CS118 Winter 2014 Project 1

Overview

In this project, you will learn about basic concepts of network applications and work through the difficulties related to network programming. The project is to implement a caching Hypertext Transfer Protocol (HTTP) proxy. You will be dealing with a subset of HTTP 1.1 specification, which is defined in RFC2616.

All implementation should be written in C++ using BSD sockets. No high-level network-layer abstractions (like Boost.Asio or similar) are allowed in this class. You are allowed to use some high-level abstractions for parts that are not directly related to networking, such as string parsing. We will also accept implementations written in C, however use of C++ is preferred and use of BSD sockets is required

The submitted code should be well organized, there should be proper indentation, and enough comment/documentation that allow understanding of what your code is doing. It is highly recommended that names of all classes, functions, methods are meaningful and follow CamelCase convention. If you wish, you may follow the coding guidelines described here.

Tasks specification

Code skeleton will be uploaded to the CourseWeb or you can check it out from a git repository:

git clone git://github.com/iliamo/cs118-project1

You are strongly recommended to use this skeleton. At the least, your code should be one-simple-command compilable (e.g., using make or ./waf) with optional configuration step.

Your code will be tested on lnxsrv SEAS servers. You should already have access to the machine, if not you have to apply for SEAS account and you will get an access. For more information about lnxsrv click here.

Attention! You should configure your PATH environment on lnxsrv. Add the following line to file *~/.bashrc* in your home directory on the server:

export PATH=/usr/local/cs/bin:$PATH

The skeleton code links to the BOOST library, therefore, you will also need to add the following line to .bashrc:

export LD_LIBRARY_PATH=/u/cs/grad/yingdi/boost/lib:/usr/local/**cs**/lib64/:$LD_LIBRARY_PATH

Attention! The above will work only if you are using bash. If not, export command should be changed to something appropriate, which sets environment variable. Or you can change your shell by just typing bash after login.

To set bash as your default shell interpreter, you can add following line into file "~/.login"

exec /usr/local/cs/bin/bash

Attention! You should test compilation of your code on lnxsrv before submitting source to the CourseWeb. If compilation fails, you will get zero credit. If compilation gives warnings, we will deduct 5 points.

Caching HTTP Proxy

Your first task is to build a simple WEB proxy that accepts HTTP requests, forwards these requests to remote locations, gets and caches response, and returns the response to the client. You are responsible only for implementing the GET method for the proxy. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 2616 section 10.5.2).

You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP. Server should listen on port 14886. (You can change the port number when debugging the code, but it must be changed back to 14886 when submitting the code.)

Executable should be called http-proxy without any command-line parameters.

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(on port 14886). Each new client request is accepted, and a new process is spawned using, for example, fork() to handle the request (if you prefer pthreads or any high-level threading library like Boost.Thread, it is also fine, provided the code can be compiled on lnxsrv).

There should be a reasonable limit on the number of processes that your proxy can create. In this project, your proxy should run with no more than 20 simultaneous processes, and should serve at most 20 incoming connections. Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request, but don't worry, we have provided you with libraries that parse the HTTP request lines and headers. Specifically, you will use our libraries to 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 2616, 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 proxy should also generate a type-400 message (See RFC 2616, Section 10.4).

Parsing Library We will provide library for parsing requests. Check the reference code on the Coursweb or on github.

Parsing URL Once the proxy sees a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host and port, and the requested path. See the URL (7) manual page for more info. You will need to parse the absolute URL specified in the request line using the parsing library. If the hostname indicated in the absolute URL does not have a port specified, you should use the default HTTP port 80.

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.ucla.edu/ HTTP/1.1

Send to remote server:

GET / HTTP/1.1
Host: www.ucla.edu
(Additional client specified headers, if any...)

Note that we always send "HTTP/1.1" flags to the server, so that it will keep open a persistent connection. The proxy can keep at most 100 simultaneous connections to all remote servers (in total). You proxy should also support non-persistent connection when it is specified in the header:

To add new headers or modify existing ones, use the HTTP Request Parsing Library we provide.

Returning Data to the Client 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.

Pipeline Since the proxy supports persistent connection, you also need to implement the pipeline.

Caching Same web pages may be requested by several users. The performace of proxy can be improved by caching fetched web pages. As a part of caching, conditional GET must be implemented. (see RFC 2616, Section 13)

Grading

The code should be submitted to the CourseWeb in .tar.gz archive format before the deadline. **No late submissions will be graded!**

Your .tar.gz tarball should contain:

- All the source code necessary to compile your proxy
- Compiled executables should be named http-proxy. After make or ./waf compilation executables should be either in the current directory, src/, or build/ folders.
- Makefile or wscript (if you're using waf to build the code)
- README file describing your code and design decisions
- TEAM file listing project members and specifying which parts of the code were written by whom

Your code will be graded on 100 point basis.

