Web Client And Server

Project Part A

Overview

In this part of the project, you and your partner will build a simple web client and a succession of servers to which it can connect. The goal is to slowly introduce you to Unix and socket programming and get you to a stage where you will be able to tackle the subsequent parts of the project. The last server that you will construct will have roughly the same structure as the TCP and IP layers of the network stack you will build. There is also an extra credit server you may build that is structured like a real high performance web server or cache, such as Inktomi's Traffic Server or the Squid cache.

HTTP and HTML

The combination of HTTP, the Hypertext Transport Protocol, and HTML, the Hypertext Markup Language, forms the basis for the World Wide Web. HTTP provides a standard way for a client to request typed content from a server, and for a server to return such data to the client. "Typed content" simply means a bunch of bytes annotated with information (a MIME type) that tells us how we should interpret them. For example, the MIME type text/plain tells us that the bytes are unadorned ASCII text. You will implement a greatly simplified version of HTTP 1.0.

HTML (type text/html) content provides a standard way to encode structured text that can contain pointers to other typed content. A web browser parses an HTML page, fetches all the content it refers to, and then renders the page and the additional embedded content appropriately.

HTTP Example

In this project, you will only implement HTTP, and only a tiny subset of HTTP 1.0 at that. HTTP was originally a very simple, but very inefficient protocol. As a result of fixing its efficiency problems, modern HTTP is considerably more complicated. It's current specification, RFC 2616, is over a hundred pages long! Fortunately, for the purposes of this project, we can ignore most of the specification and implement a tiny subset.

The HTTP protocol works on top of TCP, a reliable stream-oriented transport protocol, and is based on human-readable messages. Because of these two facts, we can use the telnet program to investigate how HTTP works. We'll use telnet in the role of the client and www.cs.northwestern.edu in the role of the server. The typed content we'll transfer is the CS department's home page. This is essentially the same as fetching the home page using your favorite web browser.

The following shows what this looks like for the URL http://www.cs.northwestern.edu/index.html. The text in bold is what you would type, while the text in italic are the parts of the response that we'll talk about.

```
$ telnet www.cs.northwestern.edu http
Trying 129.105.99.240...
Connected to Godzilla.cs.nwu.edu (129.105.99.240).
Escape character is '^]'.
GET /index.html HTTP/1.0
(blank line)
HTTP/1.1 200 OK
Date: Tue, 12 Sep 2000 14:55:18 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Wed, 06 Sep 2000 22:57:54 GMT
ETag: "194822-2ab9-39b6cbf2"
Content-Length: 10937
Accept-Ranges: bytes
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<html>
…

</html>


Connection closed by foreign host.
$
```

The first thing to notice is that we are opening a TCP connection to port 80 (telnet looks up the service "http" in the list /etc/services and discovers that it is a TCP service that runs on port 80.) Telnet does a DNS lookup on the host www.cs.northwestern.edu and finds that it is at IP address 129.105.99.240. It then does a reverse lookup on the IP address to find the canonical name of the machine (www is an alias for godzilla). It then opens the connection and lets us type.

"`GET /index.html HTTP/1.0`" is the most basic form of an HTTP 1.0 request, and the form that you will implement. It says "please give me the file that you think of as /index.html using the 1.0 version of the HTTP protocol." The blank line demarcates the end of the request. This is necessary because a more complex request may place further conditions (on additional lines) on what the client is willing to accept and how it is willing to accept it.

The response always begins with a line that states the version of the protocol that the server speaks ("`HTTP/1.1`" in this case), an error code ("`200`"), and a textual description of what that error code means ("`OK`"). Next, the server provides a bunch of information about the content it is about the send as well as information about itself and what kinds of services it can provide. The most critical lines here are "`Content-Length: 10937`", which tells us that the content will consist of 10937 bytes, and "`Content-Type: text/html`", which tells us how to interpret the content we shall receive. A blank line demarcates the end of the response header and the beginning of the actual content. After the content has been sent, the server closes the connection.

