**CPSC 261: Lab 8**

**Lab 8 - Network Programming - Sockets**

* [Goals](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#goals)
* [Introduction](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#introduction)
* [Overview](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#overview)
* [The Details](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#details)
* [Provided Materials](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#provided)
* [Handing It In](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab8.html#handing)

**Goals**

The goals of this lab are to:

* Give you experience using sockets to communicate between processes across the Internet.

**Introduction**

This lab will help you explore network programming using sockets. You will be writing two client programs that communicate with existing servers in the Internet.

**Overview**

This lab has two parts:

* You will write a client program to communicate with a simple server that I have written. Your client will communicate using the TCP transport protocol and have a short simple conversation with the server.
* Secondly, you will evolve your client program so that it can communicate with real Internet servers. In particular, your evolved client will be capable of communicating with web servers and extracting data from them.

The work necessary to do this is really rather small, as it turns out!

**The Details**

* Step 1 - I have a server running at TCP port 11261 on lulu.ugrad.cs.ubc.ca. You need to write a client program that will communicate with my server, send it messages formatted in a particular way, and receive the replies from the server.

Your client program should be called client.c (this is what the Makefile expects). Your client should take two arguments, the hostname and port number of the server that it is to connect to. Your client program must create a socket, bind it to a local address (the port number that your client uses will not be important) and connect to the specified server (at the given machine address and port number. There are a number of resources on the web (and elsewhere) that describe the code that is necessary to create, configure, and connect sockets. [Beej's guide](http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html) is still my favourite although it has gotten much longer and more difficult to read over the last few years. Thanks to the [Internet Archive](http://archive.org/index.php) we can find the older version of the guide, which I have copied to [here](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/Beejs-guide-Jan-1999.pdf).

I strongly suggest that you separate this task into two pieces:

* + First, write a function tcp\_connect that given a hostname and a port number creates a socket, binds it to a local port, and connects to the given machine and port, finally returning the connected socket. You are provided with tcp.h and tcp.c files that will get you started. They contain the relevant functions from the sample program that we've talked about in class.
  + Second, complete your client program which will use your tcp\_connect function to establish the connected socket.

Once your client program is able to establish a connection to the server, it needs to send a pair of messages and receive the corresponding replies. The first message that I want you to send will look like this:

Mon Mar 17 08:37:14 PDT 2014

More precisely, the message consists of just the current time, formatted as done by the standard Linux command date, followed by a newline character..

You can also send

Mon Mar 17 08:37:14 2014

which is probably easier to generate.

After sending this message to the server, you must receive the server's reply, which will consist of a single long (64 bit) positive integer, formatted using ascii characters, followed by a newline character.

After receiving this message, you must send another message to the server, which consists of the number that you just received minus 1, a single space, and the string "My name is Norm Hutchinson" followed by a newline character, except that it should contain your name, not mine. After sending this message, you must shutdown the sending half of the socket connection, indicating that you will be sending no more data to the server. This is done by calling the function shutdown which is described in section 2 of the Linux manual. You want to shutdown the write half, so will pass the argument SHUT\_WR.

After sending the message, you must read the reply which will come back from the server.

Whenever you are reading network messages, you must be aware that the message may be broken into multiple network packets and so you may need to call the recv function multiple times before you have received the entire message. When you have completed this conversation with the server, you should record the messages that you sent as well as the replies that you received in your README.txt file.

* Step 2 - Your client program is almost all that it needs to be to communicate with real servers in the Internet. In this step you will evolve your client program so that it can "speak" a simple subset of the Hypertext Transfer Protocol (HTTP), the lingua franca of the Internet.

Copy your client.c program to webclient.c and edit webclient.c so that instead of having the little conversation that you had with my server, you send a real HTTP message to a real web server and process the reply that you receive.

Again, there are lots of guides on the Internet to the HTTP protocol. We only need to use a very small subset of it to construct a minimal request and decipher the reply. The first little bit of [this guide](http://www.jmarshall.com/easy/http/) should provide all the information that you need. Just look at the section entitled Structure of HTTP Transactions.

Your client program in this instance will take 3 arguments:

* + The name of the server to talk to; something like www.google.ca, for example.
  + The port number that the HTTP server is listening at. This will almost always be 80 for real web servers.
  + The name of the document that you are requesting. A good initial document to get from any web server is "/index.html", which is the standard name for the top-level document of any server. So a good invocation of your web client program will be:

./webclient www.google.ca 80 /index.html

The request that you send to the server should look like this:

GET the-name-of-the-document HTTP/1.0

Host: the-name-of-the-web-server

After you have a working webclient program, I want you to use it to successfully access at least 5 web servers, and record the request that you sent as well as all of the header lines that you receive (the header lines are the ones in the reply before the first blank line, they will all start with a header keyword followed by a colon(:)) in your README.txt file. A successful access to a web server is defined for this step as one that gives you back some interesting data. An access that gives you back an error response (Document not found, Permission denied, Document moved, etc) is not considered successful, but you may be able to turn such a request into a successful one by following the instructions in the unsuccessful reply. For example, when I run my webclient thusly:

./webclient google.ca 80 /index.html

I get back a reply that says that the document has permanently moved:

HTTP/1.0 301 Moved Permanently

Location: http://www.google.ca/index.html

Content-Type: text/html; charset=UTF-8

Date: Fri, 22 Mar 2013 16:02:16 GMT

Expires: Sun, 21 Apr 2013 16:02:16 GMT

Cache-Control: public, max-age=2592000

Server: gws

Content-Length: 228

X-XSS-Protection: 1; mode=block

X-Frame-Options: SAMEORIGIN

The Location: header line tells me where it went, so I can try again with:

./webclient www.google.ca 80 /index.html

Which gives me back:

HTTP/1.0 200 OK

Date: Fri, 22 Mar 2013 16:03:52 GMT

Expires: -1

Cache-Control: private, max-age=0

Content-Type: text/html; charset=ISO-8859-1

Set-Cookie: PREF=ID=12dcd2a12e297ec5:FF=0:TM=1363968232:LM=1363968232:S=7Z5x5L8\_z2Qput59; expires=Sun, 22-Mar-2015 16:03:52 GMT; path=/; domain=.google.ca

Set-Cookie: NID=67=jeYSsS379hNCrxFsNP7wc\_SNhcZxgx9U3\_ISF1VKGjn0ek13el8CyC6G9xcREgej7yUMgpanpKmT81iFGFLNZn9D1xVz-2CpNkeps6e4e7woah4amc3\_BPl4QSLB-ZRn; expires=Sat, 21-Sep-2013 16:03:52 GMT; path=/; domain=.google.ca; HttpOnly

P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for more info."

Server: gws

X-XSS-Protection: 1; mode=block

X-Frame-Options: SAMEORIGIN

Some web servers give rather long replies to the request that you will be sending. To capture the reply easily, just redirect the output from your webclient command to a file and then edit the file:

./webclient www.google.ca 80 /index.html > log

**Provided Materials**

You should use subversion to check out the lab8 directory from your repository, in the same way that you did for the previous labs.

Your lab8 directory should contain the following files:

* Makefile

The provided Makefile contains instructions for compiling and linking the programs client.c and webclient.c.

* tcp.h

The provided tcp.h contains external declarations for some helpful functions that are in tcp.c, as well as the tcp\_connect function that you will be writing.

* tcp.c

The provided tcp.c contains definitions of some useful functions from the socket program that we discussed in class, as well as a stub implementation of tcp\_connect for you to complete.

You will need to create all of the other files.