CSC/ECE 573 – Internet Protocols

Project #1
Due Date: October 25, 2017

Project Objectives
In this project, you will implement a simple peer-to-peer (P2P) system with a distributed index (DI). Although this P2P-DI system is rather elementary, in the process I expect that you will develop a good understanding of P2P and client-server systems and build a number of fundamental skills related to writing Internet applications, including:

- becoming familiar with network programming and the socket interface,
- creating server processes that wait for connections,
- creating client processes that contact a well-known server and exchange data over the Internet,
- defining a simple application protocol and making sure that peers and server follow precisely the specifications for their side of the protocol in order to accomplish particular tasks,
- creating and managing a distributed index among multiple peers, and
- implementing a concurrent server that is capable of carrying out communication with multiple clients simultaneously.

Peer-to-Peer with Distributed Index (P2P-DI) System for Downloading RFCs
Internet protocol standards are defined in documents called "Requests for Comments" (RFCs). RFCs are available for download from the IETF web site (`http://www.ietf.org/`). Rather than using this centralized server for downloading RFCs, you will build a P2P-DI system in which peers who wish to download an RFC that they do not have in their hard drive, may download it from another active peer who does. **All communication among peers or between a peer and the registration server will take place over TCP.** Specifically, the P2P-DI system will operate as follows; additional details on each component of the system will be provided shortly.

- A **registration server (RS)**, running on a well-known host and listening on a well-known port, keeps information about the active peers. The RS does **not** keep any information about the RFCs that the various active peers may have.
- When a peer decides to join the P2P-DI system, it opens a connection to the RS to register itself. If this is the first time the peer registers with the P2P-DI system, it is given a **cookie** by the RS which identifies the peer. The cookie is used by the peer in all subsequent communication with the RS. For the purposes of this project, the cookie can be a small integer, unique to each peer. The peer closes this connection after registration. When a peer decides to leave the P2P-DI system, it opens a mew connection to the RS to inform it, and the RS marks the peer as inactive. Note, however, that a peer may leave the system without issuing a leave request, e.g., because the peer host crashed or the user turned off the system.
- Each peer maintains an **RFC index** with information about RFCs it has locally, as well as RFCs maintained by other peers it has recently contacted. It also runs an **RFC server** that other peers may contact to download RFCs. Finally, it also runs an **RFC client** that it uses to connect to the RS and the RFC server of remote peers.

- When a peer P_A wishes to download a specific RFC that it does not have locally, it opens a new connection to the RS and requests a list of active peers. In response, the RS provides P_A with a list of all peers who are currently active; if no such active peer exists, an appropriate message is transmitted to the requesting peer. Peer P_A then opens a connection to one of the other active peers, say, P_B, and requests its RFC index. When P_A receives the RFC index from P_B, it (1) merges it with its own RFC index, and (2) searches its new RFC index (after the merge) to find any active peer that has the RFC it is looking for. If the RFC index indicates that some active peer P_C has the RFC, P_A opens a new connection to the RFC server of P_C to download the RFC (note, however that P_C may have left the system by this time, and this connection may be unsuccessful). If P_A does not find any peer that has the RFC, or if its connection to P_C is unsuccessful, then it contacts another peer, say, P_D, in the list of active peers it received from the RS. This process continues until either P_A successfully downloads the RFC or the list of active peers is exhausted.
- The **RFC server** at a peer listens on a port *specific to the peer*; in other words, this port is not known in advance to any of the peers. The RFC server at each peer must be able to handle multiple simultaneous connections for downloads (of the RFC index or an RFC document) by remote peers. To this end, it has a main thread that listens to the peer-specific port. When a connection from a remote peer is received, the main thread spawns a new thread that handles the downloading for this remote peer; the main thread then returns to listening for other connection requests. Once the downloading is complete, this new thread terminates.

The Registration Server (RS)

The RS waits for connections from the peers on the well-known port `65423`[1]. The RS maintains a **peer list** data structure with information about the peers that have registered with the RS at least once. For simplicity, you will implement this data structure as a linked list; while such an implementation is obviously not scalable to very large numbers of peers, it will do for this project.

Each record of the **peer list** contains seven elements:

1. the hostname of the peer (of type string),
2. the cookie (of type integer) assigned to the peer (if the peer is on the list, it must have registered, therefore it has a cookie),
3. a flag (of type Boolean) that indicates whether the peer is currently active,
4. a TTL field (of type integer); it is initialized to a value of 7200 (in seconds) every time a peer contacts the RS (to register or ask for the peer list), and is decremented periodically so that whenever it reaches 0 the peer is flagged as inactive,
5. the port number (of type integer) to which the RFC server of this peer is listening; note that this field is valid only if the peer is active,
6. the number of times (of type integer) this peer has been active (i.e., has registered) during the last 30 days, and
7. the most recent time/date that the peer registered.

