CSC/ECE 573 Internet Protocols Project 1 Report:

Team Members:

1. Vivek Nadimpalli (vvnadimp)
2. Sarath Chandra Chaparla (schapar)

Message Formats:

Message Header Size is fixed to 100 bytes. This can be modified if needed. Spaces are filled at the end of header if it's length is less than 100 bytes. Message body follows the header.

Messages from Peer to RS:

Format:

<message name> <sp> P2P-DI/1.0 <cr><lf>

Host: <sp> <hostname>:<rfcserver port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> <Length of the message body excluding the header> <cr><lf>

<sp>

<message body>

Message body format:

For Register message:

<cookie> <cr><lf>

<hostname of the rfcserver> <cr> <lf>

<rfc server port>

For PQuery message

<cookie> <cr><lf>

Example Messages:

"Register P2P-DI/1.0

Host: vivekMac:65400

OS: linux2

Content-Length: 23

-1

vivekMac

65400”

“PQuery P2P-DI/1.0

Host: vievkMac:65400

OS: linux2

Content-Length: 1

6”

“KeepAlive P2P-DI/1.0

Host: vivekLinux:65400

OS: linux2

Content-Length: 1

6”

“Leave P2P-DI/1.0

Host: vivekLinux:65400

OS: linux2

Content-Length: 1

6”

Message Responses from RS to Peer:

Message Format:

Successful Response:

OK <sp> <message name> <sp> P2P-DI/1.0 <cr><lf>

Host: <sp> <RS hostname>:<RS port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> <Length of the message body excluding the header> <cr><lf>

<sp>

<message body>

Error Response:

Error <sp> <message name> <sp> P2P-DI/1.0 <cr><lf>

Host: <sp> <RS hostname>:<RS port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> <Length of the message body excluding the header> <cr><lf>

<sp>

<message body>

Message body format:

For Register Response message:

<cookie> <cr><lf>

For PQuery Response message:

<peer hostname>:<peer rfcserverport> <cr><lf>

<peer hostname>:<peer rfcserverport> <cr><lf>

….

If there are no activePeers, PQuery contains content-length as 0. This would never be the case as the peer querying the RS would have to be an activePeer itself and that gets returned in activePeerList. However, peer ignores entries in activePeerList related to self.

Example Message Response:

"OK Register P2P-DI/1.0

Host: vivekMac:65423

OS: linux2

Content-Length: 1

6"

"OK PQuery P2P-DI/1.0

Host: vivekLinux:65423

OS: linux2

Content-Length: 126

vivekLinux:65405

vivekLinux:65401

VivekLinux:65402

vivekLinux:65403

vivekLinux:65404

vivekLinux:65400 “

“OK Leave P2P-DI/1.0

Host: vivekLinux:65423

OS: linux2

Content-Length: 1

2”

“OK KeepAlive P2P-DI/1.0

Host: vivekLinux:65423

OS: linux2

Content-Length: 1

2”

Messages from Peer to Peer:

Message Format:

GET RFC-Index:

GET <sp> RFC-Index <sp> P2P-DI/1.0 <cr><lf>

Host: <sp> <hostname>:<rfcserver port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> 0 <cr><lf>

GET RFC:

GET <sp> RFC <sp> <RFCNumber> <sp> P2P-DI/1.0 <cr><lf>

Host: <sp> <hostname>:<rfcserver port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> 0 <cr><lf>

Example Messages:

"GET RFC-Index P2P-DI/1.0

Host: VivekLinux:65400

OS: linux2

Content-Length: 0"

"GET RFC 8243 P2P-DI/1.0

Host: vivekLinux:65400

OS: linux2

Content-Length: 0"

Message Responses from Peer to Peer:

Message Format:

RFC-Index Response Format:

"OK <sp> RFC-Index <sp> P2P-DI/1.0

Host: <sp> <hostname>:<rfcserver port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> <Length of the message body excluding the header> <cr><lf>

<sp>

<peer hostname>: <peer rfcserver port>: <rfc number>

<peer hostname>: <peer rfcserver port>: <rfc number>

….

