CSC/ECE 573 – Internet Protocols

Project #1- Report

Team Members:

Nikhila Nathani (nnathan)

Rithish Koneru (rkoneru)

Implementation:

Registration Server

The server is multi-threaded, in order to handle requests simultaneously from many clients. Whenever a client requests a connection, it opens a new thread, and assigns it to the particular client, while it waits on the port 65423.

The server allows the client to register, and checks if it is previously registered, and if not, it assigns a unique cookie to it, and also responds to leave-message, or keepalive message or to any download query request, that is PQuery in this case. All the requests are given a response, if the peer is in active status, otherwise the server returns an error.

Each peer is identified through the peer list data structure, maintained by the server. It checks with the hostname and the port number to which the RFC server of the peer is listening to, and if it finds the peer to be a match, it retrieves the old cookie, otherwise it creates a new cookie and sends it as a response.

Peer

Since the peer can request other peers, and can also respond to requests it gets, it should have an RFC client as well as an RFC server. Apart from this, the peer should be able to handle requests, and should be able to keep the connection alive, for the time it is performing the task that is requested.

RFC Client:

It takes up an individual thread, and maintains the local RFC index, which has the information about RFCs which are only on that peer, and when it connects with another peer, it gets the RFC index(Get RFC_index) of that peer, and merges it with the local copy, and also the index of where it actually can be retrieved from, in this way, it can redirect any other peer asking for those RFCs, to the original peer that holds them.

With this, the peer can update the requiredRFCs if they are in the merged RFCs. Once all the downloads are done, the thread returns. If the client needs to do more downloads, it searches for an active peer, otherwise it queries the server, through PQuery.

RFC Server:

It does the same as the server, which runs on an individual thread, and continues to run, so it can accommodate any peer requests. Since it acts as a server for multiple peers, it

handles the request through multiple threads. Thus it eliminates the need to connect to a registration server for each request, instead, peers can share and transfer data between each other.

Message Format:

The customized message format used in this code, will be in a high level readable format. The request and response messages are configured in a way that, they are the same for both the registration server and the peers, since each peer also has an RFC server, that performs the server actions, in a peer-to-peer system. So the ResponseMessage and RequestMessage are created as a separate file, and are called when required.

RequestMessage Format:

private Mtype mtype;
private Integer size;
private String RFCServerSocketHostname;
private Integer RFCServerSocketPortNumber;
private String content;
private Integer fileNumber;

Mtype:
It is the message type categorized as: Register, Leave, PQuery, KeepAlive, RFCQuery, and GetRFC. They are classified based on the requirements of the project tasks

RFCServerSocketHostname:
It serves as the index for the peer or server on the other side, when we request. It is used to maintain the activePeerList and the PeerList data structures

Content:
It is the additional message that the source intends to send to the destination. It can be of general sentences for reference

fileNumber:
This is used by the GetRFC to request that particular one, by specifying the file number accordingly

ResponseMessage Format:

```
private MessageStatus mStatus;
private Integer size;
private String fromAddress;
private String toAddress;
private String content;
private HashSet<String> activePeers;
private HashMap<Integer, RFCObject> rfcIndex;
private File rfcDocument;
```

MessageStatus:
It stores the status code of the response message. They can be OK, ERROR, UNAUTHORIZED, DBG

fromAddress:
It is the address of the peer from which the request has come, and the response needs to be sent to

toAddress:
It is the address of itself, indicating that the client on the other server should send any requests to this toAddress

Content and activePeers is the same as the request message

rfcIndex:
It contains the RFC number, and the hostname, along with the TTL value of each peer that has accessed it. It might have multiple records for each peer

Examples:

Peer to RS request message format

RequestMessage [
Messsage Type = Register
, size = null
, RFCServerSocketHostname = 0.0.0.0
, content = null
, Requested file Number = null
]

RS to peer response message format

return Message is=ResponseMessage [
mStatus = OK
, size = null
, fromAddress = /127.0.0.1
, toAddress = /127.0.0.1:49728
, content = The peer is now registered with cookie value:1
, activePeers = null
, rfcIndex = null
, rfcDocument = null
]

Peer to Peer request message format

RequestMessage [
Messsage Type = GetRFC
, size = null
, RFCServerSocketHostname = null
, content = null
, Requested file Number = 8060
]