[1]The VCL/Eos RHEL5 machines block all ports except the ones in the range 65400-65500. Therefore, you will need to use ports in this range for this project.

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. Upon a request from a remote peer, this peer provides a copy of the whole RFC index it maintains (i.e., including local and remote RFCs it knows about). For simplicity, you will implement the RFC index as a linked list.

Each record of the **RFC index** contains four elements:

- the RFC number (of type integer),
- the title of the RFC (of type string),
- the hostname of the peer containing the RFC (of type string), and
- a TTL field (of type integer). For RFCs maintained locally, this value is set to 7200 (in seconds) and never modified. For RFCs maintained at a remote peer, the TTL value is initialized to 7200 at the time the RFC index from this remote peer is received, and is decremented periodically thereafter.

Note that the index may contain multiple records of a given RFC, one record for each peer that has a copy of the RFC document.

The Application Layer Protocol: P2P
You must design a protocol for peers to communicate with the RS and among themselves. In particular, the protocol must run over TCP and support at least the following types of messages:

- 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.
  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.
  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.
  4. `KeepAlive:` a peer periodically sends this message to the RS to let it know that it continues to be active; upon receipt of this message, the RS resets the TTL value for this peer to 7200.
- For **peer-to-peer communication:**
  1. `RFCQuery:` a peer requests the RFC index from a remote peer.
  2. `GetRFC:` a peer requests to download a specific RFC document from a remote peer.

You may define the protocol as a simplified version of the HTTP protocol we discussed in class. Suppose that peer A wishes to communicate with peer B running at host `somehost.csc.ncsu.edu`. Then, A may send to B a request message formatted as follows, where <sp> denotes "space," <cr> denotes "carriage return," and <lf> denotes "line feed."

```
method <sp> document <sp> version <cr> <lf>
header field name <sp> value <cr> <lf>
header field name <sp> value <cr> <lf>
<cr> <lf>
```

In this case, there is only one method defined, `GET`, that can be used to implement both the `RFCQuery` and `GetRFC` message above. You may also define certain header fields, e.g., `Host` (the hostname of the peer from which the RFC is requested) and `OS` (the operating system of the requesting host). In this case, the `RFCQuery` request message would look like this:

```
GET RFC-Index P2P-DI/1.0
Host: somehost.csc.ncsu.edu
OS: Mac OS 10.4.1
```

and the `GetRFC` request message would look like this:

```
GET RFC 1234 P2P-DI/1.0
Host: somehost.csc.ncsu.edu
OS: Mac OS 10.4.1
```

The response message may also be formatted similarly:

```
version <sp> status code <sp> phrase <cr> <lf>
header field name <sp> value <cr> <lf>
header field name <sp> value <cr> <lf>
...
<cr> <lf>
data
```

where the `data` field contains the RFC index or RFC document text file, depending on the request message. You are free to define header fields (e.g., for date, OS, last modified, content length, content type, etc), and status codes to signal errors, (e.g., bad request, file not found, version not supported, etc).

Similarly, you will have to define the request and response messages for peer-to-RS communication.

Typical Sequence of Messages

Figure 1 depicts the steps required for Peer A to register with the RS and download an RFC from Peer B. Before it joins the system, A first instantiates an RFC server listening to a local port in the range 65400-65500. In Step 1, Peer A registers with the RS, provides the local port number for its RFC server, and receives a cookie (if this is not the first time that Peer A registers with the RS, then it provides the cookie it received earlier in its `Register` message). The RS updates A's record to active and initializes the corresponding TTL value; if this was the first time A registered, then the RS creates a new peer record for A and adds it to its peer index. In Step 2, Peer A issues a `PQuery` message to the RS, and in response it receives a list of active peers. Recall that the list of peers returned by the RS includes the port number used by each peer's RFC server. Let us assume that Peer B is in this active list, and that B has the RFC that A is looking for. In Step 3, A issues an `RFCQuery` message to B, and in response it receives the RFC index that B maintains. A merges B's RFC Index with its own index. Since we have assumed that B has the desired RFC, A then in Step 4 issues a `GetRFC` message to B and downloads the RFC text document. Finally, in Step 5, Peer A sends a `Leave` message to the RS and leaves the system; the RS updates A's record to inactive.

Note that a peer sends each message (request, query, keep alive, etc.) to the RS by opening a new TCP connection; it receives corresponding responses over the same TCP connection. The TPC connection is closed at the end of each message exchange. Similarly, a peer opens separate TPC connections to a remote peer to request the RFC index and to download an RFC.

You will run two experiments to compare P2P and centralized file distribution, as explained in the next two tasks.

