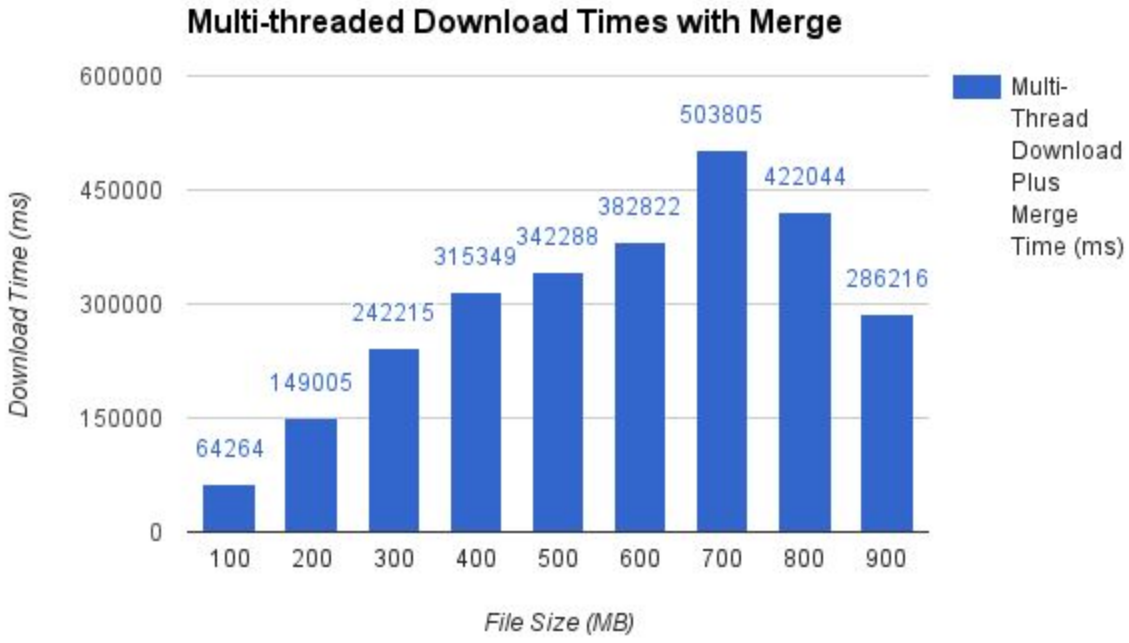
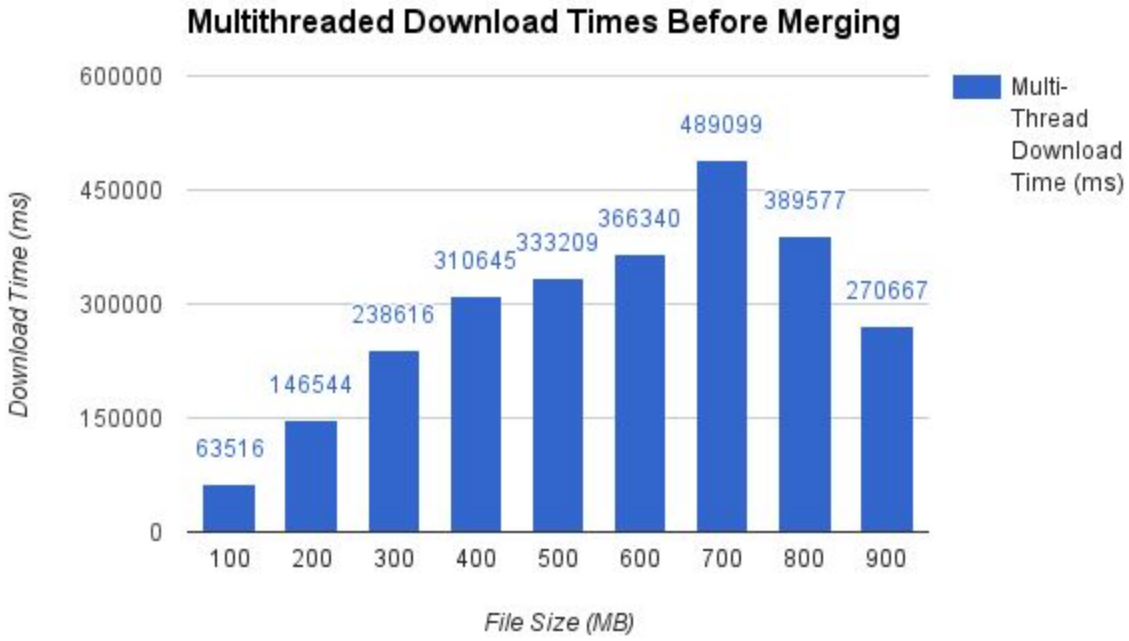
