Michael Landes
CS 8300 (GOS)
10/2/12

CS 8803 (GOS): Project 1: Simple HTTP Server (Boss-Worker)

Introduction:

For this project, we were asked to implement a multi-threaded web server (serving a *very* small subset of the HTTP protocol). The premise is that multi-threading can increase performance when I/O is present in a system.

To examine the hypothesis, I constructed two programs, one server, and one client. The most important feature of the server is that it implements a boss-worker model. We can vary the number of threads in the worker pool as a run-time parameter, which will help us determine the effect of threading in such an environment. The most important features of the client are the ability to vary the number of threads making requests simultaneously, and the number of requests each thread will make. The number of simultaneous threads simulates that number of requests into the server at a given moment of time, and increasing the number of requests each thread makes extends the time period of the load long enough to measure the effects on the server.

The Simulation Process:

There are two key variables I am interested in to examine the hypothesis. First is throughput, or, how many megabytes does the server return per second. The second variable is accuracy, or how many bytes did the server return in respect to how many bytes it should have returned. In my system, the client times the request and receive process, and records the amount of bytes received per thread, and system-wide. We can extract from these data points the throughput and the accuracy.

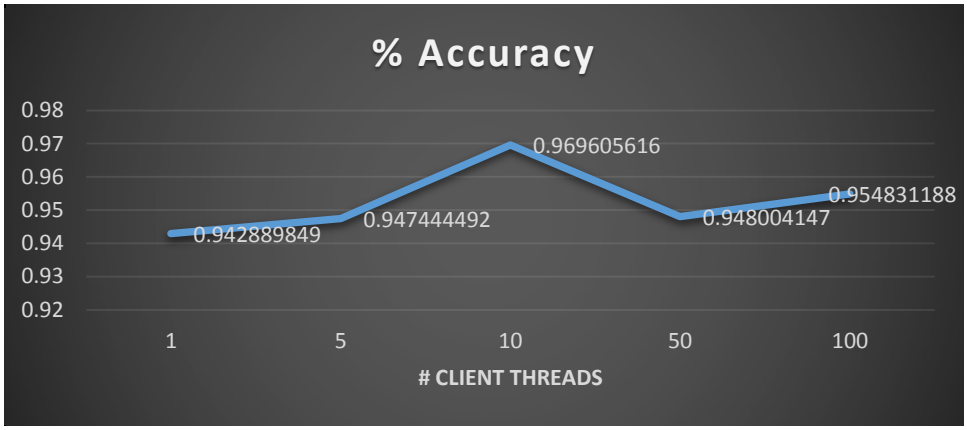
I will measure a number of axes of variation including total load (in terms of number of threads the client runs simultaneously), number of threads used to run the server (which is directly correlated to the premise), file-size (which puts more strain on each server thread), and finally number of different files requested (which causes the server to be more I/O-bound since it is theoretically using the cache less).

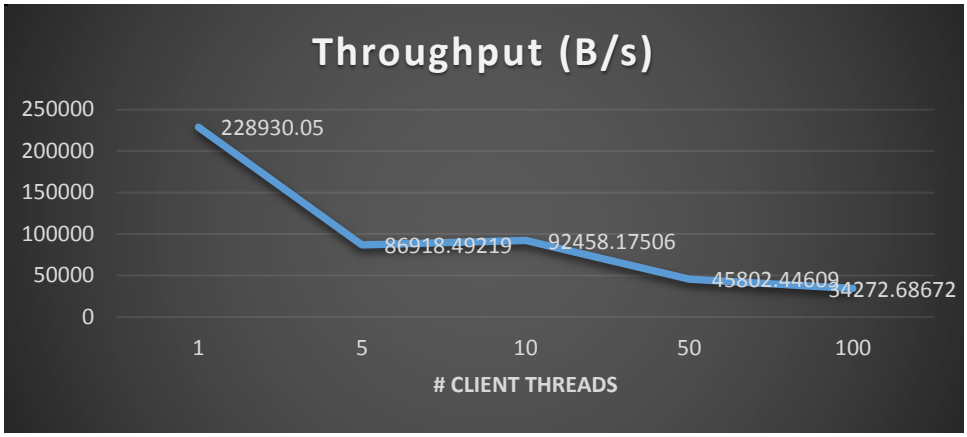
One final note on experimentation: all results reported are averages of at least *five* repetitions of the exact same experiment.

