Multiplayer Game Development: Technical Report

CI628 – Multiplayer Game Development

Oliver Andrew Lee – S.N.: 14813961
14th January 2021

University of Brighton

Contents

1. Introduction

The visual design for the Client implemented for this project is intended to replicate the provided Java server view along with any additions made. However, different textures have been created and additional sounds utilised. When the Client is started up the User is presented with a 'Menu' screen from which they can join the server. Additional screens present include an error screen, game over screen and pong game screen.

During the Pong game, a player can see both player scores and what role their client has been assigned; with Player One, Player Two or Spectator being rendered appropriately. In respect to the Spectator role, this can be changed if one or both of the players are not present thus allowing control to be passed on to them. Once both players are present, a countdown begins before the game commences although players can still move prior to this. This countdown is resumed should a player disconnect and be replaced.

Included with this report are the modified files to both the Server Java code and Client C++ code alongside assets used. An unlisted YouTube video with a brief voice over demonstrating five clients connecting to the server alongside the presented server view has also been made available at: *https://youtu.be/7mgE7VKEn-c*

2. Implementation

2.1 Connection Termination

When a Client exits with user intent (either by closing the window or through the provided Escape key) the server did not register this connection as disconnected. This was partially remedied by having the client send a specified message ("*CON_CLOSE*") to the server which would then terminate the connection.

However, there would still be the problem of the server trying to send messages to these terminated clients. This is because of the use of `server.getConnections()` that still has the terminated connections stored. An ArrayList of connections was chosen to circumvent this. Connections are added to this array when they first connect and removed when `connection.isConnected()` returns as false.

Using an ArrayList to store the active connections facilitates the created method that sends to all active connections, It is also used in the Active Connection Check. However, an alternative implementation of the provided `MessageReaderS` class has been identified that would negate the use of this ArrayList entirely and is described in the Critical Review.

This data structure is also utilised in the `sendToActive()` method. With the current implementation of the `server.getConnections()` storing all connections, whether they have been terminated or not, this method theoretically increases the performance of the server over using `server.broadcast()`. It does so because instead of going through all possible connections, it only tries to send messages to connections that are known to be active.

2.2 Active Connection Check

This is an addition made to the provided Java server code that periodically checks the registered active connections to see if the client is still present. This was implemented via a Daemon Thread (Paraschiv. E, 2019) started when the server is initialised. The use for this check became apparent while testing the ability of a 'Spectator' to take over a player (described in section 2.3); while a player was correctly reassigned if the player intended to exit or close the client, this wasn't the case

if this client had crashed or otherwise disconnected without intent. Using a Thread that can be put to sleep allows the connected Clients required time in order to send a response and confirm they are still present.

The first thing that this Thread does is copy the previously mentioned ArrayList of active connections. It does this in order to avoid any potential concurrency errors that may occur if mutating this shared structure directly were to happen; for example removing an active connection while the main thread is trying to send a message to all of them.

Once a specified message has been sent to these registered connections and five seconds have passed, it removes any connections that have sent the correct reply from the copied ArrayList before terminating those that remain. Client connections that have replied are handled in the `onReceive()` method by putting the connection into a BlockingQueue. This structure was also chosen in an effort to avoid potential concurrency errors since this periodic check is performed on a separate Thread. This BlockingQueue has been created with a capacity of 25 meaning that a maximum of 25 active connections will be maintained.

2.3 Connected Client Roles

In order to allow for Clients to send identical messages but control different Entities, their roles are assigned on the server. This is done by storing the connection information in one of two variables of type Connection and is performed when each connection is first established. Implemented in this way, a suitable reaction in `onReceive()` can be performed. A "*_DOWN*" message from the Client assigned as player two would manipulate the relevant Entity via mocking keys I and K, with the same being true for the Client assigned as player one with W and S; allowing players to share a control scheme.

A third role is also present and is the 'default' role should both players be assigned and connected. Communication from Clients that are not assigned as either players have no affect on the game with exception to the "*TAKE_OVER*" message; should either player be disconnected then the 'Spectator' takes over the available position when the server receives this.

By implementing Client roles in this way, it also opens the possibility to add additional gameplay features such as pausing play if either player disconnects. Although not utilised in this project, it also allows for Clients to be sent messages that are specific for them if required, without needing to send it to all connected Clients.

2.4 Client Receive

It was discovered during development that too many calls to send clients a message in quick succession often resulted in messages being combined (See Appendix A). In the provided server code this can be seen clearly every time the ball hits the left or right wall. Whilst the first message containing scores comes through separately, the subsequent "*HIT_WALL_*" and "*GAME_DATA*" are presented to the Client as a combined command.

