Luke Brodwski & Kevin Daley

Dr. Williams

CS490 Networking Spring 2015

27th April 2015

System Documentation

For our data structured we used a hash table. The reason we chose a hash table was because we would be able to index the table by using the name of the song as the key. This way multiple entries of the same song would not exist. They key points to an Entry object which we created. This object holds an Array List of IP’s. These IP’s are ones of the peer’s who have the file (key). An array list is perfect for this because the Array List will adjust it’s indexes when we add/remove entries from it so the 0th index will always have a peer. This way when the server needs to make a response with the peer IP, it can just grab it from the 0th index. If the Array List of IP’s is empty when the hash table is updated then the entry inside the Hash Table is removed since no peers have this file in the network. For our Hash table we had to use a Concurrent Hash Table, which doesn’t throw any exceptions when multiple threads try to access or edit the table. Since both the peer and server are multithreaded it is the right option.

One of the design choices we made was to split all our GUI classes from our network based classes. The reason we did this was to keep our program more organized which made it easier to isolate any problems. It also makes it really easy to update now because if we wanted to create a new GUI we wouldn’t have to touch any of the networking classes. The GUI files have certain ActionListerner’s which create networking objects and pass them into Threads. One key aspect of all our threads is instead of extending off the class Thread, we instead implemented Runnable in the needed classes. The reason for this was since we were not changing any of the default behaviors of a Thread, there was no need to change/override/add any methods. Implementing the Runnable interface meant that we just needed to add one simple run() function which did all the dirty work for us. This way we can create the Runnable object first, pass any parameters or change any options and or configurations with the object. Then we can spawn a new Thread object, pass it that runnable object and let it do its thing. If we wanted to stop the through we would be able to by changing the looping variable through the Runnable object we created prior. This made working with Threads very easy, and added lots of functionality to our application. For all our UDP/RDT connections we put all our methods into one class which has different configurations depending on wether a client, or server is using this object. The reason for this was the server and client both at one point in our program have to have a RDT stream. So instead creating multiple classes with the same functions, we added configuration variables and functions to the RDT so they both can share RDT receive, and transfer. For all our TCP connections we isolated them in our Peer class. This class listens for any other Peers wanting to make a connection. Once a connection is made it spawns a new Thread and passes the socket into the Thread. This thread is then able to either download, or upload a file to a peer in the background while the SocketServer goes back to listening for more peers. This design lets us accept multiple peers downloading files from one peer, as well as the peer downloading multiple files from other peers.

It is the peer’s responsibility to update its list of remote files by asking the server for the latest list. At the same time, it tells the server what files are currently in its shared folder. The peer is also required to select its shared folder and enter a valid IP for the server before an attempt is made to join the network. Once it joins the network is receives from the server a list of available files on the network. It can then query a file to the server again and the server would respond with a peer who has this file. The peer does not maintain a database of remote files. Instead, it processes a list provided in an HTTP message from the directory server directly into a graphical table. This choice was only made to take advantage of the inherent functionality of the GUI. The name of the file is grabbed from the GUI and sent as in a REQUEST HTTP packet to the server which would then again respond with the IP of a peer.

In our programs we used basic algorithms to parse, and create HTTP packets. Our HttpUtil is able to move bytes of data around and create a HTTP packet up to HTTP/1.1 specifications. It is able to create request headers, response headers, and is also able to take headers and data and form one packet. In our networking side we adjusted the Timeout period for UDP packets being received by adjusting to the time it takes a packet to make a RTT. We start off by estimating a RTT, then adjust this number by timing the time it takes for us to send a packet, and receive it’s ACK. This lets us communicate with a client or server reliably even if the network is really busy and there is a high deviation in the RTT.