Peer-to-Peer with Distributed Index (P2P-DI) System for Downloading RFCs

We have built a peer-to-peer (P2P) system to download RFCs files. The RFC files are not present on a central server, instead they are distributed over several peers in the system. This information is maintained using a Distribution Index (DI) on a Registration server. The system allows clients to register to the P2P and query this DI to download the required RFC files.

All the communication in the system takes place over TCP.

Componenets of the system:

1. Registration Server - Maintains information about the peers in the system
2. RFC Server - Serves RFC files requested by peers, if it contains the file
3. RFC Client - Joins the P2P-DI and registers with RS, uses the peer index information to downloads RFCs from the active servers

Implementing Componenets of the system:

1. Registration Server (RS)

The RS waits for connections from the peers on the well-known port 65423. The RS maintains a peer list data structure with information about the peers that have registered with the RS at least once.

As suggested, we have used a linked list data structure to maintain the peer list. The nodes of linked list are defined as struct data type called : *peer_info*. Each record of the peer list contains these elements:

```
typedef struct __peer_info {
    char hostname[20];      // hostname of the peer
    int cookie;             // cookie assigned to peer
    bool active;            // flag to indicate active
    int ttl;                // ttl field
    int server_port;        // RFC server port of peer
    int reg_count;          // # of times peer was active
    struct tm *reg_time;    // recent registration time/date
}peer_info;
```

2. The Peers

Each peer maintains a local RFC index, that initially contains information only on RFCs stored locally at the peer. When the peer retrieves the RFC index of a remote peer, it merges it with its local copy, i.e., it updates its RFC index to include information about RFCs maintained by the remote peer.

We have used linked lists to implement the RFC index. And the nodes are defined as struct data types called : *RFC_index*. Each record of the RFC index contains four elements:

```
typedef struct RFC_index{
        int RFC_num;               // the RFC number
        char RFC_title[200];       // the title of the RFC
        char hostname[20];         // the hostname of the peer-
                                   // -containing the RFC
        int ttl;                   // a TTL field
}RFC_index;
```

.

The Application Layer Protocol: P2P

noindent We have defined a protocol for the peers to comminucate among themselves and for the communication with the RS server. The application runs on top of TCP and support the following types of messages:

Header Format : Generic

$< sp >$ marks the boundary of one field to next.
$< cr >< lf >$ marks the end of a header line.
$< cr >< lf >$ alone in a line - marks the end of header.

* For **peer-to-RS communication:**

1. Register: the peer opens a TCP connection to send this registration message to the RS and provide information about the port to which its RFC server listens.

Header Format of the Register request:

```
 REGISTER_REQUEST <cr> <lf>
 hostname: <sp> 127.0.0.1 <cr> <lf>
 server_port: <sp> 37191 <cr> <lf>
 cookie: <sp> 1 <cr> <lf>
<cr> <lf>
```

The cookie feild will be present only if client already has a cookie from the rs server. Hostname can be used, but we are using IP instead here.

Header Format of the Register Response:

```
REGISTER_RESPONSE <cr> <lf>
cookie: <sp> 1 <cr> <lf>
<cr> <lf>
```

The cookie value will be different only if client is registering for the first time. Otherwise server response will have the same cookie value the client had sent in request.

2. Leave: when the peer decides to leave the system (i.e., become inactive), it opens a TCP connection to send this message to the RS.

Header Format of the Leave Request:

```
LEAVE_REQUEST <cr> <lf>
hostname: <sp> 127.0.0.1 <cr> <lf>
server_port: <sp> 35261 <cr> <lf>
contentlength: <sp> 0 <cr> <lf>
cookie: <sp> 1 <cr> <lf>
<cr> <lf>
```

There is no leave response from the RS server according to our design. The assumption here is that the client peer will leave the P2P and might not wait for the RS server to acknoweldge.

3. PQuery: when a peer wishes to download a query, it first sends this query message to the RS (by opening a new TCP connection), and in response it receives a list of active peers that includes the hostname and RFC server port information.

Header Format of the PQuery Request:

```
PQUERY_REQUEST <cr> <lf>
hostname: <sp> 127.0.0.1 <cr> <lf>
contentlength: <sp> 0 <cr> <lf>
cookie: <sp> 2 <cr> <lf>
<cr> <lf>
```

4. KeepAlive: a peer periodically sends this message to the RS to let it know that it continue

Header format of the KeepAlive request:

```
KEEPALIVE_REQUEST <cr> <lf>
 hostname: <sp> 127.0.0.1 <cr> <lf>
 contentlength: <sp> 0 <cr> <lf>
 cookie: <sp> 2 <cr> <lf>
<cr> <lf>
```

There is no KeepAlive response from the RS server according to our design. The assumption here is that the RS will update the peer status and timers on its database but wont waste its time and resourses in sending a response since there might be al large set of peers sending keepalives intermittently.

