CSC/ECE 573 (002) – Internet Protocols
Sample Project
Fall 2016

Project Objectives

In this project, you will implement a simple peer-to-peer (P2P) system with a centralized index (CI) using UDP. Next we first describe the P2P-CI system and then describe the file transfer protocol between peers

Peer-to-Peer with Centralized Index (P2P-CI) System for Downloading RFCs using UDP

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-CI 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 will take place over UDP, however the communication between a peer and the server will take place over TCP. Specifically, the P2P-CI system will operate as follows; additional details on each component of the system will be provided shortly.

- There is a centralized server, running on a well-known host and listening on a well-known port, which keeps information about the active peers and maintains an index of the RFCs available at each active peer.
- When a peer decides to join the P2P-CI system, it opens a TCP connection to the server to register itself and provide information about the RFCs that it makes available to other peers. This connection remains open as long as the peer remains active; the peer closes the connection when it leaves the system (becomes inactive).
- Since the server may have connections open to multiple peers simultaneously, it spawns a new process to handle the communication to each new peer.
- When a peer wishes to download a specific RFC, it provides the RFC number to the server over the open connection, and in response the server provides the peer with a list of other peers who have the RFC; if no such active peer exists, an appropriate message is transmitted to the requesting peer. Additionally, each peer may at any point query the server to obtain the whole index of RFCs available at all other active peers.
- Each peer runs an *upload server* process that listens on a port *specific to the peer*; in other words, this port is not known in advance to any of the peers. When a peer A needs to download an RFC from a peer B, it communicates with peer B via peer B's upload port via UDP, provides the RFC number to B, and B responds by sending the (text) file containing the RFC to A; once the file transmission is completed, the connection is closed.

The Server

The server waits for connections from the peers on the well-known port 7734 (this is well known only to us☺). The server maintains two data structures: a list with information about the currently active peers and the index of RFCs available at each peer. For simplicity, you will implement both these structures as linked lists; while such an implementation is obviously not scalable to very large number of peers and/or RFCs, it will do for this project.

Each item of the linked list of peers contains two elements:

1. The IP address of the peer, and
2. The port number (of type integer) to which the upload server of this peer is listening.

Each item of the linked list representing the index of RFCs contains these elements:

- The RFC number (of type integer),
- The title of the RFC (of type string), and
- The IP address of the peer containing the RFC (of type string).

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

Initially (i.e., when the server starts up), both linked lists are empty (do not contain any records). The linked lists are updated as peers join and leave the system. When a peer joins the system, it provides its hostname and upload port to the server (as explained below) and the server creates a new peer record and inserts it at the **front** of the linked list. The peer also provides the server with a list of RFCs that it has. For each RFC, the server creates an appropriate record and inserts it at the **front** of the linked list representing the index. If a peer acquires a new RFC at any time, it must update the server that it now has this new RFC as well.

When a peer leaves the system (i.e., closes its connection to the server), the server searches both linked lists and removes all records associated with this peer. As we mentioned earlier, the server must be able to handle multiple simultaneous connections from peers. To this end, it has a main process that initializes the two linked-lists to empty and then listens to the well-known port 7734. When a connection from a peer is received, the main process spawns a new process that handles all communication with this peer. In particular, this process receives the information from the peer and updates the peer list and index, and it also returns any information requested by the peer. When the peer closes the connection, this process removes all records associated with the peer and then terminates.[1]

The Peers

When a peer wishes to join the system, it first instantiates an upload server process listening to any available local UDP port. It then creates a connection to the server at the well-known port 7734 and passes information about itself and its RFCs to the server, as we describe shortly. It keeps this connection open until it leaves the system. The peer may send requests to the server over this open connection and receive responses (e.g., the hostname and upload port of a server containing a particular RFC). When it wishes to download an RFC, it communicates with the remote peer at the specified upload port, requests the RFC, receives the file and stores it locally.

