Technical Competency Demonstrator

Team Decided

Link to the Technical Competency Demonstrator
Link to the networking code library (RaftNetworking)

After analysing the project, the most fundamental technical process which we identified is the sending of messages between nodes. All functionality of distributed consensus is built upon the ability to send/receive messages from nodes in a timely fashion. After our initial findings of there not being a publically available non-bloated networking library which suited our needs, we determined we needed to write our own. This is what we'll be doing for our Technical Competency Demonstration (TCD).

This decision was not taken lightly, as writing socket level programming in any language is fraught with error/difficulty and technical programming traps for new players. Fortunately, one of our team members has prior experience writing libraries like this, in C, C++, and the C# language as well.

When estimating performance bottlenecks in the raft consensus protocol running across distributed nodes, the largest factor which impacts consensus time of the network is the node's network latency from one another. Therefore, we chose UDP based networking over TCP, this was due to UDPs lack over acknowledgement message overhead for each message reducing overall message latency. The risk of running UDP however is that as the packets are connectionless, messages may be lost after transmission without the TCP resolving them in its host-to-host OSI layer (layer 4). This has been deemed not an issue for our project, as the raft consensus algorithm itself handles internally any packet/message loss issue.

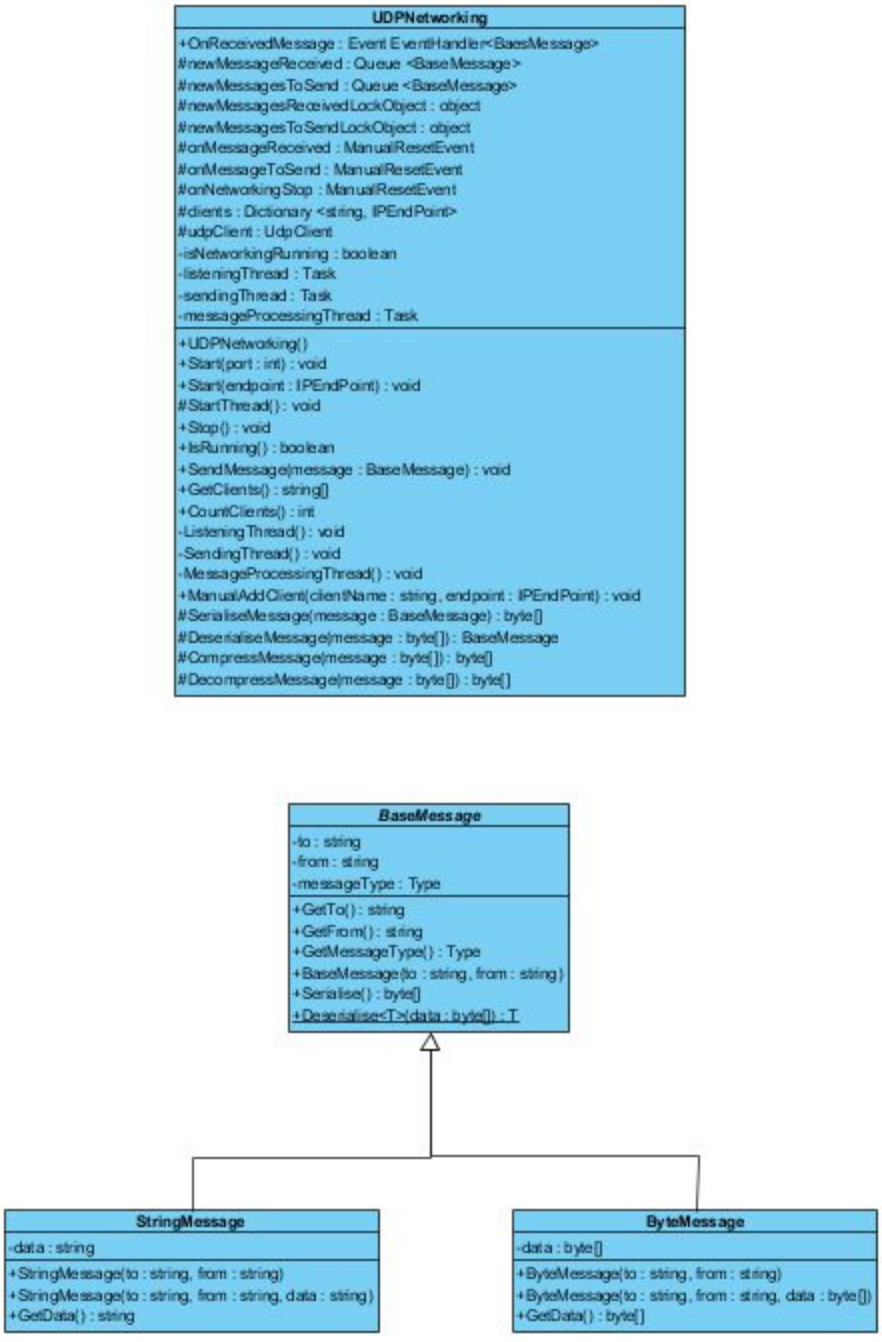
Initially we thought that we'd be caught out with a maximum packet size issues, as with UDP networking over the internet you are only minimally guaranteed 576 byte messages due to historic networking device buffer sizes, and a likely modern packet size in the 1472 byte range. At such small packet sizes, we'd need to implement our own packet reassembly protocol into the messages to support our larger message requirements, however upon testing we found that Microsoft's socket wrapper UdpClient was already implementing this and allowed us to send messages with a total length of 65KB, much more than we'll reasonably require.