Part 1: HTTP Client

Write a client program that supports the following command line and semantics.

```
http_client server_rname server_port server_path
```

When run, `http_client` should open a connection to port `server_port` on the machine `server_name`, and then send an HTTP request for the content at `server_path`. It should then read the HTTP response the server provides. If the response is that `server_path` is valid and includes the data, `http_client` should write the data out to standard out and exit with a return code of zero. You can then view this output using a web browser such as netscape or lynx. If there is an error, `http_client` should write the response to standard error and exit with a return code of –1. For example,

```
http_client www.cs.northwestern.edu 80 index.html
```

should print the CS Department's home page to standard out and return zero, while

```
http_client www.cs.northwestern.edu 80 junk.html
```

should print the response to standard error and return –1.

Part 2: Connection-at-a-time HTTP Server

Write an HTTP server that handles one connection at a time and that serves files in the current directory. This is the simplest kind of server. The command-line interface will be

```
http_server1 port
```

You will then be able to use http_client, telnet, or any web browser, to fetch files from your server. For example, if you run

```
http_client host port http_server1.cc
```

you should receive the contents of your source file.

It is important to note that you will not be able to use port 80. Ports less than 1500 are reserved, and you need special permissions to bind to them.

Your server should have the following structure:

1. Create a TCP socket to listen for new connections on (What packet family and type should you use?)
2. Bind that socket to the port provided on the command line. We'll call this socket the accept socket.
3. Listen on the accept socket (What will happen if you use a small backlog versus a larger backlog? What if you set the backlog to zero?)
4. Do the following repeatedly
    a. Accept a new connection on the accept socket (When does accept return? Is your process consuming cycles while it is in accept?) Accept will return a new socket for the connection. We'll call this new socket the connection socket. (What is the 5-tuple describing the connection?)
    b. Read the HTTP request from the connection socket and parse it. (How do you know how many bytes to read?)
    c. Check to see if the file requested exists.
    d. If the file exists, construct the appropriate HTTP response (What's the right number?), write it to the connection socket, and then open the file and write its contents to the connection socket.
    e. If the file doesn't exist, construct a HTTP error response and write it back to the connection socket
    f. Close the connection socket.

Part 3: Simple Select-based Multiple-connection-at-a-time Server

The server you wrote for part 2 can handle only one connection at a time. Try the following. Open a telnet connection to your `http_server1` and type nothing. Now make a request to your server using your `http_client` program. What happens? If the connection request is refused, try increasing the backlog you specified for listen in http_server1 and then try again. After `http_server1` accepts a connection, it blocks (stalls) while reading the request and so is unable to accept another connection. Connection requests that arrive during this time are either queued, if the listen queue (whose size you specified using listen) is not full, or refused, if it is.

Consider what happens if the current connection is very slow, that it is running over a modem link, for example. Your server is spending most of its time idle waiting for this slow connection while other connection requests are being queued or refused. Reading the request is only one place where `http_server1` can block. It can also block on waiting for a new connection, on reading data from a file, and on writing that data to the socket.

Write an HTTP server, `http_server2`, that avoids just two of these situations: waiting for a connection to be established, and waiting on the read after a connection has been established. You can make the following assumptions:

- If you can read one byte from the socket without blocking, you can read the whole request without blocking.
- Reads on the file will never block
- Writes will never block

It is important to note that if you have no open connections and there are no pending connections, then you should block. The components of the networking stack you will build later in the quarter will also make these assumptions when they communicate with each other. In that case, the assumptions are reasonable since they will all run on a single machine communicating with a mechanism called fifos or named pipes (man fifo).

To support multiple connections at a time in `http_server2`, you will need to do two things:

- Explicitly maintain the state of each open connection
- Block on multiple sockets, file descriptors, events, etc.

It is up to you to decide what the states of a connection are and how you will maintain them. However, Unix, as well as most other operating systems, provides a mechanism for waiting on multiple events. The Unix mechanism is the `select` system call. `select` allows us to wait for one or more file descriptors (a socket is a kind of file descriptor) to become available for reading (so that at least one byte can be read without blocking), writing (so that at least one byte can be written without blocking), or to have an exceptional condition happen (so that the error can be handled). In addition, `select` can also wait for a certain amount of time to pass. We have provided you with a version of select called `minet_select`. `minet_select` has precisely the same semantics as `select` (man select), but it makes it easy to choose between the kernel network stack and the user-level Minet stack.

