all players
- if game ended, who won
Broadcast sender
- At startup, broadcasts on local network to join maze game
- Server responds with accepted or rejected, if accepted, server sends its TCP and multicast details - Address/Port (client connects to these)
TCP Sender
- Sends player movements to server
Multicast Receiver
-Receives game status
- Receives movements of all players (updates positions of all players on the game GUI)
Maze Client

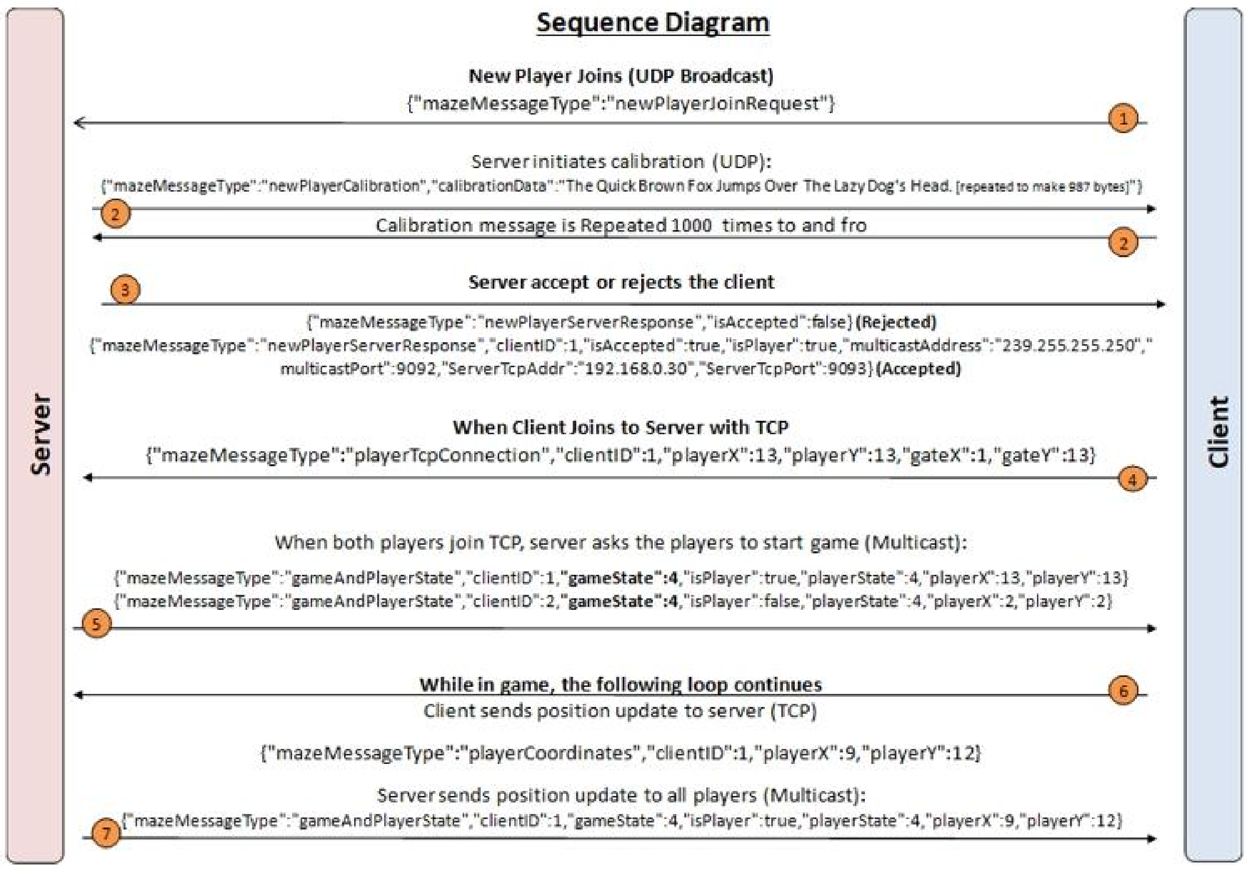
Fig 2. Network Overview Design

Working of the game with the network protocols

When the server starts:

- It creates 3 sockets, one each for broadcast (UDP), multicast (UDP), and TCP.
- § It starts listening on the commonly known broadcast IP address 255.255.255.255, on port 9091
- It prepares to send multicast data on IP address 239.255.255.250, on port 9092
- It gets the host name and IP address of the machine it is running on, and starts listening on this IP address and port 9093

So when a client listens on a multicast it can connect with the server on the tcp ip port and address that it receives from the server.