* For **peer-to-peer communication:**

1.RFCQuery: a peer requests the RFC index from a remote peer.

Header format of the RFCQuery request:

```
RFC_GET_QUERY <cr> <lf>
 hostname: <sp> 192.168.1.10 <cr> <lf>
 contentlength: <sp> 0 <cr> <lf>
<cr> <lf>
```

Header format of the RFCQuery response:

```
RFC_GET_QUERY_RESPONSE <cr> <lf>
 8514 <sp> 8744 <sp> 8114 <sp> ... <sp> 8113 <sp> 8183 <cr> <lf>
<cr> <lf>
```

2.GetRFC: a peer requests to download a specific RFC document from a remote peer.

Header format of the GetRFC request:

```
GET_RFC_CONTENT <sp> RFC# <cr> <lf>
 contentlength: <sp> 0 <cr> <lf>
<cr> <lf>
```

Header format of the GetRFC response:

```
TAKE_RFC_CONTENT <sp> RFC# <cr> <lf>
 contentlength: <sp> size_in_bytes <cr> <lf>
<cr> <lf>
 data
```

```
We have defined different names for the Request messages and responses fo
 RFC_GET_QUERY 8
```

RFC_GET_QUERY_RESPONSE	9
GET_RFC_CONTENT	10
TAKE_RFC_CONTENT	11

Task 1: Centralized File Distribution

1. Initialize Peers Po t0 P5; P0 will have 60 RFCs
2. All 6 peers register with RS and receive peer list
3. Peers P1 to P5 query P0 and get the index of 60 files
4. Peers P1 to P5 starts a loop to download 50 RFC files from peer P 0 , one file at a time.
5. plot the cumulative download time against the number of RFCs for P1 to P5.

We tested the download times using 60 RFCs in a single server. To fully download all the files, a single peer took 2.2463 seconds. The chart in Fig.1 shows the download time trend with increasing number of RFCs.

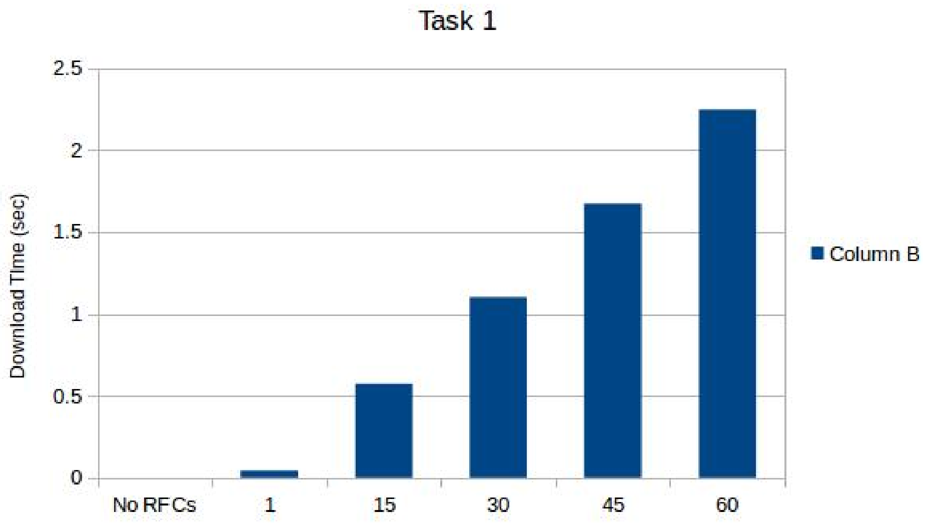

Figure 1: Task 1

Task 2: P2P File Distribution

1. Initialize Peers Po t0 P5; each will have 10 RFCs
2. All 6 peers register with RS and receive peer list
3. each of the six peers queries the other five peers and obtains their RFC index and merge them
4. each of the six peers starts a loop to download the 50 RFC files
5. plot the cumulative download time against the number of RFCs for each of the six peers

We tested the download times using 50 RFCs in a p2p setup. To fully download all the files, a single peer took 2.1608 seconds. The chart in Fig.2 shows the download time trend with increasing number of RFCs.