The Application Layer Protocol: P2P

The protocol used to download files among peers is a simplified version of the HTTP protocol we discussed in class. Suppose that peer A wishes to download RFC 1234 from peer B running at host <some IP address>.
Then, A sends to B a request message formatted as follows, where <sp> denotes "space," <cr> denotes "carriage return," and <lf> denotes "line feed."

method <sp> RFC number <sp> version <cr> <lf>
header field name <sp> value <cr> <lf>
header field name <sp> value <cr> <lf>
<cr> <lf>

There is only one method defined, GET, and only two header fields, Host (the hostname of the peer from which the RFC is requested) and OS (the operating system of the requesting host). For instance, a typical request message would look like this:

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

[1] Since multiple processes may manipulate the linked lists of the server simultaneously, these lists must be locked so that at most one process may access them to read or modify at any one time. For simplicity, we will not require that you implement locking.

The response message is formatted as follows:

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

Four status codes and corresponding phrases are defined:

- 200 OK
- 400 Bad Request
- 404 Not Found
- 505 P2P-CI Version Not Supported

Five header fields are defined: Date (the date the response was sent), OS (operating system of responding host), Last-Modified (the date/time the file was last modified), Content-Length (the length of the file in bytes), and Content-Type (always text/plain for this project).

A typical response message looks like this:

P2P-CI/1.0 200 OK
Date: Wed, 12 Feb 2009 15:12:05 GMT
OS: Mac OS 10.2.1
Last-Modified: Thu, 21 Jan 2001 9:23:46 GMT
Content-Length: 12345
Content-Type: text/text

(data data data ...)

Application Layer Protocol: P2S

The protocol used between a peer and the server is also a request-response protocol, where requests are initiated by peers. The format of a request message is as follows:

method <sp> RFC number <sp> version <cr> <lf>
header field name <sp> value <cr> <lf>
header field name <sp> value <cr> <lf>
…
<cr> <lf>

There are three methods:

- ADD, to add a locally available RFC to the server's index,
- LOOKUP, to find peers that have the specified RFC, and
- LIST, to request the whole index of RFCs from the server.

Also, three header fields are defined:

- Host: the hostname of the host sending the request,
- Port: the port to which the upload server of the host is attached, and
- Title: the title of the RFC

For instance, peer <some IP address> who has two RFCs, RFC 123 and RFC 2345 locally available and whose upload port is 5678 would first transmit the following two requests to the server:

ADD RFC 123 P2P-CI/1.0
Host: thishost.csc.ncsu.edu
Port: 5678
Title: A Proferred Official ICP

ADD RFC 2345 P2P-CI/1.0
Host: thishost.csc.ncsu.edu
Port: 5678
Title: Domain Names and Company Name Retrieval

Once a peer downloads a new RFC from another peer, it should transmit an ADD request to add a new record into the server's index.
A lookup request from this host would like this:

LOOKUP RFC 3457 P2P-CI/1.0
Host: thishost.csc.ncsu.edu
Port: 5678
Title: Requirements for IPsec Remote Access Scenarios

while a list request would be:

LIST ALL P2P-CI/1.0
Host: thishost.csc.ncsu.edu
Port: 5678

The response message from the server is formatted as follows, where the status code and corresponding phrase are the same as defined above.

version <sp> status code <sp> phrase <cr> <lf>
<cr> <lf>
RFC number <sp> RFC title <sp> hostname <sp> upload port number<cr><lf>
RFC number <sp> RFC title <sp> hostname <sp> upload port number<cr><lf>
...
<cr><lf>

In other words, the data part of the response lists one RFC per line, along with the information about the host containing the RFC.

The response to an ADD simply echoes back the information provided by the host:

P2P-CI/1.0 200 OK
RFC 123 A Proferred Official ICP thishost.csc.ncsu.edu 5678

The response to a LOOKUP may contain multiple lines for a given RFC, each line containing information about a different peer having the RFC. Finally, the response to a LIST lists all the records in the server's database, one per line. If the request contains an error or the requested RFC is not found in the index, an appropriate status code and phrase is returned, and the data part of the response is empty.
Note: the peer should print the responses it receives from the server to the standard output in the same format.

Simple File Transfer Protocol (Simple-FTP)

You will implement Simple-FTP to transfer RFCs between peers. Simple-FTP will use a custom file transfer protocol that uses UDP to send packets from one host to another, hence it has to implement a reliable data transfer service using the Go-back-N scheme. Using the unreliable UDP protocol allow you to implement a "transport layer" service such as reliable data transfer in the application layer.

Client-Server Architecture of Simple-FTP

When a peer receives a file from another peer, the peer that requested the file is essentially a client and the peer that sent the file is essentially a server for this particular file transfer.

