Computer Networks- Home Assignment-1
Mini Web Server

Instructor: Maya[illegible]andey

Assigned on: 20-08-2010

Submission Deadline: 04-09-2010

Abstract

In this assignment, you are going to write a web server. This web server will be single threaded and non-blocking, and will handle static content. This assignment is to be done in groups, and the work you submit must be your own.

Introduction

During the course of this project, you will do the following:

- Learn and implement a small subset of the Hypertext Transfer Protocol (HTTP)
- Become familiar with socket programming
- Write a web server
- Get very good at debugging and testing network programs

Project specifications

Background

An HTTP web server is a program that listens for incoming TCP connections (typically, but not always, on port 80). Client requests and server replies flow on each connection in a mixture of human-readable text and binary data until either the client or the server decides to close the connection. The HTTP protocol has many optional features that allow an implementation to be bewilderingly complicated and bug-ridden.

The web server

This is the main component of the project. Your server must be HTTP/1.1 compliant (see RFC 2068). In that it must support GET and HEAD requests, as well as persistent connections. Note that persistent connections are not listed the HTTP/1.0 specification.

Invocation

Your web server program must accept the following command line options, in any arbitrary order. If an argument is left out when running the server, then use the specified default.

- -p port: Overrides the default port the server should run on.
- -root path: Sets the root directory of the web server. This is where the files are served from. For example, if the server is run with -root. /networks/website, then a request for http://www.myserver.com/index.html will result in serving the file. /networks/ webs ite/index.html. If this option is not present, than use. /www as the default root directory.
- -debug: When this option is present, you may print out debugging messages to the console (stdout). When this option is not present, your program may not print any output to the console.

HTTP Operations

HTTP is a fairly simple protocol. Clients send the server a request and the server processes that request and send back a response. The format of an HTTP request is <method>< request-uri> <http-version>, followed by an arbitrary number of additional header lines, and terminated with two CRLFs (\r\n); one on the last header line, and one by itself on the next line. HTTP/1.1 defines seven methods: GET, HEAD, POST, PUT, DELETE, TRACE, and OPTIONS. You will only need to support the first two, GET and HEAD. The request URI field is the path to the object the client wants (RFC 1630 has more information about the format of a URI). Finally, the http version field tells the server what version of HTTP the client supports, and will be HTTP/1.1 in your assignment.

The format of an HTTP response is <http-version><status-code><reason-phrase>, followed by an arbitrary number of additional header lines, and terminated with two CRLFs; one on the last header line, and one by itself on the next line. The HTTP version is the same as above. The status code is an integer representing the action the server took, or an error code if something went wrong. For example, you should be familiar with the ubiquitous 404 status code, meaning a file was not found. Section 6.1.1 of RFC 2068 gives a complete listing of status codes. You will only need to implement a small subset of these, namely, 200, 400, 404, and 501. You may also want to implement 500, since it may help with debugging. Finally, the reason phrase is a piece of English text describing the status code. For a status code of 200, the reason phrase will be OK. RFC 2068 gives the reason phrases for all of the status codes.

Below is more information on the two HTTP methods you need to support.

1. GET

   The simplest HTTP operation of all is GET. This is a simple two-message exchange. The client initiates it by sending a GET message to the server. The message identifies the object the client is requesting with a Uniform Resource Identifier (URI, see RFC 1630). If the server can return the requested object, it indicates success with an appropriate response (in this case a 200 OK status code along with the requested object). If the server cannot return the requested object (or chooses not to), then it can return any number of other status codes.

2. HEAD

   The HEAD operation is just like a GET operation, except that the server does not return the actual object requested, just the HTTP headers of the response. HEAD is just a short form of header. Clients usually use a HEAD message when they want to verify that an object exists, but don't need to actually retrieve the object. Programs that verify links in web pages and cache servers also use the HEAD operation.

A GET request from the web client will look like this:

```
  GET /index.html HTTP/1.1
Host: some_host:1234

User-Agent: mini-webclient/1. 0 (Unix)

Connection: close
```

Assuming this request is successful; headers for the response from your web server will look like this:

```
  HTTP/1.1 200 OK
Date: Mon, 02 Sep 2010 15:31:31 GMT

Server: mini-webserver/1.0 (Unix)

Content-Length: 94

  Connection: close
  Content-Type: text/html
```

Note that these are just sample headers. A real web browser may emit more headers, such as Accept-Encoding or Accept-Charset, which you will not have to process. Also note that the time is in GMT, and your server should send all timestamps in GMT. You can do this using the gmtime () and strftime () functions.

Persistent connections

Obtaining an HTML document typically involves several HTTP requests to the web server (to fetch embedded images, etc.). In HTTP/ 1.0 and earlier, TCP connections were closed after each request and response, so each resource to be retrieved required its own connection. Opening and closing TCP connections, however, takes a substantial amount of CPU time, bandwidth and memory. In practice, most web pages consist of several les on the same server; so much can be saved by allowing several requests and responses to be sent through a single persistent connection.

HTTP/1.1 enables browsers to send several HTTP requests to the server on a single TCP connection. In anticipation of receiving further requests, the server keeps the connection open for a configurable interval after receiving a request. This method amortizes the overhead of establishing a TCP connection over multiple HTTP requests, and it allows the client to send several requests in series (called pipelining). Moreover, sending multiple server responses on a single TCP connection in short succession avoids multiple TCP slow-starts, thus increasing network utilization and effective bandwidth perceived by the client. For more information on persistent connections in HTTP/1.1, read section 8.1 of RFC 2068.

If an HTTP/1.1 client sends multiple requests through a single connection, the server MUST keep the connection open and send responses back in the same order as the requests. If a request includes the Connection: close header, then that request is the final one for the connection and the server should close the connection after sending the response. Also, the server should close an idle connection after some timeout period (can be anything, but yours should be 15 seconds).

