Alterac Valley 2019

Technical Design Document: Networking

Object-oriented analysis and design
February 5th, 2019

Table of Contents

Team

- Aditya Dutta
- Samuel Bujold-Bouchard
- Mike Wells
- Eric Balas
- Raphael Rodriguez
- Spencer Marling

Overview

This document contains the technical information about the network design for Alterac Valley 2019. It will focus on how the TCP and UDP servers will function and will contain four core diagrams. The first will be a UML diagram of the different classes that will be created for this game. The second and third are sequence diagrams showing the flow of information between the server-client for both TCP and UDP connections. The fourth is a game loop diagram which will demonstrate how the flow of information will move during login and gameplay.

Detail protocols for UDP and TCP

TCP Server

The TCP (Transmission Control Protocol) server will be used when the player logs in or out of the game or selects what race and character class they wish to play. We will use TCP for these connections because it is the safest and most reliable protocol for passing sensitive information (ie. player account information such as passwords and emails) between the client and the server. Furthermore, when sending sensitive information the data will have to hashed and/or salted (salt is a randomly generated text that adds another layer of encryption to passwords) to ensure that the user's information is kept secure from outside connections. In addition, TCP has error checking via a two-way connection between hosts (see Figure 1 for a diagram), which can ensure that information is passed accurately or guarantees a player's character selection is read properly by the server. While TCP is slower than other protocols, the error checking and increased reliability as well as ensuring that packets are received in the correct order outweighs the slower communication speed. To set up the TCP server, we will use SDL_Net version 2.0.1 <https://www.libsdl.org/projects/SDL_net/>.

Anytime a new connection is requested, a new thread will be created to handle the connection. To create fast and reliable threads, we will be using the built-in threads in the Standard Template Library (part of the <thread.h> file). The advantage of using multiple threads is to ensure that the core game can run smoothly on the client while accepting packets of data from the server in parallel. Without these separate threads, it is possible that a network connection (specifically TCP connections as the client and server must communicate back and forth for error checking) can block the main thread from continuing until the next asset packet is received. This can cause significant delays and result in very choppy gameplay. As such, any time a network connection is made (either TCP or UDP) a new thread will be created.

Information passed with TCP:

1. Player information (username, password, email).
2. Player statistics (wins, loses).
3. Character faction and class.

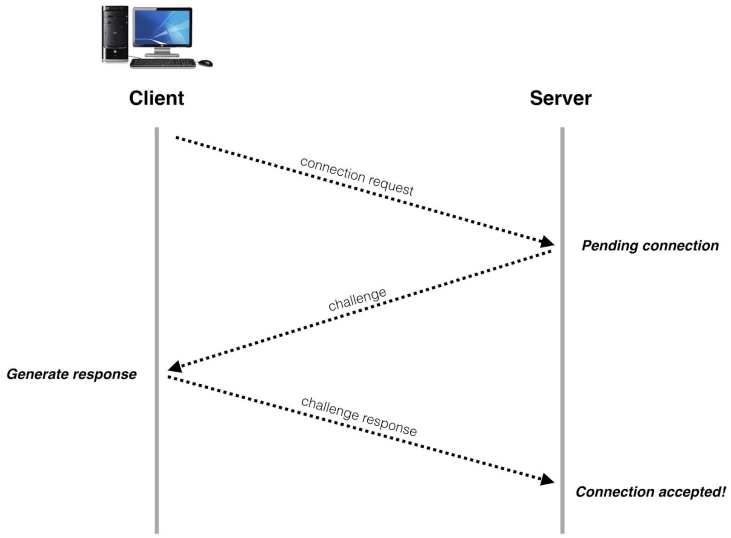

Figure 1: Client-server handshake for safe and reliable connections. From Client Server Connection by Glenn Fielder: <https://gafferongames.com/post/client_server_connection/>

Authentication

As we will be storing critical player information (eg. passwords, emails) we will have to use hashing. Starting in SQL Server 2012, they offer several algorithms for hashing passwords including MD2, MD4, MD5, SHA, SHA1, SHA2_256 and SHA2_512. For our game, we will be using the newer SQL Server 2017. Hashing a password will not make it completely invulnerable to attacks but will make it significantly more difficult to decrypt. Using newer algorithms, in this case SHA2_512, will improve the strength of the hash and increase security.

