Project One: The ACME HTTP Proxy Server[1]

Assigned: Thursday, September 20th, 2001
Due: Friday, October 5th, 2001, 5 p.m.

When the CIO convinced ACME Fudge & Bolt Co.'s CEO of the importance of joining the Internet revolution, he thought he was doing the company a huge favor and a big bonus would be his for the taking. Unfortunately, his predictions that ACME's profits would climb through the roof, employee productivity would soar, and the company's image would shine like never before have proven to be, how shall we put this mildly... disastrously inaccurate. A series of major problems have arisen:

- the Internet load quickly overwhelmed the initial ISDN link, so ACME had to change to a new and more expensive internet service provider and install an expensive T3 link;
- employee productivity has been steadily declining - on several occasions, ACME's CEO caught marketing twerps playing Doom, wasting time surfing the net, and tying up the internet link for long periods of time downloading material from `www.playboy.com`. People in other departments have seen even worse.
- company profits are down, and the new web server has proven to be more of a marketing embarrassment than a marketing bonanza. The final *coup* came in July, when a prankster hacked their way into ACME's network and caused computers to display nothing but pictures of snow-covered mountains, Santa Claus, and reindeer for an entire week.

After the "Christmas server" incident, ACME's CEO went ballistic. She told the CIO that he had until the end of September to solve the problems or find a replacement. A firewall was installed immediately, which helped prevent outside hackers from breaking in. The next step involves controlling ACME's Internet link to restrict its abuse by ACME employees. You have been hired as a contractor to write a configurable HTTP proxy server that would allow ACME employees access to the Web on a *need-to-browse* basis.

1 Overview

HTTP requests consist of roughly three steps: (i) the client application parses the desired URL to determine the machine name and TCP port number of the desired web page, and makes a TCP 'socket' connection to that machine and port; (ii) the client application sends a request message to the server on the new socket connection, specifying what operation it wants performed (most often, a GET operation to get the contents of a web page), and; (iii) the server application performs the desired operation and returns the resulting data (most often a web page in html format) to the client on the established TCP connection.

When a client application is configured to connect via a *proxy server*, the client establishes its connection to the proxy server's machine and TCP port and forwards its complete request to the proxy server rather than the "real" web server. The HTTP proxy server accepts local HTTP connections and forwards them to their final destination – essentially it introduces an extra "hop" between the client browser and the web server. There are several reasons why you would use a proxy server instead of connecting to the web site via the Internet directly: you might be behind a firewall where direct connections are not allowed; a proxy can be used to locally cache data in

[1]This project was developed at the University of Utah and is being used with their permission.

order to reduce the amount of global traffic; or—in our case—the proxy is used to control access to the Web on a per host/per location basis.

The objectives of this assignment are to help you learn how to write networking code using the Berkeley sockets API, to familiarize you with one of the most widely used Internet application protocols (http), and to give you experience in building client/server applications. An interesting thing about proxy servers is that they behave like a server when they accept requests from local clients, but they act like a client when they forward their local client requests along to the real web server. Thus, by implementing a proxy server, you will get a feel for how both sides of a client/server application are implemented.

The following sections detail the set of operations that your proxy server must perform to solve ACME's internet problem. In addition to this document, we have provided a number of other resources to help you implement the project, including:

- RFC 1945: Hypertext Transfer Protocol (HTTP 1.0). This document describes the format of HTTP protocol requests and responses, which is what your server will be receiving and generating.
- RFC 1738: Uniform Resource Locators (URL). This document describes the format of URLs, which your server will need to parse. Most of the legal URLs specified in RFC 1738 are *not* supported by the ACME proxy server, however, so consider this text supplemental.
- RFC 1808: Relative Uniform Resource Locators. This document builds on RFC 1738. Like that document, RFC 1808 is supplemental.
- Berkeley sockets user guide and examples: This collection of documents and sample programs should prove useful for getting started on programming your proxy server using Berkeley sockets. It is accessible via the class web page.
- Unix man pages: in addition to the high level primer described above, you should be sure to read the relevant Unix man pages on `gethostbyname()`, `gethostbyaddr()`, `byteorder`, `socket()`, `bind()`, `listen()`, `accept()`, and `select()`.
- Unix Network Programming sampling: The book, "Unix Network Programming" by W. Richard Stevens is considered the quintessential guide to programming network applications (clients or servers) on Unix. It's on reserve at the library. You might consider owning or borrowing a copy of the complete text if you want more background material or examples.

2 Requirements

Your proxy must implement a subset of the HTTP/1.0 protocol. The protocol is described in RFC 1945. In this section, we describe the steps that your server must perform whenever a client request arrives.

Upon startup, your server should read a configuration file, open a TCP socket on a port indicated in the configuration file and start accepting requests. Whenever a connect request arrives, you should translate the client's IP address to a host name and check whether the client is allowed to access the Web. If the client is not authorized, your server must refuse the connection immediately – those marketing dweebs have been cut off the net entirely until further notice!

