Project Overview

CS 327 - Peer 2 Peer Lab

Andrew Lucas

The first thing I thought about was what language I wanted to try and use for this project. I wasn't sure what would be best, and I ultimately settled on using C#. Since I had spent a lot of time using C# when working on previous Unity projects, it was the language I had the most experience with, and I felt confident using it. It turned out to be a good choice, at least as far as I'm concerned. The MS Docs for C# were surprisingly helpful for this project.

The first part of the project I worked on was sockets. I had no previous experience using them, but luckily the MS Docs had an example of a client and server socket. I was able to use that example to learn the basics of how sockets work, and how I could potentially implement them in my project. After a bit of tweaking and testing, I was able to successfully connect a client socket to a server socket, but only locally. Still, it was a good start.

Next I considered how to try and discover nodes over the LAN. The first part of that was how to find other devices on the LAN. When looking into it, I found that pinging the network seemed to be a common and easy approach. I found a couple different methods for setting up the pings, and ended up using the simplest one. First I needed to get the IP of the host the app was running on. Then, I could break up that IP, keeping the first 3 octets, and then generating 256 IP's (0-255) for the last octet. Then I could use those 256 IP's to create pings. If an IP responded before the timeout, it was added to the list of active devices. Also, each ping was sent asynchronously, so I didn't have to wait for one ping to finish before going to the next. However, that did mean I needed to add a way to wait for the pings to finish before continuing, but that was simple enough to add.

One problem I ran into with the pings is that for some reason my laptop and desktop could not ping each other, but other devices on the network were pinging. I tried a couple of fixes, and it did end up working, but I'm not sure which of the fixes did it. One was my laptop was set to public networks instead of private, so I switched it to private. I also found a setting in the firewall settings that related to receiving and sending pings, so I turned that on as well. After those two fixes, it was working. So now the first assumption for my project is that it assumes ping's will find all active nodes on the LAN.

Now that I have a list of device IPs to check, I can attempt to connect to a node for each IP. For the first setup of sockets I had them all in one class, so at this point I created

separate classes, one for server sockets, and one for client sockets. Then I modified the client socket to use the list of IPs. For connecting, I used a specific port, so the client could then check that port for each IP. If a server is active at that IP, the client will connect to the server node, otherwise it won't have a node to connect to. At this point I was able to establish a connection between my laptop and desktop, so I successfully created a client and server socket, and was able to discover them over the LAN and connect to them.

The next part I worked on was handling files. The first thing I had to figure out was how to go through all the files in a directory. Thankfully, I learned a good way to do this from my CS 342 class this semester. For that class we needed a function that would go through every file in a directory, and all the subdirectories as well. So I was able to use that exact function to iterate through all the files in a given directory. For the purposes of this project, I created a sync folder in the directory of the executable so that the path would always be known.

Now that I could easily go through all the files, I had to figure out what to do with them. I knew I would need to compare files on one end to files on the other, so I figured creating a list of files would be a good start. Then I thought about what would need to be compared. The first obvious choice is to compare the file paths, looking only at the portion of the path starting at the sync folder, as anything before that could be different. If the file path exists, then that file exists. But if the file already exists on both ends, then another comparison is needed. At first I considered comparing the file size, but on thinking about it more I realized it is definitely possible that two files that are different could still have the same size. Then I thought about trying to hash the file data itself and comparing the hashes. I was able to find a built in MD5 hash function, so I used that to hash the file data. After testing to ensure the hash would be the same for the same data, regardless of other aspects like file name and such, I was confident in using it to compare. But now I couldn't just use a list, so I created a dictionary with the keys being the file paths, and the values being the file hash for that file path. Now I was able to iterate through all the files and add them to the dictionary along with their hashes.

When I first started the project, I made it a Windows Form project instead of a console app. I had intended to try and add some file management to the form using buttons or some way of showing the files and such, but eventually decided that was beyond the scope of what was needed for this project. After working on the file manager, looking through all my code was difficult. A lot of it was essentially duct taped together, and hard to follow. It was very messy. So, since I no longer needed the form part of the project, and it was disorganized anyways, I decided to start a fresh project as a console app instead. By doing so, I was able to ditch the previous mess and keep things more

organized and logical. I'm glad I did too, because when implementing it the first time, I could see that it worked, but there were parts I didn't fully get how it worked, it just did, but reimplementing what I had before, I was able to more slowly go through it piece by piece and really understand what each part was doing. I was also able to better tweak it to work for what I needed, and to leave out the extra bits I wasn't using.

After ensuring the implementation worked as well as before, it was time to work on the syncing function. One difficult part when working on this was sending and receiving messages in the right order to ensure both the client and server were actually executing the correct part of the sync. First the client would send a file path. Then the server would send back if it had that file path or not. If not, it needed that file. If it did have the file, then the client would send the hash next. The server would compare hashes, and return if they were the same or not. If the same, nothing needed to be done as they were already synced. If different, then I needed another comparison to figure out which of the files to use. I decided to use the last write time, so whichever of the two was most recently written to would be the one to sync. At first the sync function was one way, meaning that the client would send all of its files to the server, so the server would be synced with the client, but the client would be missing the server files. After making sure the syncing worked how I wanted it to, it was easy to enable the server to then send its files to the client so both would be synced.

The last big feature I worked on was files being updated while the app is running. I was able to add a built in C# class that can watch for file changes and raise events when they are. That way, when a file is altered the file manager can update the dictionary. Then it was a simple matter of having the client resync every minute. Originally I was going to have it check if it needed to sync every minute, and only sync if something changed, but then I needed a way for both the server and client to handle changes. With having the client sync automatically every minute, it's guaranteed any changes will be synced, so it was easier to do it that way instead.