Due Date: Wednesday, May 6, 2015, 11:59pm (Late days may be used.)

This project must be done in groups of 2 students.

1 Introduction

This assignment introduces you to the principles of internetwork communication using the HTTP and TCP protocols, which form two of the most widely used protocols in today's Internet. In addition, the assignment will give you some insights into how to construct web services that are based on the popular REST [3, 4] architectural style, and show an example of how to implement a concurrent server that can handle multiple clients simultaneously.

2 Functionality

In this part of the assignment, you will implement a basic HTTP web server that provides access to web services and files. The web server should implement persistent connections as per the HTTP/1.1 [2] protocol.

2.1 System Status Web Service

The system status web service is a basic HTTP web service that publishes a Linux system's status as reported by the kernel via the /proc file system. The web service must provide this information in JSON [1] format.

The web service shall respond to requests for at least the following resources:

Service URL	Example JSON Output	Based On
/loadavg	{"total_threads": "174", "loadavg": ["0.00", "0.00", "0.03"], "running_threads": "1"}	/proc/loadavg
/meminfo	{"SwapTotal": "5799928", "SwapFree": "5793240", "MemFree": "2434304", ... }	/proc/meminfo

In addition, you must support query arguments as per RFC 2616, Section 3.2.2. You must support a 'callback' argument. If given, you must return proper syntax for a JavaScript function call in which the value of the field appears as function name and the JSON object appears as argument. You should check that the argument provided to the 'callback' field consists of only alphanumeric characters, underscore (_), and periods (.). You must ignore all additional argument=value pairs. For example, a request to

`/loadavg?callback=jsonp1258749550540&_=1258749554624`
would return `jsonp1258749550540({"total_threads": "174", "loadavg": ["0.00", "0.00", "0.03"], "running_threads": "1"})`.

(The second field `_` is ignored.)

Note that the order of the attributes within the returned JSON object (e.g., SwapTotal, SwapFree, etc.) is arbitrary. You can see a demo at http://cs3214.cs.vt.edu:9011/loadavg and http://cs3214.cs.vt.edu:9011/meminfo.

You should return appropriate error codes for requests to URLs you do not support.

2.2 Serving Files

Your web server should, like a traditional web server, support serving files from a directory (the 'root') in the server's file system. These files should appear under the `/files` URL. For instance, if the URL `/files/widget/plugins/jqplot.barRenderer.js` is visited, and the root directory is set to the parent directory (widget), the content of the file `widget/plugins/jqplot.barRenderer.js` should be served. You should return appropriate content type headers, based on the served file's suffix. Support at least .html, .js, and .css files; see /etc/mime.types for a complete list.

Make sure that you do not accidentally expose other files by ensuring that the request url does not contain .., such as `/files/../../../../../etc/passwd`.

2.3 Synthetic Load Requests

A system's load average as well as its physical memory use (and the resulting statistics) are influenced by the actions and resource needs of the processes running on it. You should implement a set of paths that serve as entry points for synthetic load requests. When visited, your web server should induce a synthetic load that causes a change in a system's CPU or memory load (which would then be reflected in the data reported by the system status service.) You should implement the following synthetic load requests:

1. `/runloop` When this URL is visited, start a thread or process that spins (loops) for 15 seconds. This loop should be started in parallel, without delaying the response to this request. The expected result will be a temporary increase in the load average. Recall that the load average is a weighted average over samples that represent the number of ready or running threads.

2. `/allocanon` method should force the system to allocate physical memory devoted to anonymous virtual memory. "Anonymous" virtual memory is the memory backing a process's heap or areas mapped via mmap's MAP_ANONYMOUS feature. It's called anonymous because it is not related to a known file on disk. Each request should result in the allocation of 256MB. If a system still has free physical memory,

this should reduce the amount of physical memory by this much, which should be displayed in the memory statistics. If physical memory is exhausted, the necessary physical memory will come from the eviction of pages holding either cached file data or anonymous memory.

Keep in mind that Linux, like all modern operating systems, allocates physical memory on demand. Use mmap() to design a function that has the desired effect. You should store a list of the blocks of memory you've allocated.

3. `/freeanon` method should force the system to deallocate a chunk of physical memory. To that end, you should munmap() the last block of memory you've allocated as a result of a visit to `/allocanon`.