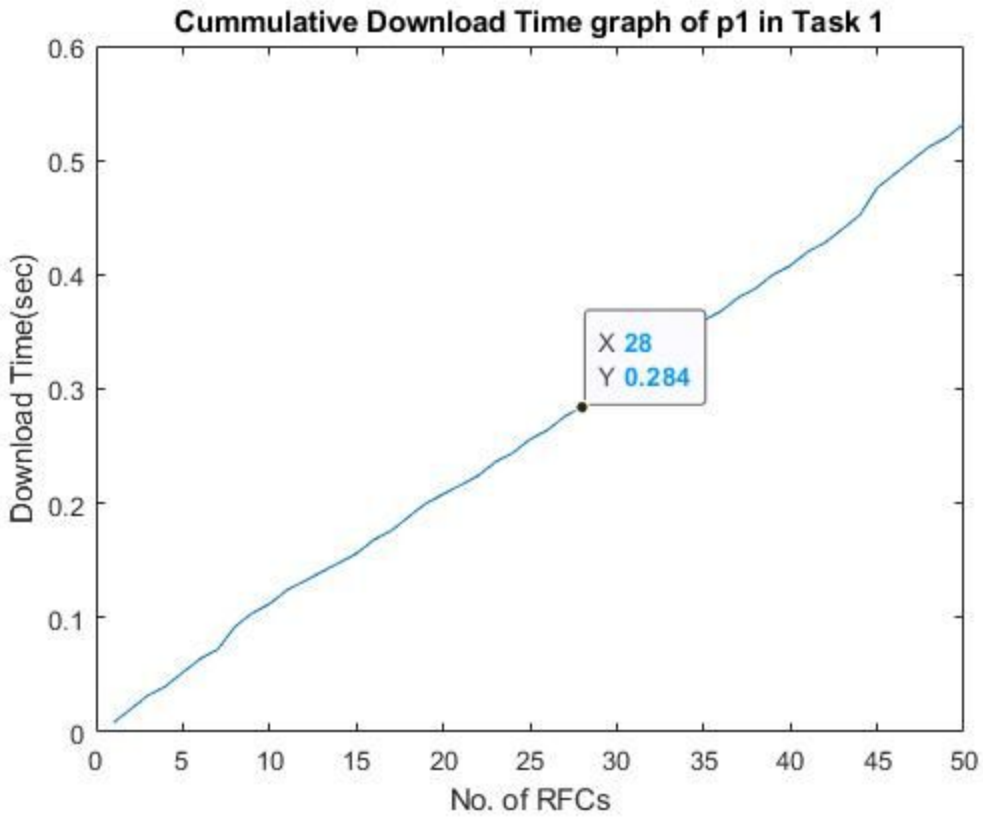
Peer to Peer response message format

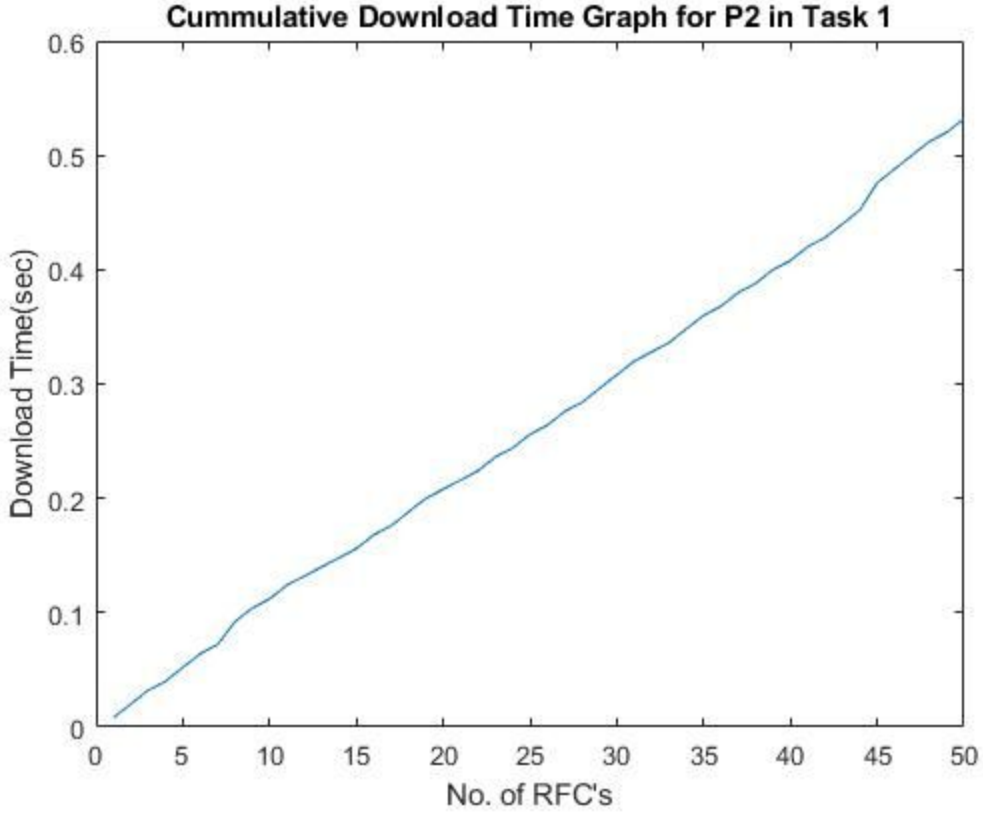
ResponseMessage [
mStatus = OK
, size = null
, fromAddress = /127.0.0.1
, toAddress = /127.0.0.1:50552
, content = null
, activePeers = null
, rfcIndex = null
, rfcDocument = P0/rfc8059.txt
]