"

RFC response Format:

"OK <sp> RFC <sp> <rfcnumber> P2P-DI/1.0

Host: <sp> <hostname>:<rfcserver port> <cr><lf>

OS: <sp> <OS name> <cr><lf>

Content-Length: <sp> <Length of the message body excluding the header> <cr><lf>

<sp>

<entire RFC content goes here>"

Example Messages:

"OK RFC-Index P2P-DI/1.0

Host: vivekLinux:65405

OS: linux2

Content-Length: 520

<rfc index list goes here>

"OK RFC 8243 P2P-DI/1.0

Host: vivekLinux:65405

OS: linux2

Content-Length: 68547

<entire rfc content goes here>"

Implementation:

Peer:

RFCServer: runs in a separate thread and continues to run all the time. Every time a peer connects to the server, a new thread is spawned, request served and the connection closed and thread exits. There can be multiple peers making connections simultaneously and thus multiple threads handling peer requests would be running simultaneously.

RFC Client: runs in a separate thread and does the following:

Initializes local RFCIndex from the files stored in its folder. Registers with the RS. In the register message, sends a cookie value -1. On getting a new cookie assigned in the response, it'd use that for subsequent communication with RS.

Then it queries for active peers with PQuery message. The response includes itself and it skips that while requesting for RFC-Indexes. For each peer in the list, sends Get RFC-Index message and on getting response, merges with its own RFCIndex, and checks if the requiredRfcs are in the RFCIndex received and if so, sends Get RFC messages to each peer to retrieve the RFCs one by one. As it downloads each RFC, it removes that from the requiredRFC list maintained internally. Once it downloads all the needed ones, client thread returns.

If there are no active peers and it still needs to download some RFCs, it'd wait for 2 seconds and queries the RS again for active peers. It does this in a loop. The wait time can be changed.

KeepAlive:

A separate thread for keepalive is also spawned at the time of initialization which wakes up every 10 seconds (time can be modified as needed), sends a keepalive message to RS, updates the TTL of

RFCIndexes obtained from remote peers and the also of activePeers. Currently coded to exit the peer process after 2 such iterations (20 seconds) after sending “Leave” message to RS as the test for task1 or task2 would complete within that time. This can be modified and rfc server can continue to run by commenting out the os._exit() function in peer.py.

Code is common for task1, task2 and also handles demo scenario without any changes to the code.

For running demo, a keyword “demo” needs to be passed to peer.py while invoking the program.

RS:

Runs in main thread. For every connection from a peer it spawns a new thread. A peer is identified by its hostname and its rfcserver port combination. If a peer sends Register message, it checks internally if it was already registered before. If so, it just returns the cookie already assigned to it in the response. So, when a client process exits and restarts with the same port and host name, if it still remained in the RS’ list of registered peers, it gets the old cookie, and thus wouldn’t create a duplicate entry. If not registered, it assigns a new cookie and sends it in the response.

When it receives any message from a peer when its isActive status is false, it returns an error.

Test Results:

Task1 (Centralized):

This tests the centralized server scenario where Peer 0 (P0) contains all the RFCs and the remaining 5 peers download 50 RFCs from P0. In this scenario, P0 will not download anything as it has all the RFCs. So there is no graph for download times for P0 for task1.

In parallel, there is a keepalive thread running which sends KeepAlive messages every 10 seconds (can be changed to any value) to RS.

It has been coded for convenience of testing the program to “Leave” the network after 2 KeepAlive messages are sent and then peer exits.