Sequence Diagram
Server
Client
New Player Joins (UDP Broadcast)
1
Server initiates calibration (UDP):
2
Calibration message is Repeated 1000 times to and fro
2
3
Server accept or rejects the client
When Client Joins to Server with TCP
4
When both players join TCP, server asks the players to start game (Multicast):
5
While in game, the following loop continues
6
Client sends position update to server (TCP)
Server sends position update to all players (Multicast):
7

```
{"mazeMessageType":"newPlayerJoinRequest"}
{"mazeMessageType":"newPlayerCalibration","calibrationData":"The Quick Brown Fox Jumps Over The Lazy Dog's Head. [repeated to make 987 bytes]"}
{"mazeMessageType":"newPlayerServerResponse","isAccepted":false} (Rejected)
{"mazeMessageType":"newPlayerServerResponse","clientID":1,"isAccepted":true,"isPlayer":true,"multicastAddress":"239.255.255.250","
multicastPort":9092,"ServerTcpAddr":"192.168.0.30","ServerTcpPort":9093} (Accepted)
{"mazeMessageType":"playerTcpConnection","clientID":1,"playerX":13,"playerY":13,"gateX":1,"gateY":13}
{"mazeMessageType":"gameAndPlayerState","clientID":1,"gameState":4,"isPlayer":true,"playerState":4,"playerX":13,"playerY":13}
{"mazeMessageType":"gameAndPlayerState","clientID":2,"gameState":4,"isPlayer":false,"playerState":4,"playerX":2,"playerY":2}
{"mazeMessageType":"playerCoordinates","clientID":1,"playerX":9,"playerY":12}
{"mazeMessageType":"gameAndPlayerState","clientID":1,"gameState":4,"isPlayer":true,"playerState":4,"playerX":9,"playerY":12}
```

Fig 3. Sequence diagram of messages between server and client

Prediction

Since we didn’t have linear motion, a different approach of optimisation and prediction was used.

Calibration

1. When the server receives a "join" request from a client, before accepting the client to play the game, it starts a "calibration" cycle to measure the speed of data exchange between client and the server.
2. At the beginning, the current time (in milliseconds) is taken.
3. The server then sends a fixed "calibration" message of approximately 1024 bytes.
4. The client, upon receiving the calibration message, sends it back immediately.
5. This cycle is repeated for 1000 times, and the end time is measured similarly.
6. If the difference is more than a set value (2000 milliseconds), the client is rejected from playing the game.

Local Moves

1. When the game starts, every time a move is made, the client informs the server of the new coordinates. The server then multicasts all the clients of the new player coordinates and these new coordinates are then rendered by all clients.
2. For smooth GUI experience for the client, the client also renders its own changed coordinates locally, in case there is a slight delay in receiving the details from the server.

Testing

During normal conditions of testing, calibration tests for 1000 packets took around 281ms - 332 ms. When a lag of 50 and drop rate of 30% was introduced the server was still able to handle it and the calibration check took 340 ms. It is only when drop rates were at 90% that the server was no longer able to let clients join since it took calibration longer than our preset 2000ms to process the connection and it would timeout.

Tests for adding more clients were also tried and our checks to ensure no clients can join after 2 players are already in the game were successful. It was also successfully run across two different machines as well.

Conclusion

This project allowed for a deeper understanding of networks at an underlying level and was quite challenging in terms of having to optimise a game from all angles of networking while trying to ensure that we enforced good programming practices as well. Some further advancements that I would have liked to try would have been to optimise some known issues with the maze generation at random on two different machines. All in all the project was implemented almost as outlined in the proposal along with learning different optimisation techniques.

Special thanks to Dr Laith Al-Jobouri, Dr Andrei Boiko, and Gareth Robinson for their teaching and assistance during this project and module.

References

1. Blog.actorsfit.com. 2022. *[Linker error] undefined reference to WSAStartup@8' solution in Eclipse c++ - actorsfit*. [online] Available at: <https://blog.actorsfit.com/a?ID=01200-b2933e82-250d-4f6c-b32c-2ce435dfd84c> [Accessed 11 January 2022].
2. Titanwolf.org. 2022. *Solution for [Linker error] undefined reference to WSAStartup @ 8 'in Eclipse c ++*. [online] Available at: <https://titanwolf.org/Network/Articles/Article?AID=48248659-7870-4a9b-8ee6-8d37307e9278> [Accessed 11 January 2022].
3. Tangentsoft.net. 2015. *Winsock Programmer's FAQ: Get the Local IP Address(es)*. [online] Available at: <https://tangentsoft.net/wskfaq/examples/ipaddr.html> [Accessed 18 January 2015].
4. Lebeau, R., 2010. *Determining the IP address of a connected client on the server*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/4282369/determining-the-ip-address-of-a-connected-client-on-the-server> [Accessed 27 November 2010].

5. Docs.microsoft.com. 2021. *Winsock functions - Win32 apps*. [online] Available at: <https://docs.microsoft.com/en-us/windows/win32/winsock/winsock-functions> [Accessed 1 July 2021].
6. Cs.unc.edu. 2022. *Multicast sockets - programming tips*. [online] Available at: <http://www.cs.unc.edu/~jeffay/dirt/FAQ/comp249-001-F99/mcast-socket.html> [Accessed 11 January 2022].
7. Tenouk.com. 2022. *A practical and hands-on network programming tutorials with C code samples and working C program examples*. [online] Available at: <https://www.tenouk.com/Winsock/Winsock2example4.html> [Accessed 11 January 2022].