2.1 Processing an Incoming Request

For authorized clients, you must accept the request and parse it. A request consists of a request line, followed by request header fields. The request line has three fields: method, URL and protocol version (which in your case will be the string `HTTP/1.0`). Your server must implement the `GET` and `POST` methods. In addition to these basic http methods, we require that you implement one non-standard method, `QUIT`, to help us grade your work. The purpose of this method should be obvious: whenever you get a `QUIT` request, your server should exit immediately.

After determining the method, the next thing you need to do is parse the URL (uniform resource locator). The URL notation is almost as complicated as the HTTP protocol itself. The complete details on the formation of URLs are available in RFCs 1738 and 1808, both of which are available in the class handouts directory and via the class web page, but as with the HTTP protocol, we require you to support only a subset of the legal URLs. In particular, your server only needs to understand the `http:` scheme; if the request contains any other scheme (e.g., `file:`, `ftp:`, or `gopher:`), it should be rejected.

Part of the URL is the host name of the machine serving the requested web page (e.g., for the URL `http://www.cs.rochester.edu/courses`, `www.cs.rochester.edu` represents the host name of the machine exporting the web page). Your server must differentiate between Web locations; some locations can be blocked entirely, while other locations might be redirected to alternate servers. Before contacting the destination server, your proxy must consult the configuration file to determine whether access to the requested location is blocked or redirected to an alternative location. Requests to URLs that refer to blocked locations cause your server to return an error code to the client.

If the request location is not blocked, your server should attempt to connect to the server specified in the server URL, or the new server if a redirection was performed. The default port for HTTP access is TCP port 80; note however that the `http:` scheme allows for alternate port specification (as in `http://www.foo.com:1223/mumbo.html` – in this case the desired TCP port is 1223). You must honor alternate port requests. When the connection is established, your proxy server should forward the original request to the destination server.

As already noted, an HTTP request consists of a request line, followed by header fields.

Header fields consist of a keyword followed by a colon and an arbitrary sequence of tokens (field value) terminated by a CRLF sequence[2]. Your proxy can be instructed to change the contents of header fields on the fly, as the request is forwarded. The strategy employed here is to choose the appropriate transformation as a function of the header keyword and do the substitution of the field value (see below).

2.2 Processing a Reply

The connection established with the remote server is used both to transfer the forwarded request and receive a reply (i.e., your proxy does a `write` and a `read` on the same socket descriptor). When the remote server responds to your request, you will receive a response message consisting of a status line, a sequence of response header fields and a response body.

The HTTP protocol specifies the data type it is sending in the response body. The approach is similar to the MIME conventions: header field `Content-Type` describes the returned document using the *media type* convention, defined in the MIME standard. An example of a media type is `text/html`.

The final piece of functionality that your proxy must implement is *media filtering*: it must be possible to apply external filters (Unix programs) to the reply body based on its media type. For example, it must be possible to pass all text data through a four-letter-word-removal filter, or pass

[2] Yes, the HTTP protocol, like many other Internet application protocols uses the CRLF sequence to represent an end of line. You should correctly implement this requirement, i.e., just a newline or a carriage return is not sufficient to terminate a line.

all composite (multipart) messages through an image removal filter. Your proxy server will *not* need to perform the actual filtering! We will provide a set of filters, or you are welcome to test with some of your own. However, your proxy must examine the media type of returned documents to determine if the message body should be handed to a filter that has been registered for that type of data, and if so, replace the returned message body with the output of the specified filter. The interface for connecting to filter programs is described below.

2.3 Other Requirements

Your server should be capable of handling multiple connections concurrently. This means that you must be very careful when using system calls that could block the server (i.e., reading or writing from/to a socket). There are two common ways to handle multiple connections concurrently: (i) forking off a child process for each connection and (ii) using the `select(2)` system call to perform asynchronous operations. You may use whichever technique you prefer. Note that you are not required to fork a process or use select during the resolution of host addresses using the DNS interface. Although functions such as `gethostbyname` or `gethostbyaddr` block waiting for an answer, you may block the server while performing these operations.

The HTTP protocol defines *persistent connections* that are used to send and receive more than one request.

If you investigate HTTP requests produced by various client browsers, and Netscape Navigator in particular, you will see the header field `Proxy-Connection: Keep-Alive`. This indicates that you should not close the connection when the last byte of the reply is sent. We do *not* require that you implement persistent connections – you may close socket connections as soon as you finish using them.

3 The Configuration File

The runtime behavior of the ACME HTTP proxy server is controlled by means of a configuration file. Figure 1 illustrates a sample proxy configuration file.