Michael Landes
CS 8300 (GOS)
10/2/12

Increasing the Server Load:

First we examine load on the server. We will start the server running with just 1 worker thread, and would expect that as load increases, this one thread degrades in its capacity to handle the requests. We will run the client, varying the number of threads used, and having each thread send 200 requests. The server will be serving 50 different small documents.





First notice accuracy. The server seems to be pretty accurate. The server does indeed send all the bytes it is asked to send. This is also a result of the implementation of the client. The client doesn't have a timeout in the wait, and will receive all the bytes that are sent eventually (unless the socket closes). Therefore I have a fairly accurate server implementation.

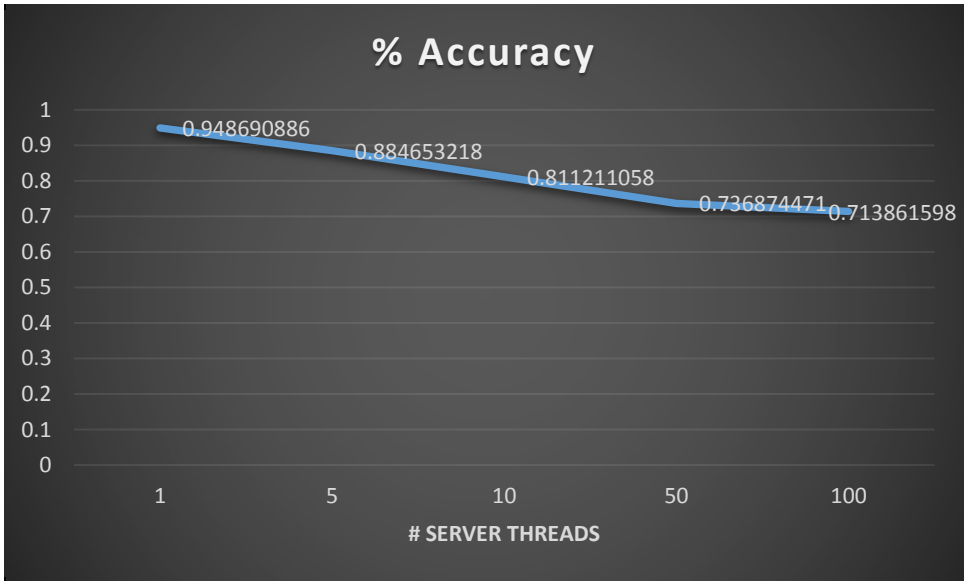
Now notice throughput. The leftmost point looks like an outlier, and it is, but not for statistical reasons, but for mathematical reasons. This is one request from one thread, and happens so fast, that the elapsed time approaches zero, which makes the throughput approach infinity. This data point is fairly useless for our analysis. Next, notice that there does indeed appear to be a decline in throughput as the load increases on the server. We cannot be sure though that this decrease is the client's fault, or the server's. The client remember is launching up to 100 threads on one machine, and the decline in throughput may simply be the result of a decline in request rate from the client as the machine gets overloaded.