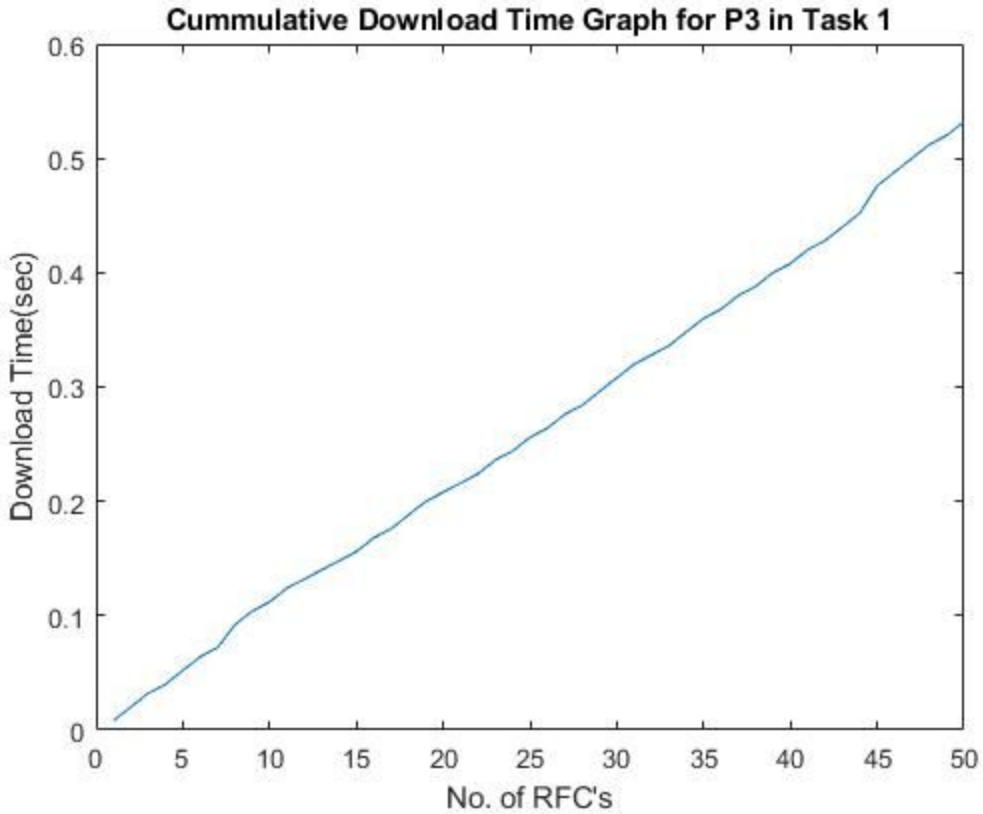
Download Time Graphs:

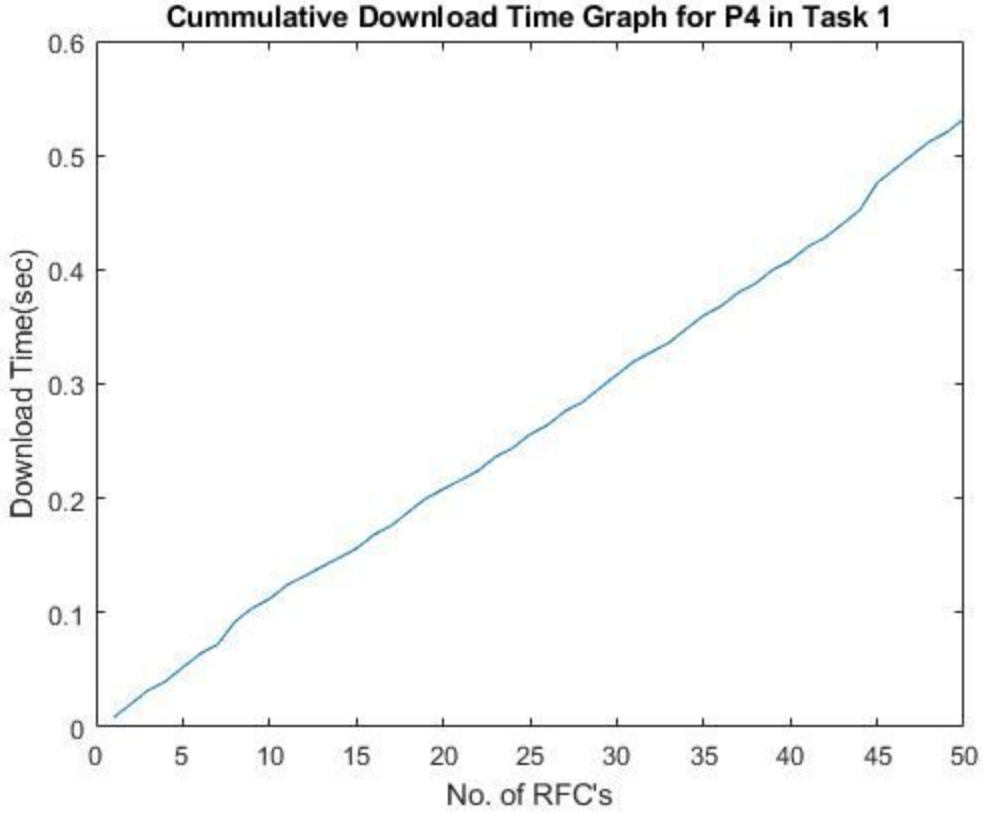
Task1:

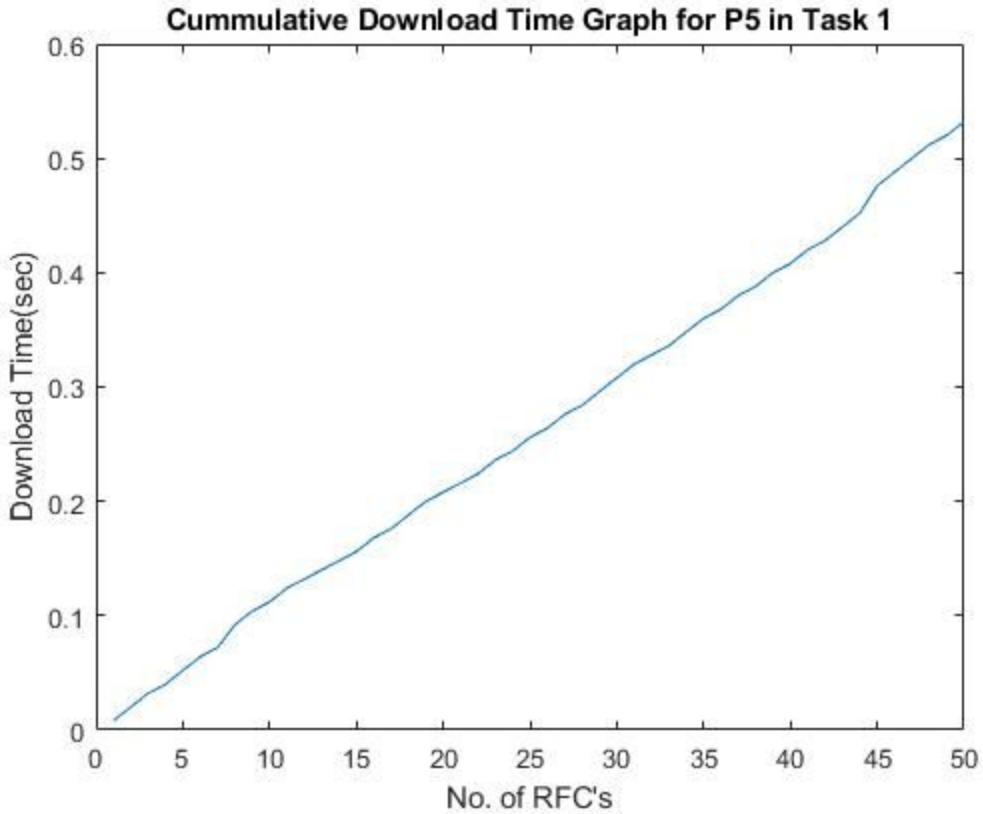
It is the centralized server case, where the P0 becomes the server which contains all the 60 RFCs and the other 5 clients p1 to p5, don't have any RFCs. The other 5 clients query for the required RFCs and obtain them from the P0, using the RFC Index. Since the P0 doesn't download anything, it doesn't have a download time curve in task1. And since all other peers download the same number of RFC's from the client, they have an almost similar cumulative download curve.