A configuration file consists of a sequence of lines containing comments or commands. Comments are initiated by a pound sign (`#`) and extend to the end of the line. Line continuation is done by means of a backslash character (`\`) . Each non-empty line consists of a sequence of tokens that make up a command (clause). If a token contains white space, it must be enclosed in double quotes (`‘‘’’`). No special escape sequences are recognized in strings.

3.1 Configuration File Contents

There are six clauses that can appear in a configuration file:

`port` *n* If present, this clause specifies the TCP port number your server should listen on. If omitted, the port number is indicated by the environment variable `PROXY_PORT`. If the clause is not present and `PROXY_PORT` is undefined, the default http port (80) should be used.

`refuse` *pattern* This clause specifies hosts from whom connections should be refused. Argument *pattern* is an *extended regular expression*; for more information about regular expressions, consult the `regex(5)` man page.

`block` *pattern* This clause specifies that all URLs matching *pattern* are blocked – an attempt to access them should return an error message.

```
# proxyrc -- sample HTTP proxy configuration file
#
# This clause defines the port where proxy should listen on.
# The value is accessible via the proxy_port() macro in *host* order.

port 5000

# A list of refused clients and blocked/redirected URLs goes here.
# The string that should be submitted to proxy_location() should
# consist of the <host> part followed by '/' and the <location> part.
# Note that you should (a) extract port number (if any) and append
# a trailing '/' even if the <location> part of the URL is empty.

refuse marketing.acme.com               # blocks complete 'marketing' group

block www.nytimes.com                 # news surfing is not allowed

redirect www.netscape.com/(.*) \
         "www3.netscape.com/\1"         # www3 server is preferrable to www.

# This clause probably doesn't have any real use, except if you try to
# fool server or browser into believe something that isn't there.
# This clause, for example, rewrites 'User-Agent' header fields in
# such a way that any substring 'Netscape foo.bar ...' is replaced
# by 'Mozilla foo.bar ...' (Remember, it's spelled M-o-z-i-l-l-a :-)

rewrite User-Agent "Netscape (.*)" "Mozilla \1"

# This is how you filter incoming data.
# The HTTP protocol defines a bunch of content-types which have
# exactly the same structure as used in MIME mail.
# You have to parse the 'Content-type' header field, extract the media
# and pass it to proxy_filter().  If a filter has been defined, you'll
# get as a return the string representing a shell command to be executed.
#
# Sample entry here says that all text should be piped through an imaginary
# scramble filter which groks text on stdin and dumps a swedish-chef
# version of it on stdout.  Note that the complete command line has
# to be enclosed in double quotes if spaces are embedded.

filter text/(.*) "scramble -swedish -\1"
```

Figure 1 Sample proxy configuration file

Note that you must parse the original URL to extract the destination machine before attempting to match the request URL to *pattern*. In other words, you should strip off the scheme prefix (`http:`), port specifier, and any other extraneous information from the requested URL prior to performing the regular expression check. Note that a "/" marks the end of the host part of the URL (possibly with a port extension), so stripping out the hostname should be straightforward.

`redirect` *pattern subst* The redirect operation is similar to the block operation, except that instead of returning an error message, you should redirect the request to a new location. This is done by *substituting* the hostname and port number in the request URL with one specified in the configuration file. The substitution string *subst* specifies the new location. It is possible to include substrings of the original string in the replacement string. Substrings are delimited in the *pattern* string by using parentheses; substrings of the original URL are available as escape sequences `\1`, `\2`, ... Escape sequence `\0` returns the original string.

`rewrite` *hfield pattern subst* This clause specifies that you should perform a substitution on the request header field *hfield* by replacing the field value with the substitution string specified in the configuration file.

`filter` *media filter* This clause specifies that for each media type matching *media* pattern, a *filter* command is to be executed by passing the message body to standard input and forwarding the filter's output to the client.

3.2 Configuration Library

Since the parsing of configuration files and dealing with regular expressions is not an important part of the networking experience, we have provided a library of utility functions to perform most of the grunge work for you. The following functions are supported by the utility library:

- `int proxy_init(`*proxy_t* p, const char* rcfile*`)` : This command reads the configuration file *rcfile*; it returns `PROXY_OK` if the file was successfully read, and otherwise prints error diagnostics on stdout and returns `PROXY_ERROR`. If the *rcfile* is `NULL`, the environment variable `PROXY_RCFILE` is used to determine the configuration file name; if the variable isn't defined, a default filename `proxyrc` is used.

- `int proxy_port(`*proxy_t* p*`)` : This command returns the TCP port number where the proxy should listen for incoming connections. Port number is in *host* order.

- `int proxy_client(`*proxy_t* p, const char* host*`)` : This command checks whether a particular client is allowed to send HTTP requests. String *host* is the canonical host name from *struct hostent*. The return value of `proxy_client()` is `PROXY_OK` if client is allowed to access the Web, and `PROXY_REFUSE` if not. If a client is refused, error code 403 (Forbidden) should be returned.

