Ubermicro Phase 1

CS4210

Version 1.0

February 20, 2006

Jose Caban & Puyan Lotfi

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Jose Caban	Spellchecked and proofread	02/20/2005
.99	Jose Caban	Document Complete, spell check next	02/20/2005
.9	Jose Caban	Report on Flood and Client tests complete	02/20/2005
0.0	Jose Caban	Layout created	02/20/2005

Contents

Figures

1 Introduction

1.1 Overview

To start off, let us explain the name. We titled our server "ubermicro-httpd" because since we used bits of code from microhttpd, it has some similar functionality (mime types, directory listings, ../ prevention, and symbolic link traversal). Why would be add a prefix "uber"? Well because our server is in everyway cleaner and more advanced.

1.2 Microhttpd: the horror

Microhttpd is 300 lines of hellish K&R style code, designed to be executed by inetd, piped some data, and piped back the response via stdout. Since the executable it self was designed to run for a short period and then terminate, the author though that was a good enough reason to not close file descriptors and not free memory. In addition, since the author saw the need to make his static arrays enormous he assumed a large stack size. Using Microhttpd for the project was a good lesson in how not to code. Therefore, a good portion of our work with Microhttpd was cleaning the original programmer's memory leaks, closing file descriptors, and making stack allotted variables small enough so that they would not overflow the thread stack. What's more interesting is how we managed to integrate this code.

The process of integrating Microhttpd could have been done rather clumsily. There are many ways it could have been done, but when we found that we could create a filestream from the socket descriptor and change the printf() call to fprintf() calls on the stream we knew that this was the ticket. The primary disadvantage to this is that streams are not full duplex like regular UNIX IO so there are some pitfalls.

2 Installation and Execution

2.1 Installation

- Extract the provided ZIP file into whatever directory you desire
- Run "make" from the directory installed to build the server executable
- Run "cd client" and then "make" to build the client executable (alternatively, the client may be built and executed from Visual Studio .NET 2005 Win32)

2.2 Execution

2.2.1 Server

The server can be run by executing "bin/l33t_server" from the directory containing its makefile. The server searches its current directory for its configuration file: "l33t_server.conf". If this file is not found, the server will report an error and quit. The config file format is easy to understand:

```
#COMMENT
VariableName:Value
```

Figure 1: Config File format

Proper execution of the server should look similar to the following:

```
asskoala@Faye Project 1 % bin/l33t_server          [21:25:26 on 02/20/06]
* Beginning l33t_server Initialization *

** Using 256 threads
** Using home_dir: /tmp/gtg184g/
* l33t_server started *

** Initializing Thread Pool
*** Listening on Port: 1337
```

Figure 2: Executing the Ubermicro server

Please look at the provided configuration file for example settings.

Finally, to quit the server, simply type q and then enter. All other input will be ignored.

2.2.2 Client

The client can be executed much like the server, simply run "bin/l33t_client ADDRESS PORT_NUMBER TESTFILE". Note that the client does not support DNS resolution and, as such, the ADDRESS must be specified as an ip address (usually 127.0.0.1). Some example test files are included in the /client/tests directory. These files use standard XML formatting. A simple test file is shown in Figure 3:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- XML File for test execution-->

<Tests>
      <simpleTest>
      <!-- The number of threads to spawn to make requests -->
      <numthreads>32</numthreads>

      <!-- The number of times the request will be made per threads -->
      <iterations>10</iterations>

      <!-- The files to retrieve -->
      <filesToRetrieve>

         <!-- The file names, use / for directories -->
         <file>/index.html</file>
         <file>/pwner.jpg</file>
         <file>/makefile</file>
         <file>/special.zip</file>

