Assignment - In progress

Add attachment(s), then choose the appropriate button at the bottom.

Title	Project 3 - RPC-Based Proxy Server
Due	Mar 27, 2014 11:55 pm
Status	Not Started
Grade Scale	Points (max 10.0)
Modified by instructor	Mar 14, 2014 1:50 pm

Instructions

Project 3: RPC-Based Proxy Server

Introduction

Remote Procedure Calls (RPC) are a powerful and commonly-used abstraction for constructing distributed applications. Apache Thrift is a modern technology for making remote procedure calls that is both efficient and extensible. Other RPC systems include XML-RPC, JSON-RPC, Sun/ONC RPC, plus web services and systems that build RPC-like functionality on top of RESTful interfaces (such as Java RESTlets).

In this project, you will do the following:

1. Write a simple web 'proxy server' using Apache Thrift.
2. Investigate and implement different caching schemes for your service.
3. Evaluate the performance of your service under different load conditions and using different caching schemes.

This project has two basic goals: First is introducing you to programming with a real remote procedure call system. Second is exploring the principles and performance of caching schemes in a distributed application.

You should work in groups of two for this project.

Specific Details

You should do the following:

1. Implement a simple RPC server and client using Apache Thrift. This should be a "Hello World" style program. You should create a single RPC method that the server implements (e.g., return the current time), create a server that implements, and then create a client program that calls the server method and prints the result. When testing, be sure your client and server can run successfully on two different machines - the point of RPC is that you can call a funciton across the netwrok. For more information about Thrift, see below.
2. Turn your server into a basic proxy server. (Don't worry about the caching part yet.) Modify your server's RPC method to take a URL parameter and return the body of the document located at that URL. Now change the server implementation so that it makes an HTTP GET request for the provided URL, receives the HTTP response, and returns the document body from the HTTP response. You can use the libcurl library to make the HTTP GET request - you do not need to implement any part of the HTTP protocol yourself, unless you want to! More information about using libcurl is below. To test this, you can modify your client to take a URL on the command line, call the RPC method on the server, and print the resulting body text. Then test it with some familiar URLs.
3. Now that you have a proxy server that can retrieve web documents and respond to RPC requests for a document at a URL, you should add a caching mechanism to your proxy server. A web cache in your proxy server can improve performance for much the same reason that the local web cache in your browser does - it's faster to retrieve a document from memory (or disk) than to go across the Internet to get it. Although the distance from the client to the proxy server must still be traversed, the scenario for using this is that they would be on the same local network, making it still much faster to get a document from a web cache on the proxy server than to go to its original source on the Internet. Here are some parameters for your web cache:
    1. You will be asked to implement more than one caching policy (details below), so it will save you trouble if you create

a nice interface that separates your cache from the rest of the proxy server (where the Thrift code and curl/HTTP code are), because the rest of the proxy server doesn't need to change. Then you can just recompile with a different proxy server source file (but the same server code) to create a proxy server that implements a different caching policy. Take a moment to think about a "web cache" abstraction and what kind of methods it would need to provide in order for the proxy server to use it.

2. When the RPC method is called, the server should first check the web cache to see if the document is present, and if so it can return that copy of the document. If not, it will need to fetch the document from the Internet (as it did before you added the web cache), add the document to the web cache (which may require evicting one or more documents form the cache to make room), and then return the document.
3. Implement your web cache as an in-memory cache. Don't worry about saving anything on disk.
4. Your cache should have a size limit (e.g., 1024 KB) that the total size of all cached documents should not exceed. That's only the size of the document bodies; any metadata about the documents maintained by your cache need not count against the limit. Better still, make the cache size a configurable option; this will make your life a little easier when you need to run experiments later (details below).
5. You don't have to worry about canonicalizing URLs for uniqueness. (In other words, don't worry about the fact that *http://www.google.com* and *http://www.google.com/index.html* are actually the same thing.)
6. Your cache lookup shouldn't be completely brain-dead: You shouldn't need to look at every item in the cache to determine if a document is present in the cache. (In other words, cache lookup should be sub-linear, or computational complexity should be *better* than O(N)). This means you should probably use some sort of search tree or a hashing-based approach to lookup.
7. For this project, you are to implement at least three policies (that is, the policies that determine which document(s) to evict from the cache when adding a document for which there is not room):
    1. **Random** (a document is chosen at random from the cache to evict - this is a good "baseline" to compare against
    2. Your choice, one of: **FIFO** or **Least Recently Used** (this can be an approximation based on some fixed history or "absolute" based on time-stamps)
    3. **Your choice** of any one other policy from the list below

