Design

This program consists of a single architecture that acts as both a client and a server. Its design features allow it satisfy multiple types of topographies simultaneously as was requested by the design specification. It fully complies with both a star topology and a 2D-mesh Topology. The core features behind this architecture allows a user to start any number of clients with either a specified configuration that statically creates the topology or you are also allowed to manually create it using the available commands.

To begin a user may choose to either create their own configuration file in the form of a bash or shell script which would make use of command line arguments to create the desired topology. This is also achievable directly from the command line. The necessary and available arguments are peer –l –p. The –l argument accepts an integer value and represents the listening port you would like to bind that peer to on the local machine. The p option accepts a full address in the form of:

<fully qualified domain name>:<port>

You may only specify a single listening port with the –l option but you can specify as many neighboring ports using –p as you would like. An example of this might looks something like: *peer.py -l 65355 -p localhost:65350 -p localhost:65351*. This would mean we are binding this peer to port 65355 of the local machine and setting up the peers who are bound to ports 65350 and 65351 as neighbors in the topology. The other option is simply running the *peer.py* program with no arguments in which it will bind to a randomly available port on the host machine. At this point you will have access to the peer interface of commands and you can manually use the connect and disconnect commands to create a custom topology if you desire. These commands do require you to know the fully qualified domain name and binding port of the peers you wish to connect to. This implementation is not limited to working solely on a local machine it can connect to peers who are on different machines and domains as well so long as the necessary ports and securities of that machine allow for it. There is no centralized file repository or list for that matter in this implementation as that was not a requirement of the specification. It is also possible to leave the topology at any time and perform a graceful exit that will notify all neighbor peers that the active peer is leaving the network. This command is simply *exit*.

Along with being able to manually or automatically create your topologies the program allows the user to see what all peers are neighbors to the current active peer by issuing the *neighbors* command. It will then print out a list of all neighbors for the peer who invoked the command. Anytime a peer adds a connection to another peer this list is updated for both peers. The same is true as well for the use of the disconnect command. These two commands share a very similar command structure. For the connect command

the structure is: *connect <fully_qualified_domain_name>:<port>* the same is true of disconnect with a simple adjustment to the primary command word.

Besides managing it's 'location' and 'neighbors' in a network the primary use a peer would have within this cluster is the ability to transfer files. The download command in this implementation is called *fetch*. To keep this very simplistic the user does not need to know what peer has the file, they simply need to know the file name. The appropriate usage for this command is: *fetch <filename>*. Upon invoking this command the file look will propagate across all peers and will produce a list of peers who currently have this file, the user will automatically begin downloading the file from one of these peers. The system is also robust enough to handle multiple file requests across multiple peers at the same time. It tracks these requests using a unique identifier not only for the peer, but for the file, and the request number to ensure that everything gets delivered as it should.

All interactions between the peers is handled through sockets, and for almost all transactions that occur there is both a message sent and a message received by both parties. This 'conversation' allows us to provide visual que's and details on both ends of a transaction to better identify to users the work flow of what is occurring. To further alleviate pains of using the system there exists a help command that gives a list of what commands are available. Taking this a step further every command that a user might wish to execute comes with a usage statement similar to a linux man-page to explain the appropriate usage of the command as well as any arguments that might be available to be passed to it.