      </filesToRetrieve>
   </simpleTest>
</Tests>
```

Figure 3: Sample Test file

Currently, the client only executes the first test in the file. Explanation of how it works is in section 3.2.

3 Implementation

3.1 Server

3.1.1 Networking

Our use of sockets was done in a way that allowed for cleaner and more reusable code. As time goes on, we will refractor more and more pieces of code so that we can have more code reuse. The idea was that we could isolate the TCP connection process into three major components: setting up socket data structures and binding to a port, accepting connections, and handling accepted connections. These are separated into various functions. We also loop our recv() calls to ensure all the data we are expecting arrives properly from the TCP bytestream.

3.1.2 Threading

The threading model uses a simple producer/consumer model with a linked list queue of (relatively) infinite length to store sockets. The producer establishes the connection and places the socket in the queue. The workers wait until the queue has a socket and handle the connections as soon as they can. The queue is atomically operated upon and is the only place in which mutexes are locked within the source code.

3.2 Client

The client takes in three parameters, one being the server's IP, port number, and an xml file. The xml file specified how many threads the client should spawn, what files to request, and how many times the file should be requested. Therefore, the file requesting is divided among the threads, who each request the same file a certain number of times.

4 Testing and Benchmarks

4.1 Introduction

The server was initially stress tested using apache testing tools (http://httpd.apache.org/test/), particularly the apache flood test, http://httpd.apache.org/test/flood/. After the stress testing, benchmarks were created for the time taken to pass the flood "stay alive" test, tests with our own client, and tests with the HTTPerf tool.

4.2 Test Beds

Samwise & Boromir:
Dual 3.06GHz Xeons
1GB RAM
SCSI disks
RedHat Enterprise Linux AS 4

Idaho:
1.8GHz Pentium4
512MB RAM
IDE disks
RedHat Enterprise Linux AS 4

4.3 Flood

The following test was run on *Boromir*. As can be seen from the graph (Figure 4), the performance decreases from 16 threads to 64 threads then drastically improves at 128 and 256 server threads. However, the actual performance difference between the worst case (64 threads) and best case (256 threads) is a 4% difference. This difference is nominal as it is below 5% and can be correlated to lower loads due to user interaction or background processes on the test system.

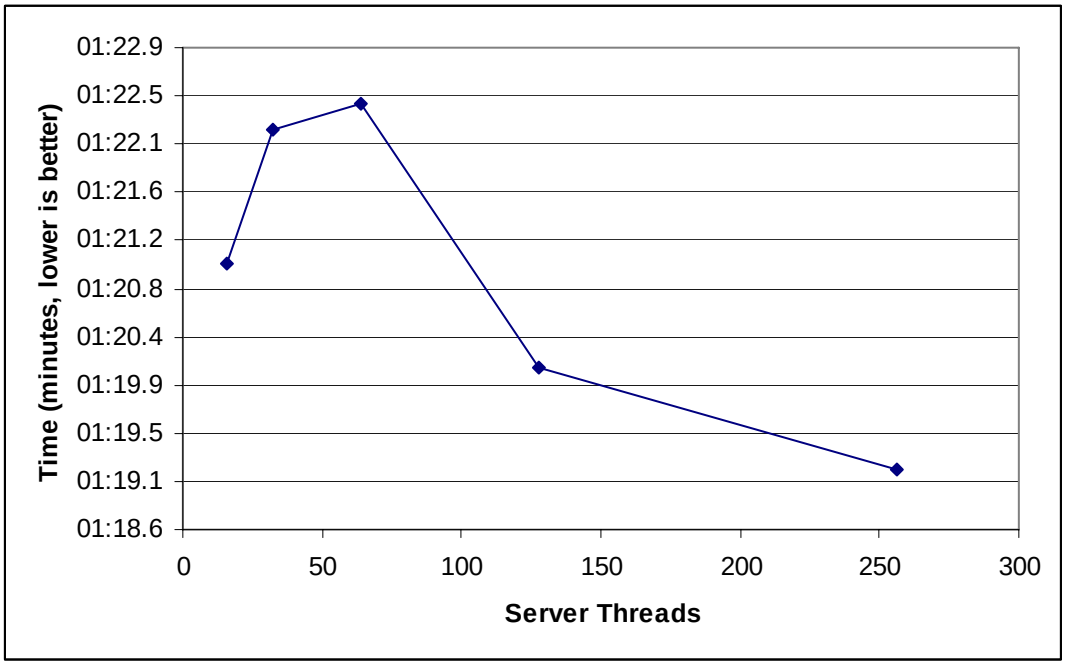

Figure 4: Flood runtime

4.4 Client tests

4.4.1 Single File Test

