Project 1

Report

Version 1.0

February 20, 2006

Vladimir Urazov

Omar Zakaria

Contents

1 Building The Sources

The source code for the project is broken up into three directories:

common - this directory contains the code common to both the client and the server, such as some of the networking functionality, the multi-thread-safe list data structure, etc. This directory contains its own makefile, which builds a static library name libcommon.a from all of the source files. This library can then be included in the build of the server and the client to provide the necessary functionality.

server - this directory contains the http server code. Unlike the common library, it actually builds into an executable. The make file in the server can be used to build it (provided the common library has already been built).

client - this directory contains the code for the testing client, used to do performance and robustness tests on the server. It also builds into an executable with the make file, but assumes that the common library has already been built.

The simplest way to go about building the entire project is by typing 'make' in the root directory of the project. That will create a 'bin' folder there and will build all the components of the project into it. Hence, once the make process completes, the directory will contain the following files: client, libcommon.a, server. The client and the server can then be immediately run.

2 Running The Executables

The project build process creates two executables: server and client.

The **server** is the HTTP server. It supports the following command-line parameters:

port - the port that the server will listen for connections on. The default value for port is 1337.

pool-size - the number of threads to create to handle client requests. The default value is 16. Note that due to per-user constraints Linux imposes on the number of available threads, the server will fail creating enough worker threads if this parameter is set too high.

home - the path to the root document directory, relative to the directory from which we are starting the server. All the documents will be served relative to the home directory.

So, for example, if we are in the root directory of the project, after running make, we can start the server as follows:

```
bin/server port=80 pool-size=32 home=./home
```

(Note there should not be spaces between the name of the parameter and the parameter values).

Note that the server has certain security restrictions, for instance it will not allow clients to go up the directory hierarchy, and also if will respond with forbidden HTTP status, if the user running the server has no permissions reading the requested file.

The **client** executable is the testing client used for exercising the server. In order to run, it requires a URL file, which is basically a plain text file with URLs that the client will request from the server. These URLs can have either host names or dotted IP addresses with them - the client can handle both kinds of URLs. The client supports the following options:

job-count - the number of requests to make to the server. The default value is 1. Note that this is the total number of requests to be made, and this will cycle through the URLs in the URL file. So, for example, suppose this parameter is set to 100, and the URL file contains two URLs, then each URL will be requested 50 times.

thread-count - the number of threads to spawn to make requests to the server at the same time. Note that these threads will handle as many requests as the job-count specifies, so, for example, if the job count is 100, and thread-count is 10, then likely each thread will make 10 requests to the server (though that's not necessarily the case, since for example if one of the workers is taking a long time to receive the response from the server, then the others will pick up the slack).

url-file - the name of the file that contains the URLs to request. If no value is provided for this parameter, then the URLs will be read from the console.

So, if we have built the project from the root directory of the project, we can run the client as follows:

```
bin/client job-count=1000 thread-count=32 < urlfile
```

3 Tests

3.1 Test Types

The server was put through four main batches of tests. These were:

- *Dual Core, Same Machine* – in these tests, both the client and the server were run on a dual-core machine in the Enterprise lab. In these tests, we varied the number of threads in the server, and the number of threads in the client. We also requested up to four different files of varying sizes. The raw data from these tests is available in *results/dual-same*. These tests were run on *frodo*.
- *Dual Core, Different Machines* – in this batch of tests, the server and the client were run on different machines in the Enterprise lab. The server was tested under similar conditions as above. The raw data from these tests is available in *results/dual-diff*. In these tests, the server was run on *legolas*, and the client was run on *frodo*.
- *Dual Core, Same Machine, Thousand Files* – in these tests, the client and the server were run on the same dual-core Itanium machine in the Enterprise lab. The client requested a thousand different very small files from the server. The raw data from these tests is available in *results/dual-thousand-same*. These tests were run on *elrond*.
- *Single Core, Different Machines* – in this batch of tests, the client and the server were run on two different single-core machines in the States lab. We varied the number of threads in the client and the server, and up to four different files were requested multiple times. Note that originally, we tried to perform the tests on the same machine, however the tests took a long time to run on a single-core computer (which we think might have been the case due to the large number of threads we tried to run at the same time – 32 for the server, and between 4 and 128 for the client). The raw data from these tests is available in *results/single-diff*. In these tests, the server was run on *michoacan*, and the client was run on *yucatan*.

