Louis Le, Stephen Worley

Peer2Peer-DI
Elementary Peer 2 Peer System with distributed index.

SETUP

This project is best used with a VirtualEnv. Please use this if you wish by running in your terminal(bash):

```
source ./bin/activate
```

When you start the registration server(RS), nothing will happen but it is running; it will print out respective messages when a peer contacts it.

For the RFC server, you must provide the hostname, location of the peer's RFC files, and the port number when you run the python code. Be sure that the RFC files are in the folder before running the file. On startup, the server will create an RFC index based off of what RFC files are in its folder. Once the server is running it will prompt the user for a command with the inviting words: "Enter command: ".

After running this program to download the RFC files, you will find that in the designated folder for each peer, it will have two csv files: "t1_<folder location>_st.csv" and "t1_<folder location>_tt.csv", these two files hold time records of the download times. The t1 of the two files indicate was made for naming purposes to complete task 1. If one wants to change that, you can edit header_text variable in the main.

The st file holds the time it takes to download each file, it will be in the format: "<rfc num>,<time to download>\n"

The tt file holds the cumulative time it takes to download the files. It will be in the format: "<num of files downloaded>, <time total>\n"

To run the registration server:

```
python reg_server.py
```

To run a peer server:
    python rfc_server.py <hostname> <RFC location> <port>

     Example: python rfc_server.py localhost ./peers/P0 1234

INPUTS

Once the registration server is running, it will not take any inputs from the user, but will respond to the RFC servers.
THe user will interact only with the RFC servers and registration server will regulate on its own.

Below are the commands a user can enter:

Note: these commands below are simplified for ease of use for the user. When the RFC server contacts the RS or peer it will include more parameters such as port number or the cookie number, but since that can be a hassle to take care of we decided to make as easy as possible for quicker commands.

"Register" - the RFC server will automatically register itself with the RS by sending "Register: <port>" and the RFC server should print "Peer recieved" and "Cookie: <cookie num>"

"PQuery" - the RFC server will request from the RS a list of all active peers by sending "PQuery". It will print out the query for the user to see

"KeepAlive" - the RFC server will tell the RS to reset its TTL to 7200 by sending "KeepAlive: <cookie>". It will print "Refreshed" for confirmation

"Leave" - the RFC server will tell the RS that it wants to be turned inactive by sending "Leave: <cookie>". The RFc server will print "Left"

"RFCQuery: <hostname> <port>" - the RFC server will reach out to another peer with the specified hostname and port to request its RFC index, it will then merge it with its own. The peer that is giving the index will print "RFC Query Request" and then "RFC Query Sent"

"GetRFC: <hostname> <port> <rfc_num>" - the RFC server will contact the given peer to get its RFC file. It will add the file to its folder and add to its own RFC index.

"Search: <rfc_num>" - this command allows for the user to have the RFC server run through the above commands to get the RFC file. It will perform a PQuery, then RFCQuery until it finds the right peer and use GetRFC.

"Search: <start>-<end>" - this commands lets you search for a whole array of RFC files from a start number to an ending number inclusively.