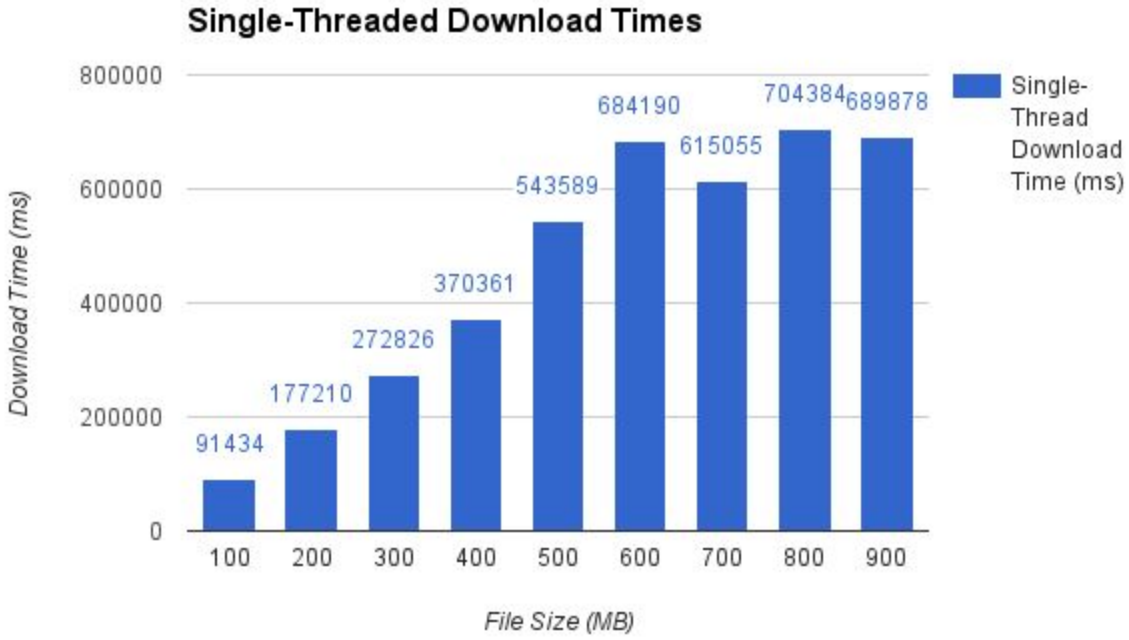
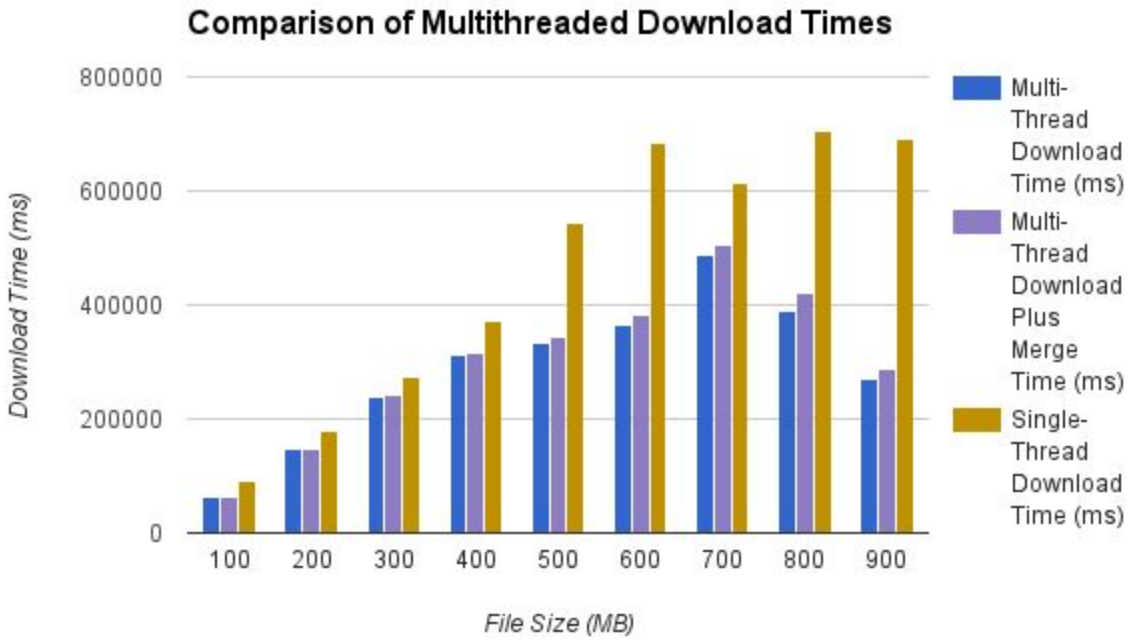
Graphs