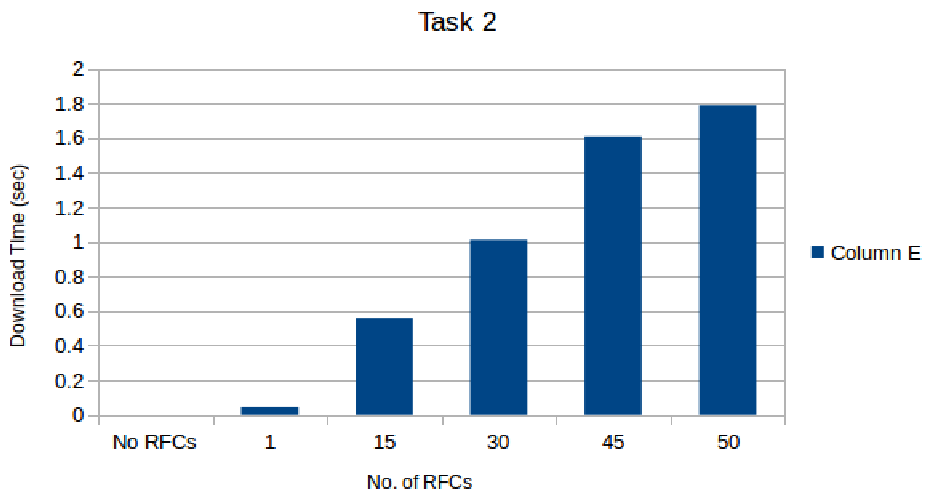

Figure 2: Task 2

Observation and Conclusion :

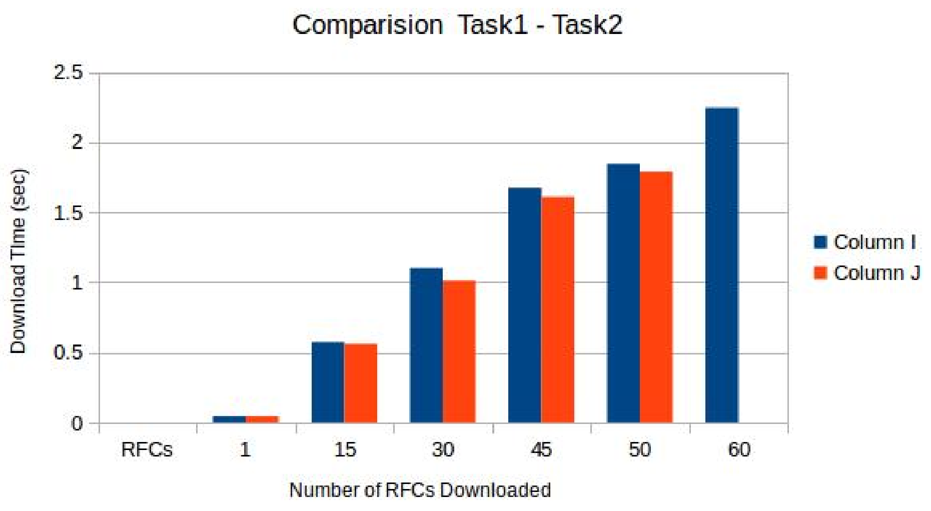

Figure 3: Comparision of Task 1 and Task 2 Download times

For downloading files in a client server architecture we see that as the number of files are increasing , the time taken to download them is more. This is as expected since network has to transfer and receive more data.

With increase in corresponding data bytes , proportional increse in download time is observed thus confirming our implementation coherence with client server architecture.

In the peer to peer transfer, we see that the time is almost the same as client server architecture. While this is may sound counter intuuitive, the reason for this is after receiving pquery response from RS server , we sequentially contact each peer for its data to download data. Thus, after finishing download from one peer we access second peer.

The reason we dont access clients in parallel is network could redundantly download same files from each of the peers before it is being written. We avoid this by the above approach.

The P2P download graph will show a ddownward trend if the peers contact all the peers in PQuery response parallelly with proper synchronization.

Steps to compile and run the code

We have included a make file in the src directory. Follow the steps to complie and run the code:

1. $tar - zxvftvnargun_hramach2.gz$
2. cd src
3. Run Make
4. make server # to complie only the client
5. make client # to compile only the server
6. ./server to run RS Server
7. ../testing/client1/src/socket_client #to run peer 1
8. ../testing/client2/src/socket_client # to run peer 2
9. ../testing/client3/src/socket_client # to run peer 3
10. ../testing/client5/src/socket_client # to run peer 4
11. ../testing/client6/src/socket_client # to run peer 5
12. ../testing/client6/src/socket_client # to run peer 6