CS 4480: Computer Networks - Spring 2014

Programming Assignment 1
HTTP Web Proxy Server
Three parts:
PA 1: A – Due on Jan 24th 2014
PA 1: B – Due on Feb 7th 2014
PA 1: Final – Due on Feb 21st 2014

1 Background

1.1 HTTP Proxies

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy. The proxy then opens a connection to the server, and passes on the client's request. The proxy receives the reply from the server, and then sends that reply back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).

Why use a proxy? There are a few possible reasons:

- **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.
- **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.
- **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or other client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

2 Assignment details

The goal of this programming assignment[1] is to develop a simple caching HTTP Web Proxy Server. The proxy should be capable of serving multiple concurrent requests. Your proxy only need to support the HTTP

[1]Credit: This programming assignment was derived from similar assignments at Stanford (Nick McKeown) and Princeton (Jennifer Rexford - COS-461) and as described in Kurose and Ross. Code skeleton from Kurose and Ross companion website.

GET method.

Very basic skeleton code in Python and Java is provided if you choose to make use of it. There is also C-code for parsing HTTP messages which might be useful if you want to code in C. The skeleton code is incomplete: To get the skeleton code to work you need to add your own code at places marked as `/* Fill in */` or `# Fill in start, #Fill in end`. You then need to extend the functionality to realize the programming assignment.

Note that you are not required to use the skeleton code. It might in fact be simpler to start from scratch with your own code.

You are allowed to use a language different from python or java to complete the assignment. However, (i) you are not allowed to use any libraries other than the standard socket libraries for that language (you may use the provided C message parsing code), and (ii) you will essentially be on your own, i.e., if you get stuck you will likely get limited (if any) support from the teaching staff.

It is strongly recommended that you approach the assignment as a multi-step process: First develop a proxy that is capable of receiving a request from a client, passing that through to the origin (real) server and then passing the server's response to the client. Then extend this basic capability so that your proxy is capable of serving multiple clients concurrently. Then enhance the functionality of the basic multi-client proxy to create a caching proxy.

2.1 Basic multi-client proxy

Basics. Your first task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy MUST handle concurrent requests. E.g., by using the `multiprocessing` package in python, or `threads` in java. You will only be responsible for implementing the GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).

You should not assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP.

Listening. When your proxy starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections. Each new client request is accepted, and a new process/thread is spawned to handle the request. There should be a reasonable limit on the number of processes/threads that your proxy can create (e.g., 100). Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request. Specifically, you should ensure that the proxy receives a request that contains a valid request line:

```
<METHOD> <URL> <HTTP VERSION>
```

All other headers just need to be properly formatted:

```
<HEADER NAME>: <HEADER VALUE>
```

In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2) – as your browser will send if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxies that some ISPs deploy, unbeknownst to their users). An invalid request from the client should be answered with an appropriate error code, i.e., "Bad Request" (400) or "Not Implemented" (501) for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your client should also generate a type-400 message.

Getting Data from the Remote Server Once the proxy has parsed the URL, it can make a connection to the requested host *(using the appropriate remote port, or the default of 80 if none is specified)* and send the HTTP request for the appropriate resource. The proxy should always send the request in the relative URL + Host header format regardless of how the request was received from the client:

Accept from client:

```
GET http://www.google.com/ HTTP/1.0
```

Send to remote server:

```
GET / HTTP/1.0
Host: www.google.com
Connection: close
(Additional client specified headers, if any..)
```

Note that we always send HTTP/1.0 flags and a "Connection: close" header to the server, so that it will close the connection after its response is fully transmitted, as opposed to keeping open a persistent connection. So while you should pass the client headers you receive on to the server, you should make sure you replace any Connection header received from the client with one specifying close, as shown.

After the response from the remote server is received, the proxy should send the response message as-is to the client via the appropriate socket.