The single file test consists of a single thread requesting the same 1-Megabyte file 10 times. Results are shown in Figure 5. The file was created using the "dd" command in UNIX. The test was executed on *Boromir*. Unlike the flood test, the 64-thread server performs the best with the 128-thread server performing the worst. The additional overhead of extra threads could be the cause of the 128 and 256 thread servers performing worst. However, once again, the performance difference between the worst case and best case is less than 5%, at a value of 1.3%. These changes, therefore, can be attributed to varying load on the test machine.

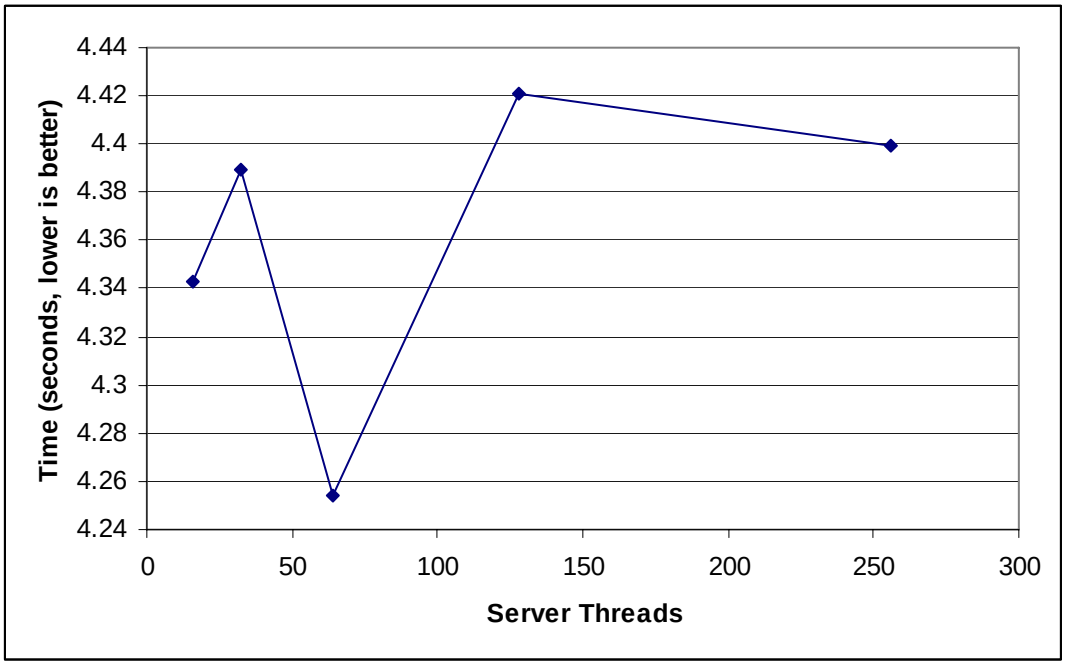

Figure 5: Single 1MB File Test

4.4.2 Single File 10 Thread

The single file 10 thread is identical to the single file test except that 10 threads concurrently request the file (therefore a total of 100 requests for the file are made). This test was run on *Boromir*. The time differences here become considerably different. The first thread always completes first due to the time advantage it gets from the other threads still being spawned in the background. The best performance in this test looks to be with the 128-thread server. Theoretically, there should be little difference since only 10 clients are making requests. However, a roughly 19% difference exists between the 128 thread server and the 64 thread server.

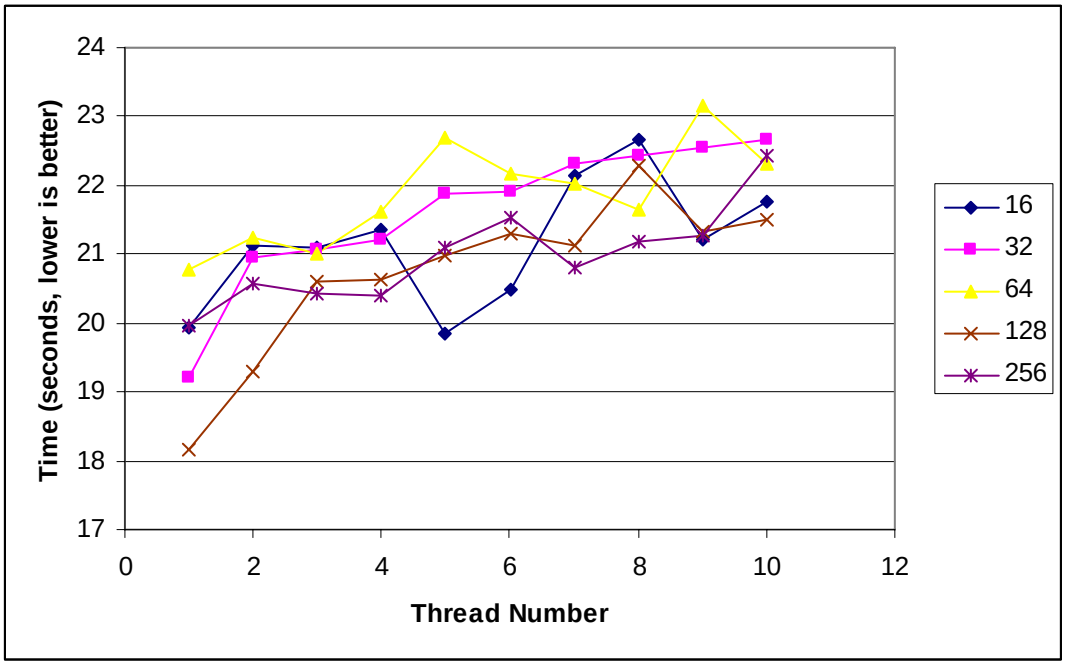

