# CSE461 Project 2: TCP/HTTP Proxy

**Out:** Friday January 31, 2014
**Due:** Monday February 10, 2014 by **11:59pm**.
**Teams Allowed:** Yes
**Teams Encouraged:** Yes
**Ideal Team Size:** 2

## Summary

There are two parts to this project. In one, you'll write a simple HTTP proxy, which will introduce you to both the client and server sides of TCP, and give you a tiny bit of exposure to HTTP. In the other part, you'll experimentally determine how the programming environment you're using exposes to your code the operations done to the TCP connection by the other side. That is, we'll want to answer questions like, "What can happen in my code when I invoke `read` on a TCP connection?" Knowing that will help us write robust code.

You'll hand in code for the HTTP proxy component and a short writeup for the TCP behavior component of this project.

## TCP Overview

If you do a web search for "`tcp socket` *`yourlanguage`*" you'll probably find dozens of pages that all contain essentially the same information. These pages are useful, and you should use them, as they give you the details for your language. Here is a general overview, though.

TCP distinguishes between the "server side" and the "client side" of a connection. The server must come up first, creating a socket to which the client can connect. Imagining for the moment that the server side is up, the client does this:

- create a STREAM socket
- connect it to the server's ip and port
- start reading and writing using that socket

The server side is a bit more complicated. The server needs to wait around for clients to connect to it. It uses a special kind of socket for this. Each time a client connects, a new socket is generated on the server side. That socket is a regular old socket, just like the client uses. This simplifies the server code, because it ends up with a 1-1 relationship between (regular old) sockets and clients, making it easy to keep their data streams separated.

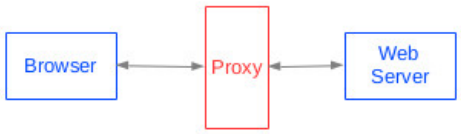
Here's what a server does. (Your programming language may combine some of these steps.)

- Creates a STREAM socket
- Binds it to an ip address and port
- Calls `Listen` to indicate that this socket will be used to wait for incoming connections, not for reading or writing data
- Sits in a loop calling `accept`. `accept` blocks until some client connects. When that happens, it returns to the server code a new (regular old) socket, connected to that client.
- Now handles the client interaction by reading and writing on that socket. It's common to use a separate thread to do this, so that a slow client doesn't prevent other clients from connecting concurrently.

## One Part: HTTP Proxy

HTTP is the protocol used to transfer information between browsers and web servers. HTTP is transmitted using TCP as the transport protocol.

An HTTP proxy is a program that can accept and reply to requests that would normally be directed to some web server. Proxies are an example of the use of "interposition" - placing something between two things that communicate using a well-defined interface -- as shown in the figure below. Interposition is a generally useful technique. When possible, it allows new functionality to be injected into existing code with little or no modification to that code. For example, an HTTP proxy might be used for monitoring or debugging (by capturing a log of browser requests and server responses), to improve performance by maintaining a cache of web pages, or to enforce some policy about which sites can be accessed.



The requirements for our proxy are very modest: it merely prints out the first line of each HTTP request it receives from the browser, then fetches the requested page from the sourcing web server and returns it to the browser. This means that, for the most part, you don't have to know anything about HTTP; you simply read what the browser sends, print out (only) the first line, and pass that and all subsequent lines on to the web server. On the other side, you read everything the web server sends and pass it back to the browser. You keep forwarding data in this way, in each direction, until you detect that the source has closed the connection. To use the proxy we must configure the browser to send all its requests to the proxy, instead of directly to the web servers. In Firefox you do this using Edit/Settings, then the the Advanced icon, then the Network tab, then the Settings button for "Connection." Configure the proxy manually.

While that's the basic operation, there are two details that require a bit of processing of the HTTP stream. To make what follows more concrete, here's an example of what Firefox sent when I requested the page `www.my.example.page.com`. (I obtained this by running `nc -l 46103` to set it listening for TCP connections on port 46103, and then configuring Firefox to use a proxy located at `localhost:46103`.)

```
$ nc -l 46103
GET http://www.my.example.page.com/ HTTP/1.1
Host: www.my.example.page.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:26.0) Gecko/20100101 Firefox/26.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

**Determining the web server's address**

When the browser sends an HTTP request to your proxy, you need to forward it on to the appropriate web server. You determine the web server by recognizing the `Host` line in the HTTP header. In the example above, the host is `www.my.example.page.com`. You should be insensitive to the case of the keyword `Host`, and you should be tolerant of white space anywhere it might plausibly appear. In general, the host name may be given as `hostname:port` or `ip:port`. If no port is specified, you should use 80, the default.

The HTTP specification says that lines of the header are terminated by CRLF:

```
CR             = <US-ASCII CR, carriage return (13)>
LF             = <US-ASCII LF, linefeed (10)>
```

You should be lenient in interpreting this, though. For instance, you might see headers where the lines are terminated by a single LF.

HTTP does not require any particular ordering for the lines of the header, except that the request line (which is always of the general form shown in the example above) must be first.

The HTTP 1.1 specification requires that a `Host` line be provided in an HTTP request (but not in a reply). Your code does not have to work with HTTP 1.0, which doesn't require these lines. (But, I'd guess you'd have a hard time finding a browser that wanted to speak HTTP 1.0 in any case.)

**Turning off keep-alive**

The HTTP `Connection: keep-alive` line can be used to indicate that the browser (or server) wants to keep the TCP connection open even after the current HTTP request has been fully satisfied. This is a performance optimization: if the browser issues additional requests to the same server within a short time, the overhead of closing the current TCP connection and opening a new one is avoided.

Supporting `keep-alive` greatly complicates the proxy, because it needs to do enough HTTP parsing to understand where one HTTP request ends and the subsequent one begins (and similarly for responses coming from the server). HTTP doesn't have a simple framing mechanism for marking these boundaries. To avoid that, you should filter the request and response streams, turning `Connection: keep-alive` into `Connection: close`. That causes the browser and web server to close the TCP connection after each request. Each HTTP request now starts with the creation of a new TCP connection and ends with TCP close, making things simple for the proxy.

**`run` Script**

To help us test your code, provide a `run` script that will build and invoke your proxy. The script should take a single argument: the port number the proxy's server socket should bind to. The proxy's output should be the first line of each HTTP request it receives from the browser. Execution is terminated by Ctrl-C.

**Sample Output**

Most interesting web pages these days are complicated enough that the set of HTTP requests issued to fetch them will differ from fetch to fetch. We show here some sample output to give you a general sense of what your output should look like, though.

| | |
|---|---|
| CSE Winter Quarter 2014 Time Schedule | Sample Output |
| www.whitehouse.gov | Sample Output |
| www.cnn.com | Sample Output (Truncated to the first approximately 40 seconds.) |

## The Other Part: TCP Behavior

Let's imagine you have a typical application that uses TCP. You have a thread that sits in a loop trying to read from the socket. It may time out periodically, but most of the time it's blocked on the read, waiting for input to arrive. You also have another thread that performs most of the application logic. From time to time it writes to the socket, to send data to the other end.

Your goal is to determine experimentally what your code sees when the other end manipulates the TCP connection. The answer to this can be quite system and language dependent.

One thing that can happen at the other end is that it crashes abruptly. On any reasonable system this should cause any TCP connections it has open to be closed, and so the behavior you see is likely to be the same as a graceful close.

`close` `shutdown`

In general, there are two operations the other end can do to the connection: and . Typically, a socket is closed when no further reading or writing is intended. `shutdown` is a bit more detailed. It takes an argument that indicates whether the connection is being shut down for reading or for writing, individually. *Exactly* what `close` and `shutdown` do is a mystery, and probably system/implementation dependent. See, for example, the man page for `shutdown`, the bulk of which is this illuminating line:

> The shutdown() function shall cause all or part of a full-duplex connection on the socket associated with the file descriptor socket to be shut down.

Perform experiments that let you determine what your program sees for each operation it can be doing and each operation the other side might do. Your program can be blocked on a read when the remote operation occurs. It can also be performing a write after the remote operation occurs. That write might be the first one after the remote operation occurs, or the second, or the third. (Behaviors of those three could differ.) You might get a return code that indicates success, or one that indicates failure. You might get an exception. You might receive a signal (http://en.wikipedia.org/wiki/Unix_signal, especially `SIGPIPE`), which might cause your program to simply stop execution without any apparent error having taken place.

Ultimately, our goal is to be able to write robust code, which means that the code must be prepared to deal with everything that could happen to the TCP connection because of the behavior of the other end of the connection. Create a table of actions your program may take and things that could happen on the other end that indicates how your program can detect that they have taken place, so that it could deal with them without crashing or hanging.

For this part, hand in a short report as file `tcp.pdf`, if in the preferred pdf format, or `tcp.txt` if in the alternative acceptable format, text.

## Turn in

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `proj2`. There should be no other files in that directory.
2. Create a README.TXT file that contains the names, student numbers, and UW email addresses of the member(s) of your team.
3. Put the README.TXT file, your HTTP proxy solution source code, your `run` script, and your `tcp.pdf` report file in the `proj2` directory.
4. While in the directory that is the parent of `proj2/`, issue the command `tar czf proj2.tar.gz proj2`.
5. Verify that the tar file contains the files you intend to submit: `tar tf proj2.tar.gz`.
6. Submit the `proj2.tar.gz` file to the course dropbox.