   If all pages of the virtual memory block are still present in physical memory at that time, the expected result is an increase in the amount of available physical memory.

All requests should respond with a short HTML message describing the outcome of the operation. This part of the project is intended to allow for a demonstration of the sysstat web service, and is also intended to give you a deeper understanding of the concepts of CPU and memory load in a system using virtual memory.

2.4 Multiple Client Support

For all of the above services, your implementation should support multiple clients simultaneously. It must be able to accept new clients and process HTTP requests even while HTTP transactions with already accepted clients are still in progress. You should use a single-process approach, either using multiple threads, or using an event-based approach.[1] If using a thread-based approach, it is up to you whether you spawn new threads for every client, or use a thread pool. You may modify or reuse your thread pool implementation from project 2, if this is useful.

To test that your implementation supports multiple clients correctly, we will connect to your server, then delay the sending of the HTTP request. While your server has accepted one client and is waiting for the first HTTP request by that client, it must be ready to accept and serve additional clients. Your server may impose a reasonable limit on the number of clients it simultaneously serves in this way.

2.5 Robustness

Network servers are designed for long running use. As such, they must be programmed in a manner that is robust, even when individual clients send ill-formed requests, crash, delay responses, or violate the HTTP protocol specification in other ways. *No error incurred while handling one client's request should impede your server's ability to accept and handle future clients.*

[1] For the purposes of this project, a multi-process approach is not acceptable.

2.6 Performance and Scalability

New this semester, we will benchmark your service to figure out the maximum number of clients and rate of requests it can support. Note that for your server to be benchmarked, it must obtain a full score in the robustness category first. We will publish a script to benchmark your server.

2.7 Protocol Independence

The Internet is currently undergoing a transition from IPv4 to IPv6. This transition is spurred by the impending exhaustion of the IPv4 address space as well as by political mandates.

Since IPv4 addresses can be used to communicate only between IPv4-enabled applications, and since IPv6 addresses can be used to communicate only between IPv6-enabled applications, applications need to be designed to support both protocols and addresses, using whichever is appropriate for a particular connection. For a TCP/UDP server, this requires to accept connections both via IPv6 as well as via IPv4, depending on which versions are enabled on a particular system. For a TCP/UDP client, this requires to identify the addresses at which a particular server can be reached, and try them in order. Typically, if a server is reachable via both IPv4 and IPv6, the IPv6 address is tried first, then a fallback onto the IPv4 address is performed.

Ensuring protocol independence requires avoiding any dependence on a specific protocol in your code. Fortunately, the socket API was designed to support multiple protocols from the beginning as its designers foresaw that protocols and addressing mechanisms would evolve. For instance, the bind() and connect() calls refer to the addresses passed using the type `struct sockaddr *` which is an opaque type that could refer to either a IPv4 or IPv6 address.

To implement protocol independence, you need to avoid any dependence on a particular address family. Accordingly, you should use the `getaddrinfo(3)` or `getnameinfo(3)` functions to translate from symbolic names to addresses and vice versa and you should avoid the outdated functions `gethostbyname(3)`, `getaddrbyname(3)`, or `inet_ntoa(3)` or `inet_ntop(3)`.

An excellent tutorial on how to write protocol independent network code is given in this resource: `http://www.akkadia.org/drepper/userapi-ipv6.html` The information given in this guide replaces the (unfortunately) out of date chapter in our textbook.[2]

Ensuring that your server can accept both IPv4 and IPv6 clients can be implemented using two separate sockets, one bound to either family. Two separate threads can then be devoted to these sockets to accept clients that connect using either of the two protocol families.

[2]The 3rd edition will cover protocol-independent programming.

However, the Linux kernel provides a convenience feature that provides a simpler facility for accepting both IPv6 and IPv4 clients. This so-called dual-bind feature allows a socket bound to an IPv6 socket to accept IPv4 clients. Linux activates this feature if /proc/sys/net/ipv6/bindv6only contains 0. You may assume in your code that dual-bind is turned on. [3]

2.8 Widgets

Widgets are HTML elements that, when inserted into HTML documents, function as placeholders for enhanced information or functionality. This functionality is usually provided by supporting JavaScript code that interprets the widgets and their parameters. I have designed two small widgets that can interact with the web service you'll create. These widgets graphically display the load average and memory usage of the machine on which your service runs. They also include the functionality to access the synthetic load request entry points via appropriate AJAX requests.