Figure 6: Single File Test (10 Thread)

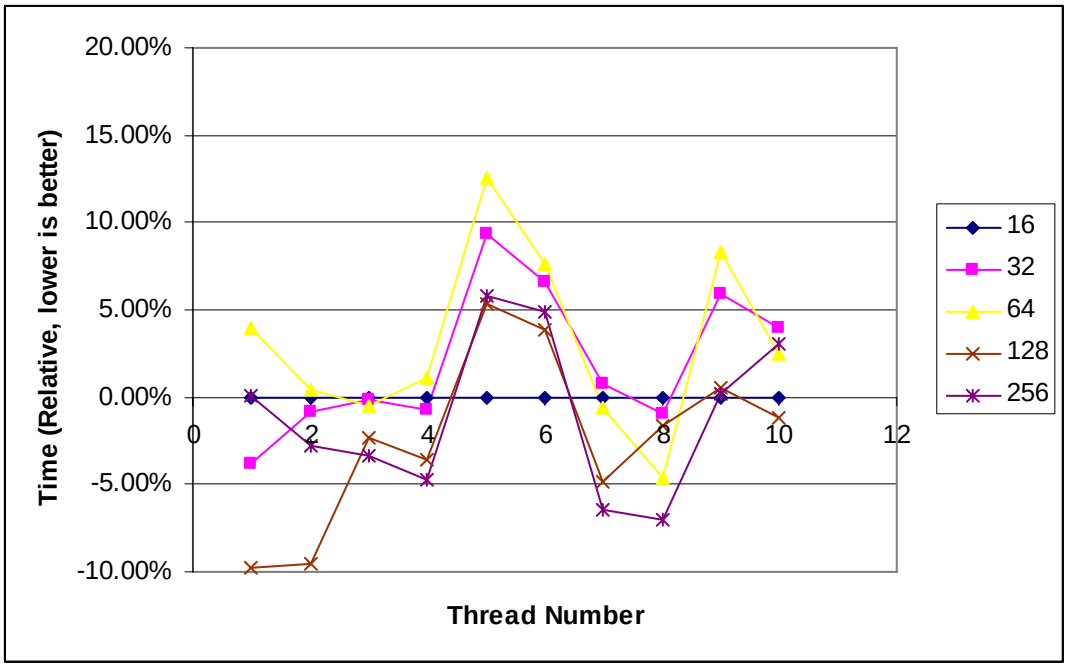

Figure 7: Single File Test (10 Thread) Relative

The relative performance of the server, compared against the 16-thread version, shows that the 128-thread server, in general, performs best in this test.

4.4.3 Single File 128 Threads

This test is the same as above except that 128 client threads concurrently try to access the server, resulting in 1,280 requests by the time the client finishes executing. Figure 8 shows the results of this test. In this case, the 128-thread server and the 256-thread server struggle to achieve the best performance. Theoretically, the 128-thread server should have slightly higher performance, however in practice this difference averages out to be relatively equal. The relative performance is shown in Figure 9. This test was run on *Boromir*.

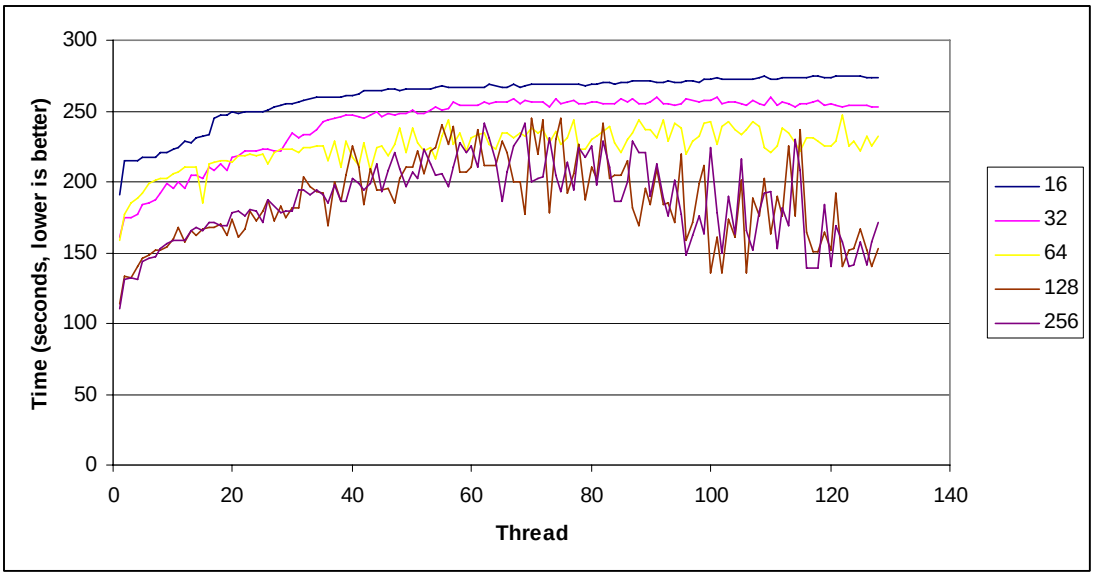

Figure 8: Single File Test (128 Thread)

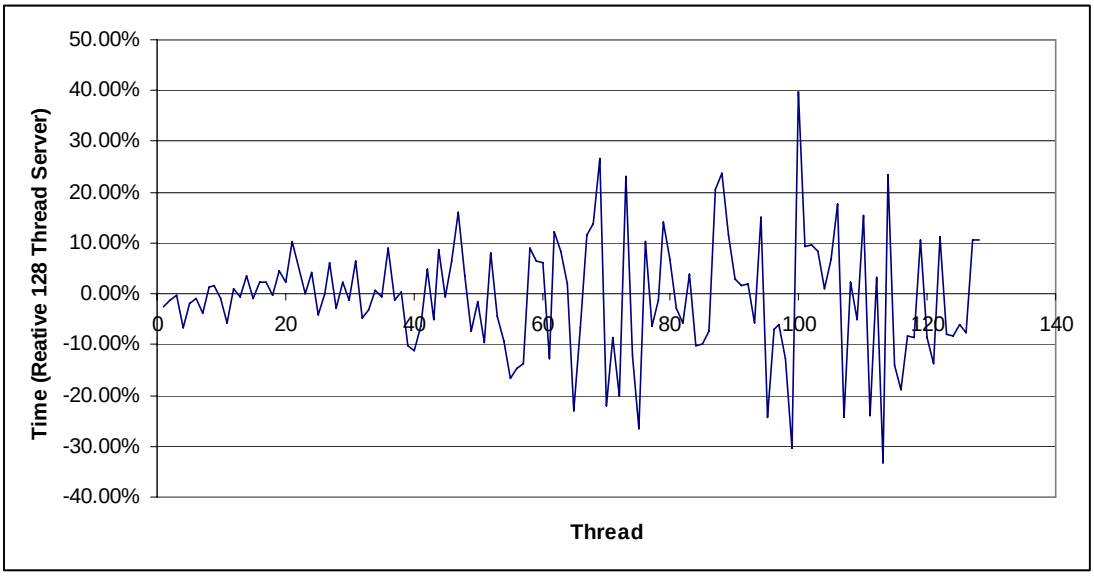

Figure 9: Single File Test Relative to 128 Thread Server Performance

4.4.4 256 Small File Requests

The 256 small file test spawns 256 threads to make concurrent requests 3 times each for the small 4K file contained on the server. This test was run on *Samwise*. The results are shown in Figure 10:

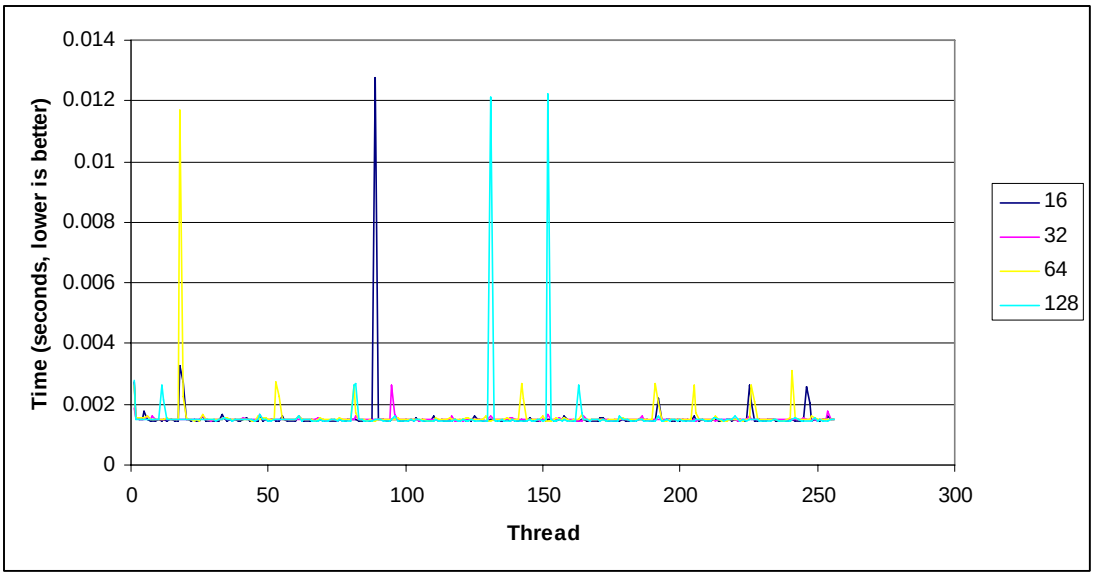

Figure 10: 256 Small File Request Results