4. Before you can measure your cache policies, it would be helpful for the client to be able ot make a bunch of requests instead of just one, so let's do that now. You can use a list of URLs; it's probably most convenient to read the list from a file so you can build and change the list easily, but it's up to you. Then have your client make a number of requests from the URLs in the list.
5. Next you want to think about your metrics. What would be a good measure of success for a cache? There are man options - pick one or two that you think are good choices. When you do your writeup, clearly describe what you metric(s) are and why you chose them. You may discuss ideas for good metrics with other groups - since there are a number of options, not everybody will necessary reach the same conclusions. (Justifying your choice will be as important as the choice itself, though.) Now modify your client and/or server to measure your chosen metrics.
6. When you implemented the URL list for your client, you may have done something simple, like going through the list in order. However, different workloads (e.g., orders of requests) may produce better or worse results on different cache policies. Alternate workloads could include going through the list in some pattern, e.g., A-B-A-C-A-D-A-etc. or choosing URLs at random. You could even implement a statistical distribution so your radom selection might not be uniformly random. Other methods for creating a workload may be possible as well. You should come up with two different workloads to use in your experiments. You should select these workloads such that you get different performance results with them, and also such that your different cache policies produce different performance results. (So you may want to take some time to think about under what conditions each cache police would perform well, and under which they would perform poorly.) Your writeup should clearly describe the two chosen workloads. Implement both workloads so that you can measure your chosen metrics with each. (If it turns out easiest to write two different client programs - one for each workload, then you may do so. Depening on what the workloads are, this may or may not be the case, though.)
    1. To implement workloads, you may find helpful to give an index number to each URL. (The index number doens't have to be in your URL file; you can just assign them in order as the URLs are read from the file.)
7. Now run experiments to measure the performance of your caching proxy. You will evaluate your proxy under both workloads with no caching and then with each of your cache replacement policies. (That should make 8 combinations total.)
    1. "no caching" is equivalent to simply having a cache size of zero for many policies, so you may not need to implement a separate "no caching" policy if you've built your other policies to do the sensical thing when cache size is set to 0.
    2. You should see how the cache size affects the performance for all your cache policies (except, of course, the "no caching" policy).
    3. Your cache should be relatively small compared to the total set of possible requested documents because you want a lot of cache contention.
    4. Your client and server should run on different machines for the experiments, but those two machines should be closer to each other than to any of the requested URLs. That way it should be much faster to contact the proxy and get a cached document than to go to the original server for the document. (This may mean, for example, that if both machines are on the campus network, requesting www.gatech.edu may not be the best choice, but most non-gatech

URLs should be okay.)

Replacement Policy

A cache of any kind must have a replacement policy. This is the policy that is used whenever a new item is added to a full cache (in order to decide which old item to evict). That is, the new item *replaces* some other object in the cache. Some common strategies are:

- Least Recently Used (LRU) -- remove the item that has been requested least recently. The idea is that items requested recently are more likely to be requested in the near future.
- Least Frequently Used (LFU) -- remove the item that is accessed the least frequently. The idea is that the statistical behavior will continue over time, and thus that items used frequently in the past will be used frequently in the future.
- First-In First-Out (FIFO) -- remove the item that has been in the cache the longest. The idea is that pollution by old items should be prevented. (This is different from LRU because it measures when the item was added to the cache; LRU measures when the item was last read.)
- Random (RAND) -- remove an item at random. The idea is that, even if it does not perform maximally well, it will not perform maximally abysmal under any load.
- Largest Size First (MAXS) -- remove the item which has the largest size, assuming that users are less likely to re-access large documents because of the high access delay associated with such documents.
- LRU-MIN -- a variant of LRU which tries to minimize the number of documents replaced and gives preference to small-size documents to stay in the cache. If an incoming document with size S does not fit in the cache, the policy considers documents whose sizes are no less than S for eviction using the LRU policy. If there is no document with such size, the process is repeated for documents whose sizes are at least S/2, then documents whose sizes are at least S/4, and so on. See the paper Caching Proxies: Limitations and Potentials for more details about LRU-MIN.
- Greedy-Dual-Size (GDS) -- removes the item with the lowest value of H. Each item starts off with a value of H = (cost of bringing the item to cache / size of the item). When an item is replaced, decrement all of the other items H values by the replaced item's H value. When items are accessed again, restore H to the original H value (cost / size). The cost function is parameterized.
- Greedy-Dual-Size-Frequency (GDSF) -- even more complex modification of GDS. See Improving WWW Proxies Performance with Greedy-Dual- Size-Frequency Caching Policy

Note that performance of different policies will depend on the queries that a client generates. Please specify how you generated thoese queries, e.g., using random / uniform distribution, and relate them to the performance measurements in your report.

Apache Thrift

To install Thrift, you may be able to use your Linux package manager (e.g., apt-get). If it's not available there, you can download it from the Thrift site. (There is a Windows compiler too, but you should do this project on Linux.) Installation instructions are here.

You can find Thrift examples here in many languages. Although Thrift offers cross-language support, all your code should be C++. (If you're more comfortable with C than C++, don't feel you need to do anything object oriented beyone the stub code that the Thrift compiler will generate for you anyway.)

You can read the whitepaper on the Thrift site if you want to see how Thrift is extensible. However, it's probably not necessary for this assignment, since we're just using the options Thrift comes with out-of-the-box.

libcurl

libcurl is a powerful library for communicating with servers via HTTP (and FTP, LDAP, HTTPS, etc.). It supports HTTP GET/PUT/POST, form fields, cookies, etc. The purpose of using it in this assignment is to simplify your life. Instead of writing lower-level sockets code to talk HTTP to a webserver and request a page, you can simply use libcurl (which can do the same in a few function calls). In fact, you can just use the example code demonstrating how to perform a simple HTTP GET request. If you want to implement your own socket-based code to talk to the remote webserver, it will mean more work for you but you're welcome to do so (just remember that it is not the focus of this project).

example.c is sample code that uses libcurl to perform an HTTP GET of a specified URL (the Makefile shows you how to compile it, but basically all you need to remember is to add -lcurl to link with the curl libraries). The resultant data (just the data, not the headers) is captured into a dynamically allocated buffer and then written to stdout. You can use this code with few modifications

directly in the proxy to perform the proxy GET requests.

If you have trouble with servers returning errors complaining about the lack of an HTTP User-Agent field, add a curl_easy_setopt(handle, CURLOPT_USERAGENT, "<agent name>"); before the curl action is performed.

Resources

The following documents have useful information on Thrift, libcurl, and relevant network protocols. You will need to have a cursory understanding of the protocols in order to do this project, but do not bother becoming extremely knowledgable about them just for this project.

- libcurl C Interface overview
- Apache Thrift official site
- Thrift: The Missing Guide - contains more details about Thrift (including how Thrift types map to C++ types)
- ...looking for more resources. If you find something that's really helpful, post it on Piazza! That will help other students, plus the TAs may add it here as well.

Writeup Guidelines

Some helpful guidelines on what to be sure you include in your writeup will be added here shortly...

Submission Instructions

You should submit a single tarball which includes:

- All your code (proxy server and client, plus any testing harness)
- Makefile
- README (explain how to build and how to run the experiments)
- Your write-up (PDF)

** You must submit to **T-Square** by the deadline, in order to be considered on time. **

Additional resources for assignment

example.c (2 KB; Mar 3, 2014 12:57 am)
Makefile (2 KB; Mar 3, 2014 12:57 am)

Submission

This assignment allows submissions by attaching documents only.

Attachments

No attachments yet

Select a file from computer Choose File no file selected