I ran the experiment.py at my home in Springville, UT. I used the following files in my test:

"small","http://corbt.com/files/design-philosophy-sigcomm-88.pdf",
"medium","http://corbt.com/files/Delta-Rae-Morning-Comes-Live.mp3",
"large","http://corbt.com/files/poster.pdf"

My methodology was as follows:

- Use the header from the URL to get the content-length. Divide the content-length by the number of threads, in order to know how many bytes each thread needed to get.
- I used the number of bytes needed for each thread and incremented it, so that each thread downloads approx. the same number of bytes, and that all of the bytes add up to the content-length.
- For each thread I created a 'DownThread' class. Which I then called run on. In this run method I downloaded the data and stored it in the 'DownThread' object (i.e., self.data = data).
- I then joined all of the threads. Once they were joined I opened up a file with that writes bytes and iterated through the threads writing the data to the file (i.e., f.write(thread.data)).
- After that I simply printed out the URL, number of threads, content-length, and the time that had passed since creating the threads.

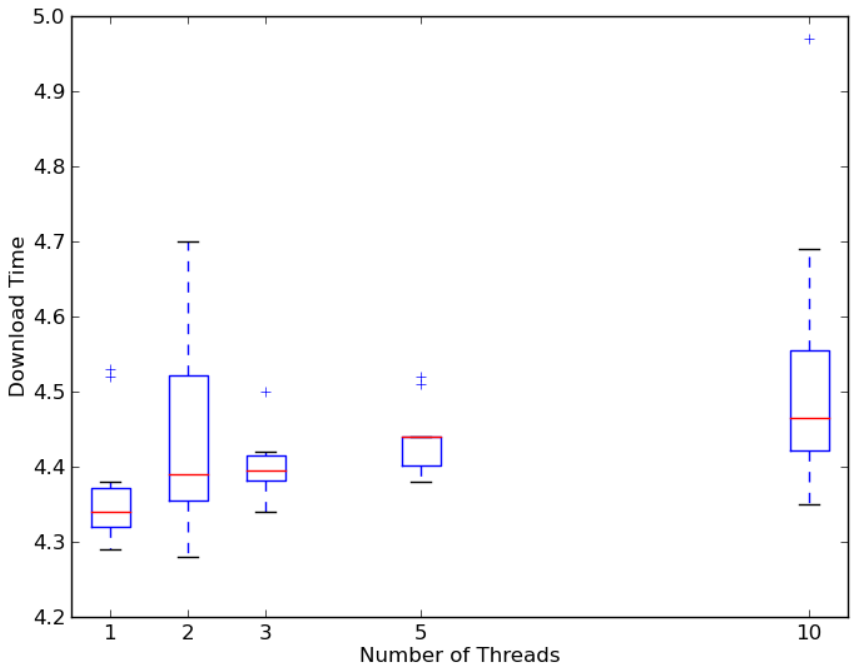
For all three sizes (small, medium, and large) there was a trend that caused the time to grow as the number of threads grew. The change wasn't much; the difference between 1 and 10 threads was about a 10% increase in time, for all file sizes. You'll notice in the "Large" graph that the upward trend is a lot smaller than it appears in the "medium" graph. In fact, in the "Large" graph the 10 thread average time was barely higher than the 5 thread average.

The reasons to do multi-threading for downloads is that you can get a greater percentage of a server's band width. For instance, if one thread was giving you 1/1000 of the server's band width then in theory having 10 threads should give you roughly 1/100 of the server's band width. Thus, you should be able to download more data in a shorter amount of time.