2.1.1 Testing Your Proxy

Basic test. Assuming your proxy listens on port number `port` and is bound to the `localhost` interface, then as a basic test of functionality, try requesting a page using telnet:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://www.cs.utah.edu/~kobus/simple.html HTTP/1.0
```

If your proxy is working correctly, the headers and HTML of a (real) simple HTML document should be displayed on your terminal screen.

Notice that here we request the absolute URL:

```
http://www.cs.utah.edu/~kobus/simple.html
```

instead of just the relative URL. The relative URL could be requested as follows:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET /~kobus/simple.html HTTP/1.0
Host: www.cs.utah.edu
```

A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response from a direct telnet connection to the remote server.

Concurrency test. You should also test the multi-client functionality of your proxy by requesting a page using telnet concurrently from two different shells. If you request a very simple page, like the one above, it might be difficult to show concurrency as the page downloads very quickly. A simple way to work around this is to download a large file so that you can verify that the file is concurrently being served to both clients. (For example, you can create a large file and serve it up with your own web server, e.g., using something like the Python SimpleHTTPServer.)

A more sophisticated test. For a slightly more complex test, you should configure your web browser to use your proxy server as its web proxy. The exact way to configure this will depend on the browser you use. For example, when using Firefox, (i) select Firefox>Preferences (or Edit>Preferences), (ii) select Advanced, (iii) select the Networking tab, (iv) select Setting for “Configure how Firefox connects to the Internet”, (iv) select “Manual proxy configuration” and enter the IP address and port number for you proxy.

2.2 Caching proxy

Basics. When a proxy is deployed for performance reasons, it typically caches web objects each time one of the clients makes a request for the first time. Basic caching functionality works as follows. When the proxy gets a request, it checks if the requested object is cached, that is, stored locally in the proxy. If the object is stored locally, it returns the object from the cache, without contacting the server. If the object is not cached, the proxy retrieves the object from the server, returns it to the client and caches a copy for future requests.

Required functionality. Add the simple caching functionality described above to your multi-client proxy. Your implementation need to be able to write responses *to the disk* (i.e., the cache) and fetch them from the disk when you get a cache hit. For this you need to implement some internal data structure in the proxy to keep track of which objects are cached and where they are on the disk. This bookkeeping data structure can be maintained in main memory; there is no need to make it persist across proxy shutdowns. *However, your cache should be be implemented by writing web objects to disk.*

Enhanced functionality. In practice, a proxy server must verify that the cached responses are still valid and that they are the correct responses to the client’s requests. You can read more about caching and how it is handled in HTTP in section 13 of IETF RFC 2068. You can implement this basic cache validation functionality in your caching proxy to earn *extra credit.*

Note that real caching proxies employ sophisticated cache replacement strategies. You do not need implement any such functionality for this programming assignment.

3 Grading and evaluation

To encourage you to start early and systematically work on the assignment, there will be three submissions as outlined below.

3.1 PA 1 - A: Basic Proxy

For this first part of the assignment, the focus will be on basic (single client) proxy functionality. Specifically, we will evaluate your code by performing a test similar to the *Basic test* paragraph Section 2.1.1.

What to hand in You should submit your completed sub-assignment electronically on Cade by the due date. You submission should consist of a **single** tarball file which contains the following:

1. All the source code for your assignment.
2. A readme.txt file explaining how to compile and/or run your program(s).

Your submission tarball must be submitted on CADE machines using the handin command. To electronically submit files while logged in to a CADE machine, use:

```
% handin cs4480 assignment_name name_of_tarball_file
```

where `cs4480` is the name of the class account and `assignment_name` (pa1_a, pa1_final etc.) is the name of the appropriate subdirectory in the handin directory. Use pa1_a for this sub-assignment.

3.2 PA 1 - B: Multi-client Proxy

For this part of the assignment you are to develop the multi-client proxy as described in Section 2.1. At this point the emphasis will be on the *multi-client* aspect of the proxy. Specifically, we will evaluate your code by performing a test similar to that described in the *Concurrency test* paragraph in Section 2.1.1.

