CMPE 156/L, Winter 2016

Programming Assignment 3
Due: Feb 17

The problem in this assignment is to develop the same parallel file transfer application of Exercise 2, using UDP sockets instead of TCP. The application should enable a client to download "chunks" of a file from multiple servers (for example, ftp mirrors) distributed over the Internet, and assemble the chunks to form the complete file. This application has two key differences with that of Exercise 2.

1. Each chunk must be transferred from a server to the client using UDP as the transport-layer protocol. Because the chunks must be transferred reliably, this means that you need to implement a reliable application-layer protocol over UDP. The TFTP protocol is a good example of such an application-layer protocol. TFTP uses the stop-and-wait protocol with a retransmission timer for reliability, and block-level sequence numbers to maintain order. Some of the more complex functions of TCP, such as flow control, congestion control and round-trip delay estimation are not required to be supported.
2. The server must support concurrent transfers of chunks of a file to the same client or to different clients. This requires implementing a scheme similar to that employed by TFTP, where the server receives the initial request on a well-known UDP port, but opens a separate UDP port for transferring data from/to each client.

To test this application, you will have a single client process on your machine transferring data from multiple servers. The design should allow the server processes to run on any machine on the Internet, although for testing you can run several server processes on the same machine. The client will obtain the contact information for the servers from a text file `server-info.text` that lists the IPv4 addresses of the server hosts and the corresponding listening well-known UDP port numbers (the well-known port numbers can be the same if the server processes run on different machines. Otherwise, the only way to distinguish multiple server processes on the same machine is to use different port numbers).

```
128.32.16.1 1234
192.71.62.5 2345
localhost 5678
```

You should start each copy of the server using the command line

```
./myserver <port-number>
```

where the argument is its listening UDP port number.

The client process is started with the command

```
./myclient <server-info.text> <num-chunks>
```

The client should first contact one of the servers listed in server-info.text and obtain the size of the file to be transferred. It should then connect to the number of servers as specified in the <num-chunks> argument and transfer a chunk of the file from each server. Each chunk is specified by a starting offset and a chunk size. The client must calculate these parameters by dividing the size of the file by the number of chunks. The client must then initiate the transfer of each chunk concurrently. When all the chunks have been received, it should assemble them into the complete file and save it on disk (so that you can do a diff with the original file).

When the number of chunks <num-chunks> is larger than the number of entries in server-info.text, the client process needs to transfer more than one chunk from the same server. For example, if the number of chunks is 8, and the number of entries in server-info.text is only 6, you need to transfer two chunks in parallel from two of the servers, and one chunk each from the remaining four servers.

You should not assume that each chunk will fit within a single UDP segment, so your application layer protocol must deal with breaking a chunk into multiple UDP segments for transmission, and re-assembling the UDP segments into the original chunk on the receive side.

Requirements

- The server must be designed to support concurrent transfers to either the same client or different clients in parallel.
- You need to specify the application protocol, including the commands, responses, message formats, etc., to be used by the client and server to achieve the desired functionality. At a minimum, the client needs to check if a certain file exists at the server, and get its size. The client should also be able to initiate the transfer of a chunk of the file, identified by a starting offset and chunk size. You should also take care of any error scenarios that can arise in the client-server interactions.
- If one of the selected servers happens to be unreachable or malfunctions, the client must use an alternate server to transfer the corresponding chunk. If all the servers in server-info.text are already in use, the client must start the transfer of a new chunk from one of the servers that is currently performing a transfer.
- The server design must be robust against errors and network failures. The application should not fail unless the servers listed in server-info.text have all failed.
- The server should have a number of files available to serve (at least 6). The files can be limited to text files, and their total sizes can be limited to 100 Kbytes. These files are to be placed in the same directory where the server program is.

What to submit?

You must submit all the files in a single compressed tar file (with `tar.gz` extension). The files should include

1. A specification of the application-layer protocol, describing the handshakes involved, message formats, error handling, etc. Please submit PDF, do not include any Microsoft Word files.
2. All source code necessary to build and run the client and server.
3. A Makefile that can be used to build the client and server binaries.
4. Any test files you used to test your design.
5. Documentation of your design in plain text or pdf. Do not include any Microsoft Word files. The documentation should describe how to use your application and the internal design of your client and server implementations.
6. A README file including your name and a list of files in the submission with a brief description of each. If your code does not work completely, explain what works and what doesn't or has not been tested.
7. Organize the files into directories (*src*, *bin*, *doc*, etc.)

Grading

Each submission will be tested to make sure it works properly and can deal with errors. Grades are allocated using the following guidelines:

Basic Functionality:	40%
Dealing with errors	30%
Documentation	15%
Style/Code structure, etc.	15%

The files must be submitted before midnight on the due date.

Honor Code

All the code must be developed independently. All the work submitted must be your own.