Whilst every effort was made in the server code to minimise unique send calls and thus receipt of these combined messages, it was decided to make amendments to the clients receive code to make it slightly more robust. Instead of looking for a specific string in the command (*cmd*) passed, the Client now checks to see if the command contains the relevant string. This allows for the client to react to multiple commands if they have been combined, such as still being able to read the game data alongside which wall has been hit.

2.5 Server Reconnection and Screens

The adjusted `run_game()` Client method results in only opening a connection to the server during the game screen. This means that if there is no server available or a connection error is experienced, the Client will not crash. Instead the user is free to attempt reconnecting to the server without opening a new instance of the Client.

This is achieved by using an Enum that represents the currently expected screen. The connection is terminated if the expected screen condition is met with `SDL_NetQuit()` being reached if this is "*EXIT*". Which screen that is expected is controlled largely in the main loop, with rendering performed by MyGame. It may be more prudent, if not just for the sake of readability then also extensibility, to change this so that each screen would have it's own class and dedicated render loop and input.

As a side affect of allowing the User to reconnect to the server, it also facilitates being able to leave a server and reconnect to a different one. This would, however, require the implementation of some method allowing a User to enter a custom IP address and port.

2.6 Separation of Concerns

True to the Server-Client architecture using an authoritative server and dumb client (Gambetta. G, 2020), all logic in relation to the game is performed in the Java server code. The Client is only responsible with displaying, or reacting to, the information that it receives from this server through textures rendered or sounds played. This helps to ensure that the view presented by each connected Client does not significantly differ which is important as they should both be playing the same game although exactly how dumb the Client is can be improved on through Client-side prediction.

3. Critical Review

The implementation of assigning differing roles to the different Clients and reacting to their respective messages accordingly allows for them to control different Entities without sending extra characters in their messages. It could be extended to add other players if required and could be used in a team based game with further development. For example, for a game that uses two teams, connections could be stored as two ArrayLists of Connection types where a Connection would be assigned based on the size of said teams and Connections removed if they are terminated.

Regularly checking the Connections that are currently considered connected by using a Daemon Thread to send a request for confirmation is implemented well for a project such as this. While connections that are closed with intent send a message telling the server to terminate their connection, this Thread correctly detects if a Client has crashed or otherwise disconnected without intent. Going forward, this could be further developed alongside changes to `MessageReaderS` (and subsequent removal of the ArrayList used as a workaround) to properly register these connections as closed as it normally would if a `EOFException` is correctly thrown when a Client disconnects with intent; this would reduce bloat in the return of `server.getConnections()` (as seen in Appendix B).

Despite having potential improvements to its implementation described later in this section, the implementation and maintenance of the active connections ArrayList is still suitable as a workaround. As there is little point in trying to send messages to connections that are no longer present, it allows the server to only attempt this on active connections as well as when checking for active connections.

Conversely, there are numerous areas where the implementation could see improvement. For example, the implementation of the Client receive method is only effective due to the relative simplicity of the Pong game being displayed; more complex games may find commands mixed with different arguments. If this is the case then it may be prudent to have commands sent with a set prefix. If a string in the arguments structure is detected to have that prefix then a recursive call of the receive method should be performed, passing that string as the command and the trailing elements as the arguments.

Secondly, while it functions perfectly well as a workaround, the implementation of the active connections ArrayList could be implemented in a much more succinct way. Although ventures were made into changing the implementation, due to time restraints and the time at which it was detected, this change could not be made succinctly. Within the defined `MessageReaderS` that is present in the server code, having it check the received message and throw an `EOFException` in `read()` on receipt of "*CON_CLOSE*" would mean that the server would correctly be informed of the disconnection and `server.getConnections()` should only contain connections that are still active (Appendix B Fig. 2). Thus the need for the ArrayList would be made moot as it would theoretically have the same contents as that returned by the server method.

Finally, despite being a seemingly small aspect of the implemented game, the countdown that initiates before play starts or resumes could be implemented on client and server both instead of it's current implementation of just on the server. This could be done alongside the player bats using Client-side prediction which would reduce the appearance of "lag" (Molinari. W, 2013). By beginning the animation of movement (or in the case of the countdown, rendering the text) whilst waiting for a response from the server and reconciliating, the client would feel more responsive. Whilst not strictly required with this implementation of a server on localhost, the need would become increasingly apparent in regards to connecting to an external server.

4. Conclusion

Many concepts have been made clear throughout this module, some of the most important ones that have been explored during this project are described. For example, there are the various architectures available ("one to one", "one to many" and "many to many") for a client server system. Many games follow an authoritative server, dumb client model which has game logic controlled on the server, following the one to many architecture. This can also help minimise certain problems such as player cheating.