You can find a demo of the widgets at
http://cs3214.cs.vt.edu:9011/files/index.html. The files served from this directory are available on the course web site. Please consult the HTML code of this file for how to include the widgets into your own page for your testing and demonstration, as discussed in Section 4.3.

2.9 Minimum Requirements

The minimum requirements include the web service functionality discussed in Section 2.1 and support for multiple clients as discussed in Section 2.4. Robust error handling is also required. A test driver will check the minimum requirements.

These minimum requirements can be met using one-thread-per-client, HTTP/1.0-only implementation. Support for protocol independence is not required to meet the minimum requirements.

2.10 Choice of Port Numbers

Port numbers are shared among all processes on a machine. To reduce the potential for conflicts, use a port number that is 10,000 + last four digits of the student id of a team member.

If a port number is already in use, `bind()` will fail with `EADDRINUSE`. If you weren't using that port number before, someone else might have. Choose a different port number

[3] I should point out, however, that this will make your code Linux-specific; truly portable socket code will need to resort to handling accepts on multiple sockets.

in that case. Otherwise, it may be that the port number is still in use because of your testing. Check that you have killed all processes you may have started while testing. Even after you have killed your processes, binding to a port number may fail for an additional 2 min period if that port number recently accepted clients. This timeout is built into the TCP protocol to avoid mistaking delayed packets sent on old connections for packets that belong to new connections using the same port number.

3 Strategy

Make sure you understand the roles of DNS host names, IP addresses, and port numbers in the context of TCP communication. Study the roles of the necessary socket API calls. You should exploit a layered design that separates your TCP support code from the HTTP layer.

For the TCP layer, make sure you handle short reads correctly, as discussed in lecture and in the book. To avoid byte ordering related bugs, avoid setting addresses directly in your program - use the getaddrinfo() function instead.

Since you will be using a multi-threaded design, use thread-safe versions of all functions.

Familiarize yourselves with the commands `wget(1)` and `curl(1)` and the specific flags that show you headers and protocol versions. These programs can be extremely helpful in debugging web servers.

Refresh your knowledge of `strace(1)`, which is an essential tool to debug your server's interactions with the outside world. Use `-s 1024` to avoid cutting off the contents of reads and writes (or recv and send). Don't forget -f to allow strace to follow spawned threads. A trick to easily verify that your Content-Length computation is correct is to issue the body of each HTTP response in a separate system call.

The book's student site contains an implementation of a tiny web server you may use as a starting point; you may also use the book's implementation of robust I/O to handle short reads. However, the book code is not implemented in a protocol-independent fashion. Study the examples provided on the course website instead.

4 Grading

4.1 Coding Style

Your service must be implemented in the C language. You should follow proper coding conventions with respect to documentation, naming, and scoping. You must check the return values of all system calls and library functions.

We will pay particular attention to how you separated the implementation of HTTP from your use of TCP sockets. We may use the helgrind checker to check your server for race conditions. Your code should compile under `-Wall` without warnings, the use of the `-Werror` flag as part of CFLAGS should have become a habit by now.

4.2 Submission

You should submit a .tar.gz file of your project, which must contain a Makefile. Your project should build with 'make clean all' This command must build an executable 'sysstatd' that must accept the following command line arguments:

- `-p port` When given, your web service must start accepting HTTP clients and serving HTTP requests on port 'port.' Multiple connection must be supported.
- `-R path` When given, 'path' specifies the root directory of your server, files in it are reachable under the /files prefix.

Submit a file called 'README' that lists group members and briefly describes your DESIGN.

Please test that 'make clean' removes all executables and object files. Issue 'make clean' before submitting to keep the size of the tar ball small. Please use the submit.pl script or web page and submit as 'p4'. Only one group member need submit.

4.3 Grade Breakdown

This project will count for 120 points; meeting the minimum requirements will yield 40 points, the remainder will be given for correctness and performance. In addition, each group member will individually have the chance to demonstrate their webservice from an EC2 instance; instructions for that will be posted separately.

Good Luck!

References

[1] Douglas Crockford. *Introduction to JSON*. http://json.org/.

[2] Roy Fielding, Jim Gettys, Jeff Mogul, H. Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Rfc 2616: Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[3] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[4] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, 2007.