Storing user information with SQL would look something similar to:

```
DECLARE @responseMessage NVARCHAR(250)

EXEC dbo.uspAddUser
          @pLogin = N'Admin',
          @pPassword = N'123',
          @pFirstName = N'Admin',
          @pLastName = N'Administrator',
          @responseMessage=@responseMessage OUTPUT

SELECT *
FROM [dbo].[User]
```

And would leave a result of :

UserID = 1 LoginName = Admin
PasswordHash = 0xCD8C29B8DEED323
FirstName = Admin LastName = Administrator

UDP Server
The UDP (User Datagram Protocol) server will be used during gameplay to pass information between the different clients in the game and the server. We will use UDP for gameplay because it is very fast and can be used to send data at high frequencies (from 21 to 128 times per second). This is important for online games where players need to send information about what they're doing to the server at all times. UDP connections are not as reliable as TCP (described above) as there is no error checking. This is acceptable for our purposes as the online game will be a non-deterministic simulation (what one player sees in their game will be very similar to what another player sees but not 100% accurate). Furthermore, once a player is authenticated by the TCP server, the player will be given a token that will be stored in the database. Whenever a player sends information to the UDP server, it will have to authenticate that data with the tokens. If the authentication fails, the action sent to the server is dropped.

Information passed with UDP:

1. Player transform information (location and rotation).
2. Core UI information (character deaths / kills for scoreboard, tower and graveyard captures, captain or general deaths).
3. Player actions (attacks, heals, captures of areas like graveyards).
4. Buffs and debuffs (let the player know if they get a buff and how long it will last. Client will handle how the buff changes their actions).
5. Character health (to be displayed on other players UI and to check for validity).

Area vs. World UDP Servers
To reduce the amount of information being passed around the UDP server during gameplay, we will be breaking up the main gameplay server (called the World server) into separate smaller servers (called Area servers). These smaller area servers will be divided up based on in-game locations that a player can explore (likely 4 to 9 areas). This is done because a player at the southern end of the game world does not need to know specific information about another player who is at the northern end of the map. As such, specific player information (transform and action information) will be passed via the area servers. General team and game information that all players need to know (like tower captures or general deaths) will be passed to the world server. As players move between areas in the game, they switch which area server they receive information.

TCP and UDP Sequence diagram

TCP Connection:
After the server finishes its initial setup, the client will need to establish a TCP connection to log into the server using their authentication information (Figure 2). As mentioned previously, TCP will ensure the information is passed properly during this phase. The server will then have to communicate and verify this information with the database. Once the result is positive, the client is informed of the success and then they can select a class and connect to / find a game. After this, the client will need to establish a UDP connection with the server and close the TCP connection.