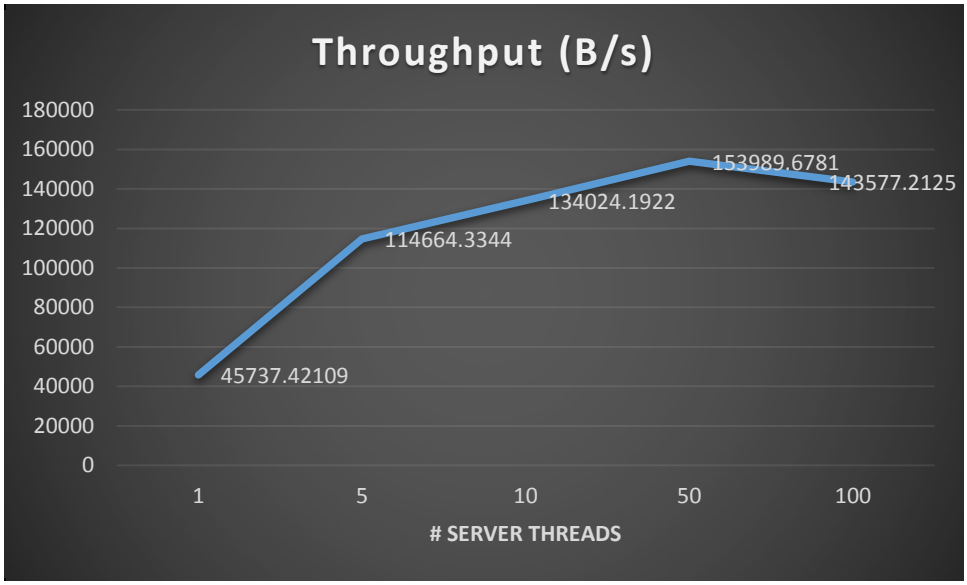
Remember that this is the performance of a server with exactly one worker thread, so there is no real multithreading going on here. Here we seem to be at an ideal state when the client uses 10 threads implying 10 simultaneous requests, however there are a number of other factors to examine.

Michael Landes
CS 8300 (GOS)
10/2/12

Multi-Threading the Server:

So now we will test the premise directly. The question we ask is whether a multi-threaded server increases performance. For this experiment set, we will hold the client constant at 50 threads sending 200 requests each for 50 different files. This time we will vary the amount of threads the server uses.





Looking at throughput, we can see that the trend is as expected. As we increase the number of server worker threads, the rate at which we return data increases. Multi-threading works! So why is there a drop off after 50? Well, remember that we have 100 threads running on a single machine with far less than 100 (or even 50) cores. There is a threshold at which managing threads surpasses the gain we get from using threads, and it seems to be 50 threads in this case.

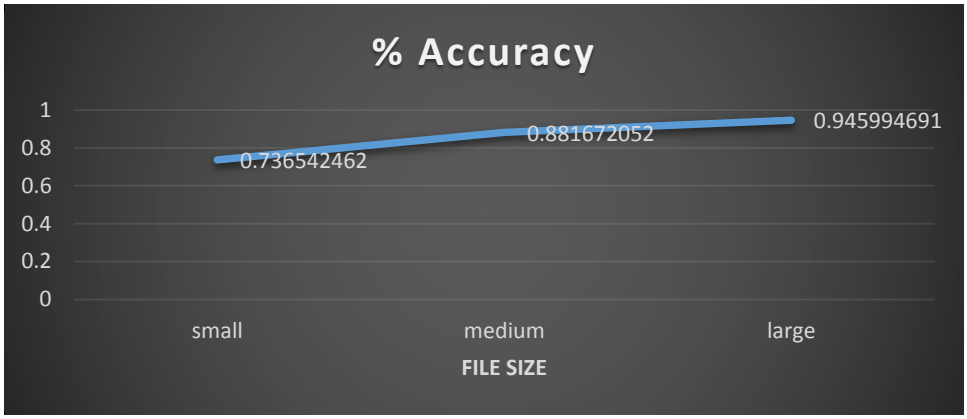
A perhaps more interesting observation is that accuracy decreases as we add more threads. In my code, this only can happen if a socket closes. The client isn't changing at all, so loading up the server with threads must result is more errors in data transfer (which is fairly rudimentary in my implementation).