The results show very little difference in performance for these small files between the number of server threads. This can be due to the size of the files. Of note, however, is the latency associated with context switching to background processes, shown in the graph in the form of "spikes". These may be also associated with latency due to the queuing structure or threading model internal to the server. The differences are so small between the number of server threads, however, that it is irrelevant.

4.4.5 Simple Test

The simple test spawns 32 threads that each make requests 10 times. These requests consist of four files (8 threads with file 1, 8 threads with file 2, etc): a very small makefile, a small html file, a jpeg file, and a somewhat large ZIP file (roughly 4MB). This test was run on *Samwise*. The results of the test are shown in Figure 11:

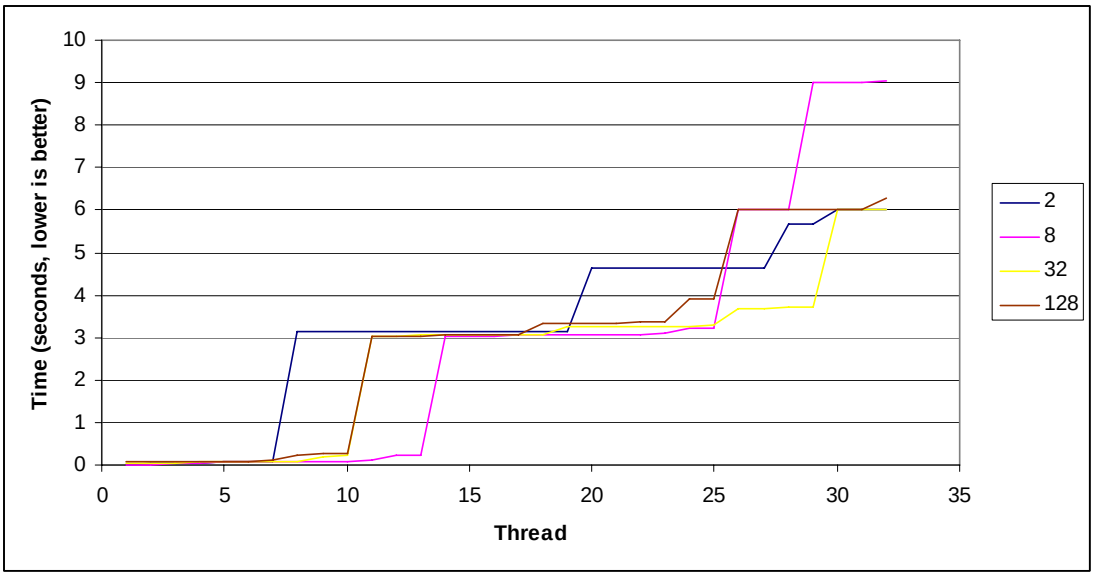

Figure 11: Simple Test

The simple test results show both expected and unexpected results. As expected, the 32-thread server and the 128-thread server performed on par with each other. Their differences are minor, much like the test in 4.4.3. However, the 2-thread server outperforms the 8-thread server. This is unexpected, as the 2-thread server should be causing starvation to the waiting requests. The tests show that no requests were dropped completely in the 2-thread case. The performance