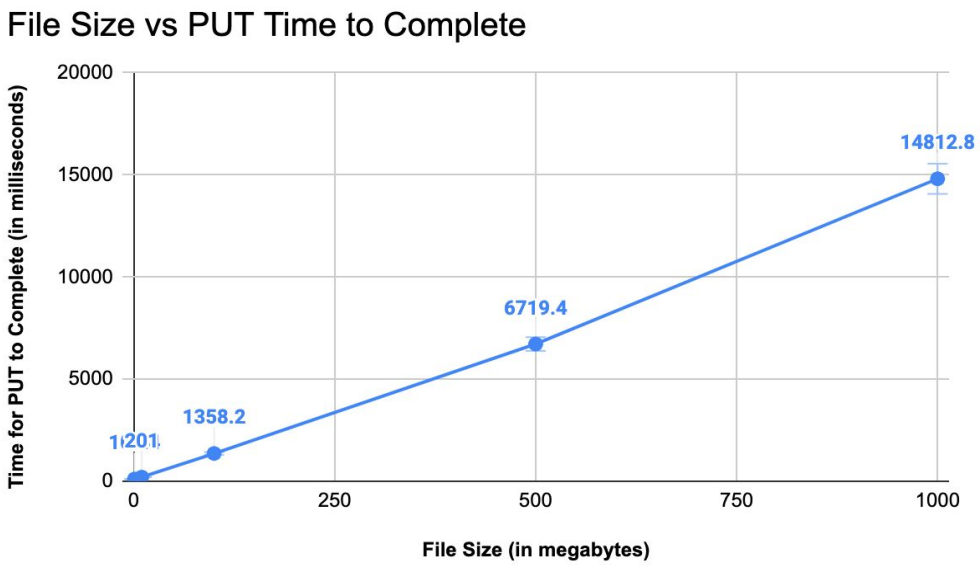
Ishaan Sehgal (netid: isehgal2) • Heesoo Yang (netid: heesooy2) • Asher Dale (netid: asdale2)

**CS425 MP2 Report - SDFS Distributed File System**

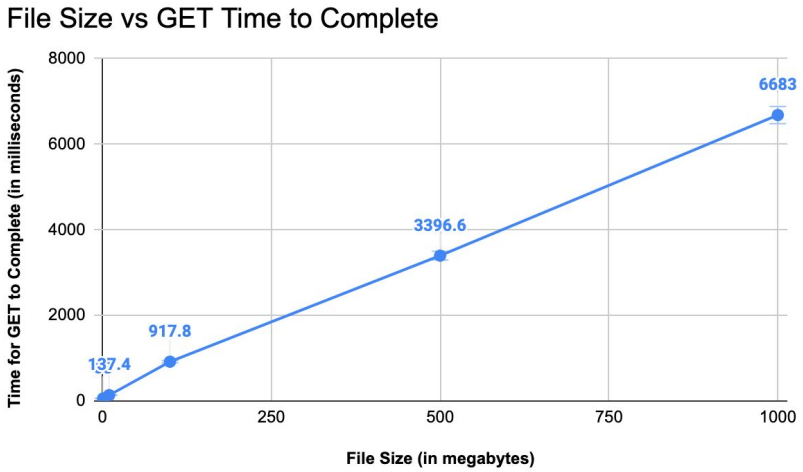
We implemented our Distributed File System using Golang, and for platform-independent messaging we used Google Protocol Buffers. We used our membership service from MP1 for maintaining membership lists, but we also added modifications to support “Block Reports” to the master (such as HDFS does) and master re-election. For “Block Reports,” we modified the gossip protocol so at each heartbeat period, each non-master node sends a message to the current master along with what files it is storing. This means that the master tracks what files are stored at each machine at all times. For master re-election, we added “MasterID” and “MasterCounter” fields to the gossip message so we can infect other members of the system with a new MasterID if the MasterCounter on an incoming message is greater than their local MasterCounter (the MasterCounter is incremented when a new master is chosen). We reach agreement on the new master by hashing each machine to a point on a hash ring and picking the lowest machine on the hash ring as the new master. Thus, each machine has a way to independently verify who the new master should be when they receive messages with higher MasterCounters.

For the file system, we are storing the full file rather than splitting it up into blocks. For large files, however, we send it in small pieces so we don’t run out of memory. For networking, we use TCP for all filesystem communication, including communication between master and replicas and between client and master. For TCP, we open connections only when a request needs to be processed and close them right after instead of keeping connections open all the time, so that we can scale more easily to more machines. We defined a new SDFSMessage struct using ProtoBuf to include all the relevant information we needed, such as the request type, filename, etc. For writes we do “ALL” and for reads we use “ANY”, because we guarantee that all replicas have the latest version of the file. To allow multiple readers and a single writer, we use Go’s built-in reader-writer mux with multiple goroutines.

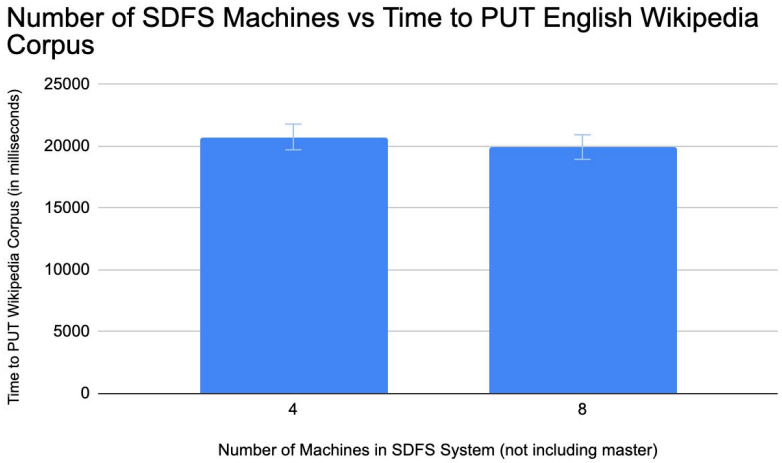
Lastly, for our replication strategy, we use the membership service’s failure detection as a trigger for replication. To initiate the replication, we take advantage of our existing request structure by simply having the master tell a newly chosen replica that they should send a GET request back to the master for a given file.



The above plot is what we expected because the time for a PUT operation to complete scales linearly as the file size increases. This makes sense because we parallelize our PUTs to multiple machines, so the time to successfully PUT a file is directly proportional to the size of the file. We PUT the file to four replicas so we can be fault tolerant up to three simultaneous failures, so even though we used 10 machines in this system for collecting data points the file only gets PUT'ed to the minimum amount of machines required for our fault tolerance.




The above plot is also as we expected, as the latency increases linearly as file size increases, but the overall latency is lower than PUT because GET does not involve fetching the file from multiple replicas. Thus we use less bandwidth, as we use an ANY policy for fetching files because our file system guarantees that all replicas have the latest version of the file. Again, we used 10 machines, but the latency should not change significantly with more machines.




This last plot is also as we expected because even though there are more machines in the system with 8 machines, we always PUT a file to a maximum four replicas, so there should be no significant difference between the PUT operations of a large file between these two systems. Thus, one can see that there is little difference between the PUT times for this experiment.