What to hand in You should submit your completed sub-assignment electronically on Cade by the due date. You submission should consist of a **single** tarball file which contains the following:

1. All the source code for your assignment.
2. A readme.txt file explaining how to compile and/or run your program(s).

Your submission tarball must be submitted on CADE machines using the handin command. To electronically submit files while logged in to a CADE machine, use:

```
% handin cs4480 assignment_name name_of_tarball_file
```

where `cs4480` is the name of the class account and `assignment_name` (pa1_a, pa1_final etc.) is the name of the appropriate subdirectory in the handin directory. Use pa1_b for this sub-assignment.

3.3 PA 1 - Final: Complete Assignment

For your final submission you should implement the remaining required functionality and optionally the enhanced (extra credit) functionality.

Your final submission will be tested more thoroughly, including evaluation with the Firefox browser, with real websites and for various error conditions.

What to hand in You should submit your completed assignment electronically on Cade by the due date. You submission should consist of a **single** tarball file which contains the following:

1. All the source code for your assignment with inline documentation.
2. A readme.txt file explaining how to compile and/or run your program(s).

Your submission tarball must be submitted on CADE machines using the handin command. To electronically submit files while logged in to a CADE machine, use:

```
% handin cs4480 assignment_name name_of_tarball_file
```

where `cs4480` is the name of the class account and `assignment_name` (pa1_a, pa1_final etc.) is the name of the appropriate subdirectory in the handin directory. Use pa1_final for this assignment.

In addition, you should submit an Assignment Report as a pdf via Canvas.

This report should include the following sections:

- *Design* that describes the program design, how it works and any design tradeoffs considered and made. You should also clearly indicate whether you have implemented the extra credit functionality.
- *Testing* that describes the tests you executed to convince yourself that the program works correctly. Also document any cases for which your program is known not to work correctly.
- *Output* that shows output that illustrates the correct functioning of your program.

3.4 Grading

Criteria	Points
Sub-assignment: PA 1 - A (works correctly)	5
Sub-assignment: PA 1 - B (works correctly)	5
PA 1 - Final Program (works correctly)	70
Inline documentation	5
Exception handling	5
Assignment report	10
Total	100
Extra credit functionality	10
Total with extra credit functionality	110

Other important points

- We will use Firefox to test your proxy against a real browser.
- Every programming assignment of this course must be done individually by a student. No teaming or pairing is allowed.
- Your programs will be tested on CADE Lab Linux machines. You can develop your program(s) on any OS platform or machine but it is your responsibility to ensure that it runs on CADE Lab machines. You will not get any credit if the TA is unable to run your program(s).
- Use TCP port numbers 6000 to 8000 for this programming assignment. These ports can be used only inside the UofU network. These are not accessible from outside of UofU.

Additional notes

For simplicity your proxy should support version 1.0 of the HTTP protocol, as defined in RFC 1945.

A significant simplification that comes from limiting the proxy functionality to version 1.0 is that your proxy does not need to support persistent connections. Recall that the default behavior for HTTP 1.0 is non-persistent connections. Note, however, that most browsers, including the latest version of Firefox automatically add the optional "Connection: keep-alive" header line. Also, with the latest version of Firefox it is no longer possible to disable this behavior. I suggest you download an older version of Firefox so that you can disable this. I tested with Firefox 15.0, although other versions might also work. (See notes below.)

IMPORTANT: Note that older versions of browsers might have security vulnerabilities. I therefor strongly suggest that if you get an older version of Firefox to use for this programming assignment, you do not use that for general web browsing, but only for the assignment.

A further implication of using HTTP 1.0 is that you can infer that you have received all the content from the server when the server closes the connection. See:

`http://www8.org/w8-papers/5c-protocols/key/key.html`

Also, as specified in the assignment the request forwarded to the server should contain a "Connection: close" header line to ensure that it closes the connection. (This might imply replacing a "Connection: keep-alive" header if the client had that in its request.)

Notes about configuring Firefox: Type 'about:config' in the title/search bar. You will be presented with a very large number of parameter settings. You should set the following parameters as shown:

```
network.http.proxy.version 1.0
network.http.version 1.0
network.http.keep-alive false
network.http.proxy.keep-alive false
network.http.pipelining false
network.http.proxy.pipelining false
```