Note that in the result directories, the result log files are named according to the number of threads the server was running (so for example, *results16.txt* is the log of requests to the server running 16 worker threads).

3.2 Test Setup

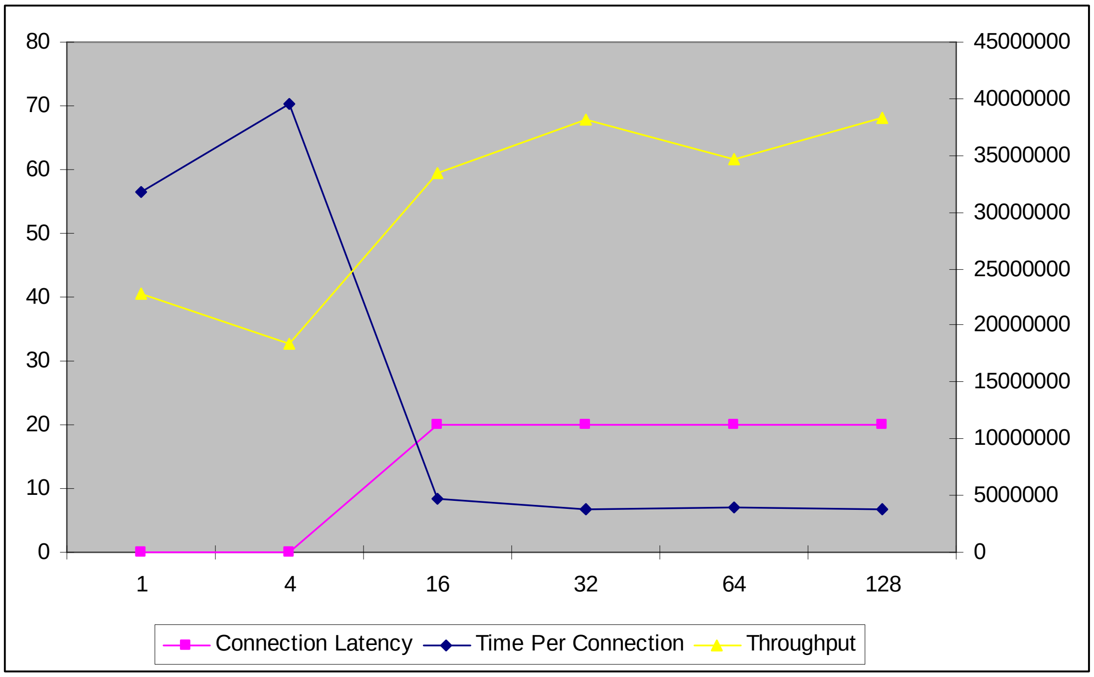
For each of the tests, we ran the server with a given number of worker threads. Then, we had a script run the client with various parameters. We started off by launching the client with a small number of threads and had it request several hundred instances of a tiny file. We repeated this several times over, requesting consecutively larger and larger files. Our final test involved requesting several hundred instances of all four file sizes to induce variation in the queries. The smallest file was about 30 bytes and the largest file was about 3 Mb. Finally, we launched the client again with a larger number of threads and repeated the file-varying sequence. The number of client threads varied in powers of two from four to 128.

The client measured several statistics about its requests to the server. The first thing the client reports upon completing all the requests is the total time for all requests in milliseconds. This is the time between when all the worker threads start requesting things from the server, to the time when every single worker thread quits when all the jobs are completed. The next thing the client reports is the total number of bytes received from the server through this session by all workers, and the throughput in bytes per second. The first metric that the client runs some statistical analysis on is the connection latency. This is the time in milliseconds from when the client tries to open the socket to the server, until when the socket is ready to send and receive data. This metric is mainly intended to measure the efficiency of the boss thread of the server, which is the one that opens up sockets and distributes them among worker threads. Finally, the client reports statistics on time per connection. This is the time between when the socket is ready to send and receive data, and when the socket is closed by the server after all the data has been sent.