Task2(P2P):

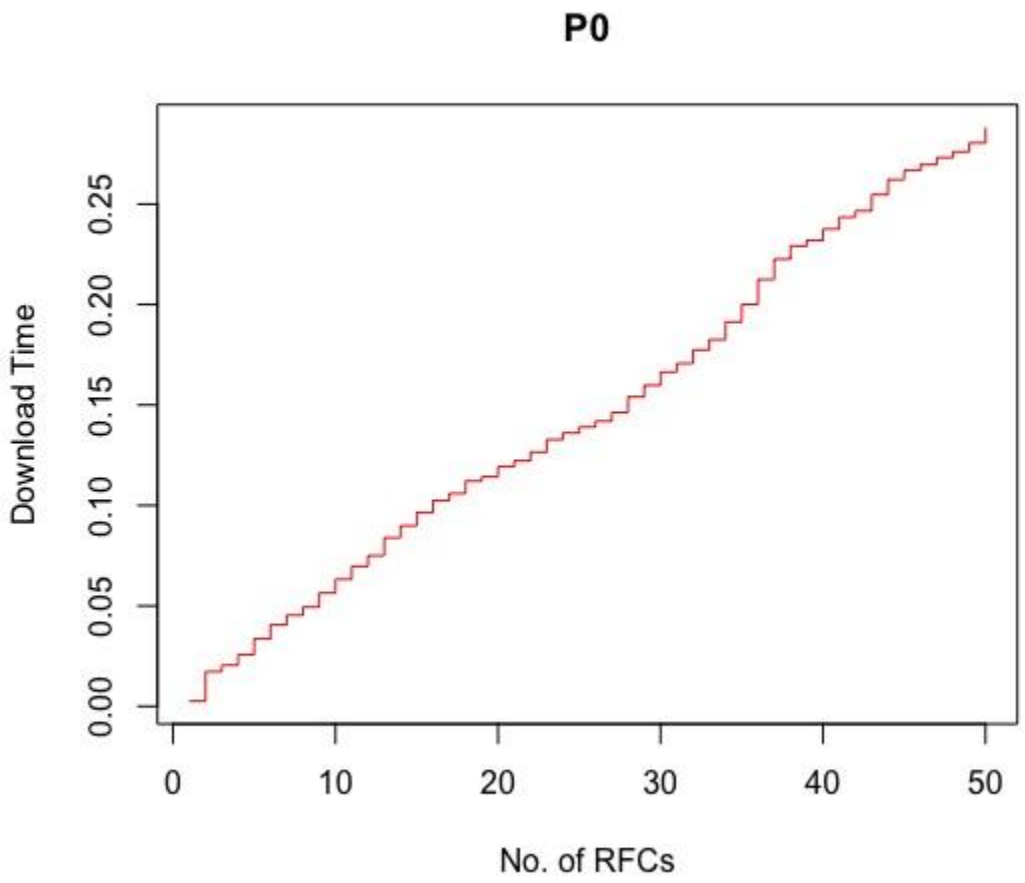
This tests the P2P scenario where each Peer contains 10 RFCs and downloads 50 RFCs from the other 5 peers.

Task Demo:

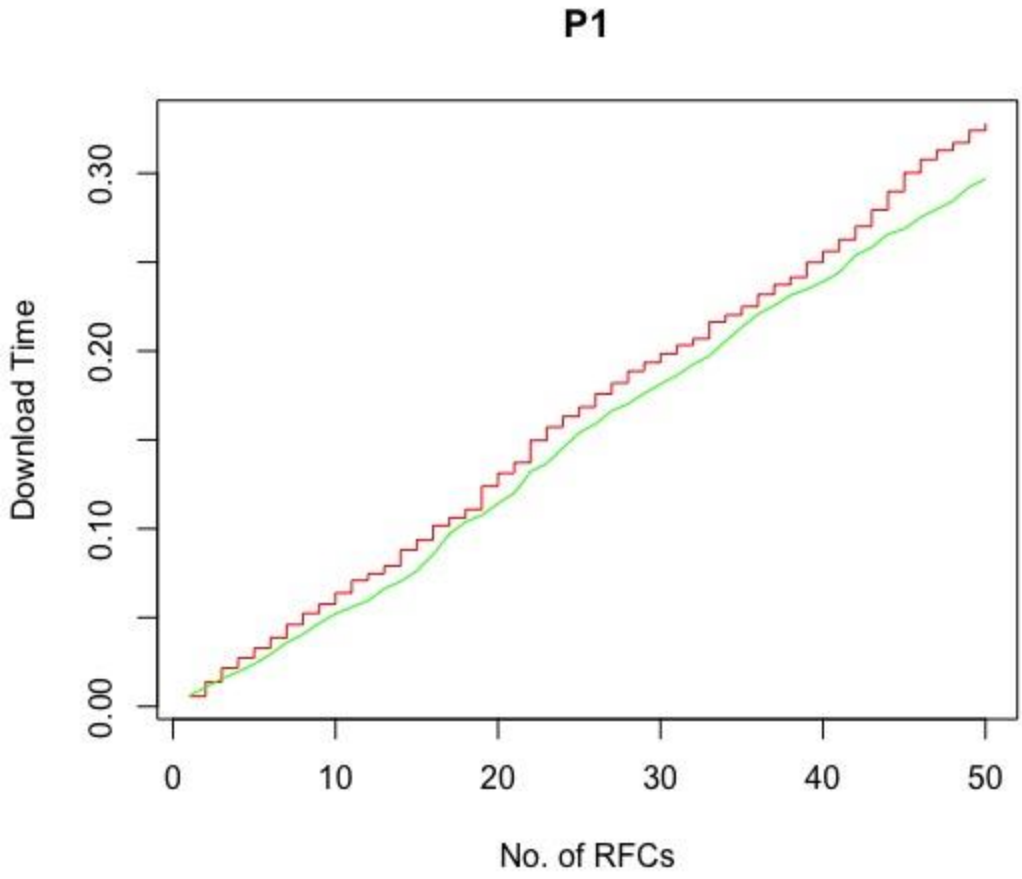
The demo task runs 2 peers with P0 containing 2 RFCs. P1 downloads one RFC from P0 and then P0 leaves the network and exits to create a scenario where P1 fails to make connection to the peer for download of 2nd RFC and requests RS for active peers again. Since there are no more active peers it waits for 2 seconds and queries RS again for activePeers in a loop. In parallel, there is a keepalive thread running which sends KeepAlive messages every 10 seconds (can be changed to any value) to RS. It has been coded for convenience of testing the program to leave the network after 2 KeepAlive messages are sent and then peer exits.

Download-Time graphs:

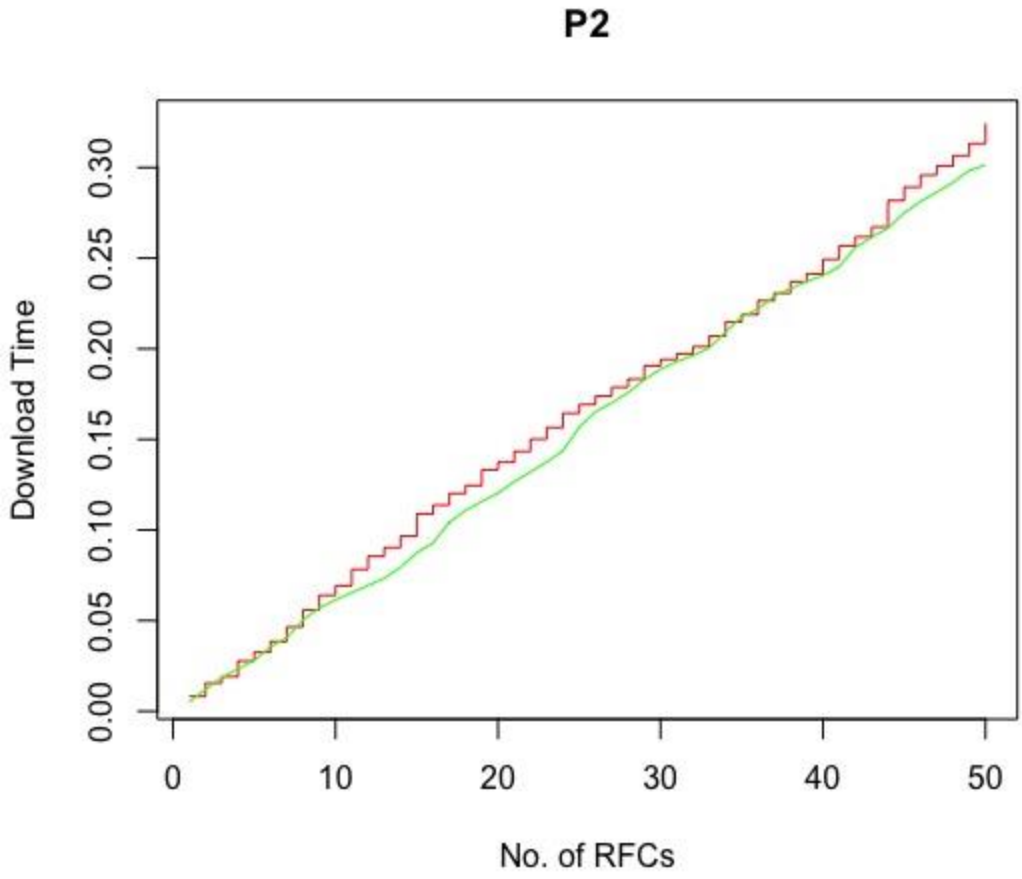
Find below graphs for each Peer with download times. Each peer graph contains an overlaid graph of both tasks for comparison purposes instead of on two separate graphs. The graph is drawn with No. of RFCs on the x-axis and the cumulative time it took on the y-axis.



