Fishnet 4: Applications (CSEP561; Spring 2005)

Due: Tue May 24 in class (Out 5/10)

In this last assignment you will develop a file sharing application that runs in the class network and collects files that are made available for download. The goal of the assignment is for you to understand the structure of network applications by developing one, and to make use of the node you have developed over the quarter. Note that this application tests your transport and routing functionality from the prior assignments, so expect to spend some time improving your code if it did not function properly.

1. Fishster, a P2P file sharing application

Develop a fishster server, which serves files in response to requests, and a fishter client, which requests available files. The application should be built into your node using an event-driven interface of your choosing to interact with your transport code. The server and client should have the following features:

- *Fishster server*. Your server should accept connections on the Transport port reserved for Fishster, port 80, receive HTTP-style requests on these connections, and respond by transferring the requested files.
    - Note that standard HTTP is implemented on TCP, which uses bidirectional connections, while our Fishnet transport protocol provides only unidirectional connections. To account, for this, we will add a new request header of the form "Response-Port: 7". The client uses this header to indicate the port it will use to listen for responses (e.g., port 7). Similarly, HTTP responses should echo the name of the requested file in a header such as "Location: index".
    - You should only support the very simplest form of "GET" requests, as in "GET index HTTP/1.0\nResponse-Port: 7\n". The server then establishes another connection in the opposite direction, and if the requested file exists in the content directory then it sends the file to the other side, again using HTTP. Use only flat names (no directory names) and use only the simplest reply format, e.g., "HTTP/1.0 200 Accepted\nLocation: index\n<file contents here>", or "HTTP/1.0 404 Not Found\nLocation: index\n" if the file doesn't exist.
    - Your server should handle a request for the file "index" specially, by returning a list of all content available at the server. The list should have the following format: "<filename1> <md5-hash-of-file1>\n<filename2> <md5-hash-of-file2>\n … <filenamen> <md5-hash-of-filen>\n". There is convenient Java support to calculate these hashes. The md5 hashes should be printed as hexadecimal. (You can use `Utility.byteArrayToHex(byte[])` for this.) This enables clients to find out what content is offered by a server by contacting it and requesting the index file.
    - Use only one file transfer per connection. (No persistent connections as in HTTP/1.1) See Peterson 9.2.2 for more on HTTP.

- *Fishster client.* Your client should occasionally connect to its network neighbors (or other nodes of the network if you prefer), determine if they have content that it does not (by downloading the index file), download any new content, and add it to the content directory and update their index file so that it becomes available to other fishnet nodes.

Take the following steps as you develop your application:

1. Test it in an emulated network consisting of two of your nodes.
2. Test it in an emulated network with our reference node to see if files can be transferred in each direction.
3. Test it in the class trawler. We suggest you use a file that is named after your CSE login and with content of your choice; we want to be able to download something from you. Make sure that each content file is at most 10KB and consists of ASCII so that it can be printed and read as a message.

Full credit will be given for achieving step 3, partial credit for steps 1 and 2.

2. Discussion Questions

1. What role do the MD5 hashes serve in your application? What role do the filenames serve in your application? Do we need both.

2. What is the pre-image of the MD5 hash `0xeffad31e450232d06bf38c2ce2e1a687`?

3. Does your node suffer from any race condition between upload and download? Explain why or why not.

4. You implemented an event-driven (asynchronous) application interface. The more common sockets API is a synchronous (blocking) interface. Describe one advantage of each.

5. How quickly will content be propagated across this P2P network? Describe two modifications to the overall design that you might make if rapid dissemination were the goal.

6. Can an unscrupulous party block the spread of your content to some regions of the network, assuming that there are multiple paths to reach each node? If yes, say how. If no, say why not.

3. Turn In

1. Electronically turn in all of the source code for your entire node.

2. In class on the due date, turn in your design document, a printout of your application code, and answers to the discussion questions.

3. Also turn in a list of the files and hashes of all content you were able to collect from the class network. This is for our interest