Project 3

David Luo

811357331

Introduction

One of the core problems that networking tries to solve is the transport of data. A protocol that aids in such a task must be able to transfer files from source to destinations as quickly as possible without sacrificing the integrity of the data in transit.

In Homework 4, we implemented a simple single-threaded file transfer protocol; however, on some systems, using only a single thread may not be using the processor to its fullest capabilities. This project extends on the previous homework by implementing a multi-threaded solution and examines the factors that may affect the performance of such a protocol.

The Protocol

The protocol relies on a coordinating server to connect receivers to senders using a generated ID number. When a receiver connects to the server, an ID number is reserved and is only released when the receiver disconnects from the server.

On the sender end, the sender connects to the coordinating server and requests the address associated with a specified ID. The sender then connects to the address, thus beginning communications between the two clients. As with the receiver, the connection to the server remains open until file transfer has been completed.

Once the two clients are connected, the sender first sends the metadata about the file: the filename, filesize, and the number of connections it wants to transfer over. The receiver responds with if the file already exists, and if not, gives the go ahead to start sending files. The sender then distributes the file among threads using a partitioning scheme that tries to ensure that each thread is given a number of bytes that was a power of 2 (e.g. 11 bytes over 3 threads would be split 4, 4, and 3). Each thread then connects to the receiver and transfers its portion of the file to a thread waiting on the receiving end. Each receiving thread receives the filename currently being written to, the offset of the data being sent, and the length of said data. In this way, the protocol is able to use many threads to read, write, and transfer from the same file.

Methodology

In the implementation, there are two major variables that may affect the transfer time of a file of a given size: the receiving buffer size (henceforth referenced as the "chunk size"), and the number of connections opened between the sending and receiving clients. The effect of these factors was measured using automated scripts (test_receiver.py and test_sender.py), automatically iterating through combinations of chunk sizes and number of connections (ranging from <16 bytes to 1 petabyte and 1 to 50 connections, respectively).

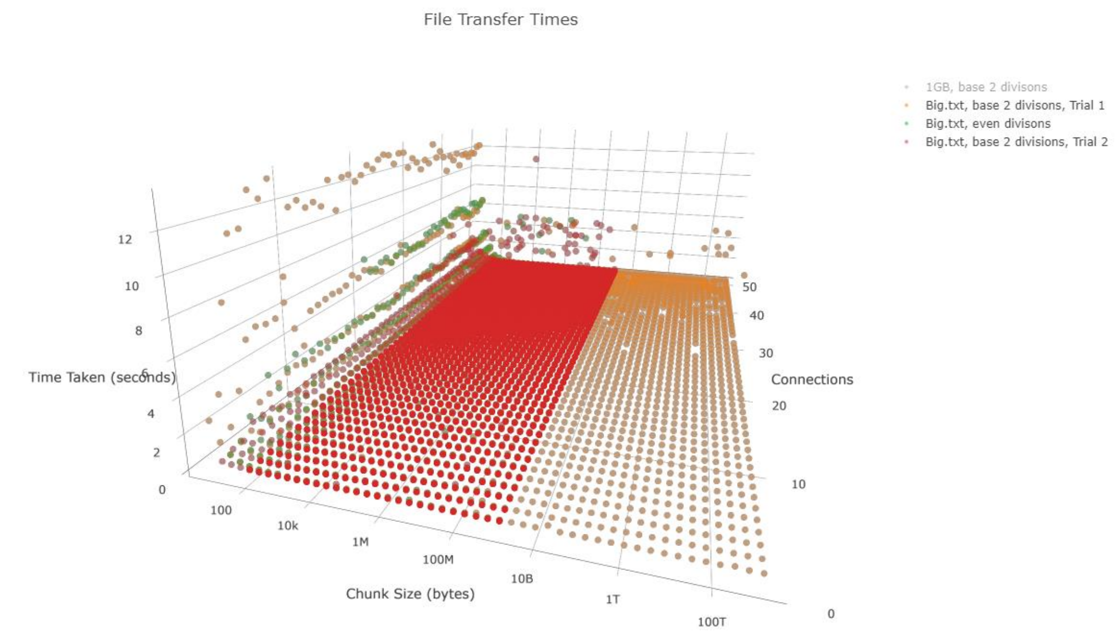
Essentially, test_receiver.py would communicate with test_sender.py to coordinate the sending of files between two different machines. For each combination of chunk size and connection number, the receiver script would initialize a new ftclient in receiver mode, ask the sender script to send over a pre-configured file, and then stop the receiving ftclient, recording the metadata of the file transfer to disk. This is repeated in a configurable number of times. The resulting file can be imported into a spreadsheet or analyzed using scripts. The experiment can easily be repeated with the aforementioned scripts using the instructions in the README.md file included with the submission of this project.