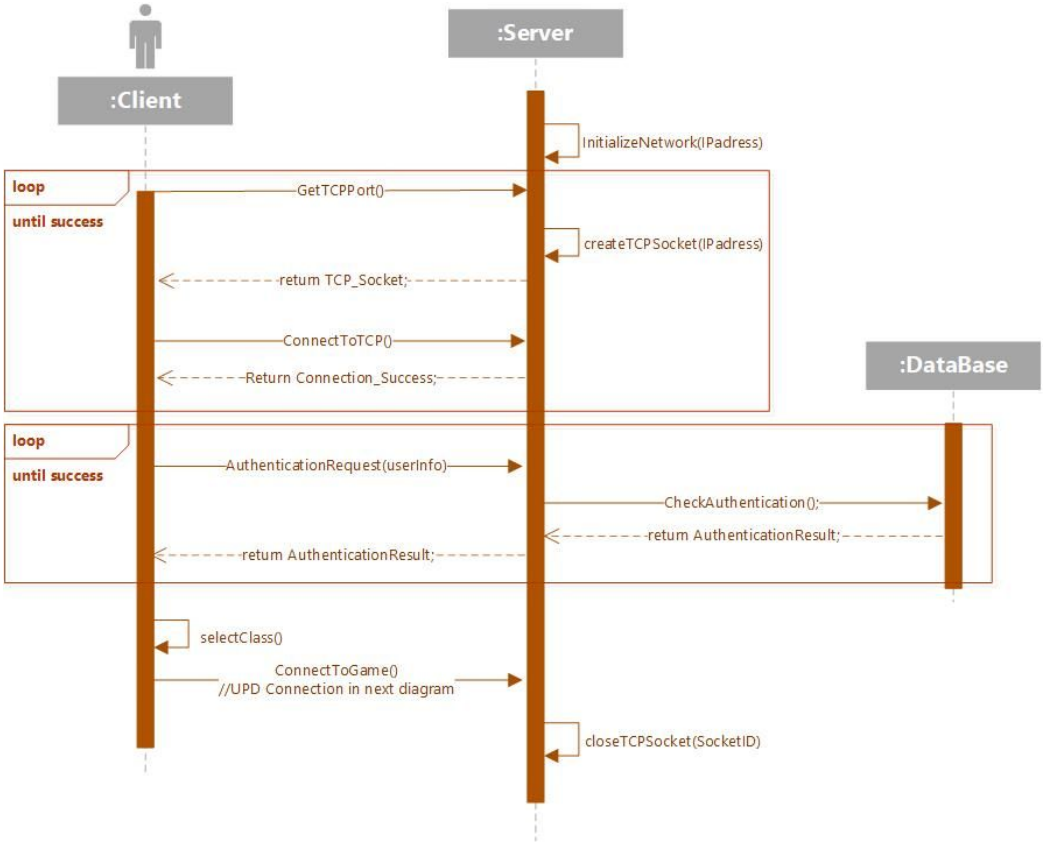

Figure 2: Sequence diagram of the initial TCP connection. Note : "closeTCPSocket(SocketID) happens in "ConnectToGame()" as shown in the next diagram (Figure 3).

Once the critical login steps are complete, the client will need to establish a UDP connection (Figure 3). This is done in a similar fashion as the TCP connection but will stay open until one side wins. The server and the client will broadcast information to each other after the connection is established.

