Jarrett Briody

Professor Chris Egert

IGME.589.02

May 8, 2019

Problem Statement:

Initially I was attempting to find a solution for defining expected packet structure both client side and server side. My expectation was that I would either use some sort of serialization or file IO. Unfortunately I quickly realized this didn’t even make sense and was not necessary, as we were just going to run the client and server through shells as DLLs, and packets would be user defined structs. Originally the expectation was that we might be able to make the server handle data based on that defined structure, but obviously this doesn’t make a shred of sense and obviously the developer is going to need to write logic for how data is handled otherwise the server won’t be authoritative.

My focus then switched to building the server DLL using a series of callbacks that would be defined by the developer, so that the DLL could call user defined logic when it received packets. I worked on this and got it to work, but then realized it was susceptible to slow developer code. I then changed my focus once more.

Finally, I focused on building a producer consumer type threaded model for the server shell and DLL, where the DLL would enqueue packets and associated information asynchronously, and provide the necessary methods for the shell to call within its own independent loop. The server would run separately from the DLL core, and the developer couldn’t write code that would slow or stall the DLL functionality.

Significance:

Building a server using the producer consumer model is probably standard for a lot of professional applications, but it is particularly necessary for libraries or packages that are going to be used by other developers who may be more inexperienced in writing optimized code that doesn’t take forever to run. This threading model also provides a sort of base for RUDP, were we to take this further.

What I Did:

See above for the history of the idea. Regarding what the idea turned out to be, I didn’t try much before I got it working. I set up an internal thread on the DLL which manages the server core for receiving messages, which enqueues packet data in a mutually excluded queue. Methods were created for dequeuing packets through the shell, checking if there were packets, among other things. Aside from programming errors or initial timing issues with threading, it worked well fairly quickly.

What I Discovered:

I discovered that this worked so well that this was more than likely a common way to implement a network manager similar to this. Slow shell code, despite allowing the queue to get filled to the brim with packets it couldn’t process, wouldn’t hang up the receiving and enqueuing of packets, and wouldn’t disallow the calling of functions in the DLL since the core ran independently. The shell that runs the DLL creates an extremely simple environment for another developer to use the library, with just a few entry points they need to worry about.

Conclusion:

Given the number of issues with the original ideas I selected, the producer consumer model for the server side worked very well and accomplished the final task I decided on. The DLL runs autonomously, and the shell is simple to implement for the developer. The queue system it uses makes the internal structure super simple for adding functionality, and is much easier to work with than just a simple array.