Your server should have the following structure:

1. Create a TCP socket to listen for new connections on
2. Bind that socket to the port provided on the command line.
3. Listen on that socket, which we shall call the accept socket.
4. Initialize the list of open connections to empty
5. Do the following repeatedly
    a. Make a list of the sockets we are waiting to read from the list of open connections. We shall call this the read list.
    b. Add the accept socket to the read list. Having a new connection arrive on this socket makes it available for reading, it's just that we use a strange kind of read, the accept call, to do the read.
    c. Call `minet_select` with the read list. Your program will now block until one of the sockets on the read list is ready to be read.
    d. For each socket on the read list that `minet_select` has marked readable, do the following:

i. If it is the accept socket, accept the new connection and add it to the list of open connections with the appropriate state
ii. If it some other socket, performs steps 4.b through 4.f from the description of http_server1. After closing the socket, delete it from the list of open connections.

Test your server using telnet and `http_client` as described above.

Extra Credit: Complex Select-based Multiple-connection-at-a-time Server

`http_server2` can handle multiple connections at a time, but there remain a number of places where it can block. These are implicit in the assumptions we have made. In general, almost any system call can block. In particular, if `select` tells us that a file descriptor is readable, it only means that at least one byte can be read. Reading any subsequent byte may block. The same holds true for writes.

To avoid unnecessary blocking, then, the program must check each system call that may block, and certainly read and write, before it executes the system call. Does this mean that we have to call `select` before we read or write each byte? Not necessarily. We can instead using non-blocking I/O. If we set a file descriptor to operate in non-blocking mode, then system calls on that file descriptor will fail with an `EAGAIN` error instead of blocking. `EAGAIN` means "I can't do that right now because doing so would block you and you asked me never to let that happen." To read more about non-blocking I/O, see the man page for `fcntl`. `fcntl(fd,F_SETFL,O_NONBLOCK)` is one way to set a file descriptor to non-blocking I/O. To learn how to retrieve error codes from system calls, check out the man page for `errno`.

For extra credit, you can build an HTTP server, `http_server3`, which uses `select` and non-blocking I/O to provide availability even in the face of blocking on any of the reads, writes, and accepts, as well as dealing with partial reads and writes. The overall structure of the code is as follows.

1. Create a TCP socket to listen for new connections on
2. Bind that socket to the port provided on the command line.
3. Listen on that socket, which we shall call the accept socket.
4. Initialize the list of open connections to empty. You should associate with each connection its state and the file descriptor for the file it is reading, etc.
5. Do the following repeatedly
    e. Make a list of file descriptors we are waiting to read from the list of open connections. This will include both sockets and file descriptors for files you are in the process of reading. We shall call this the read list.
    f. Add the accept socket to the read list.
    g. Make a list of sockets we are waiting to write from the list of open connections. We shall call this the write list.

h. Call `minet_select` with the read list and the write list. Your program will now block until one of the sockets on the read list is ready to be read or written.
i. For each socket on the read list that `minet_select` has marked readable do the following
    i. If it is the accept socket, accept the new connection, set its socket to be non-blocking, and add it to the list of open connections with the appropriate state
    ii. If it's some other socket, look up its connection in the list of open connections, figure out how much you have left to read, and then read until you get an `EAGAIN` or you've read the whole request.
        1. If you get the `EAGAIN`, update the connection's state accordingly.
        2. If you've read the whole request, open the file, set its file descriptor to non-blocking, add it to the connection state, and update the state to note that you're in the process of reading the file.
    iii. If it's some other file descriptor, look up its connection in the list of open connections, figure out how much you have left to read, and then read until you get an `EAGAIN` or you've read the whole file.
        1. If you get the `EAGAIN`, update the connection state to reflect you much you have read.
        2. If you've read the whole file, close the file, update the connection state to reflect that you are ready to start writing the contents to the socket.
j. For each socket on the write list that `minet_select` has marked writeable do the following.
    i. Look up its connection in the list of open connections, figure out how much you have left to write, and then write until you get an `EAGAIN` or you've read the whole request.
        1. If you get the `EAGAIN`, update the connection state to reflect how much you've written.
        2. If you've written the whole file, close the socket and remove the connection from the list of open connections.

