**Final System Outline**

The final system created is a networking middleware system, named ‘TrakWorks’. This system works as a sort-of middleman between the server code and the user-defined code, it does this by allowing the user to add any arbitrary data to a clients’ network data storage class.

**Development Issues**

The first issue encountered during the development of this system was creating the network data class, as taking in any arbitrary data to be passed through RakNet’s Peer Interface proved more difficult than I thought. There were many approaches taken to try and overcome this issue, some failed at the first hurdle while some failed at the finish line so-to-speak.

The first approach taken to overcome the universal data class issue, was to use templated functions and store the data in an std::map<typeid, std::any>. This approach while in theory works, did not prove successful in practice as; C++ does not support the storage of an object’s type, and retrieving data from an std::any requires you to already know the type or possible type. The std::any was a promising idea at first but when looked further into, from my understanding the std::any is essentially a pointer to some bytes so you need to tell the compiler exactly what type it is to retrieve the correct amount of bytes. For these reasons this approach was quickly terminated.

The next approach taken to overcome the issue was to maintain the templated functions and instead store the data in an std::map<const char\*, unsigned char\*>. This approach again in theory should work, but in practice did not as sending an unsigned char pointer across the network, in hindsight obviously wouldn’t work. This is because if a pointer was pointing to a float stored at 0xFFFFFF on one machine on another this same memory address might instead be a string for example. There was also an issue with sending a const char\* as occasionally the string read in would be mangled, but the main issue with this approach was the data not being sent correctly or at all, so again this approach was terminated.

The final, successful, approach taken was to again maintain the templated functions but this time instead of relying on the std::map to store a key-value pair, we would manually store a key-value pair of <const char\*, std::vector<unsigned char>>. This way we would have complete control over how the data’s name and data’s value was stored in our network data class. Internally when inserting new data or overwriting old data in the network data class, we are storing the name of the data as a const char\* but for the data value we are first converting it to its raw byte form then storing it as an std::vector<unsigned char>. This approach while almost tripling the size of the network data header file, was the best approach in terms of reliability and controllability.

The other major issue encountered during the development of this system was when trying to implement it into an application, namely a networked recreation of ‘Pong’, as there were many minor issues that combined to form one major issue that had to be resolved / manipulated to allow ease-of-use of the system. A few of these issues were just functional and backend systems that had to be adapted to allow the user to manipulate the client’s settings without breaking any of the backend logic or creating edge cases.

The first minor issue to be rectified was to allow the user to modify the network delay and rate of the client update loop. At first this was done simple by making setters for private constants; NETWORKFRAME and FPS, while this seems like a sound solution there was an underlying flaw with the client’s update logic. Basically, the client’s update loop was being run on its’ own thread to allow it to be slept without affecting the rest of the application. But while it was slept the actual demo application would change some of the network data and then the next time it ran the client would overwrite it with a mangled / old data pair. This flaw led to some data getting corrupted eventually leading to the application randomly crashing without warning, and without support for debugging two threads at once. This was rectified by not having the update loop on its’ own thread and changing it to have to be called by the user’s code first and the network delay was modified from frame based to time-based delay.

The next minor issue to be rectified was to modify the way the client sent a new game object over the server. Originally this was designed to create a game object with only position, size, and velocity in mind, which meant the new game object was limited to only containing data for position, size, and velocity, obviously flawed. The change was simple but cumbersome, as the client’s function for spawning a game object was changed to take in a game object rather than the individual data. But the server-side code also had to be slightly re-written to receive the additional data. This in turn lead to having to modify the game object’s read and write functions to allow for other RakNet::MessageID to be taken in by the function.

The last minor issue to be rectified was not having enough helper functions for the client, these helper functions were mainly Booleans and references to client member variables. A few of these functions are listed below:

|  |  |
| --- | --- |
| Function Declaration | Function Use |
| IsConnected() | Returns true if the client had successfully connected to a server. |
| IsServerFull() | Returns true if the client could not connect because the server was full. |
| OtherObjects() | Returns a reference to the member variable storing other client game objects. |
| OtherData(\_id) | Returns a reference to the network data of the client with \_id in the stored map. |
| Data() | Returns a reference to this client’s game object’s stored network data. |
| ID() | Returns the id of this client’s game object’s stored id. |
| NetworkFrame() | Returns true when network delay reaches 0. |

**System Performance**

The main optimization made to the network backend was to allow changing of when network data was sent to the server, limiting the overall amount of data sent and improving performance for the end client. This small optimization allowed for the user to manipulate the network delay to fit their average network conditions and allow for saving on processing power required to maintain a stable and smooth application.

Some areas where improvements could be made would be to the interpolation algorithms as they do work, but are not very smooth in terms of gameplay / application. The linear interpolation is the most reliable but breaks down as the user updates the velocity of the object that is being interpolated. In contrast the cosine interpolation is smooth for the client-side game objects (i.e. the player) but for any server-controlled game objects (i.e. the ball) that send their data once a second it breaks down as the networked speed is not constant.