Cummulative Download Time Graph for P2 in Task 1
Download Time(sec)
No. of RFC's
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30
35
40
45
50

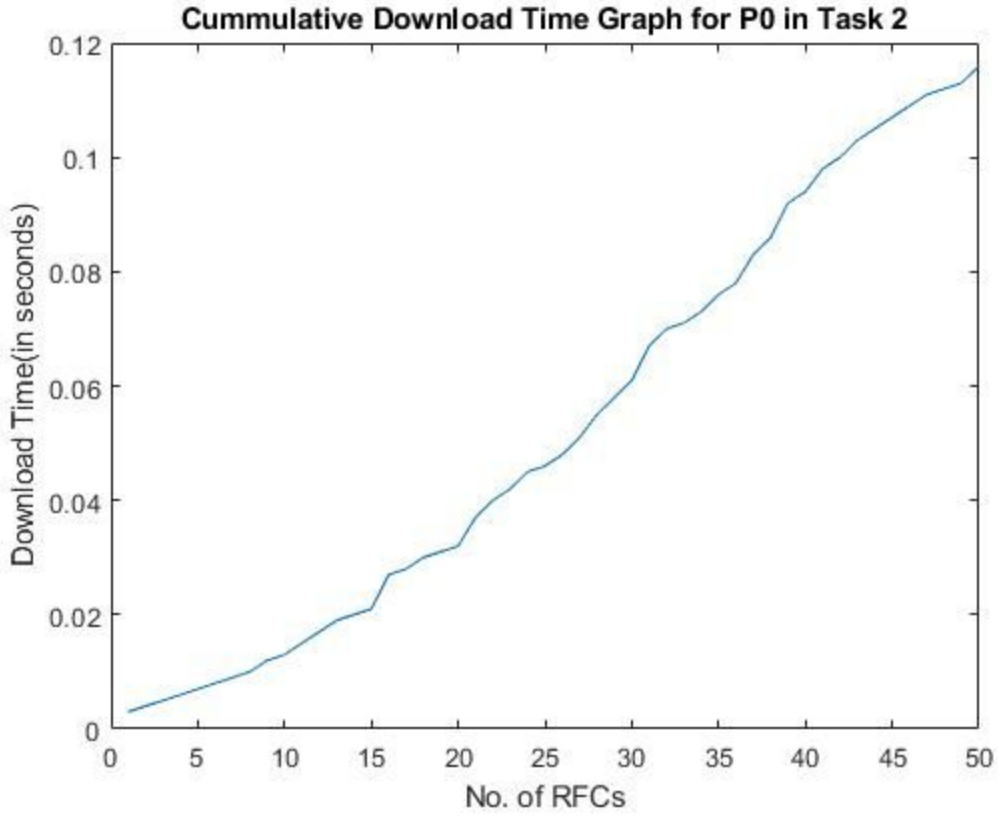

Cummulative Download Time Graph for P3 in Task 1
Download Time(sec)
No. of RFC's
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30
35
40
45
50

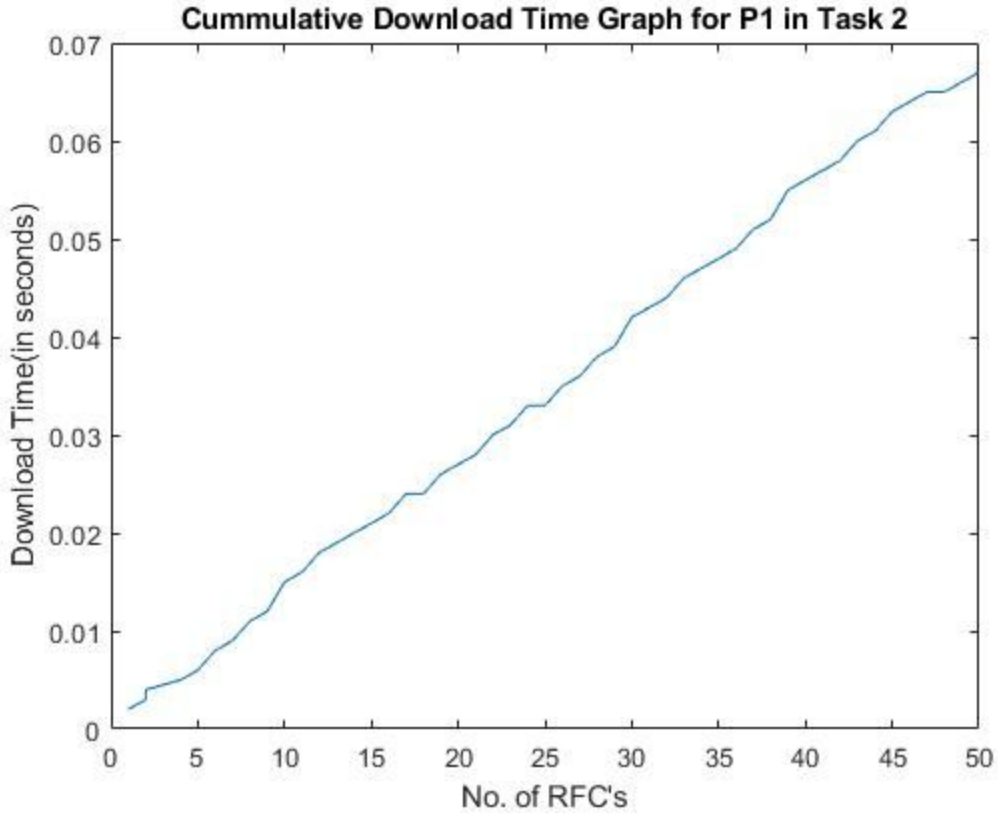

Cummulative Download Time Graph for P4 in Task 1
Download Time(sec)
No. of RFC's
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30
35
40
45
50