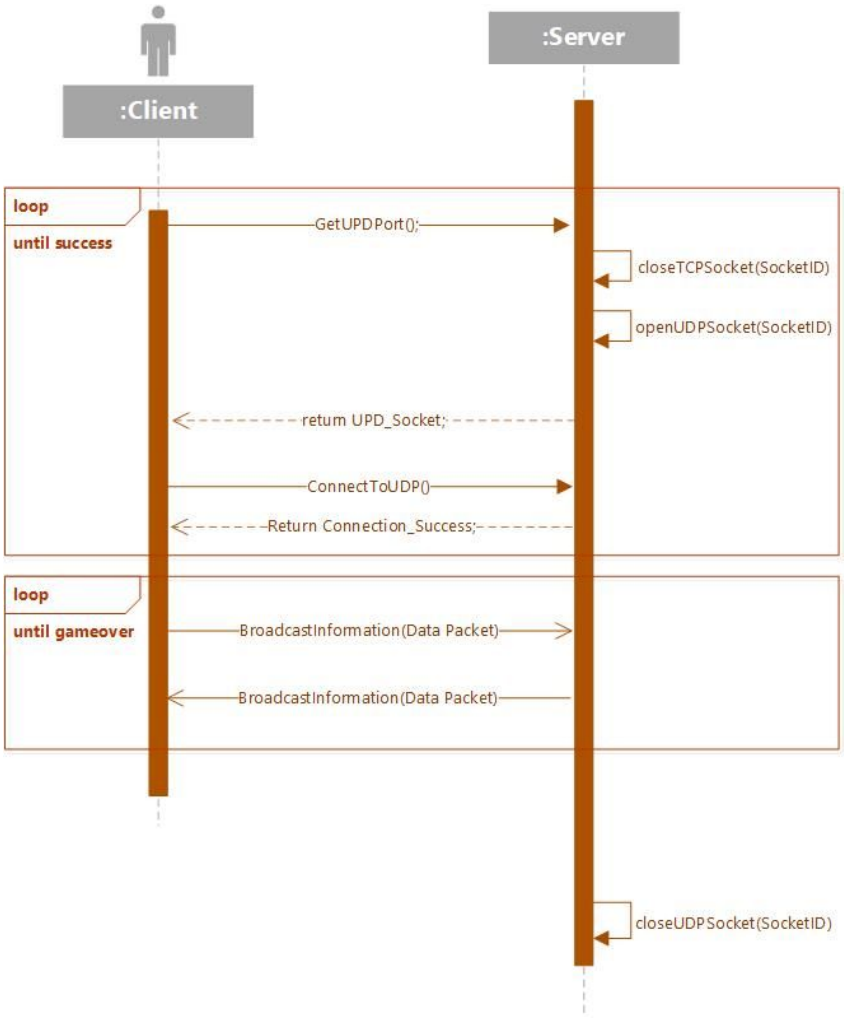

Figure 3: Sequence diagram of the initial TCP connection

UML Diagram

There will be several classes created to allow for efficient and safe networking between players and the different servers during gameplay of Alterac Valley. The two core classes detailed below (Figure 4) are the Client class and the ServerUtilities class, which contain the methods and variables of the network and will both require access to the methods of SDL_net. The ServerUtilities class is abstract as both the TCP and UDP servers will require similar variables and methods to run properly. The TCP server will require some association with the Database and will indirectly check user login information against the userData structs stored inside the Database. The UDP server is split into the world server and area server, whose differences are detailed above (see Detail protocols for UDP). These two servers will be in frequent communication with the DataPooling class, which passes collected data to the DataProcessing class, which subsequently passes this information to the GameState. The GameState will then update both the area and world servers.