The Simple-FTP Server peer (Sender)

The Simple-FTP server peer implements the sender side in the reliable data transfer. When the server peer starts, it waits for request on its upload port. As soon as it receives a request to send a particular RFC, it calls the rdt_send() (which you will implement) to transfer the data to the Simple-FTP client peer. For this project, we will assume that rdt_send() provides data from the file on a byte basis. The server peer also implements the sending side of the reliable Go-back-N protocol, receiving data from rdt_send(), buffering the data locally, and ensuring that the data is received correctly at the client. The server peer also reads the value of the maximum segment size (MSS) from the command line. The Go-back-N buffers the data it receives from rdt_send() until it either has at least one MSS worth of bytes or it does not receive any further data from application process for next 1 second. At that time it forms a segment that includes a header and MSS bytes of data.

The server peer also reads the window size N from the command line, and implements the sending size of the Go-back-N protocol. Specifically, if less than N segments are outstanding (i.e., have not been ACKed), it transmits the newly formed segment to the client in a UDP packet. Otherwise, it buffers the segment and waits until the window has advanced to transmit it. Note that if N = 1, the protocol reduces to Stop-and-Wait.

The header of the segment contains three fields:

- a 32-bit sequence number,
- a 16-bit checksum of the data part, computed in the same way as the UDP checksum, and
- a 16-bit field that has the value 0101010101010101, indicating that this is a data packet.

For this project, you may have the sequence numbers start at 0.

The server peer implements the full Go-back-N protocol as described in the book, including setting the timeout counter, processing ACK packets (discussed shortly), advancing the window, and retransmitting packets as necessary

The Simple-FTP client peer (Receiver)

The client peer implements the receive side of the Go-back-N protocol, as described in the book. Specifically, when it receives a data packet, it computes the checksum and checks whether it is in-sequence, and if so, it sends an ACK segment (using UDP) to the client; it then writes the received data into a file. If the packet received is out-of-sequence, or the checksum is incorrect, it discards the packet and ACKs the most recently received packet that it did not discard.

The ACK segment consists of three fields and no data:

- the 32-bit sequence number that is being ACKed,
- a 16-bit field that is all zeroes, and
- a 16-bit field that has the value 1010101010101010, indicating that this is an ACK packet.

Generating Errors

Despite the fact that UDP is unreliable, the Internet does not in general loose packets. Therefore, we need a systematic way of generating lost packet so as to test that the Go-back-N protocol works. To this end, you will implement a *probabilistic loss service* at the client (receiver). Specifically, the client will read the probability value p, $0 < p < 1$ from the command line, representing the probability that a packet is lost. Upon receiving a data packet, and before executing the Go-back-N protocol, the client will generate a random number r in $(0, 1)$. If $r \leq p$, then this received packet is discarded and no other action is taken; otherwise, the packet is accepted and processed according to the Go-back-N rules.

Logistics

1. Your code should be straightforward for the TA to execute. Please include a readme file describing exact steps that the TA should follow to execute your code.
2. To simulate multiple peers and a central index server, you will use VMWare. VMWare is freely available to students (https://www.csc.ncsu.edu/vmap/). Each peer must run in a separate virtual machine with a unique IP address.

Demo Setup

For the demo, you are encouraged to bring your own laptop. However, the TA will have a computer with VMWare installed. If you plan to use TA's computer, you need to set up a time with TA before the demo date and copy your VMs on TA's computer, so that no unnecessary time is wasted during the demo. The TA will give you some RFCs in .pdf format and ask you to save those files in different peers (running in different VMs). He will then ask you to give commands to different peers to obtain those files the peers that have them. When you type a command to request a file, say RFC1234 in the VM of a peer, say peer A, the CI server running in its own VM should print that peer A has asked it the name of the peers that have the RFC1234. You will have to show the TA that peer A has successfully obtained the requested RFC from another peer.

To show that your error generation module is working, the client peer should print the following line in its own VM whenever it discards a packet with sequence number X

Packet loss, sequence number = X

Whenever the server peer times out for a packet with sequence number Y, the server should print the following line in its own VM:

Timeout, sequence number = Y

If your code is working fine, you should see same values of sequence number at both server and client peers in exchanging a file.