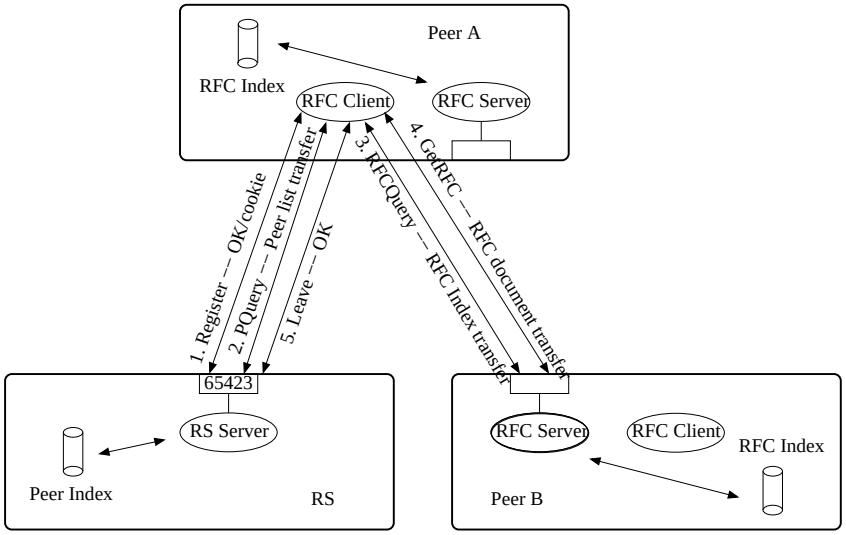

Figure 1: Sequence of messages in the P2P-DI system

Task 1: Centralized File Distribution
In this task, you will create six peers, $P_0, P_1, \ldots, P_5$, and initialize them such that P_0 contains the files of **60** RFCs, while peers $P_1, \ldots, P_5$ do not contain any RFC files. For consistency in the results, you are to use the *60 most recent RFCs*. In the first step, all six peers register with the RS and receive the peer list. In the second step, peers $P_1, \ldots, P_5$ query P_0 and obtain its RFC index; clearly, the index simply contains the 60 RFC files that peer P_0 holds. In the third step, each of the peers $P_1, \ldots, P_5$ starts a loop to download **50** RFC files from peer P_0, one file at a time. Each peer also records the time it takes to download each RFC file from peer P_0, as well as the cumulative time. You are to plot the cumulative download time against the number of RFCs for each peer $P_1, \ldots, P_5$. Note that this operation emulates a centralized server (i.e., P_0) that maintains all the information and handles simultaneous requests from clients (i.e., $P_1, \ldots, P_5$).

Task 2: P2P File Distribution
Again you will create six peers, $P_0, \ldots, P_5$, but each will be initialized such that it contains exactly 10 RFCs, for a total of 60 unique RFCs, as in the previous example; again, use the 60 more recent RFCs. In the first step, all six peers register with the RS and obtain the peer list. In the second step, each of the six peers queries the other five peers and obtains their RFC index which it merges with its own; at the end of this step, each peer will have the whole RFC index containiing 60 RFC files which are evenly distributed among the peers. In the third step, each of the six peers starts a loop to download the 50 RFC files it does not have from the corresponding remote peer. You are again to plot the cumulative download time against the number of RFCs for each of the six peers. It is not difficult to see that there is a best-case and worst-case scenario in this operation; implement both and provide download time figures for both.

Submission and Deliverables
You must submit your report and source code, as explained below, using the submit facility by 11:59pm on October 25, 2017. There are several weeks until the due date for you to work on this project, therefore no late submissions will be accepted.

You will carry out Tasks 1 and 2 offline, and submit a report (PDF or Word file) with the results and your explanation/discussion of the findings. In particular, your report should include the following information:

- the exact format of the messages exchanged between the peers and the RS and between peers;

- the download time curves for Task 1;
- the download time curves for Task 2; and
- a discussion of the differences you observed in the above download time curves and any conclusions you may draw regarding the scalability of P2P versus centralized systems for file downloading.

In order for us to test your program, you will also submit the source code (*no object files*!) separately. We would like to ensure that the TA does not spend an inordinate amount of time compiling and running your programs. Therefore, make sure to include a `makefile` with your submission, or a file with instructions on how to compile and run your code. Therefore, if you fail to include such a file, we will **subtract 5 points** from your project grade.

Testing Scenario
For us to test the operation of your program, you will need to implement the following simple scenario. There are two peers, A and B, initialized such that B has two RFCs and A has none. Both peers register with the RS, and then peer A queries the RS for the peer list. Once A get the response from the RS, it connects to B and downloads one of the RFCs. B then leaves the system by sending a leave message to the RS. A queries the RS again for the peer list, but since B has left, it should receive an indication that no peer is active. Throughout all communication in this scenario, the two peers and the RS print to the screen the format of the messages they receive.

Grading

RS code (compiles and runs correctly)	30 Points
Peer code (including concurrent RFC server)	40 Points
Application message format	10 Points
Task 1	10 Points
Task 2	10 Points
	100 Points