Some time was also spent looking into Client-server Prediction where the client can reduce the appearance of lag that may be caused by latency. It can do this by changing the visual representation of the player whilst waiting for a response from the server. This can be further expanded, for games with several players, through Replication (Lysenko, M. 2014)

Active Replication would be to send player inputs to each connected Client so they so they can also show a level of prediction until the position is confirmed by the server. On the other hand, Passive Replication would be to only send Client Input to the server with the other Clients having no knowledge of it and therefore entirely dependant on a response from the Server to display other player locations.

Replication as described in the lecture material also involves choosing what information to send and when so information is only replicated on a Client when it is needed. Less resources are required by sending only what is required when it is required.

5. Estimated Grade

Demo: A-, *Visual Design:* B, *Report:* B+.
Estimated Total: B+.

References

Research

Gambetta. G, (2020), *Fast-Paced Multiplayer (Part I): Client-Server Game Architecture*, Available at: https://www.gabrielgambetta.com/client-server-game-architecture.html Last Accessed: [12th January 2021].

Lysenko. M, (2014), *Replication in networked games: Overview (Part 1)*, Available at: https://0fps.net/2014/02/10/replication-in-networked-games-overview-part-1/ Last Accessed: [14th January 2020].

Molinari. W, (2013), *Gamedev Glossary: What Is "Client-Side Prediction"?*, Available at: https://gamedevelopment.tutsplus.com/articles/gamedev-glossary-what-is-client-side-prediction--gamedev-3849 Last Accessed: [10th January 2021].

Paraschiv. E, (2019), *Daemon Threads in Java*, Available at: https://www.baeldung.com/java-daemon-thread Last Accessed: [20th December 2020].

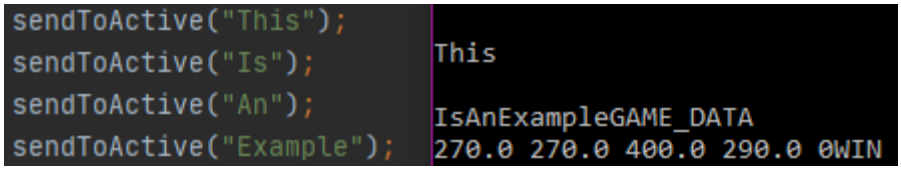
Third Party Assets

Gabovitch. I, (2009), *Click*, Sound, Available at: https://opengameart.org/content/click Last Accessed: [12th January 2021]. Used under: Public Domain Dedication (CC0 1.0)

NoiseCollector, (2005), *hit_bat. hit_wall*, Sounds, Available at: https://freesound.org/people/NoiseCollector/sounds/4391/ and included with the provided Java Server files. Last Accessed: [13th January 2021]. Used under: Attribution 3.0 Unported (CC BY 3.0)

Licenses

Appendix A – Command Combination Example



This figure depicts four of five calls to send a message and the console print out from the Client. As can be seen, the first message sent has successfully sent as a separate command but subsequent calls have been held in the queue and sent at the same time. This results in "IsAnExampleGAME_DATA" being recognised by the Client as one command. Ensuring each command and argument is sent with a proceeding ","will prevent this but will necessitate a change in how the Client reacts to received messages.

Appendix B – Output from Java Server

```
[com.almasb.fxgl.net.tcp.TCPConnection@49c12578]
No reply from connection: 1. Terminating.
22:41:07.222 [Thread-0                ] DEBUG Connection           - Closing connection 1
22:41:07.222 [Thread-0                ] DEBUG Connection           - Connection 1 was correctly closed from local endpoint.
[com.almasb.fxgl.net.tcp.TCPConnection@49c12578]
[com.almasb.fxgl.net.tcp.TCPConnection@49c12578]
```

Figure 1. *Depicts the output from the Java Server captured whilst testing. The server is set to print out the result from server.getConnections() every five seconds. As can be seen the connection in question is still being stored by the server despite being closed.*

```
Recv message: CLIENT_DATA,CONFIRM
Recv message: CLIENT_DATA,CON_CLOSE
22:50:56.039 [TCPServer_RecvThread-1   ] DEBUG Endpoint             - Connection 1 was correctly closed from remote endpoint.
22:50:56.039 [TCPServer_RecvThread-1   ] DEBUG Connection           - Closing connection 1
22:50:56.040 [TCPServer_RecvThread-1   ] DEBUG Connection           - Connection 1 was correctly closed from local endpoint.
22:50:56.040 [TCPServer_RecvThread-1   ] DEBUG Endpoint             - TCPServer connection (1) was closed
[]
[]
```

Figure 2. *Depicts the output from the Java Server captured whilst testing. Similarly to Figure 1, the results of server.getConnections() every five seconds. The primary difference is that it shows a connection that has closed with intent from the Client but by throwing an EOFException in read() (MessageReaderS). server.getConnections() no longer stores the connection information.*