During one of our oversight meetings with Jim, he confirmed that our team's technical competency demonstration would have to have the following attributes:

- Serialisation/deserialisation of a message in JSON
- Sending the serialised message over the network, and deserializing it to write the output
- Single threaded

We decided to go above and beyond these specifications and implemented a multithreaded networking library consisting of 3 synchronized threads. We realised it wasn't worth writing code for a TCD to just throw it away after, and that we could take this time to write useful code now and then we'd be able to use it later on in the project.

Class diagram

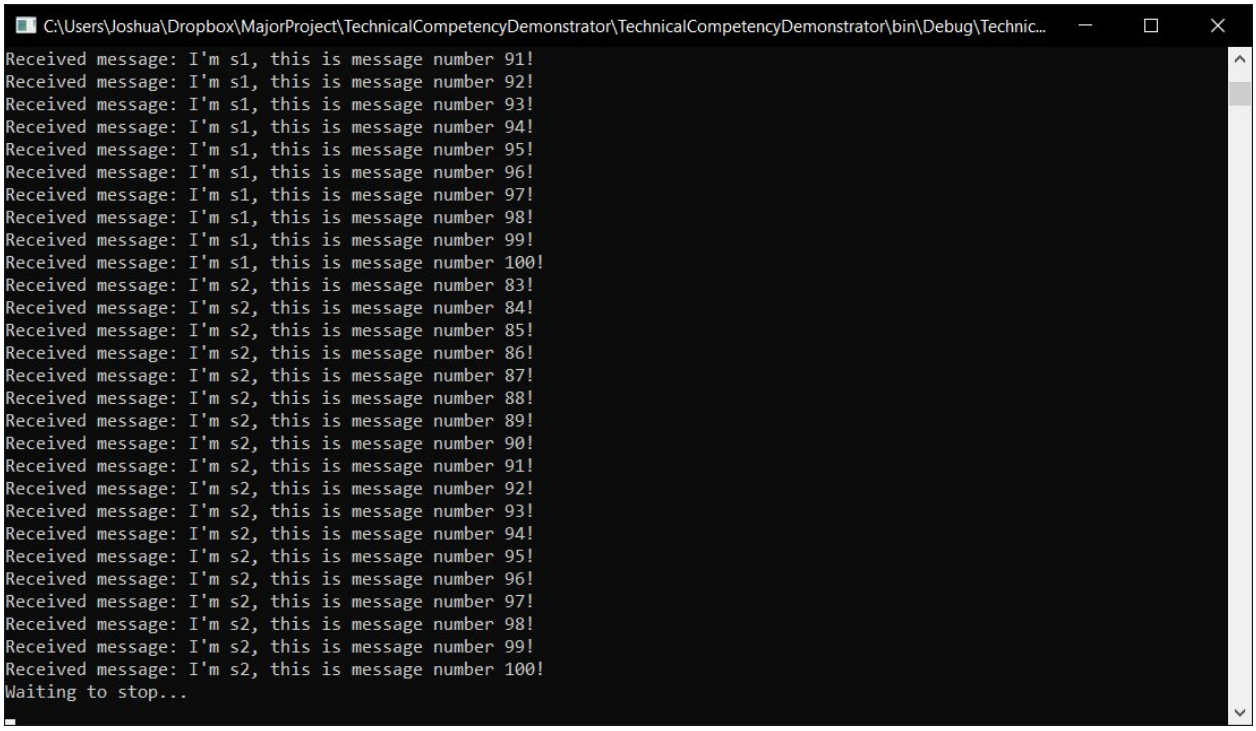


As you can see from the class diagram, the networking class itself is a single class which is interfaced with through it's minimal public methods based on an Interface, this class sends/receives messages derived from the BaseMessage class through the property of polymorphism. The library minimally only needs to send ByteMessage messages, however we've added StringMessage messages as well to simplify our example TCD program. In practise in the consensus library we'll be deriving our own messages based on our needs of message types.

Example TCD Program

```
namespace TechnicalCompetencyDemonstrator
{
    class Program
    {
        static void Main(string[] args)
        {
            UDPNetworking networking1 = new UDPNetworking();
            UDPNetworking networking2 = new UDPNetworking();
            networking1.OnReceivedMessage += OnReceivedMessage;
            networking2.OnReceivedMessage += OnReceivedMessage;

            networking1.Start(4445);
            networking2.Start(4446);

            networking1.ManualAddClient("Localhost", new IPEndPoint(IPAddress.Parse("127.0.0.1"), 4446));
            networking2.ManualAddClient("Localhost", new IPEndPoint(IPAddress.Parse("127.0.0.1"), 4445));

            for (int i = 1; i <= 100; i++)
            {
                StringMessage stringMessage1 = new StringMessage("Localhost", "Localhost", "I'm s1, this is message number " + i + "!");
                StringMessage stringMessage2 = new StringMessage("Localhost", "Localhost", "I'm s2, this is message number " + i + "!");
                networking1.SendMessage(stringMessage1);
                networking2.SendMessage(stringMessage2);
            }

            Thread.Sleep(1000);

            Console.WriteLine("Waiting to stop...");
            Console.ReadLine();

            networking1.Stop();
            networking2.Stop();
        }

        private static void OnReceivedMessage(object sender, BaseMessage e)
        {
            if (e is StringMessage)
            {
                Console.WriteLine("Received message: " + ((StringMessage)e).Data);
            }
            else
            {
                throw new Exception("Unknown message type");
            }
        }
    }
}
```