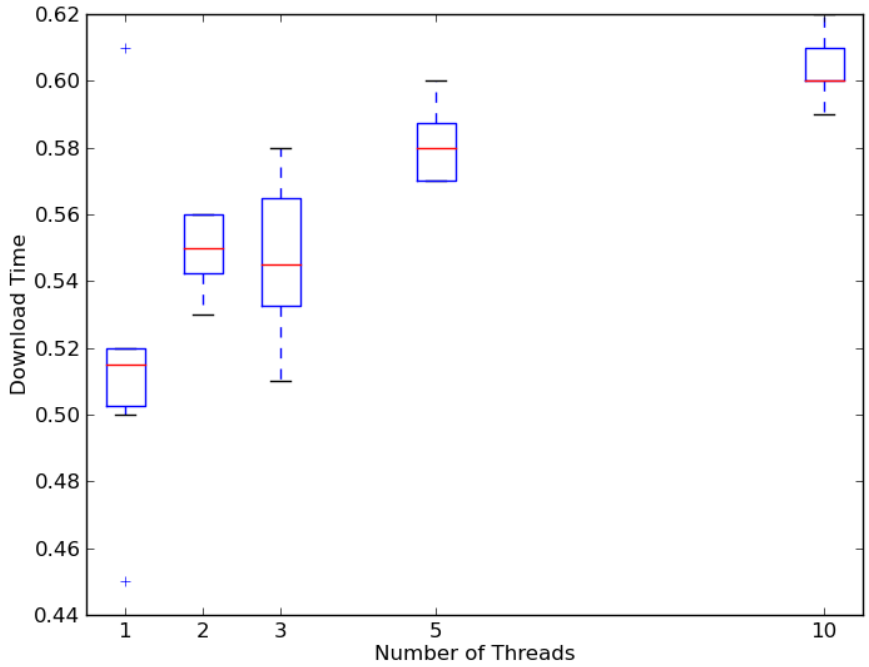
One trade off comes in with the idea that you don't necessarily need to always use more than one thread. Maybe just one call is enough to get all of the information. Another tradeoff for me was that I wanted to keep track of my threads and ensure that I wasn't downloading things and writing the bytes in the wrong order, so I used a semaphore (lock) on my threads. Because of this, I had some threads waiting for other threads to be processed. For example, maybe thread 8 returned already but we're still waiting for thread 7. Thus, I saw that there was a lot more of this wasted time creating and managing the different threads when the file was smaller.

I imagine that there was a good chance that my computer's internet connection was the bottleneck. If the bottleneck were the server that we were downloading from, one would expect (especially with the larger files) an increase the number of threads to lead to a faster the download. However, if they are giving me all the band width I can handle then increasing the number of threads won't help me, in fact, it would slow me down slightly since I would need to process those extra threads.

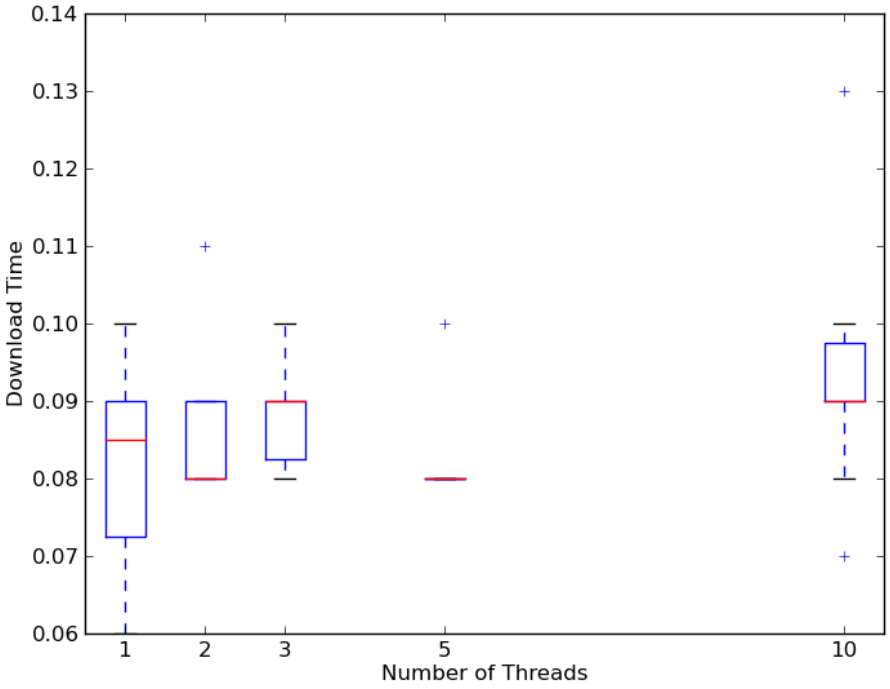
Large:



Medium:



Small:


0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06
Download Time
1
2
3
5
10
Number of Threads