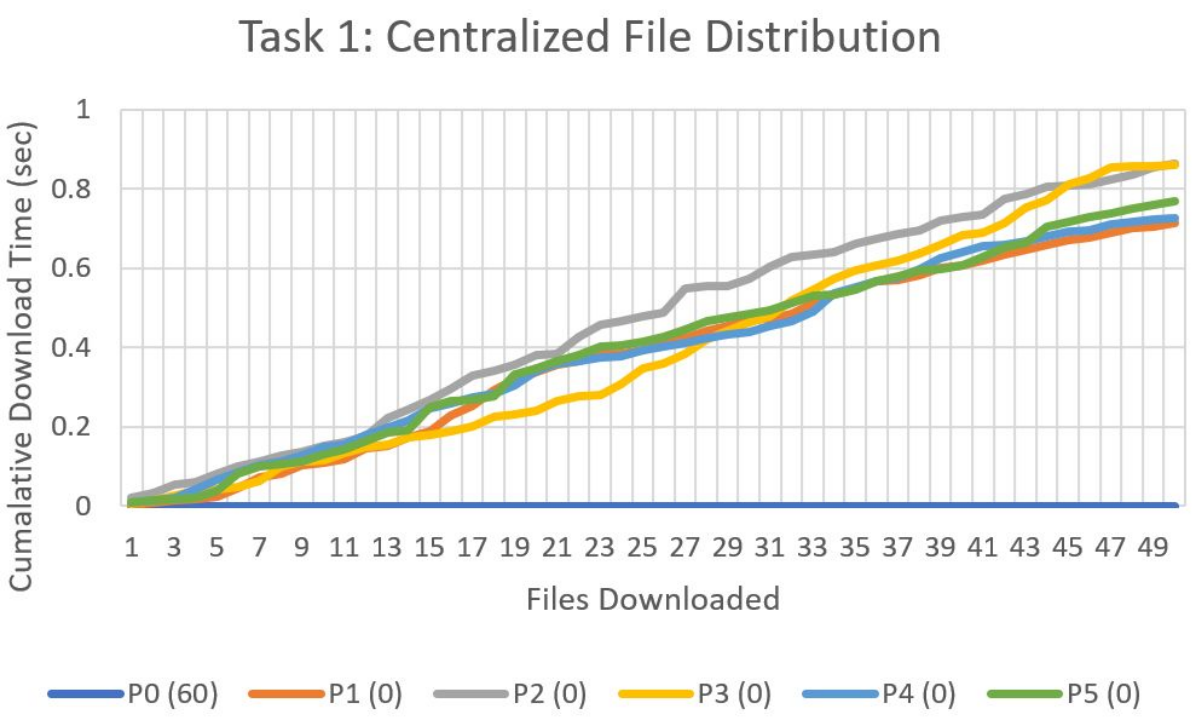
Analysis

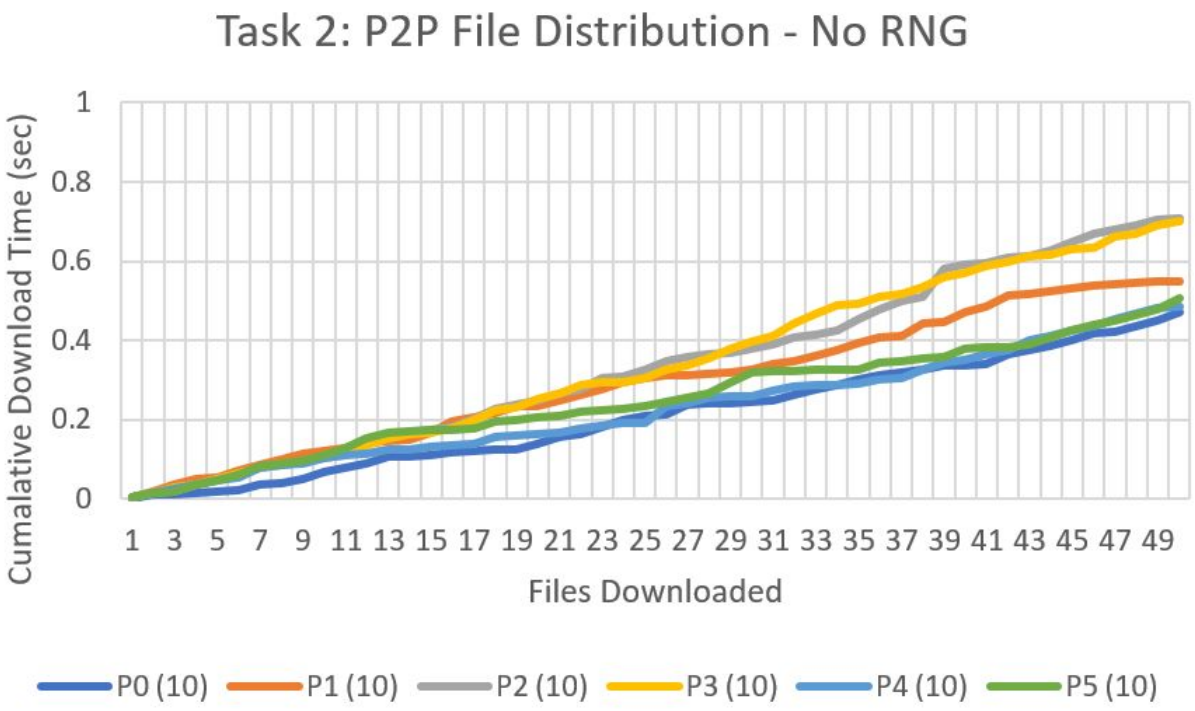
You will find that we have a folder called "times", it contains all the time records we had for both tasks. The naming convention we went with was: "<task>_<peer num>_<single or total time> "

