COMP 631: Computer Networks

Fall 2014

Assignment # 1

Using UDP sockets, develop client and server programs (recommended in C or C++) that implement a simple transport protocol that implements an error detection service based on the CRC-32. You need to first define the minimal headers that the datagrams of such a protocol would need – you can assume that you are running over UDP and should attempt to add only the minimal headers that you would need. You need to then implement client and server programs that exchange messages using this protocol as described below.

Client

Your client should accept as command-line input four fields (in order): the name of the machine on which the server is running, the port number at which the server is listening, and the names of two files. The first file contains a list of messages to be encoded by the client using a CRC-32. The second file contains a corresponding list of messages that actually get sent to the server. In both files, the messages are delimited by the following special string on a separate line:

-END OF MESSAGE-

The client reads messages from each file and:

- For each message read from the first file, it computes a CRC-32 (using the IEEE 802.3 CRC-32 polynomial given by: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$). Most programming languages have libraries with functions that compute the CRC-32 – you can use these in-built functions, rather than implementing on your own.
- It then reads the corresponding message from the second file and creates a datagram (per the design specified below) using the CRC-32 computed from the message in the first file, but the payload included from the message in the second file.
```
    Client IP (32 bits) | Client port number (16 bits) | Length
(32 bits) | CRC-32 (32 bits) | payload (variable length)
```
- The client sends the datagram to the server (at the machine and port number specified in the command line).
- The client waits for and receives an "ACK" from the server (which is sent as soon as the server receives the full datagram). The client uses this ACK to estimate the time it took to transfer the datagram to the server. The ACK is formatted as below (where Length is the length of the datagram just received by the server).
```
    10101010 (8 bits) | Length (32 bits)
```

The client repeats the above till the end-of-file is reached in either of the input files. The client should deal with error cases in each of the system calls and should output an appropriate error message on the standard output.

Server

The server accepts as command-line input two fields: first is a port number at which it runs and the second is the name of a file.

The server opens a UDP socket and waits for client requests (at the port number specified in the command line). For each datagram received from a client:

- The server first sends back an ACK formatted as described before (using the client IP and port number). This ACK should be sent as soon as the datagram is received on the server socket.
- The server computes a CRC-32 over the payload and compares it to the CRC-32 sent by the client in the datagram headers. If they match, the client writes on standard output:

  ```
  Error-free message received from <client-machine>, <client-port>
  ```

  where <client-machine> is the name of the machine on which the client is running and <client-port> is the port number of the client. If the CRC-32 do not match, the client writes on standard output:

  ```
  Error in message received from <client-machine>, <client-port>
  ```

- In both of the above cases, the server then writes the payload in the file specified in the command line, and adds the delimiter on a separate line:

  -END OF MESSAGE-

The server repeats the above for all messages received on the server socket. The server should deal with error cases in each of the system calls and should output an appropriate error message on the standard output.

Some things to remember while developing your programs:

- Remember that the programs will be run under a login different from yours (with potentially different environment variables) and in a different directory that yours.
- The design and programming for this assignment must be your own work (do not attempt to reuse socket code you find on the web or one develop by someone else --- doing so would be a strict honor code violation).
- You should follow good development practices in your programs including clean structuring, comments for major program blocks, checking return codes from system calls, etc.
- You can assume that the input files (for the client) are properly formatted --- you don't have to check for errors in the sample input (eg, encountering end-of-file before the –END OF MESSAGE- delimiter).

Evaluating the validity and scalability of your implementation:

You also need to submit a small report (no more than 3 pages) that reports on two things.

- First, you need to run validation tests to test your server and client agains the server and client written by 2 other students in the class – that is, run your client against their server, and run your server against their client. Report the results of your compatibility tests (in no more than 1 page of your report). You have to make sure that if you are testing against the code of student X, he/she is not already using your code to test against theirs. **It is important to make sure that you share only the COMPILED version of your code with fellow students – exchanging source code of your programs would be a honor code violation!** Your tests should test whether the CRC-32 check is run correctly, and whether the payload transferred and stored in the file by the server matches the one read by the client from its input file.
- Second, use your own code to study how the time it takes to compute the CRC-32 scales with the size of a message. Specifically, use your programs to send messages of different sizes and measure the CRC-32 computation time. Plot this function and comment on it. Also measure the time it takes for you to get the message across to the server and comment on how the two times compare. Please try to limit this section of the report to no more than 2 pages.

Submitting Programming Assignments and the Report:

Submission of your program should be handled as follows:

- Create a directory in your AFS home-directory space and give me (login ‘jasleen’) read and lookup permissions on the directory. If you are not familiar with AFS directory permissions, see topics “afs-intro” and “afs-security” on the department’s “howto” web pages.
- Place all the submitted materials into the directory. This should include the source files, executable files for your client and server programs ready to run on the department’s public Linux systems, and any configuration or other auxiliary files needed to run your programs. Your client executable should be named “UDP-client” and your server executable should be named “UDP-server”. No attempt will be made to recompile and link your programs so the executables must be ready to run as is.
- Send email to me giving the full pathname for the directory containing your submitted materials. The later of (i) the date on that email message, and (ii) the latest file-modified time in the directory determines the submission date for purposes of deciding is the assignment is late. WARNING: this means that if you change a file after submission, it may be considered late.
- Attach to your email, your report in PDF format (before submitting, make sure your report prints fine on a regular department printer).

Sample Inputs:

In this example, there’s a single client that runs on machine quintet.cs.unc.edu and uses port number 9678, and the server runs on classroom.cs.unc.edu and uses port number 9783.

Client example:

Sample input file: file1.txt

```
This is the first message. The text can be arbitrary (and non-ascii).
-END OF MESSAGE-
This is the second message. We’re going to simulate errors using this
one.
-END OF MESSAGE-
<EOF>
```

Sample input file: file 2.txt

```
This is the first message. The text can be arbitrary (and non-ascii).
-END OF MESSAGE-
This is the second message. Period.
-END OF MESSAGE-
<EOF>
```

Sample invocation of the client:

```
$UDP-client classroom.cs.unc.edu 9783 file1.txt file2.txt
```

Server example:

Sample invocation of the server:

```
$UDP-server 9783 file-out.txt
Error-free message received from quintet.cs.unc.edu, 9678
Error in message received from quintet.cs.unc.edu, 9678
```

Sample output file: file-out.txt

```
This is the first message. The text can be arbitrary (and non-ascii).
-END OF MESSAGE-
This is the second message. Period.
-END OF MESSAGE-
```