- If your code does not compile, you will get 0 points
- If compilation gives warnings, we will deduct 5 points.
- If your code passes provided tests (httptester.py and httptester-conditionalGET.py), you will get 40 points
- If your code passes all of the additional tests that you don't know in advance, you'll get another 30 points
- If your code is well written (good abstraction, handling of errors, is readable, have enough comments, there are no memory leaks), you will get 20 points. We will use Splint (http://www.splint.org/) and Valgrind (http://valgrind.org/) to automatically check the code. **You can get these points only if at least 70% of public tests are passed**.
- Clear description of your design decisions in a README file gets 10 points

Introduction to HTTP

The Hypertext Transfer Protocol (HTTP) is the protocol used for communication on this web: it defines how your web browser requests resources from a web server and how the server responds. You may refer to that RFC 2616 while completing this assignment.

HTTP communications happen in the form of transactions; a transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

- An initial line (a request or response line, as defined below)
- Zero or more header lines
- A blank line (CRLF)
- An optional message body.

The initial line and header lines are each followed by a "carriage-return line-feed" (\r\n) signifying the end-of-line.

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps:

A client creates a connection to the server. The client issues a request by sending a line of text to the server. This request line consists of a HTTP method (most often GET, but POST, PUT, and others are possible), a request URI (like a URL), and the protocol version that the client wants to use (HTTP/1.1). The request line is followed by one or more header lines. The message body of the initial request is typically empty (see RFC 2616 sections 5, 9, 14).

The server sends a response message, with its initial line consisting of a status line, indicating if the request was successful. The status line consists of the HTTP version (HTTP/1.1), a response status code (a numerical value that indicates whether or not the request was completed successfully), and a reason phrase (an English-language message providing description of the status code). Just as with the the request message, there can be as many or as few header fields in the response as the server wants to return. Following the CRLF field separator, the message body contains the data requested by the client in the event of a successful request (see RFC1945 sections 6, 10, 14). Depending on information in the header fields, the connection may be kept alive or closed.

It's fairly easy to see this process in action without using a web browser. From a Unix prompt, type:

$ telnet www.yahoo.com 80

This opens a TCP connection to the server at www.yahoo.com listening on port 80 (the default HTTP port). You should see something like this:

Trying 69.147.125.65...
Connected to any-fp.wa1.b.yahoo.com.
Escape character is '^]'.

Type the following:

$ GET http://www.google.com/ HTTP/1.1

and hit enter twice. You should see something like the following:

HTTP/1.1 200 OK
Date: Wed, 04 Apr 2012 21:38:09 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
...
(More headers and the actual HTML follows)

There may be some additional pieces of header information as well: setting cookies, instructions to the browser or proxy on caching behavior, etc. What you are seeing is exactly what your web browser sees when it goes to the Yahoo home page: the HTTP status line, the header fields, and finally the HTTP message body, consisting of the HTML that your browser interprets to create a web page.

HTTP Proxies

Normally, 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 infor- mation 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.

Socket programming

In order to build your proxy you will need to learn and become comfortable programming sockets. The Berkeley (BSD) sockets library is the standard method of creating network systems on Unix. There are a number of functions that you will need to use for this assign- ment:

Parsing addresses:

- **inet_addr** Convert a dotted quad IP address (such as 36.56.0.150) into a 32-bit ad- dress.
- **gethostbyname** Convert a hostname (such as argus.stanford.edu) into a 32-bit address.
- **getservbyname** Find the port number associated with a particular service, such as FTP.

Setting up a connection:

- **socket** Get a descriptor to a socket of the given type

- **connect** Connect to a peer on a given socket
- **getsockname** Get the local address of a socket

Creating a server socket:

- **bind** Assign an address to a socket
- **listen** Tell a socket to listen for incoming connections
- **accept** Accept an incoming connection

Communicating over the connection:

- **read/recv, write/send** Read and write data to a socket descriptor. **recv** and **send** are equivalents to **read** and **write**.
- **htons, htonl / ntohs, ntohl** Convert between host and network byte orders (and vice versa) for 16 and 32-bit values

You can find the details of these functions in the Unix man pages (most of them are in section 2) and in the Stevens Unix Network Programming book, particularly chapters 3 and 4. Other sections you may want to browse include the client-server example system in chapter 5 and the name and address conversion functions in chapter 9.

Multi-process (multi-threaded) programming

In addition to the Berkeley sockets library, there are some functions you will need to use for creating and managing multiple processes: fork, waitpid (or pthread create, pthread join, etc.).

You can find the details of these functions in the Unix man pages:

$ man 2 fork
$ man 2 waitpid

If you prefer Boost.Thread libraries, more information can be found here.

References

- Guide to Network Programming Using Sockets
- HTTP Made Really Easy- A Practical Guide to Writing Clients and Servers
- Wikipedia page on fork()

Acknowledgement

This project was largely based on the COS-461 class project by the Dr. Jennifer Rexford, Princeton University.