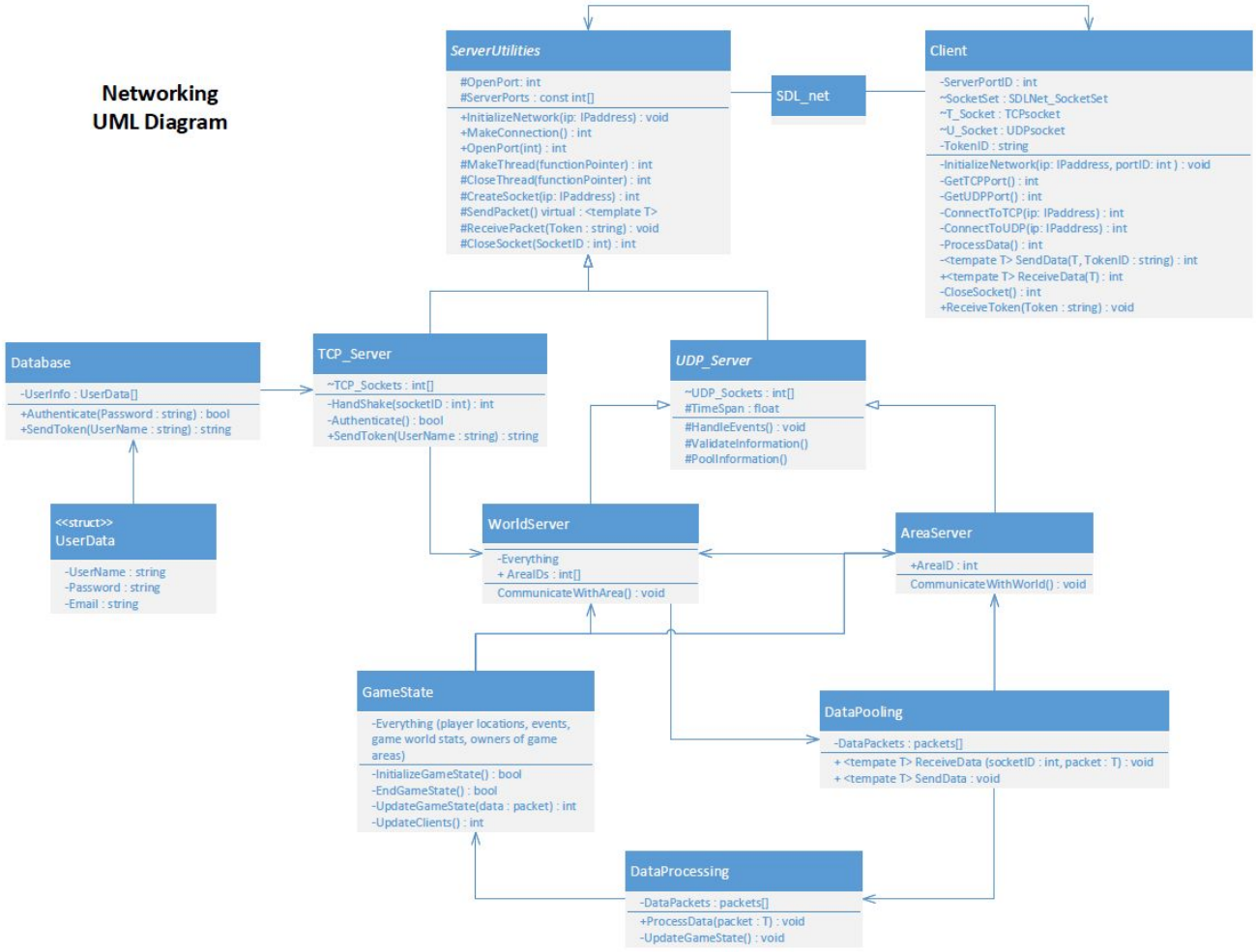

Figure 4: UML diagram of the networking different classes

Core Game Loop

The core game loop can be found in the sequence diagram (Figure 2 and 3) as it covers from the time the game boots until a game is complete. The diagram below (Figure 5) provides a more complete version of the core game loop.

Login Step

At application launch the client will connect to the login / authentication server which will then verify user credentials with the database. After verification, the server will check the client's version with the patch server and update if needed.

Game Loop Steps

The client will then connect to the world server via TCP/UDP and transfer information until the battle/game is completed. This information will vary depending on where the player is located in the playing field. This is where we will be making use of area servers to limit the amount of information the player will be broadcasted. Certains players will not receive the status of players that are not relevant to their current status.

The loop will start at the server side during the game start, sending initial information to all players. The server will then handle events from the clients, process the events and update the game state and finally broadcast it to the clients through UDP. The client loop receives events from the user and server. The events are then processed and forwarded to the update method, which updates the game state locally for the client. Finally, the new data is rendered on the screen.

Once the game is completed, the world server will communicate the results to the database (winner, score and more) and close connections with the players.

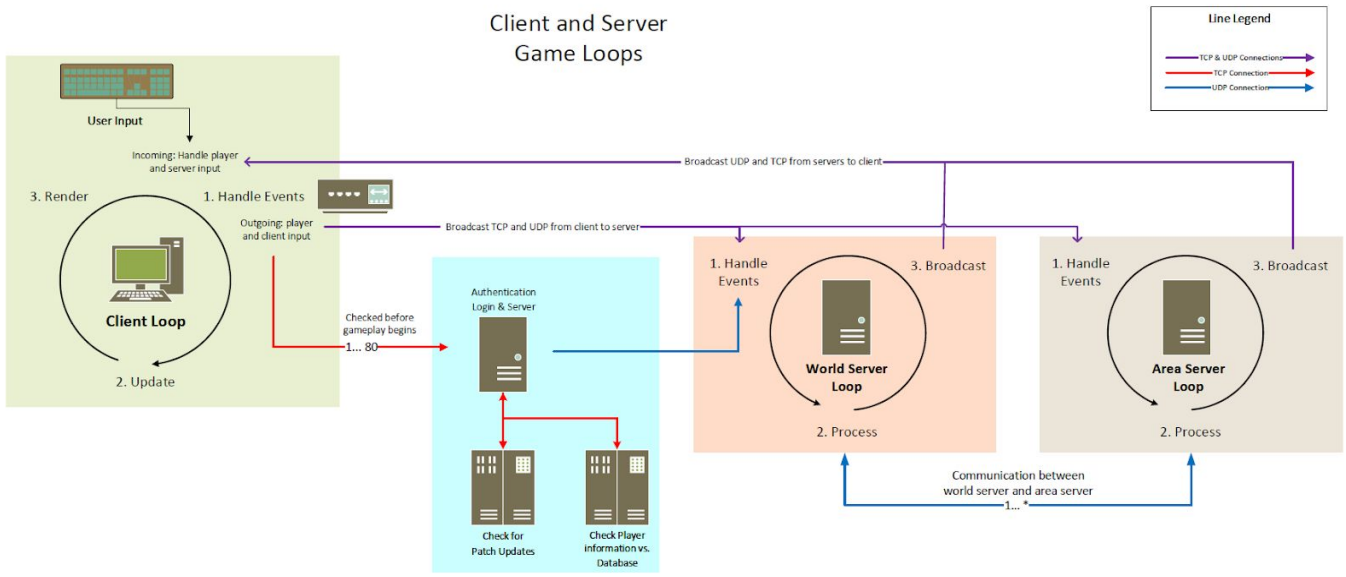

Figure 5: Core game loop diagram of the client and server during runtime.