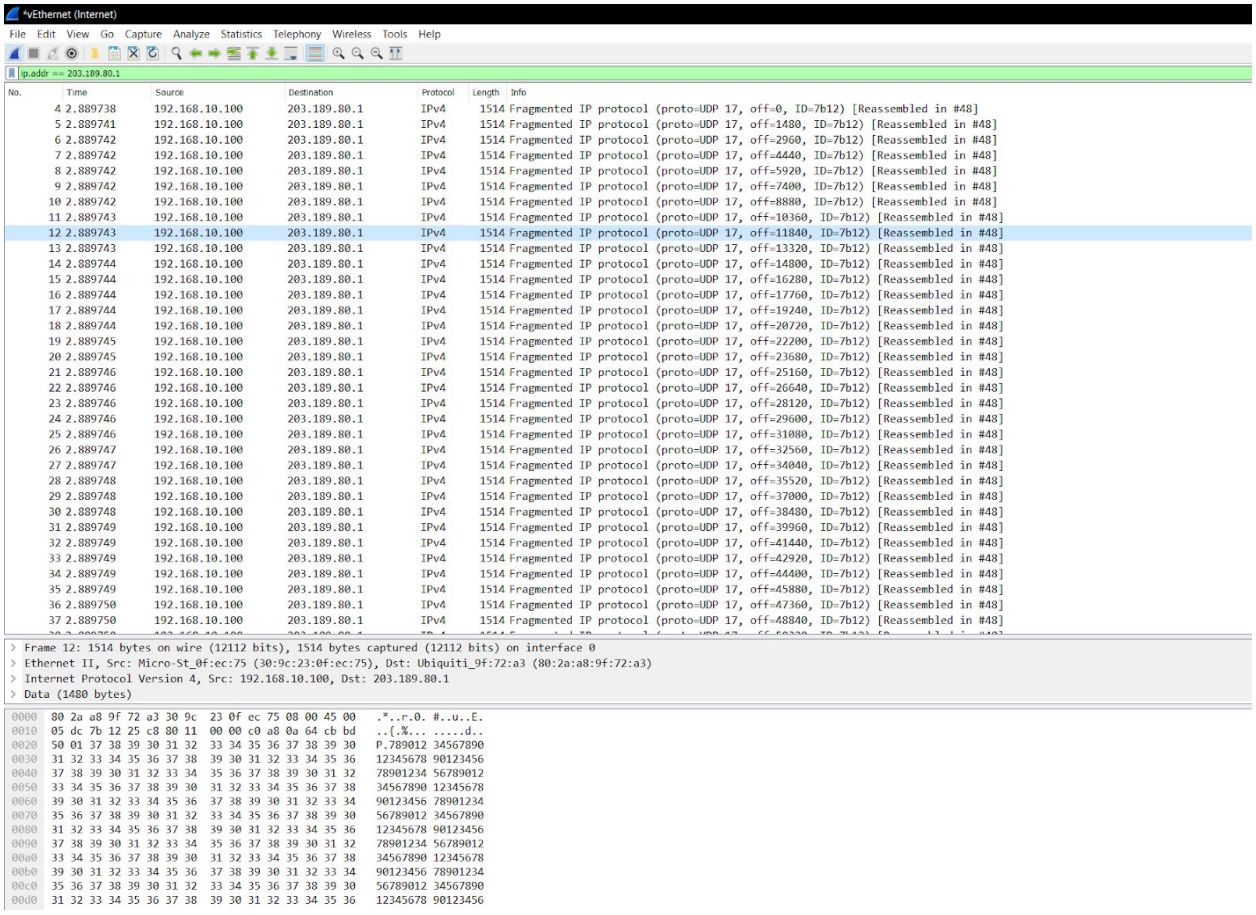
Example TCD Program output



For usability/demonstration reasons, the one example program is used for displaying the two networking classes communicate. The TCD example program references the dynamic linked library (DLL) file compiled in the RaftNetworking project. It works by completing the following steps:

- Instantiation the UDPNetworking classes so they may be used for communication
- Registering an OnReceivedMessage functions for handling received message processing
- Starting the UDPNetworking classes listening on the designated ports, this starts all their multiple threads
- Manually registering the IP details of each client with one another so they're able to communicate
- Then there is a loop of sending 100 messages both ways between the two classes. When the OnRecievedMessage function is called, you can see from it's console output the out-of-order properties of UDP messages and how they're also occurring asynchronously.
- Then we simply sleep to to allow the previous asynchronous actions to perform before writing out our "Waiting to stop…" message, this is so the message displays at the bottom with it's readline function
- After the user presses enter on their keyboard, the program then stops the networking classes, which stop their threads, and that completes the program

Network level Wireshark packet reassembly confirmation



The last image is of actual networking wire output captured through Wireshark, where it can be seen Microsoft's socket wrapper does indeed break up a message of 65KB into 1514 byte packets with 1480 byte payloads. The target IP address had to be a non-localhost host address, as localhost traffic does not pass through the networking stack where Wireshark can read it.

Although this library is fully functional for our needs as a TCD, before implementing it into our own project we've highlighted additional features we'd like to add:

- Smoothly handling all exceptions with detailed error messages
- Adding full unit testing to ensure code quality
- Perform code review and ensure the code meets style guidelines
- Add a logging ability, likely with the NLog library
- Implement a heartbeat mechanism
- Ensure all functions are thread safe
- Derive a class which implements our desired security protocols
- Cleanup various code smells, closing of listening thread and isNetworkingRunning flag