Mechanics

- Your code must use the Minet sockets layer that we will provide. This layer can be set to pass through calls to kernel sockets interface, or it can pass calls to the Minet user-level stack (the TCP and IP parts of which you will write later in the quarter!) Here you will be using it in its kernel-pass-through mode. There is a separate handout on compiling and linking with the Minet sockets layer.
- Your code must be written in C or C++ and must compile and run under Red Hat Linux 6.2 on the machines in our cluster. In particular, we will compile your code using GCC 2.95.2 and GNU Make 3.78.1, which are installed on the lab machines. You must provide a Makefile. We will expect that running "make" will generate the

executables `http_client`, `http_server1`, `http_server2`, and (if you decide to do the extra credit), `http_server3`.

Things That May Help You

- Section 2.2 of your textbook provides more information about HTTP and shows another example of simple HTTP interactions.
- You can (and should) play with www.cs.northwestern.edu or some other web server using telnet to port 80.
- RFC 2616, which you can find via http://www.ietf.org, is the specification for HTTP 1.1. This can be daunting, but it is the standard. The specification of HTTP 1.0, RFC 1945, is simpler and probably as relevant for this assignment.
- Section 2.6 of your textbook gives examples of writing TCP clients and servers in Java, while Section 2.8 gives examples of a simple web server written using Java.
- The handout "Sockets in a Nutshell".
- The handout "Minet Sockets"
- The handout "Make in a Nutshell"
- Rick Steven's Unix Network Programming book has lots of source code examples.
- The C++ Standard Template Library
- The micro_http server (http://www.acme.com/software/micro_httpd/) can show you how to parse and generate HTTP requests and responses.
- CVS (http://www.loria.fr/~molli/cvs-index.html) is a powerful tool for managing versions of your code and helping you and your partner avoid stepping on each other's toes.

Other kinds of servers

Complex select-based multiple-connection-at-a-time servers, such as the one you can build for extra credit, can provide very high performance. For this reason, this model is used in server and cache engines that must supports thousands of requests per second and more. Inktomi's Traffic Server and the Squid Cache use this approach. If you've done Windows, Mac, or X11 programming, you'll notice that select-based programming bears a strong resemblance to the event-driven model they have at their core.

We will use a `select`-based approach (albeit the simplified variant much like that in Part 3) in the remainder of this class. We do this for two reasons: because we need to support multiple connections at a time (TCP needs to talk to both the Socket layer and the IP Layer, for example), and because we want to stick with a single thread of control given that you may have never been introduced to multi-process or multi-thread programming before. Nonetheless, it is interesting to note the other approaches that are possible.

Generally, these other approaches involve having multiple threads of control. This simplifies design—it is OK for one thread of control to block because there are other threads of control that can execute. However, it also complicates design because these threads of control may have to synchronize to share data and communicate with each other.

Multi-process servers: One approach is to have each connection handled by a separate process. One process accepts connections and then hands them off to other processes to be serviced. The simplest, but most expensive, way to do this is to create a new process for each connection. Stevens gives examples of such servers on Unix, where they are implemented using the fork system call. More often, a "pool" of processes is started when the server is initialized. Each of these processes then iteratively handles connections passed to it from the process that accepts new connections. If all of these processes are busy when a new connection arrives, the connection is refused. The Apache web server works like this.

Multithreaded servers: A thread is a thread of control within a process, either implemented in the kernel or at user-level. Threads are typically much cheaper to start and switch than processes. Furthermore, threads within a process can communicate much faster and more easily than two processes can. However, they can also get in each other's way far more easily. Otherwise, multithreaded servers are similar to multi-process servers—a new thread can be started for each connection, or a thread from a thread pool can be assigned to the connection. Microsoft's Internet Information Server is a multithreaded server (it also uses multiple processes to better contain failures.)