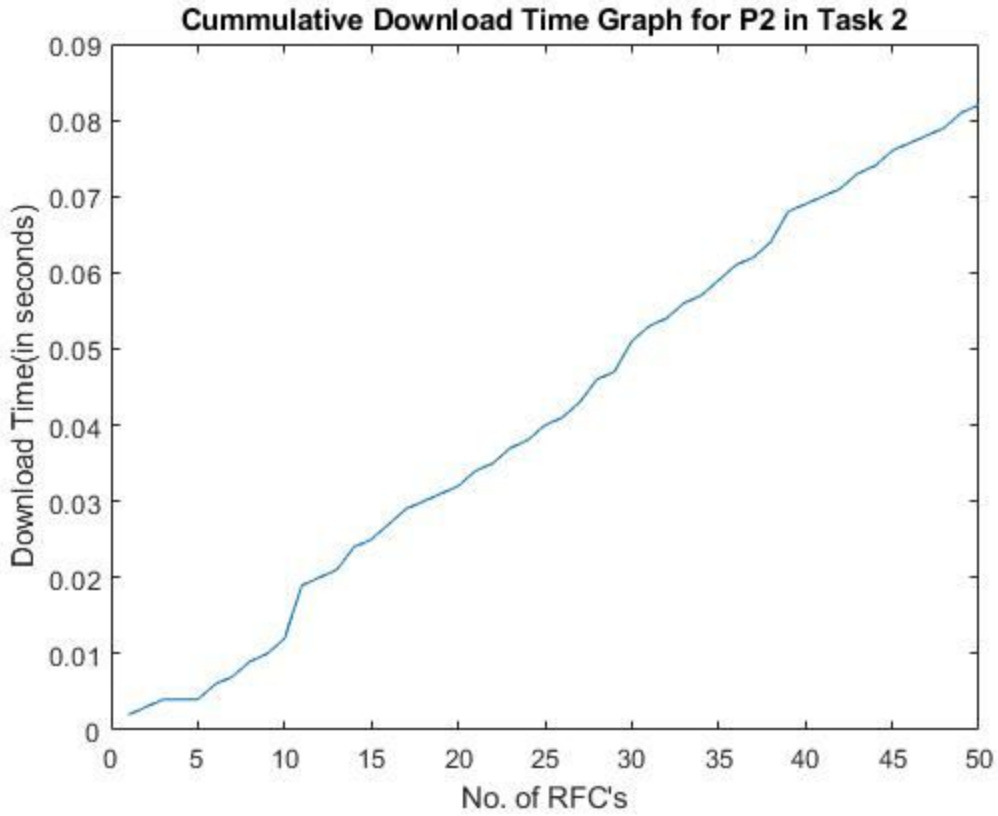

Cummulative Download Time Graph for P5 in Task 1
Download Time(sec)
No. of RFC's
0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
25
30
35
40
45
50

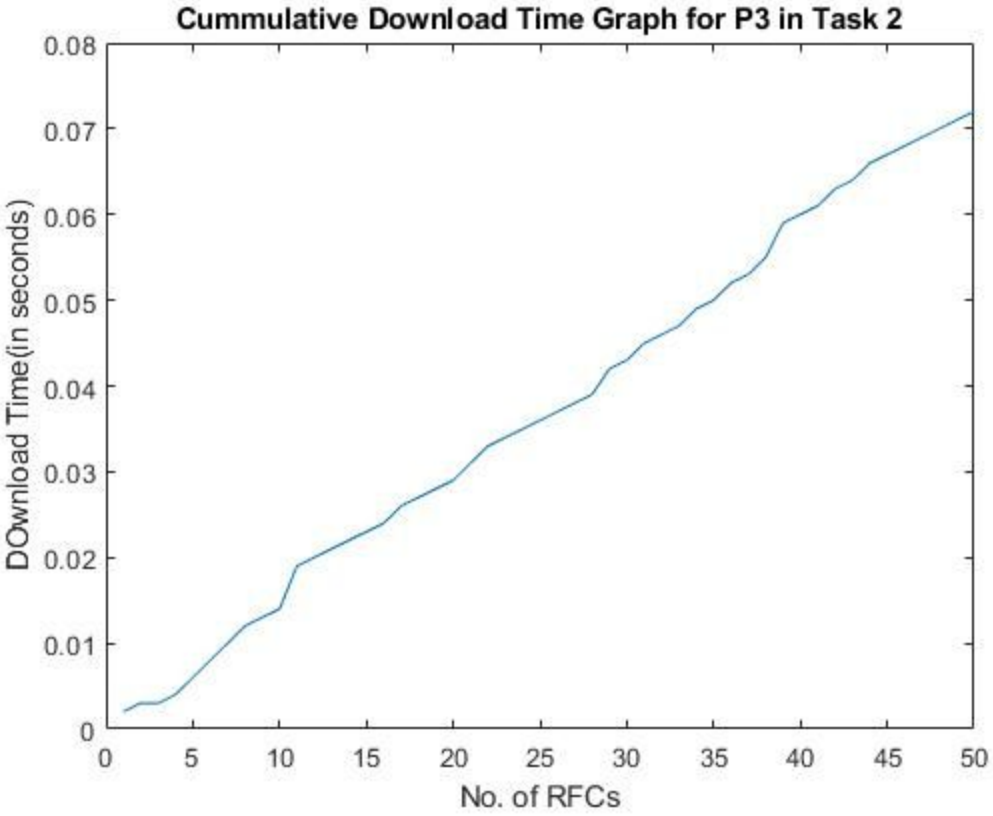
Task2:

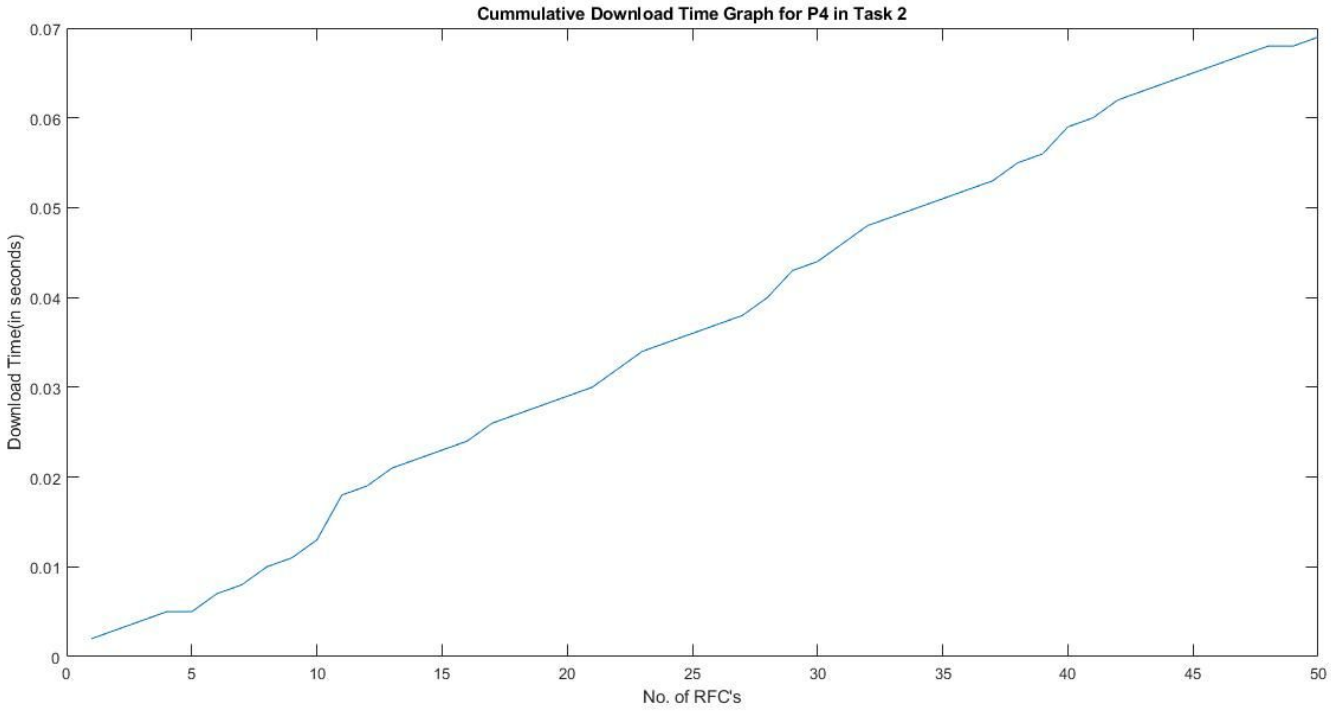
In this task, each of the 6 peers, download 10 RFCs and then retrieve the other RFCs from the containing Peer, when required. So each time it does that, the RFC Index is updated with the new indexes, and eventually, every peer index file, consists of all the RFC files.