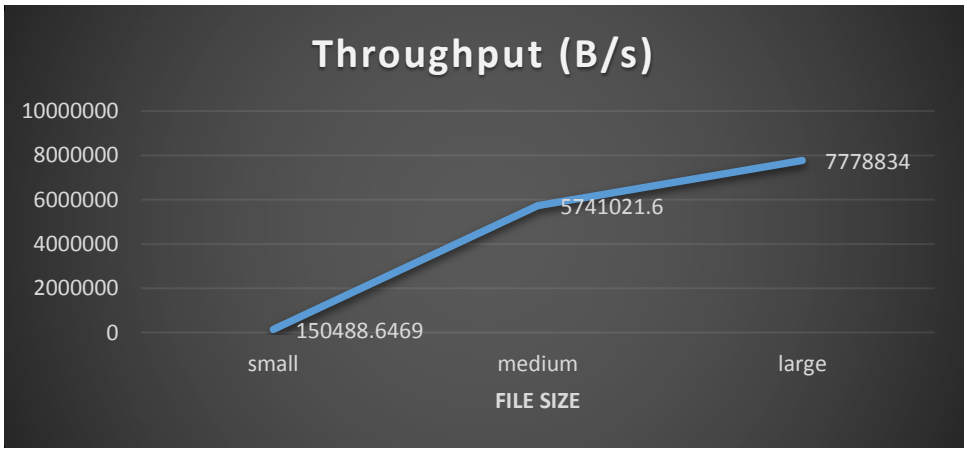
Michael Landes
CS 8300 (GOS)
10/2/12

Heavy Files:

So if using 50 threads on the server maxes out capacity, what happens when, while all other factors remain constant we increase the size of files being sent? Basically this means that each of the packets of information being sent has a different size. Before we were sending hundreds of bytes per request, let's try ending thousands, or even millions of bytes per request.

In terms of this test, a small request should return 463 B, a medium request should return 68290 B, and a large request should return 1176874 B.





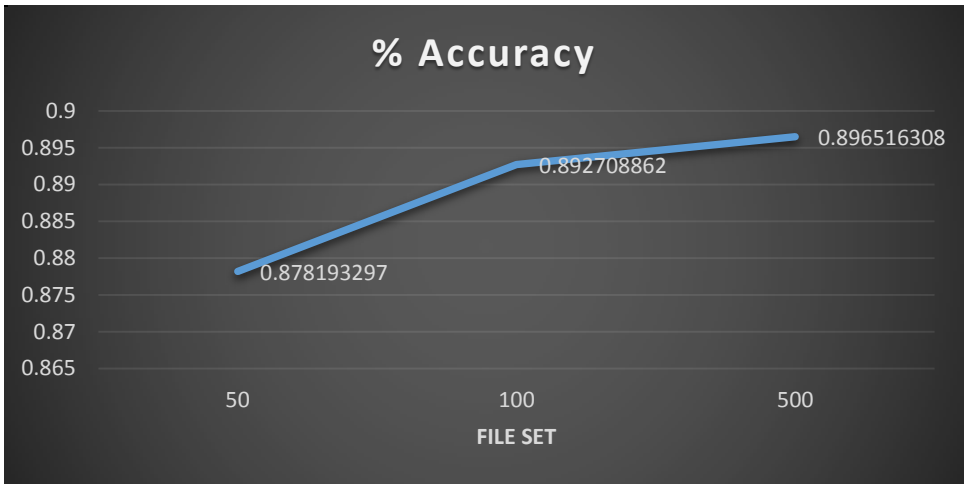
The great mystery of the accuracy graph continues! Here we see that as file size increases, accuracy actually increases. This further supports the hypothesis that the problems with accuracy are found in the socket transmission part of the code, and sockets being closed. Because here, we have less sockets with bigger files at any given moment passing through the system, and there is less chance for fault.