- `int proxy_location(`*proxy_t* p, const char* loc, char* buf, size_t bufsiz*`)` : This command checks the HTTP location; it returns `PROXY_OK` if the location is neither blocked nor redirected, `PROXY_BLOCK` if the location is on the blocked list, and `PROXY_REDIRECT` if a new location is to be used instead. In the case of the URL being redirected, *buf* contains a zero-terminated URL describing the new location. If the location is blocked, error 403 (Forbidden) should be returned.

  String *loc* should be constructed as follows: from the original HTTP URL, remove the scheme prefix `http:` and port specifier (if any); *loc* should contain only the host name and absolute path. The host name should be followed by a slash character, even if path in the original URL is empty. If the return value indicates that a new location should be used, the result string should be regarded as a full URL (without the `http:` prefix).

- `int proxy_hfield(`*proxy_t* p, const char* key, const char* value, char* buf, size_t bufsiz*`)` : This command rewrites request header fields.

  The string *key* is the header keyword – the first token before the colon. The string *value* points to the remainder of the header field, i.e., the characters after the colon. This function returns `PROXY_OK` if the modification was successful, and `PROXY_ERROR` if the buffer was too small. The new header value (without the *key* prefix) is available in *buf*.

- `int proxy_filter(`*proxy_t* p, const char* media, char* buf, size_t bufsiz*`)` :

  This command checks to see whether a filter has been specified for *media*. It returns `PROXY_OK` if no filter has been defined, `PROXY_FILTER` otherwise. If a filter has been installed, *buf* contains a string describing the filter command to be executed. Note that the command is not split into individual arguments – to execute the command, you should either construct the argument list yourself or execute `/bin/sh -c` *command*.

The file `proxy.h` contains the C header file declarations and definitions needed to use the library routines. The library code itself can be found in the files libproxy.a and librx.a.

Different versions of the library files can be created for different architectures and operating systems. In general, it is important that you use only a single architecture throughout the project (e.g., do all of your development on SunOS or Solaris or ...) or carefully clean up your links to library files and .o files whenever you switch platforms. The directory currently contains the SunOS (and Linux on the undergraduate side) versions of the library files.

4 Your Assignment

To complete this assignment, you need to implement an ACME proxy server that runs on a machine in the undergraduate (or graduate, if you are a CS graduate student) lab, and provides the functionality described above.

The ACME proxy server project will constitute approximately 20% of your final course grade. It is a major project, and you should get started on it *immediately*! Be sure to *handin* the source code files, your makefile, and the executables. Be sure to *handin every* file that we will need in order to recreate your executable, including all header files, `.c` files, and `makefile`'s. Your executable should be called `proxy`. Also be sure to *handin* any external documentation that you've written (e.g., a `README` file) and comment your code thoroughly and clearly.

To *handin* your assignment, you should -

- Put all the files relating to the assignment in one directory. Don't put anything not related to the assignment in that directory.

- Change to the directory and execute "/u/cs257/bin/TURN_IN .",
  or alternatively, "/u/cs257/bin/TURN_IN path_name", where path_name is the full path name of the directory where your project files are. This will copy all the files in the stated directory into the cs257 account. Don't execute this command until you are really done, because you cannot update the files once they have been created.

This project will be graded primarily on correctness, efficiency, and clarity of the code. Correctness will constitute 75% of the project grade, while "software engineering" issues will constitute the other 25% of the grade. The correctness portion of the grade will be based on the extent to which your proxy server supports the specified operations and passes our tests. The software engineering portion of the grade will be based on the "quality" of your solution, including such issues as (i) clarity of your overall design and organization, (ii) clear documentation (both internal and external), (iii) good coding practice (understandable variable names, well-organized functional decomposition and modular design, etc.), and (iv) the presence (or lack thereof) of grossly inefficient code and algorithms. Note that hacks that obscure the code will not be considered good programming practice.

This project is a *group* project - groups can consist of one or two people. You are on your own to form your groups — feel free to use the class newsgroup to help find partners. If you form a two-person group, let me know who your partner is via e-mail. Part of your external documentation should be a discussion of who did what part of the work. In addition, we strongly encourage you to set up a time to meet with the TA to discuss your project design prior to starting on your serious coding (i.e., a miniature design review).

5 Logistics

Your first step should be to read the relevant documentation cited above. Make sure you feel comfortable with what this assignment entails before spending an inordinate amount of time writing code. A little bit of careful design can go a long way!

Directory `~cs257/projects/proxy` is the base directory for this project. You should start by copying all the files in the `skel` directory in a directory where you intend to work.

Relevant documentation for this project is in the `doc` subdirectory. You might want to poll it every now and then for new information. Additional documentation and links are on the class home page, and I will use the class newsgroup for updates.