CS-570 Advanced Operating Systems

Assignment 2: Distributed Proxy Server

Due: Monday, 10th. of April, 11:55 PM

This assignment has been divided into two parts. The first part requires you to implement a skeletal web proxy server: a server which acts as an intermediary between a web server and web client. The second part requires you to make the proxy server a distributed program that performs computations on many different machines. You can use your web server code from Assignment-1 as the basis for implementing the web proxy server.

Instructions

- You've to implement this assignment individually
- Start as early as possible. You'll have to spare some time for experimentation and report writing. So schedule your work accordingly
- Be warned: this assignment will require more independent study of technical material than Assignment-1. This tends to be time consuming. Also, you have to decide on the design of your system.
- For signal reference, read the man page for `sigaction` and follow the pointers to other man pages.
- Write the report in Latex and plot graphs using gnuplot. For report writing, use the ACM template given on LMS. The report document should not exceed five pages including references.

1 Part One: A Skeletal Web Proxy Server

Web Proxy Server and Proxy Capable Client

A proxy server is an intermediary between a web server and a client, that is, the proxy server receives requests for pages from different clients that would normally be served by different web servers. *Note that the server can be another proxy server.* You need to support only `GET`. A `GET` request that is addressed to a proxy server has to have an absolute URI (instead of a relative URI). e.g. `http://serverName/fileName`. You client, written in Assignment-1, should support web-proxies. So, your client could be executable as:

```
web_client <#threads> [-proxy proxy_addr] <URL>
```

When web server runs on the same machine as proxy server, it is inefficient to transmit files using sockets. Instead, two processes can use shared memory to pass file efficiently. You need to implement shared memory optimization in addition. Your program should be able to enable and disable this optimization. You can use command-line parameter to do this.

Proxy server wil work as follow:

- The proxy server receives a request and extracts the address of the target web server.
- If the target web server is on the local machine, the proxy server checks to see if the web server is your own web server.
- The proxy server issues a command to the web server and the web server knows to return file data in a shared memory segment. Here is one possible implementation: The proxy server allocates the shared memory, then passes an identifier for the shared memory segment to the web server with a special-purpose request (e.g., you can invent a `"LOCAL_GET"` request). The web server can then supply the (optional) response header and file data in shared memory (in chunks, if the shared memory segment is not big enough). Note that synchronization of some kind will be needed. Moreover, all shared data between the web server and the proxy will have to be in the shared memory segment. Such shared data include, e.g., the size of the file data written, possibly a flag to show whether the web server has yet written the data or not, etc.
- Keep the shared memory segments around so that they can be used in multiple connections. Ideally, you can keep a shared memory segment per proxy worker thread, but you may run into problems with the maximum number of shared memory segments for a single process, which can be very small in some systems. If this is the case, then your threads in the proxy server will have to compete for shared memory segments! Limiting the number of threads in your proxy server is not an acceptable solution (unless the limit is over 50). This would limit the concurrency in the general case just to support a specific optimization. You can implement a synchronized queue of shared memory segments instead. The mechanism you will use to implement shared memory is standard System V shared memory system calls (shmget, shmat, etc.). There is also a Posix shared memory standard (`shm_open`, etc.).

Termination Handling

System V shared memory identifiers are persistent with respect to the process. This means that if a shared memory identifier is not explicitly removed, it will stay in the system even after your server has exited. You should clean shared memory resources after the server(s) have exited. You've to handle the exceptions like `CTRL-C`. Define and install signal handlers for some common signals, so that the server (proxy-server or web-server) cleans up when it exits.

Experiments

Using your client, compare the performance of your optimized and unoptimized web server and proxy server pairs. Show the results of experiments (give actual numbers) and explain. You should consider the following:

- Is using shared memory faster or slower?

- Is it a bandwidth issue or a latency issue? (Think about the kinds of experiments you need to perform to separate the two.)
- How do you interpret the results? (What do they tell you about the operating system and its implementation of cross-process synchronization and/or sockets?)
- Also, compare the performance of the web server and proxy server pair to just the web server. How much slower does the proxy server make the system?

2 Part Two: A Distributed Proxy Server

The end product of this part will be a proxy server that performs some useful manipulation of web pages for multiple clients. You are required to implement an application that shrinks images (to save low-bandwidth). Your proxy server can automatically convert all images it receives into a lower-quality format so that they occupy much less space and can be transferred much faster. For the purposes of the project, it is sufficient to do this for JPEG images only. You can use an existing library for JPEG image manipulation. We recommend the library by the Independent JPEG Group (`www.ijg.org`). You can download tar file, and using the APIs, implement simple routines to read JPEG data from file descriptor (file/socket) and store it in memory buffer at much lower quality.

Performance Requirements

Your proxy-server, just like in part-1 should server multiple clients simultaneously. The proxy server will need to be scalable *both* with respect to handling events without blocking and with respect to performing the computation needed. The **first** requirement is handled by the multithreaded architecture of your proxy and **second** by making the proxy server **distributed**. The distributed mechanism is quite simple: when one of the threads detects that it needs to perform something computationally intensive (e.g., fetch a JPEG file and downgrade it in quality) it should send the parameters for this computation (e.g., the HTTP address of the JPEG file or the JPEG file itself) to one of the distributed entities (e.g., it can pick one at random). The thread should then block, waiting for the results (e.g., the reduced-quality picture data).

Implementation Requirements

The implementation for this part will be an `RPC` (remote procedure call) mechanism. Using RPC has two main advantages. First, it isolates the local code from some of the details of communication (protocols, establishing connections, reading results, etc.). Second, it masks all the details of data representation on the client and server through automatically generated marshalling and unmarshalling routines. Your RPC framework should be quite simple. The only slight complication is related to deallocating memory after the proxy is done with it. The RPC server part of your proxy is dynamically allocating a buffer to hold the returned JPEG image. This buffer has to be de-allocated after the data have been transmitted to the client. (Alternatively, it can be reused for the next request, but if that does not fit, it will have to be de-allocated anyway, so this is even more complicated.) Additionally, the storage area for your RPC data has to be deallocated. You have to do both of these in an RPC `"freeresult"` or using an `"xdr_free"` routine.

Experiments

Test your proxy with a browser to ensure that it works correctly. Browse arbitrary web sites - not pages served by your own server. This is exactly how your proxy will be tested when it gets graded. You can use the same machine as both the server and the client of the RPC call, but you can also add more machines and communicate over the network. You should consider the following:

- Check whether your proxy server (both the RPC server part and the RPC client part) frees memory after it is done with it.
- Try retrieving a large JPEG image many times and watch the amount of memory held by the proxy processes (e.g., use "top"). Does your proxy server free memory correctly? (Include in your report the answer and the experiment you used to test it.)
- Vary the JPEG image size from a few KB to several hundred KB and see whether the image size matters.

3 Deliverables

You'll have to submit the following contents in single *yourRollNo.zip* file. In case of any corrupted file, you'll be marked zero.

- Complete source code of your server, client and experimentation. The code must be either self-explanatory or well commented. It should compile and run without any error or warning.
- README.txt file having your commands with arguments, with which you tested your code. You can also write some supposition(s) or comment(s) related to your approach.
- A comprehensive report of your project, explaining your experiments' observations. It'll be better you explain with some visuals. Try to answer the questions with some specific numbers and justify with your observations.