difference may simply be attributed to background process. However, it is also possible that latency have been caused by many of the threads accessing the queue at the same time in the 8-thread case. This could cause the results as shown.

4.4.6 Single File Test (Uniprocessor, 1 thread)

The single file test was run on a Uniprocessor system to evaluate performance. While this system is not comparable to the Dual Xeon systems used in previous tests, it should show the scalability on a single CPU platform. These tests were executed on *Idaho* in the states lab. The tests are shown in Figure 11.

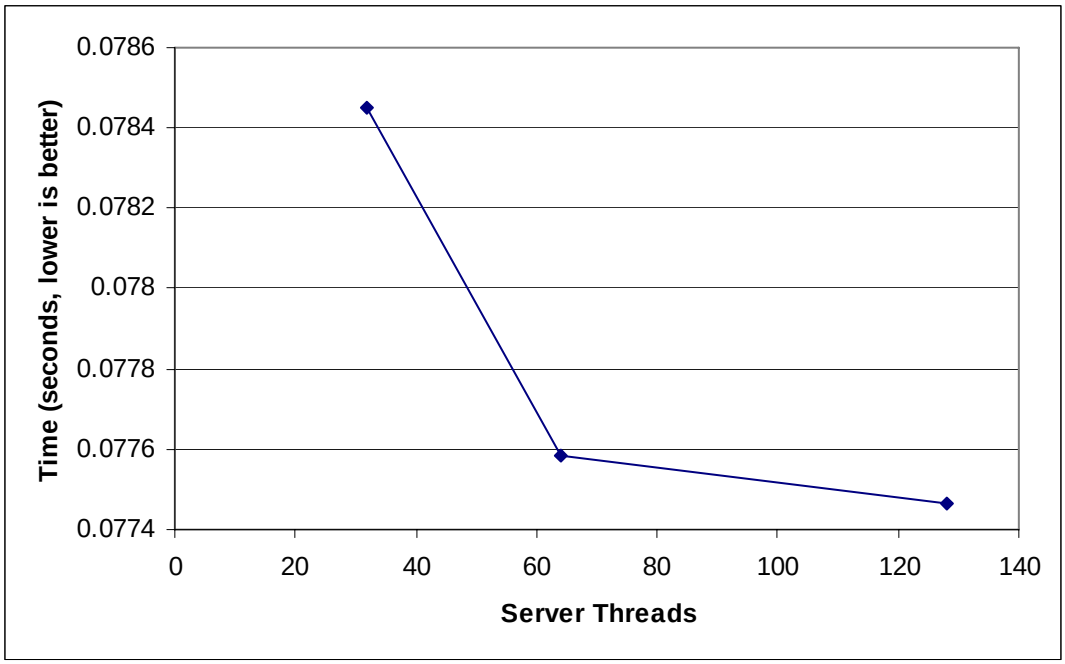

Figure 12: Single File Test Uniprocessor

Much like in the Dual Processor case, the performance difference between the number of threads is nominal at less than 2%.

4.4.7 Simple Test (Uniprocessor)

The simple test was performed, as in 4.4.5, but on the Uniprocessor system *Idaho.* The results are shown in Figure 13:

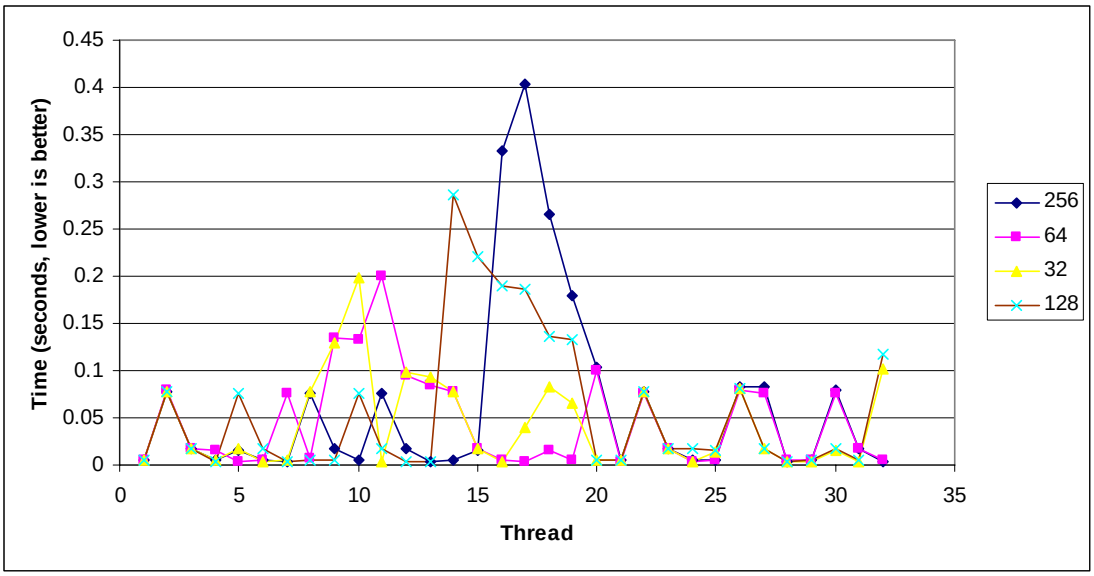

Figure 13: Simple Test (Uniprocessor)

The results show that the lowest number of server threads, at 32, performed the best in the test. These results show that increasing the number of threads does not increase the performance of the server on a Uniprocessor machine.

4.5 HTTPerf

4.5.1 Multiprocessor

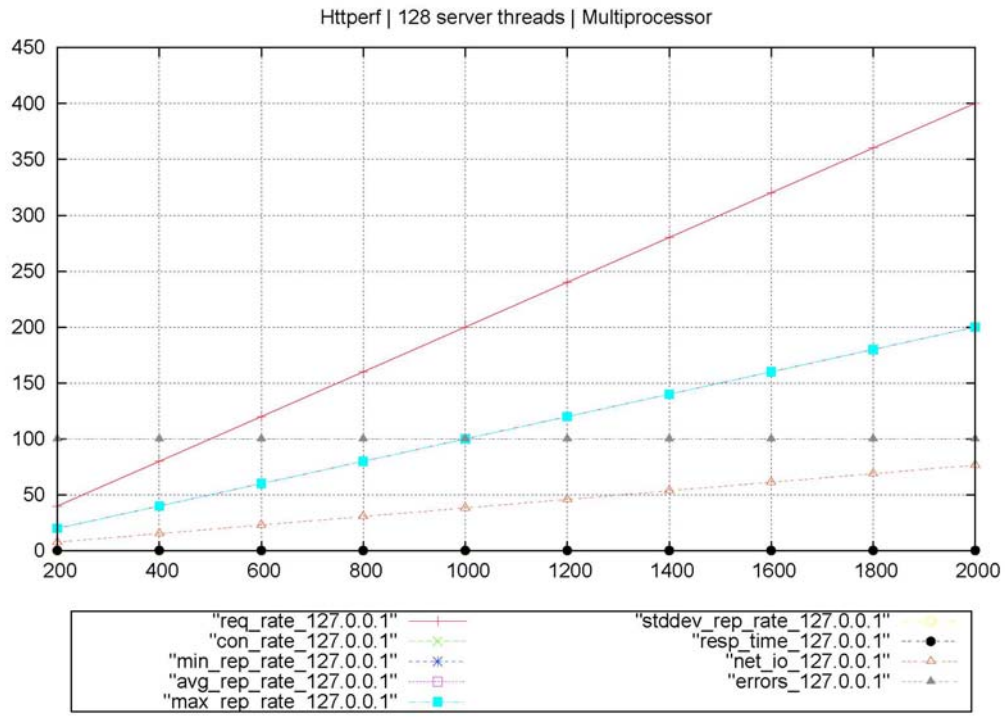

Figure 14: HTTPerf: 128 Server Threads