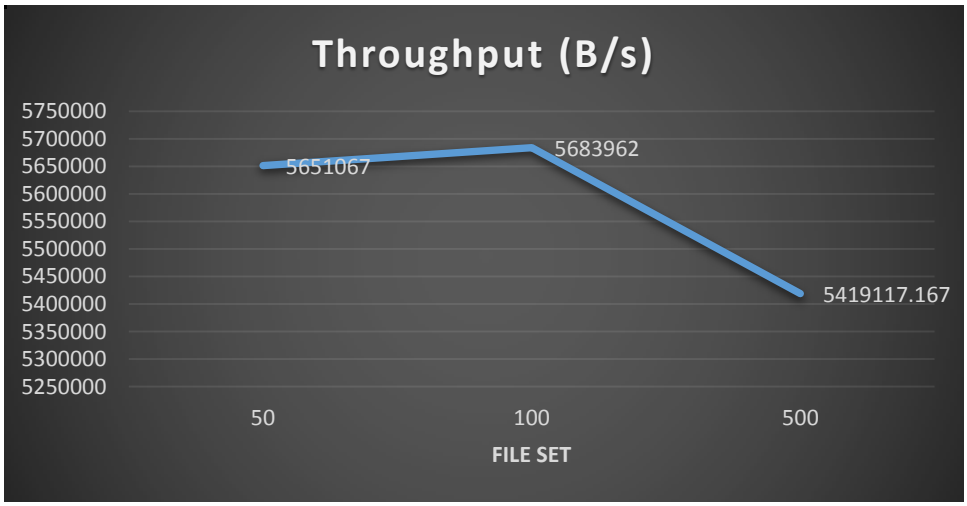
With throughput, also notice there is a vast increase in performance for the large files. Again, once the connection is set up, pumping data through it is less of a problem. The fewer connections, with more data a piece are being handled better.

Michael Landes
CS 8300 (GOS)
10/2/12

Circumventing the Cache:

Ultimately threads are most effective in an I/O bound situation. Thus far I have been testing on a server serving 50 different files, which may or may not be enough to take advantage of serving files that don't end up in the cache. Here I serve medium sized files (see above), again with 50 client threads each sending 200 requests to a server hosted by 50 workers. This time, however, I will choose from 50, 100, and 500 files, and see if there is anything interesting to notice.





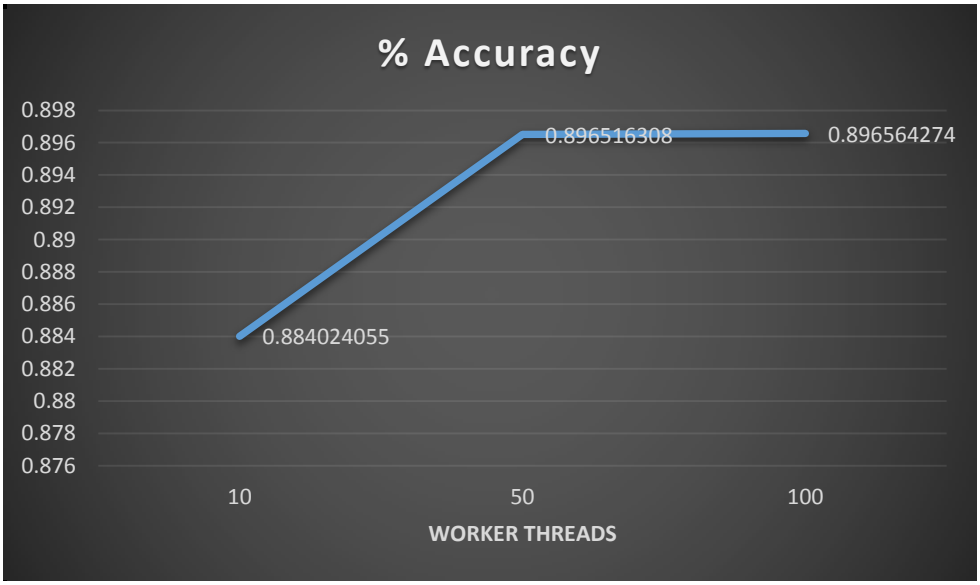
Again accuracy seems to increase as we increase the number of different files that we are trying to access. I think that because the send thread and the receive threads block while I/O is taking place, this slows them down slightly, and prevents them from pre-maturely closing sockets as often.