Multi-threaded Download Times with Merge
600000
450000
300000
150000
0
Download Time (ms)
64264
149005
242215
315349
342288
382822
503805
422044
286216
100
200
300
400
500
600
700
800
900
File Size (MB)
Multi-Thread Download Plus Merge Time (ms)


Multithreaded Download Times Before Merging
600000
450000
300000
150000
0
Download Time (ms)
63516
146544
238616
310645
333209
366340
489099
389577
270667
100
200
300
400
500
600
700
800
900
File Size (MB)
Multi-Thread Download Time (ms)


Single-Threaded Download Times
800000
600000
400000
200000
0
Download Time (ms)
91434
177210
272826
370361
543589
684190
615055
704384
689878
100
200
300
400
500
600
700
800
900
File Size (MB)
Single-Thread Download Time (ms)


Comparison of Multithreaded Download Times
800000
600000
400000
200000
0
Download Time (ms)
100
200
300
400
500
600
700
800
900
File Size (MB)
Multi-Thread Download Time (ms)
Multi-Thread Download Plus Merge Time (ms)
Single-Thread Download Time (ms)

Graph Analysis

The graphs suggest that downloading files through multiple threads is still significantly faster than downloading files though one thread. The graphs also suggest that the wireless connections in the dorms are wildly unstable, and that merge times are affected more by the spread of available space on the hard drive than the actual size of the file being downloaded.

Lessons Learned

I learned a lot of important things about networking. Not only did I learn how to send files between computers, I also learned several techniques to make file transfers more efficient, and how to circumvent limited memory space during transfer.

Problems and Adapted Solutions

I tried making the server let the user decide what file to host and making the client allow the user to decide where the final file was saved. Trying to overwrite a file through this method with the client led to each portion being written several times in a row in the same file. I solved this by making the client delete the downloaded file partitions after the client finished writing them to the final downloaded file.

When attempting to download larger files, the provided code began throwing exceptions regarding memory usage (starting after the 300mb file). The byte array that was holding the entire contents of the file to send used too much heap space, and quickly left the JVM out of memory. To counteract this, I could either dedicate mountains of RAM to the JVM to compensate, or make the program use less RAM. I opted for the latter by adding a buffer to the server threads and client threads so they could process the file in several pieces, and adapting both to use their file input and output streams without keeping the files themselves in memory. Adding the buffer instead of sending each byte as it's read also proved to increase the speed of sending and writing the files.