ECSE 414 - Homework Assignment #1 – Application Layer
Released: Friday, September 13, 2013
Due: Friday, September 27, 2013

Note: Unless otherwise noted, all assignments are due at the beginning of the lecture period on the due date. For "paper and pencil" problems (such as Problems 1, 2, and 3 below), you may submit a hard copy of your assignment in class or to the instructor's mailbox, or (preferred method) you can submit your assignment electronically via myCourses (preferred format is pdf). For programming problems (such as Problem 4), you must upload an archive file (e.g., zip, rar, or tgz) on myCourses containing the source code, executable (or .class file for Java applications), and a Readme file containing any special instructions for compiling and running your application (e.g., which version of the Java VM you used).

Problem 1 (7 marks, one per part)

A packet (call it "packet X") of size 1000 bits arrives to a router. It takes 6 μs to check if packet X has bit errors and 12 μs to lookup the appropriate next hop, based on the destination, and 1 μs to transfer packet X to queue at the output link. There are 5 packets waiting in queue when packet X arrives. Each is of length 1500 bits. The transmission speed of the outgoing link is 0.5 Mbps. The distance to the next router is 4 km. Assume that packets propagate at a speed of $2x10^8$ m/s.

a) What is the processing delay experienced by packet X?
b) What is the queueing delay experienced by packet X?
c) What is the transmission delay experienced by packet X?
d) What is the propagation delay experienced by packet X?
e) What is the maximum number of bits that could be in the link (transmitted by the sender and en route to the receiver) at any instant in time?
f) Calculate the *bandwidth-delay* product, the product of the link bandwidth with the propagation delay. Provide an interpretation of the bandwidth-delay product.
g) If you could change the distance to the next hop, at what distance would the transmission delay be equal to the propagation delay?

Problem 2 (K&R Ch 2, P10) (4 marks, two per part)

Consider a short, 10-meter link, over which a sender can transmit at a rate of 150 bits/sec in both directions. Suppose that packets containing data are 100,000 bits long, and packets containing only control (e.g., ACK or hand-shaking) are 200 bits long. Assume that N parallel connections each get $1/N$ of the link bandwidth. Now consider the HTTP protocol. Suppose that each downloaded object is 100Kbits long and the initial download object contains 10 referenced objects from the same server.

a) Would parallel downloads via parallel instances of non-persistent HTTP make sense in this case? Explain your reasoning.
b) Now consider persistent HTTP. Do you expect significant gains over the non-persistent case? Justify and explain your answer.

Problem 3 (6 marks, one per part)

Suppose you manage a local area network that has an access link of 10 Mbps and the local capacity on your network is 100 Mbps. The users of the LAN request 500 web objects per second, and the average object size is 10,000 bits. The average round-trip time from the router on the non-LAN (i.e., internet) side of the access link to a remote web server and back is 100 ms (including all sources of delay). Assume there is zero delay within the LAN.

Model the average access delay as $D/(1-BD)$ seconds, where D is the average time to send an object over the access link and B is the arrival rate of objects to the access link.

a) What are the utilizations on the LAN and on the access link?
b) What is the average access delay?
c) What is the total delay for a web object retrieval?

You currently pay your ISP $10 per kbps of traffic each month (based on the average download rate). Amazon offers you a new cooperative caching service, where you redirect all of your web requests through their system. Their cache, which has a hit rate of 65%, resides outside your LAN, in the public Internet. The round-trip time from the non-LAN side of your access link to the cooperative cache is 35 ms. When their cache does not have the object requested, they can retrieve it from the origin servers in only 15 ms on average, since they have a very high-speed backbone connection. Moreover, they optimize and compress all web objects requested before sending them back over your access link, effectively reducing the average object size to 7,500 bits.

d) With their caching service, what is the new utilization on the access link?
e) With their caching service, what is the new total average delay for a web object retrieval?
f) Amazon will charge a $8,000 setup fee, plus a service fee of $8,000 per month. If you use this service, how much money will you save or lose during the first year?

Problem 4 (Programming problem) (15 marks)

Multi-threaded HTTP Server

For this problem you will implement a simple multi-threaded HTTP server using sockets. Example code for implementing a simple TCP client is given and explained in Section 2.7 of Kurose & Ross (5th edition). Example code was also demonstrated and provided in the first tutorial for this class, and is available on myCourses.

In this problem we will develop a multi-threaded web server capable of processing multiple service requests in parallel. You will demonstrate that your server is capable of delivering an example web page provided with the assignment.

We are going to implement version 1.0 of HTTP, as defined in RFC 1945, where separate HTTP requests are sent for each object in the web page. The web server will be multi-threaded. In the main thread, the server will listen to a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread. To simplify this programming task, we'll develop the code in a few stages. In the first stage, you will write a multi-threaded serve that simply displays the contents of the HTTP request message that it receives. When this step is running properly, you will add the code required to generate an appropriate response.

As you are developing the code, you can test your server using a web browser on your machine. If your server does not run on the standard port 80 (some operating systems may throw an exception if you try to run your webserver on port 80), then you will need to specify the proper port to the browser. For example, if you are running the server on port 6789 on your personal machine, and you want to retrieve the file index.html, then you would specify the following URL in your browser:

http://localhost:6789/index.html

If you omit the ":6789" then the browser will try to connect to a server on the default port 80 which may not have a server listening on it.

When the server encounters an error, it sends a response message with the appropriate HTML code so that the error information is displayed in the browser window.