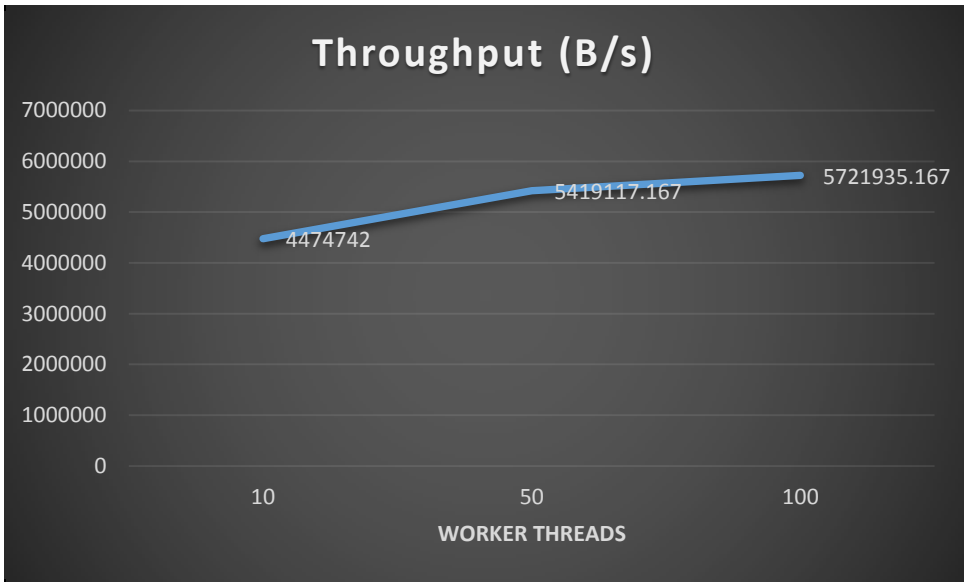
We also see that throughput increases slightly, then decreases when we move from 100 to 500 sized sets of files. The graph makes this change look more drastic than it really is, however it seems to suggest that somewhere between 100 and 500 sized sets, we hit the threshold where the cache loses its capacity to advantage our server.

Michael Landes
CS 8300 (GOS)
10/2/12

Putting it all together:

So, we have discovered that file size and file set size are important, especially because they make the whole system more I/O bound. So now let's revisit the experiment with varying the worker threads in the server and see what the improvement graph looks like now. Here we use the medium sized files, have the client request 200 files for each of 50 threads, and request from a set of 500 different files.





As you can see, now that the system is more I/O bound than before, the increase in worker threads makes an improvement as opposed to getting in the way. This is because there is more blocking, and less socket scheduling contention happening in any given moment. Also notice that accuracy is better, again due to the increase in I/O bound work. Future experiments may further test this hypothesis about accuracy.

Michael Landes
CS 8300 (GOS)
10/2/12

Answers to Questions:

The serer did not once fail while running these tests. If a connection doesn't work out, it just closes the socket, which we observed caused a slight bit of inaccuracy in the system in terms of transferring bytes. Differences were observed when varying both client and server threads.

Conclusion:

As the premise stated, multi-threading is a good way to improve the capacity overall of the server to interact with multiple clients. We observed with these experiments that this was indeed the case. There are a number of possible future questions, such as what exactly was the cause of inaccuracy. I did perform some basic tests with httperf and found that accuracy overall was pretty good, however not perfect. Using httperf, the failures seemed to be caused by timeouts in the client waiting on the server. This could be further analyzed to get a more clear idea of accuracy.

Overall, I was satisfied with the results of these experiments. They agreed with the things we have been learning in class, and my server did a great job handling the loads!