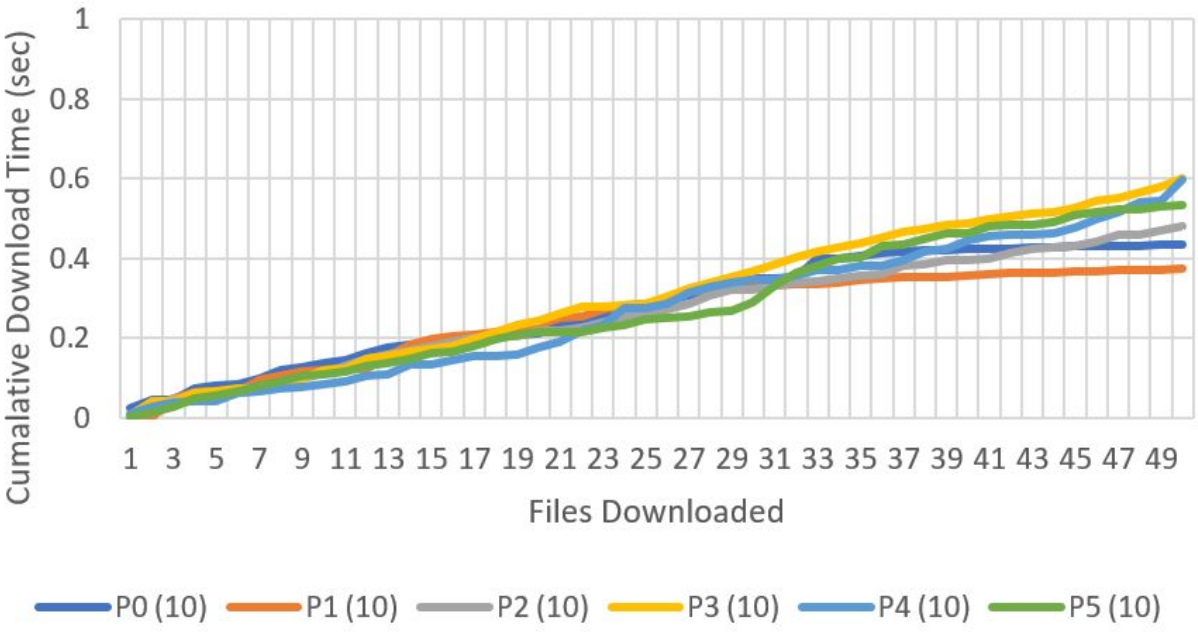
So for example, for task 2, peer 3, single times is: t2_p3_st.cvs

We found through performing Task 1 and Task 2, the times in Task 2 are all around better for every peer except for peer 0. In the first task, peer 0 had all the RFC files already so the download time for peer 0 was 0. In the second task, because all peers had 10 of the 60 files, they each download as many files but at the same time they had to find the files in each of the other peers rather than in task 1 where they were like in P0. But because of the less amount of files one must download,the times were smaller.

For task 2, we actually have two graphs because one represents the worst case scenario and the other, the best case scenario. The differences are that when choosing which files to retrieve first, the worst case is going in numerical order. If all the peers do this, then it can cause a lot of congestion. The best case is to have an RNG set so each peer will choose a random file to retrieve. As you can see the case with the RNG allowed for lower times and it seems like the download times are more closer to each other and not as volatile.


Task 1: Centralized File Distribution
Cumalative Download Time (sec)
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Files Downloaded
P0 (60)
P1 (0)
P2 (0)
P3 (0)
P4 (0)
P5 (0)


Task 2: P2P File Distribution - No RNG
Cumalative Download Time (sec)
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Files Downloaded
P0 (10)
P1 (10)
P2 (10)
P3 (10)
P4 (10)
P5 (10)


Task 2: P2P File Distribution - With RNG
Cumalative Download Time (sec)
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Files Downloaded
P0 (10)
P1 (10)
P2 (10)
P3 (10)
P4 (10)
P5 (10)