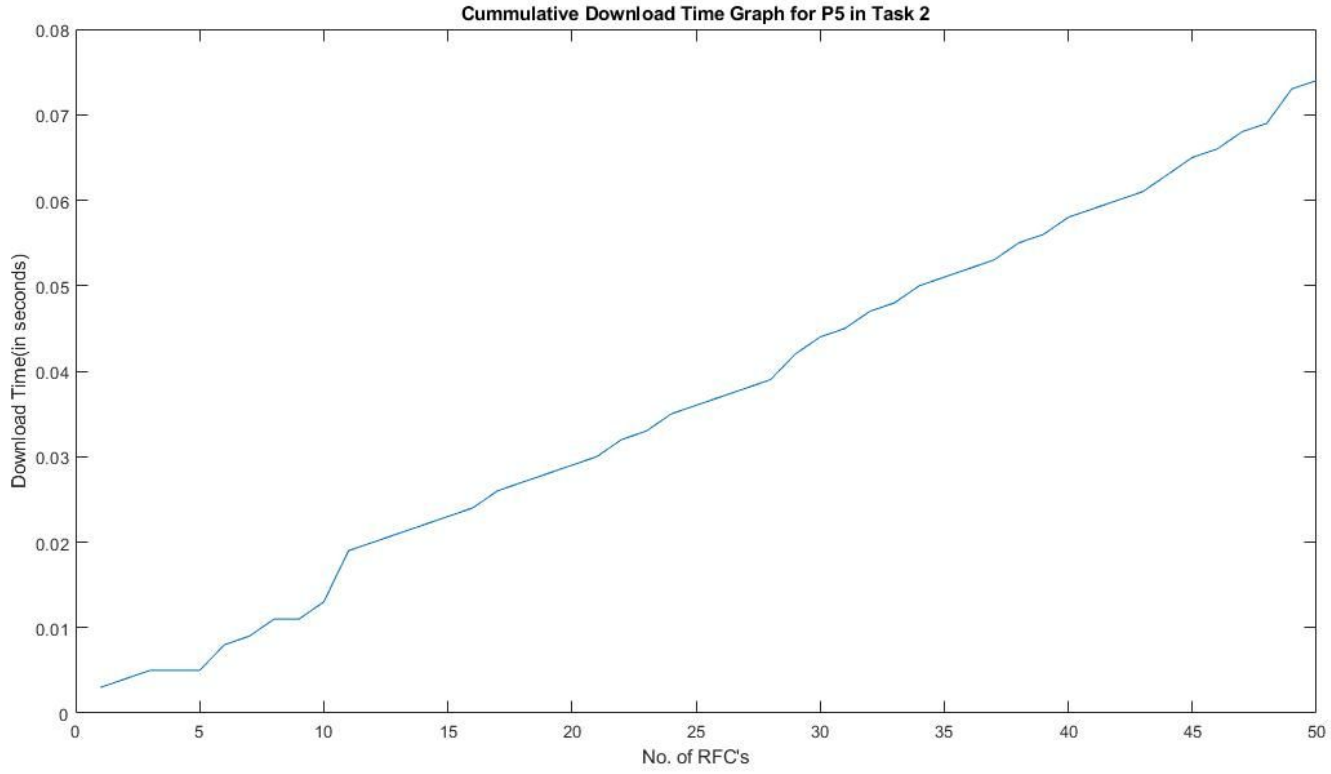



Cummulative Download Time Graph for P1 in Task 2
Download Time(in seconds)
No. of RFC's
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0
5
10
15
20
25
30
35
40
45
50


Cummulative Download Time Graph for P2 in Task 2
Download Time(in seconds)
No. of RFC's
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0
5
10
15
20
25
30
35
40
45
50


Cummulative Download Time Graph for P3 in Task 2
DOwnload Time(in seconds)
No. of RFCs
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
5
10
15
20
25
30
35
40
45
50


Cummulative Download Time Graph for P4 in Task 2
Download Time(in seconds)
No. of RFC's
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0
5
10
15
20
25
30
35
40
45
50


Cummulative Download Time Graph for P5 in Task 2
Download Time(in seconds)
No. of RFC's
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
5
10
15
20
25
30
35
40
45
50

Analysis:

It can be clearly seen that in task1 where a peer is centralized, the download time is comparatively less than the download times, in the peer-to-peer configuration. When the peers request at the same time, and during multiple runs, the P2P download times can waver towards download time being less which indicates that it's better in comparison to the centralized case.
The performance is almost similar, in both the cases, due to the small number of peers being used.

An example to show that, is that, if in a centralized case, there are k number of peers, and one peer has all the data (n files), then other peers(k-1) will be requesting the downloads. That is (k-1)*n number of downloads, which mostly depends on the load of the RFC server.

In a peer-to-peer system, k number of peers, and n-number of total files, each peer has i number of files, and need to download the remaining, that is (n-i) downloads. So the total downloads would be (k-1)*(n-i) number of downloads, in sequence.

On a large scale, the P2P gives the best case scenario, when compared to the centralized system, it is also in consideration to the maintenance of the server size system.