Your server must support these persistent connections. Please remember that a single client may issue additional requests while your server is still reading and the first request. In this case, your server must read in and process all requests before closing the connection.

Multiple connections

A web server that accepts only one connection at a time is probably impractical and definitely not very useful. As such, your server should also be written to accept multiple connections (usually from multiple hosts). It should be able to simultaneously listen for incoming connections, as well as keep reading from the connections which are already open. Note that today's web browsers may open two connections to your server (as per RFC 2068), so your server should be able to handle these multiple connections.

Many of the functions you will use for multiple connections (e.g. accept () and read ()) will block when called; they go to sleep and stall program execution until some data arrives. So, if your server is blocking on accept (), it cannot read () data at the same time. This could lead to starvation of certain clients, which means that they never get served. The reason for this behavior is because the sockets created with socket () are initially set to be blocking sockets.

Fortunately, there is a function call which lets you turn a blocking socket into a non-blocking socket. A read () call on a non-blocking socket will always return immediately with the data in the socket buffer, even if there was no data available. However, in most cases, putting your program on a busy-wait loop looking for data on a non-blocking socket consumes enormous amounts of CPU time and is a bad idea. Instead, the select () function gives you the power to monitor several sockets at the same time, blocking until there is data to be dealt with. It will tell you which sockets are ready for reading, writing and accepting, and which have raised exceptions (if you really want to know). For your server, you will want to use the select () function with blocking sockets.

Your server must deal with multiple connections. In this project, you must use the select () call to implement multiple connections in your server.

Invocation

The following is a summary of the command line options the web client accepts:

- -g url: Issues a GET request for the specified url, and prints the returned results to console (stdout).
- -h url: Issues a HEAD request for the specified url, and prints the returned results to console.
- -d: Outputs the HTTP headers received from the server before outputting the content received.
- -p port: Overrides the default port the client should connect to.

Testing

The goal of this section is to give you a better idea of how your project will be evaluated, and what you should be on the lookout for while writing your code. The following list is not guaranteed to be exhaustive:

1. Make sure your server can handle GET and HEAD requests from the client. Test requests for different types of files. Try sending HTML files, GIFs, JPEGs, QuickTime movies, MPEG movies, Shockwave files, ZIP files, etc. Your server should be able to handle all of these types of files once you write the code to handle a GET request.

2. Make sure your server can handle GET requests from different web browsers.

3. Try requesting files that don't exist, or files in subdirectories.

4. Try requesting a file outside of the server's root directory.

How to succeed in this assignment

Step 1 - Use English, not HTTP. Only use one client.

Write two programs, called mws-client.c and mws-server.c. You will run mws-client.c on one machine. You will run mws-server.c on another machine. You will need to pick a port number for your server. Your client program will need to know this port number (you will need port numbers in the future as well).

Your mws-server.c program should do the following: create a socket and bind, listen on the socket, accept and obtain a new socket descriptor, print the IP address of the client and the port number of the client , read a request from the client, print the client's request, and write a message to the client saying “I got your request".

Your mws-client.c program should do the following: create a socket, connect, and write a request to the socket, where the request says “Hello server, here is my request", read the response from the server and print that response.

Step 1a - Quick Enhancement Make your client send multiple requests on this same connection and have your server read the multiple requests and handle them all.

Step 1b - Quick Enhancement Start up multiple clients on the same machine or different machines and have them all send requests to the server . Do not use the select() call. Instead, try putting a loop around the accept() call. Make sure you understand when this approach works and when it doesn't.

Step 2 -Use English, not HTTP. Use select(). Use multiple clients. Run mws-client.c on multiple machines. Each instance of mws-client.c should send a different message to the server. Your mws-server.c code should use the select() call to figure out which client has sent a request. The server should again reply to the request in simple English. Rather than having the server simply say to the client “I got your request", the server should greet the client with the client's IP and port number. For instance, “Thank you client with IP 128.2.19.5 and port 8000, I got your request and here is my response." This will help you check that your server is doing the right thing.

Step 3 - Use English, not HTTP. Use select(). Use multiple clients. Persistent connections.

Make up a delimiter that ends each client request, e.g. \r\n. Create a scenario where a client might send 2 requests before the server has a chance to do a read. Thus when the server reads the request, the server will really be reading 2 requests. Make sure that your mws-server.c program checks for end-delimiters and can tell when the client has sent multiple requests. This is also a good time to make sure that your server can handle the case where the client request is so long that it can't obtain the whole request with a single read.

Step 4 - Use English, not HTTP. Use select(). Retrieves files. Use multiple clients. Persistent connections.

Now edit the requests in mws-client.c to be of the following format: GET filename \r\n, where filename is a fully specified path, like ./file.html. Make your server try to open the file in the specified directory. If the file does not exist, have your server reply with “Sorry, not there".

If the file does exist, your server should send back the whole file.

Step 5 - Use HTTP. Use select(). Retrieves files. Use multiple clients. Persistent connections. GET.

Repeat Step 4, except that this time, each client should send a GET file HTTP message to the server, and the server should reply with an HTTP message containing the file, or an HTTP message with the appropriate error code. Be careful to include lots of error checking in your program; for example, the server should complain if the request being sent to the server is not written in proper HTTP.

Again, allow each client to request multiple files from the server. Experiment with every odd scenario you can imagine.

Step 6 - Use HTTP. Use select(). Retrieves files. Use multiple clients. Persistent connections. GET. HEAD.

Expand your server to handle HEAD requests.

Step 7 - Use HTTP. Use select(). Retrieves files. Use multiple clients. Persistent connections. GET. HEAD.

Test your web server with a real web browser.