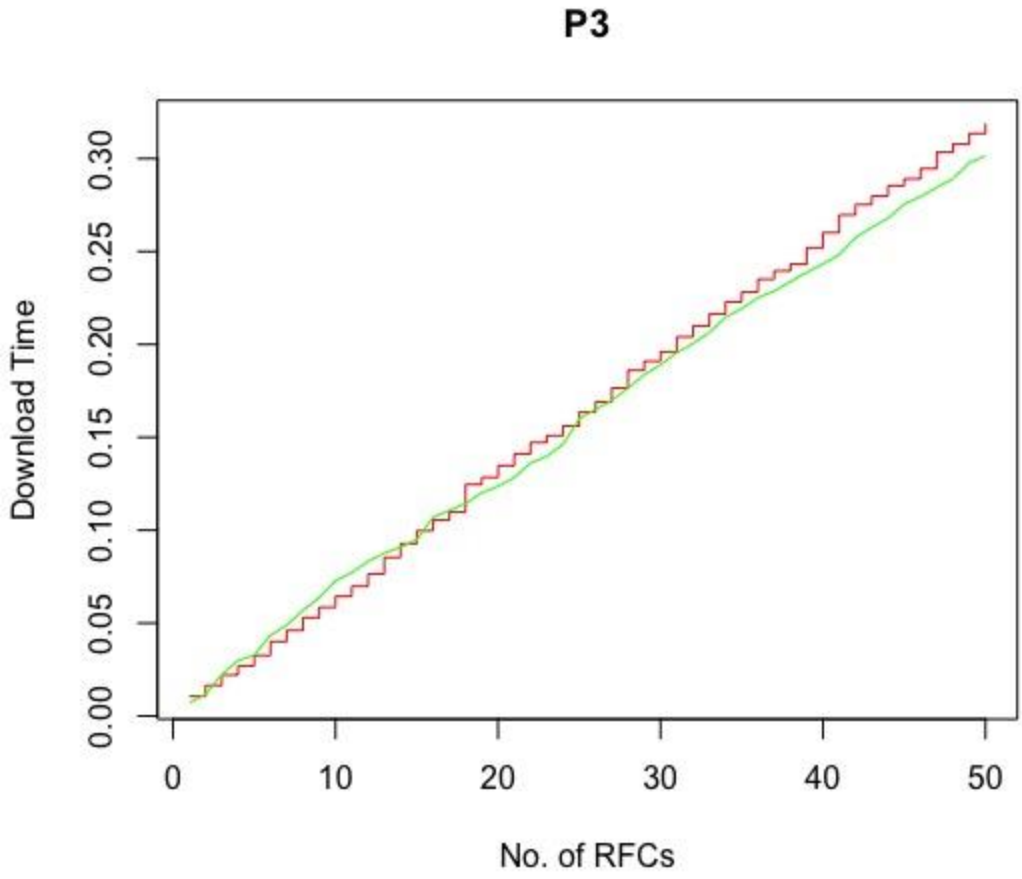
P0 downloads RFCs only in Task2 (P2P)