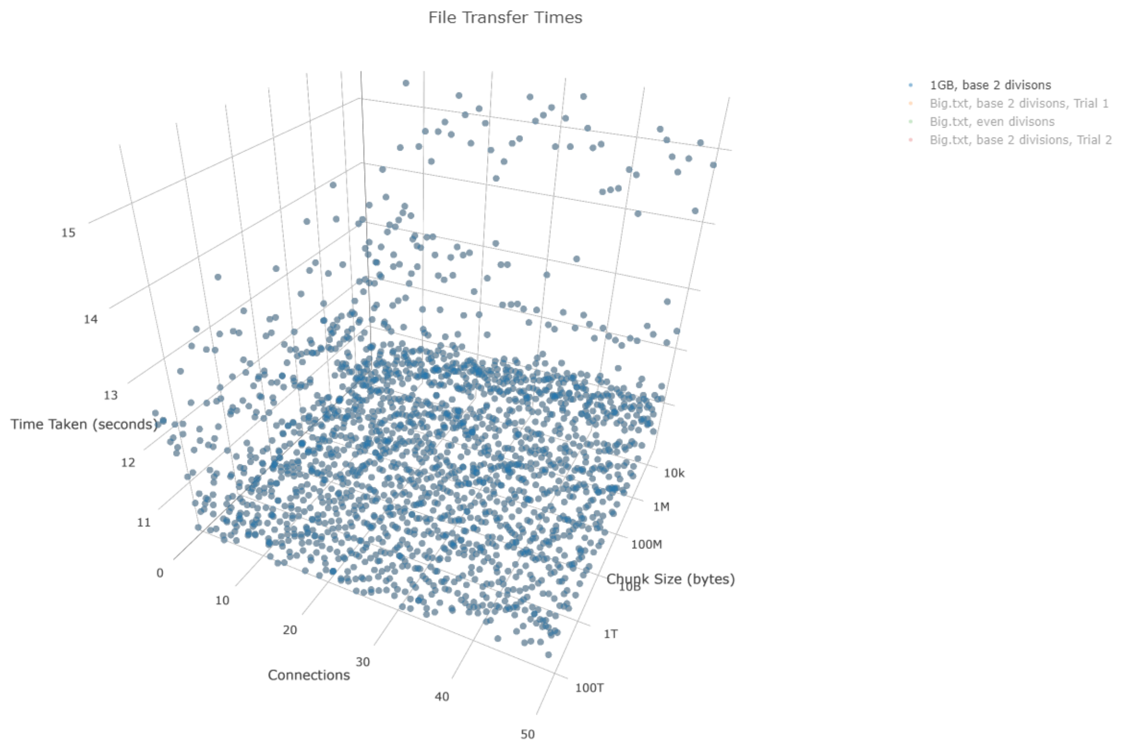
The experiment in this run was produced using one trial of transferring a 1 GB file over 1 to 50 connections with a chunk size of 2 to 2^{50} bytes and a timeout period of 20 seconds. Trials were also run with the "big.txt" file provided in Homework 4.

Additionally, while most of the trials were run with the power-of-2 partitioning scheme, another experiment was conducted using an even partitioning scheme, where every thread was given roughly the same number of bytes (e.g. 11 bytes over 3 threads would be split 3, 4, and 4), using "big.txt."

The graph on the following page was produced with these parameters.

Results


File Transfer Times
1GB, base 2 divisons
Big.txt, base 2 divisons, Trial 1
Big.txt, even divisons
Big.txt, base 2 divisions, Trial 2
Time Taken (seconds)
12
10
8
6
4
2
0
Chunk Size (bytes)
100
10k
1M
100M
10B
1T
100T
Connections
50
40
30
20
10
0


File Transfer Times
1GB, base 2 divisons
Big.txt, base 2 divisons, Trial 1
Big.txt, even divisons
Big.txt, base 2 divisions, Trial 2
Time Taken (seconds)
15
14
13
12
11
Connections
0
10
20
30
40
50
Chunk Size (bytes)
10k
1M
100M
10B
1T
100T

The above graphs are much better viewed interactively at: https://plot.ly/create/?fid=dsluo:1, or in the included plot.html file.

The raw data used to create these graphs is included with the submission of this project in CSV format.

Conclusion

Clearly there is a relationship between the chunk size and the transfer speeds. Decreasing the chunk size greatly increases the transfer speeds, but doing the opposite gives diminishing returns. The "optimal" chunk size seemed to be dependent on the file size, however, with the "leveling off" point for "big.txt" at around 512 bytes, but around 4 megabytes for the 1 GB file. This makes sense, since having too small of chunk size leads to an increased number of calls that must be made to read the entire file from the receiving buffer.

The number of connections also has a positive relationship with transfer speeds. It is particularly pronounced in the 1GB file test, where transfer times may be cut almost in half by using more connections. This probably because more connections would allow more utilization of the processor and shift the bottleneck on either the storage drive or the network.

The different partitioning schemes seemed to have no differentiable effect on the transfer times. The idea behind using a base-2-division partitioning scheme was that it would reduce the number of packets that would need to be sent, but this probably doesn't make too huge of a difference on this scale, as it only affects at most a number of packets equal to the number of connections being made. Further investigation would be required to determine if this has any effect on large connection pools or larger file sizes with better (or worse) hardware.

Clearly there is more to transfer speeds than what this experiment tested for. Extensions of this experiment may involve testing using different hardware conditions by artificially imposing limits on drive read/write speeds, on processor speeds, or on network I/O speeds, or by testing the difference on mechanical drives and solid state drives. Furthermore, more trials can be done testing the effect of different sized files or measure the effect of on-the-fly compression to save on bandwidth. The entire protocol could also be rewritten in a language that better supports threading, as in Python, the Global Interpreter Lock prevents more than one thread from executing instructions at a time.

In conclusion, this experiment showed that increasing chunk size and the number of connections generally increased transfer times, but with diminishing returns.