Project 3 (due 12/06/2012, 11:55PM)
A Distributed Proxy Server

In the previous project, you created a skeletal web proxy server. In this project you will make a proxy server that does something useful. The proxy server will manipulate data in a computation-intensive way, so that under heavy load the bottleneck is not the network but the CPU. Then it makes sense to make the proxy server a distributed program that performs computations on many different machines.

1 GENERAL INSTRUCTIONS

- Read the entire instructions carefully before you start implementing anything!

- Keep your code from Project 2! Large parts of Project 2 are orthogonal to this project: the proxy server you are creating here is optimized for different scenarios than that of Project 2. Hence, you should keep both code bases for future use.

- In this project you can choose the functionality you want to implement. There is a recommended application, but there is also a lot of flexibility.

- **Don't get stuck! If something is hard to implement as suggested, approximate it and go on!**

- If you have any questions, ask me! Use the newsgroup for broad questions.

2 DESCRIPTION OF PROJECT

There is only a single task in this project, but it is quite complex. Part 1, below, describes the functionality of your final result. Part 2 discusses the implementation needs for this proxy server. Part 3 outlines the implementation method you should use. Part 4 discusses experimentation.

1. Web Proxy Server Ideas

The end product of this assignment will be a proxy server that performs some useful manipulation of web pages for multiple clients. Proxy servers are commonly used for caching web pages, but they can have several more interesting applications. There is a recommended application for this project, but you may want to implement something else. Here are some ideas:

- *Shrink images for low-bandwidth links* (**recommended application**): 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 (e.g., over modem/wireless/slower lines). 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. I recommend the library by the Independent JPEG Group (`www.ijg.org`). I have put a link to a tarfile with the source code under the Project 3 Resource page in T-Square. To save you some time, I have also added a second tarfile with code that implements a routine reading JPEG data from a file descriptor (file or socket) and storing it in a

memory buffer at much lower quality. You may want to modify this routine (it is not the most flexible or efficient code possible). Nevertheless, it can be used for most purposes.

- *Encrypting/Decrypting proxy server*: You can create a pair of cooperating proxy servers with one encrypting data (e.g., HTML pages) and the other decrypting them. The idea is that these servers will be put on opposite sides of an unsafe network.

- *Compressing/Decompressing proxy server*: You can create a pair of cooperating proxy servers that compress and decompress data automatically for transfer over a slow channel. There are very good compression libraries available that you can use (e.g., Zlib). In the rest of this handout, I will assume that you are implementing the recommended application.

2. Performance Requirements

Keep in mind that your proxy server (just like the servers in previous projects) will need to be able to serve multiple clients simultaneously. Although any single connection may be slow (e.g., a wireless connection) there can be many of them at once, so 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. For the second requirement, consider that your proxy server is now computationally intensive: uncompressing and re-compressing a JPEG file takes nearly as long as transferring it over a slow local area network. Therefore, it makes sense to make the proxy server distributed.

The structure of such a server can be quite simple. The basic multithreaded architecture is preserved, but there will also be entities running on other machines, ready to perform the computationally intensive manipulation. 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). The distributed entities (i.e., the RPC servers) may be multithreaded, or you may have multiple distributed entities and deal with concurrency in that manner. Nevertheless, the best way to get code that is thread-safe is to declare your distributed entities to be multithreaded.

3. Implementation Requirements

The implementation mechanism for this project 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. On the other hand, RPC has its own complexities. At first sight, it may seem like overkill to use RPC for this project, as no complex data types are transferred (only strings and variable length binary data). Nevertheless, this will make your design quite extensible so that if you later want to add more functionality to your proxy server it will be relatively easy to do so, without messing with the low-level reading and writing of data from sockets. Additionally, since the network protocols are half-hidden, your code may be ready for use with protocols other than TCP (but UDP will probably not be an option, due to restrictions on the amount of data transferred). 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 (if you are using the code from the class webpage, the buffer is allocated through `malloc`). 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 de-allocated. You have to do both of these in an RPC "`freeresult`" or using an "`xdr_free`" routine. RPC mechanisms in general, and Sun RPC in particular were discussed in class.

4. Experiments and extra credit opportunities

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. 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.) Doing extensive performance experiments on lab machines is hard, because you will need to use several of them and the network is slow and heavily shared. Nevertheless, you should design and run a reasonable experiment. You can use your web client program and vary the number of machines on which your RPC server processes run. Is the network the bottleneck for transferring images, or is the decompression/compression process even slower? Vary the JPEG image size from a few KB to several hundred KB and see whether the image size matters. (Because of the way it is written, the 'lowres' code for re-compressing JPEGs at lower quality is quite slow when the image does not fit in the CPU data cache.) If you find out that this is a problem and you modify the code to address it, you will get extra credit. You will also get extra credit based on the services you choose your RPC server to provide, the design of your experiments and result analysis. You will get extra credit for a multithreaded RPC server implementation. Be warned that extra credit is purely at the discretion of the grader! I do not recommend doing the work just for the points. But if you were curious anyway, this will just serve as an extra bit of encouragement.

3 DELIVERABLES

You should turn in the following:

- Your code. Make sure it is obvious how and where to compile and run everything (i.e., supply a README file and a makefile). Submit everything as in Project 2 via T-Square.

- A report outlining your design choices and the results of your experiments. This will likely be much shorter than your previous reports for Projects 1 and 2.

Good luck! Have fun!!!