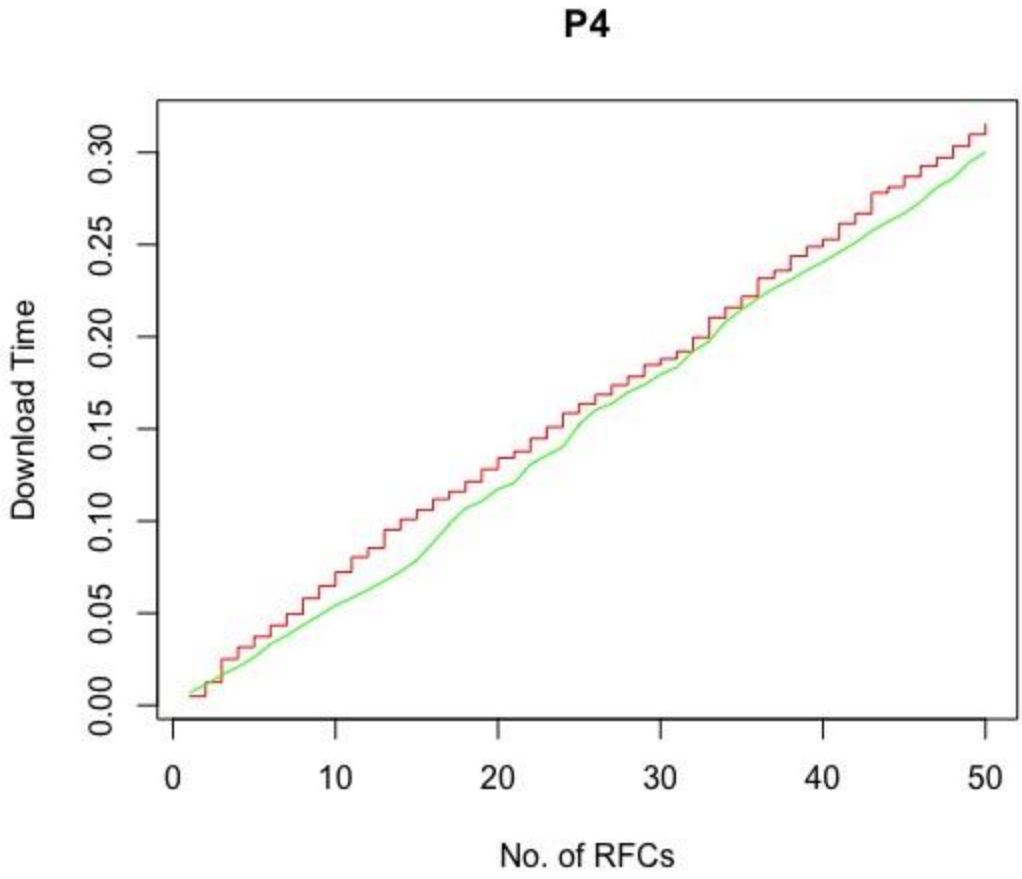
Legend: Green Represents Centralized configuration (Task 1) and Red represents P2P (Task2)