Step 1: Setting up the Basic Multi-Threaded Server Structure (3 marks)

In the following steps, we will go through the code for the first implementation of our Web server. You will be filling in missing details in the template file `WebServer.java` which is provided on myCourses in the assignment folder.

The first step is to create the server socket which will listen for incoming connections. This goes in the `main()` method of the `WebServer` class. Then, within the infinite `while` loop, listen for new incoming requests. Each incoming request will be processed by a `HttpRequest` class (a helper class, defined further down in the `WebServer.java` file). The `HttpRequest` class implements the `Runnable` interface, so it can be launched within a new `Thread` object, similar to the multi-threaded TCP server example we saw in the tutorial. You should create a new `Thread` you're your `HttpRequest`, and then start the thread.

Step 2: Parse the HTTP Request Message (5 marks)

When the new thread is started, it will invoke the `run()` method of `HttpRequest`. This method calls the `processRequest()` method, which is where most of the action happens. Next, you will begin to fill in `processRequest()`.

The first thing to do, similar to the TCP server example covered in tutorial, is to set up input and output stream filters for the socket, to make it easy to read and write to/from the socket. I suggest using a BufferedReader to handle reading from the socket and a DataOutputStream to handle writing to the socket.

Next, read the first line of the request message into the variable `requestLine`. (Hint: If you're not sure how to do this, look up the documentation for the `readLine()` method of the class `BufferedReader`.) This is the line that contains the path to the web object which is being requested. We'll print this to the command line so that we can see what info we received in the request.

The remaining lines of the header contain additional information about the server and the request. We will not do anything with these in our simple Web server, but we will read them and print them to screen. Use a `while()` loop to process all remaining header lines. You will know you have reached the end of the request message when you first encounter a line

At this point you have finished parsing the HTTP request message. Before moving on to formulating and sending back the response, this is a good time to test your application. First, scroll down to the bottom of the `processRequest()` method and add a few lines of code to close the file input/output streams you opened, and also to close the socket before the thread exits. Even though we do not fulfill the request completely, this will at least signal to the web client that we're done communicating (so the client can display some sort of error message).

After adding lines to close the socket at the end of `processRequest()`, compile and run `WebServer`. Open a web browser on your machine, and direct the browser to http://localhost:6789/index.html. Note: Make sure to use the correct port number if you changed line 27 in WebServer.java. Your browser should display some sort of error message indicating that the connection was closed, and the command line where your server is running should display the text of the HTTP request message that was received.

Step 3: Sending the HTTP response message (7 marks)

Next you will prepare and send a simple HTTP response message. Depending on whether or not the web server finds the file that is being requested, you need to send back an appropriate response. We will only include one additional header line to indicate the MIME type of the file being returned.

I have taken care of some of the work for you here. The lines under the comment "`//` STEP 3a: …" use a Java `StringTokenizer` object to parse the first line of the HTTP request in order to grab the path of the object being requested. This ends up in a variable called `fileName`, and a period is also prepended to the `fileName` variable so that the operating system knows to look for this file relative to the directory in which `WebServer` is running. For example, if the user requests the HTML object "/index.html", we want to look for "./index.html" (relative to the directory in which WebServer is running) rather than "/index.html", which would be relative to the root directory on our computer.

Note that the lines around the try-catch statement here also define and set a Boolean variable called `fileExists` which is true if we successfully find and open the requested web object, and is false otherwise.

Now, to prepare the response, you need to specify values for the two variables `statusLine`, and `contentTypeLine`, which are both initialized to null. The variable `statusLine` should contain the first line of the HTTP response. Make sure to set the code appropriately, depending on whether or not the requested file was found. The variable `contentTypeLine` should contain a line of the form "Content-type: " and the appropriate MIME type. If the file was found and will be returned, use the helper method `contentType()` to determine the appropriate MIME type string. If the file was not found, then use MIME type "text/html", since we will be sending a brief HTML error message.

Next, we are ready to send the HTTP response message. Use the `DataOutputStream` to send the status line, the content type line, and then the body of the message. Don't forget that each line of the HTTP response header (in our case, the status line and content type line) should end with a carriage return and line feed. Also, don't forget that there should be a blank line to indicate the end of the header, before the content is written. For the body of the message, if the file was found, you can use the helper method `sendBytes()` provided in the `HttpResponse` class to write the file out on the socket. If the file was not found, then send the `errorMessage` string in the body of the HTTP response.

The final thing you need to do is to finish off the `contentType()` method. Currently it only will return the correct MIME type for HTML files. Modify the method so that it also returns the appropriate types for GIF and JPEG images. You can look up the correct strings in RFC 2046. (Alternatively, you can easily find them elsewhere on the web.)

Test your Web Server

Now your server should be ready to run. Compile your code and execute it in the same directory containing the test files provided (index.html, index2.html, mcgill_0.gif, and TR02a.jpg).

First, direct your browser to http://localhost:6789/index.html. (Again, make sure to use the correct port number if you changed it.) This should display a simple web page which only contains text.

Next, either click on the link or direct your browser to http://localhost:6789/index2.html. This is a slightly more complex base HTML file which links to three additional web objects: a CSS file and two images.

Finally, try directing your browser to a file which does not exist (for example, http://localhost:6789/index3.html). You should receive the 404 Not Found message.

Et voila! Your multi-threaded HTTP 1.0 web server is complete.