3.3 Test Results

3.3.1 Dual Core, Same Machine

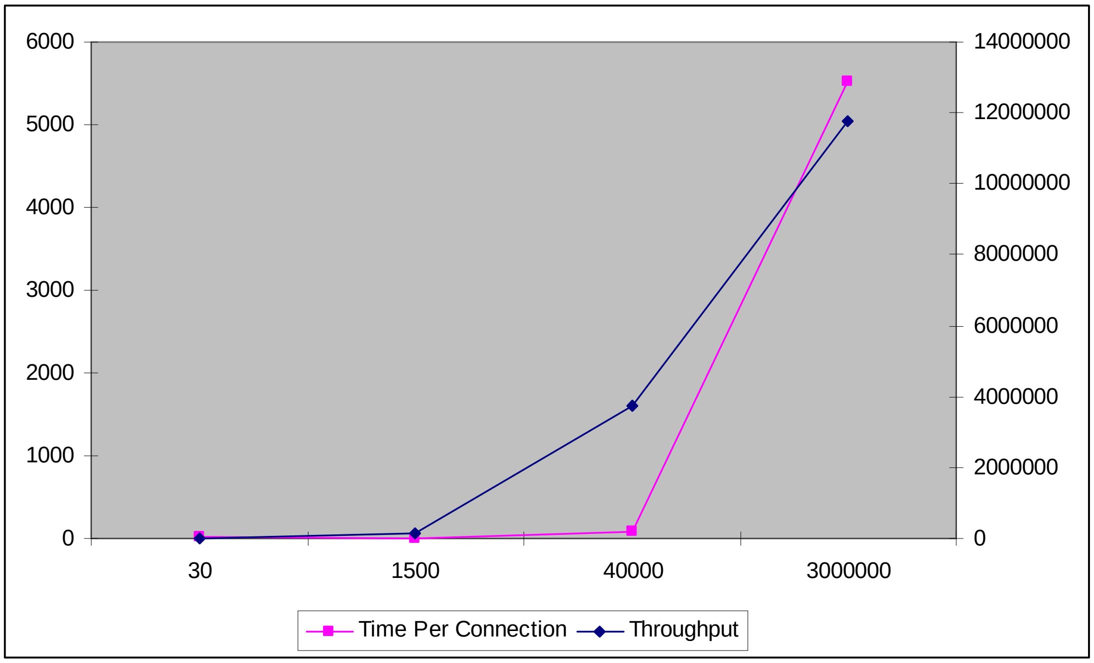
Figure 1: Request Statistics. Horizontal axis: the number of server threads. Left vertical axis: time in milliseconds for connection latency and time per connection. Right vertical axis: bytes per second, for throughput.



The figure above shows the statistics on the performance of the http server depending on the number of worker threads in the server. As the number of threads increases, so does the throughput of the server, which is to be expected. Furthermore, the time per connection also drops as the number of threads increases. We also expected this result. We did not, however, expect to see the latency of the connection increase as the server went from four to 16 threads. We are not certain why this occurred, but we are satisfied with the result; a latency of around 20 milliseconds is quite reasonable. All these statistics are based off of a client with 32 simultaneous threads requesting medium sized files. Other data can be found in the appropriate results file, but all of the results follow the same general pattern.

3.3.2 Dual Core, Different Machines

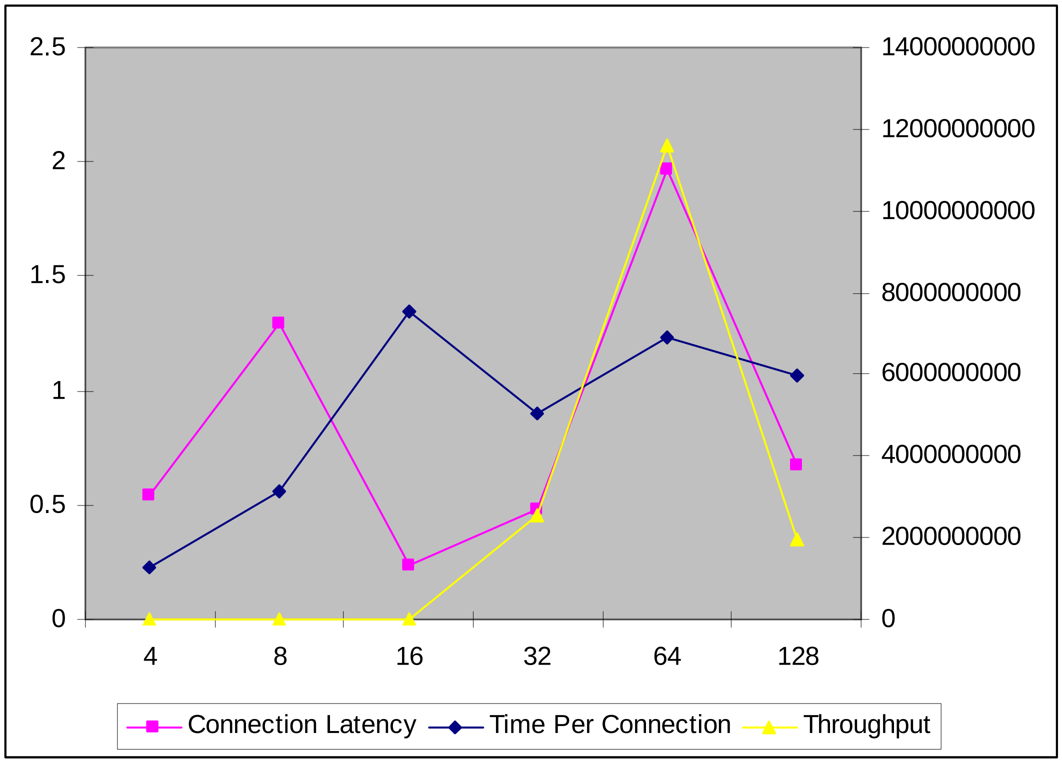
Figure 2: Request Statistics. Horizontal axis: the size of requested files in bytes. Left vertical axis: time in milliseconds for time per connection. Right vertical axis: bytes per second, for throughput.