The 128-thread server shows linear increases in performance as the HTTPerf test runs on *Samwise* (the y-axis represents requests per second and the x-axis represents concurrent connections). Figure 15 shows the 256-thread server performance:

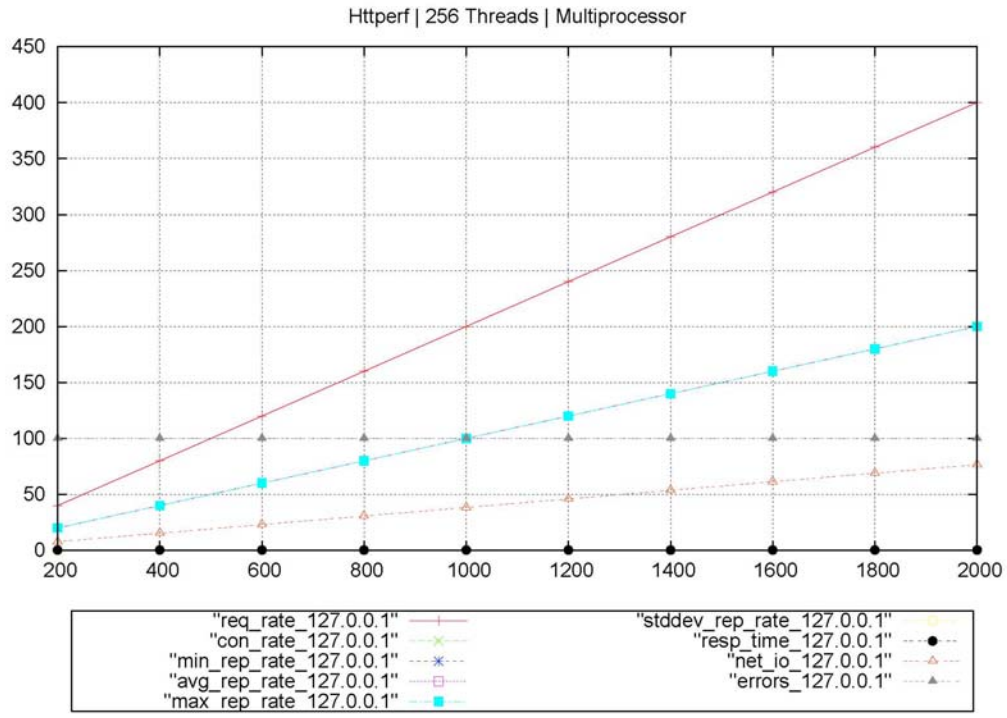

Figure 15: HTTPerf: 256 Server Threads

As can be seen from these two benchmarks, the difference between spawning 256 server threads and 128 server threads on the dual processor systems is virtually none. There is a linear increase in requests handled per second as concurrent connections increase, however, so the server scales somewhat well.

4.5.2 Uniprocessor

The HTTPerf tests were run on *Idaho* as well. The results are shown in Figures 16 and 17:

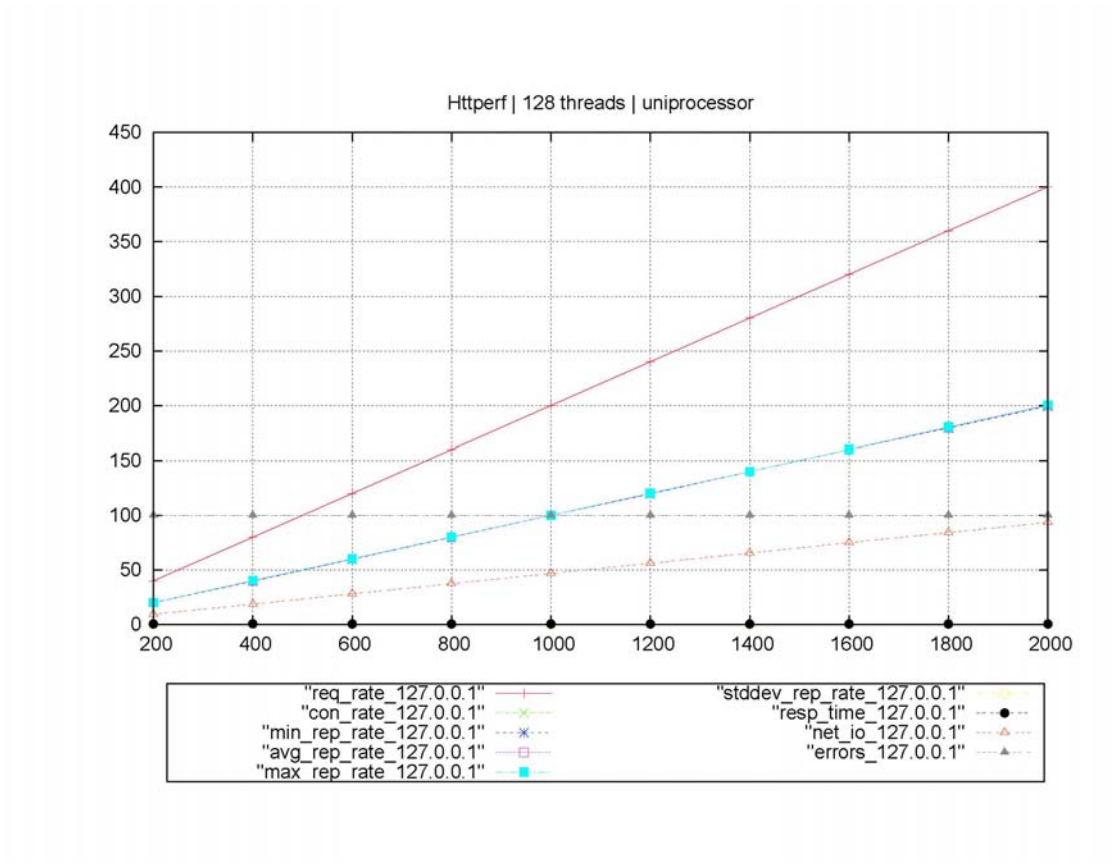

Figure 16: HTTPerf: 128 Thread Server (Uniprocessor)

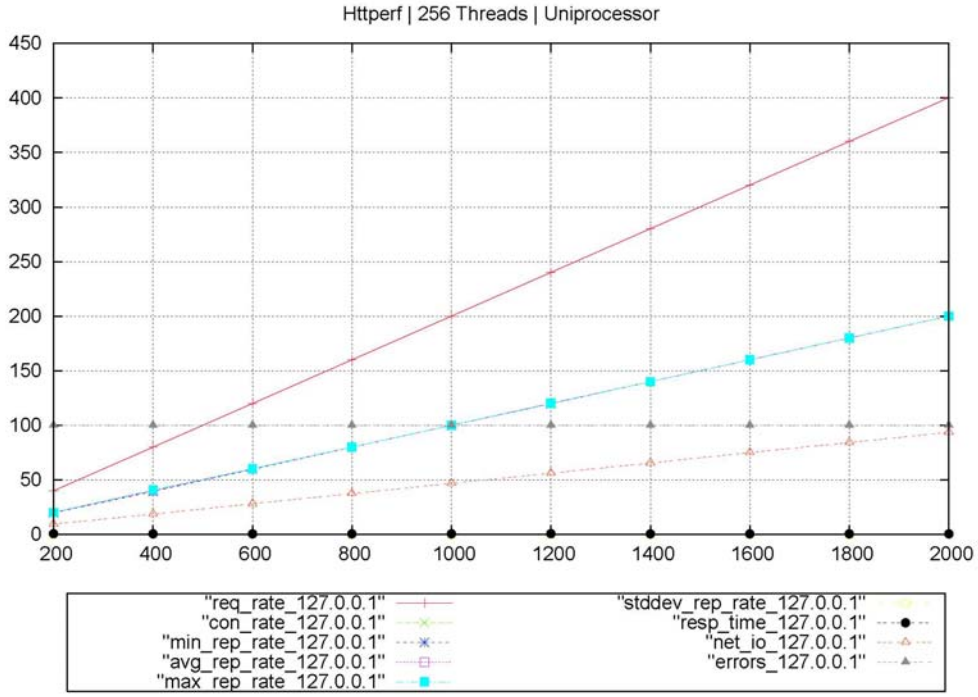

Figure 17: HTTPerf: 256 Server Threads (Uniprocessor)

The results here, as in the multiprocessor case, show that the difference between running 128 and 256 server threads is virtually none. The server continues to scale linearly, however.

5 Conclusion

The project served well to highlight the advantages and disadvantages in building a multi-threaded web server. More threads do not necessarily mean improved performance, as was highlighted with the low performance of the 256-thread server in most cases.