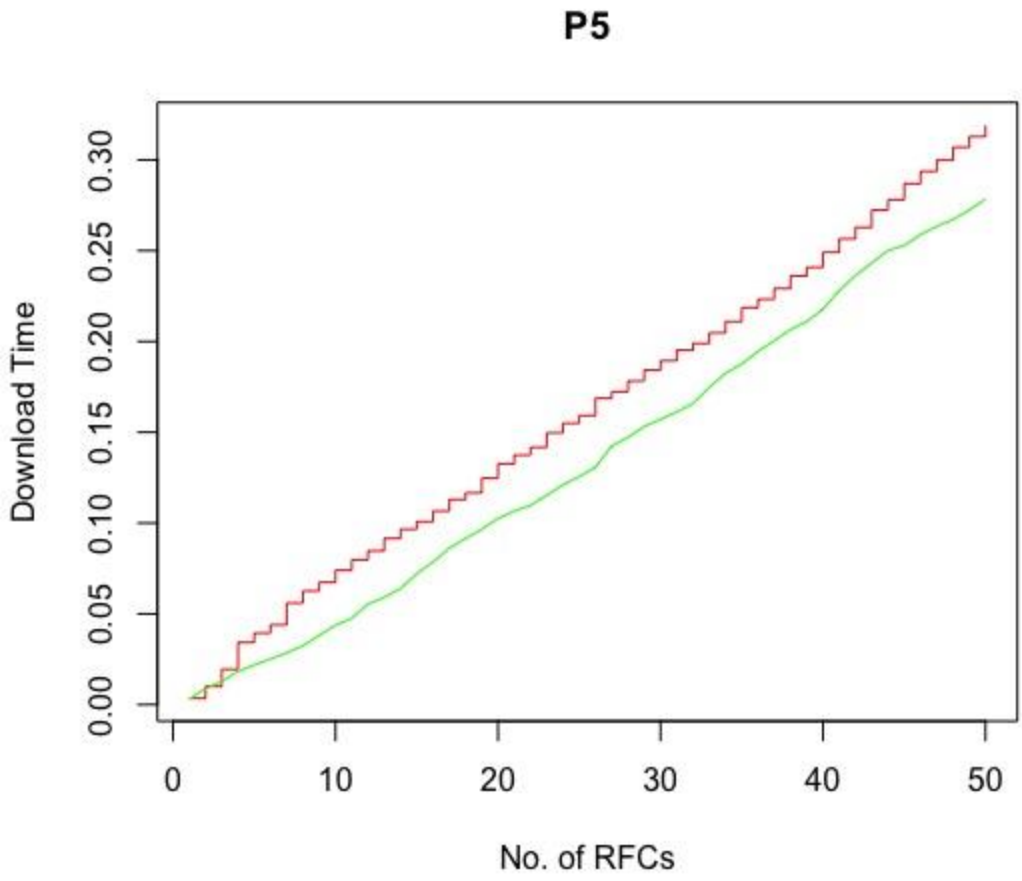
Legend: Green Represents Centralized configuration (Task 1) and Red represents P2P (Task2)



Legend: Green Represents Centralized configuration (Task 1) and Red represents P2P (Task2)



Legend: Green Represents Centralized configuration (Task 1) and Red represents P2P (Task2)



Legend: Green Represents Centralized configuration (Task 1) and Red represents P2P (Task2)

Analysis:

As can be seen, it has been observed that in the experiment with 6 peers, the centralized configuration took slightly less download times than P2P configuration for most part. On multiple runs, based on the load on the machine, at times, we observed that the P2P download times are slightly lower (better) than that of the centralized scenario.

The performance difference has been very marginal and has been so due to very small number of peers being used.

In a centralized configuration, if there are N peers and M files to be downloaded, there would be about (N-1)*M overall downloads from the centralized peer, with about (N-1) connections concurrently running for most part. Depending on the load on the system, the download time would worsen if the number of peers is large and the files are many. Download times would depend on the load on the

system serving the download, load on the peer downloading, and the network speed/latencies between peers.

In a P2P setup, for N peers and a total of M files and each peer having X files and need to download (M-X) files, each peer will serve (N-1)*X file downloads which would be much less than that of a centralized config. And also, it would make (N-1)*(M-X) downloads from other peers sequentially.

As the number of peers and files increase, files are large and are randomly distributed across peers, with each peer randomly downloading the files, which would be the case in the real world, the performance of a P2P scales much better. In such cases, centralized server gets overloaded and performance degrades when all the peers try to reach out and download all the files. Of course, this can be mitigated to some extent by having a server farm with load balancers but then maintaining such a server farm is expensive.

Overall, P2P scales much better compared to centralized servers in terms of download performance.