In the experiment shown in the graph above, we ran the client and the server on different machines. The data displayed is collected from several runs of the client, where both the server and the client had 32 worker threads each, and the client was requesting files of different sizes from the server. The data shows predictable results, that as the size of the files requested increases, so does the time per connection, and the throughput. This is especially accentuated by the fact that we are performing transfers over the network. The connection latency was omitted from the graph, since it was negligible compared to time per connection, and the pattern was not observable on the graph. Interestingly, however, the connection latency seemed to go down as the size of the files requested increased.

3.3.3 Dual Core, Same Machine, Thousand Files

Figure 3: Request Statistics. Horizontal axis: the number of client threads. Left vertical axis: time in milliseconds for connection latency and time per connection. Right vertical axis: bytes per second, for throughput.



In the graph above, we gathered the statistics from running a server with 32 worker threads on the same dual core machine as the client. The graph shows the dependence of the performance statistics on the number of worker threads in the client. As latency increases, so does throughput. We believe this to be because the boss thread in the server, who is actually accepting connections, is not getting as much CPU time because the worker threads are busy. This makes sense, since the boss must wait on IO to begin with. It is also apparent that the number of client threads, if significantly different than the number of server threads seems to have an adverse affect on throughput.

While it is not quite clear from the graph, as the number of threads increases, the total time for a connection stabilizes around a certain value, which is just over 1 millisecond. This makes sense given that we are transmitting very small files and there is a reasonable amount of overhead in switching threads to handle incoming requests. However, we would like to note that our server still behaves admirably, given that our throughput is around 2 GB a second.

3.3.4 Single Core, Different Machines

In this case, the data exhibited much the same patterns as in the dual-core different machines set of tests. However, the server seemed to start dropping connections when the file size requested started reaching 40 KB, and the number of simultaneous requests from the client went up to 32 or more threads. This was the case when the server was running 32 threads or more. And when running the server with 1 thread only, the network latency became large enough where the tests took a long time, and we felt it was unnecessary to run those tests, since they would be virtually identical to the dual-core machines, since in this case, the server could derive virtually no benefit from multiple cores. As a result, we were unable to gather complete data for the performance of the server for any number of threads other than 16.

4 Summary

Overall, we believe that our server was robust and performed well. We were unable to find any security vulnerabilities, and the server proved reasonably stable, without crashing through any of the tests performed, though sometimes dropping connections when under heavy load.

It is important to note that the testing script we used put the server under occasional severe load, asking for up to 1280 jobs with 128 concurrent threads. The single-core machines were able to keep up until we reached 32 concurrent threads and 320 requests, which actually seems quite reasonable considering they were not machines specifically dedicated to running any sort of service. Another reason we believe those single-core machines may not have performed well with many threads running simultaneously was memory usage. We found that pthreads produced large memory footprints. To test this out, we ran the server first with 16 worker threads. It took up about 300MB of RAM. When running with 32 worker threads instead, the server took up some 600MB. Therefore, on the States machines, which have 500MB of RAM, the server had to constantly page to disk, and that decreased the performance dramatically.