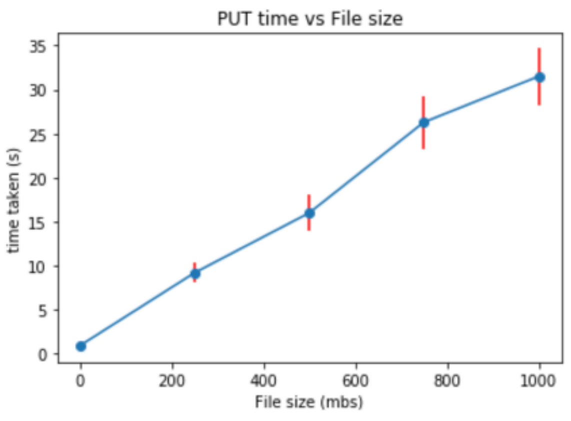
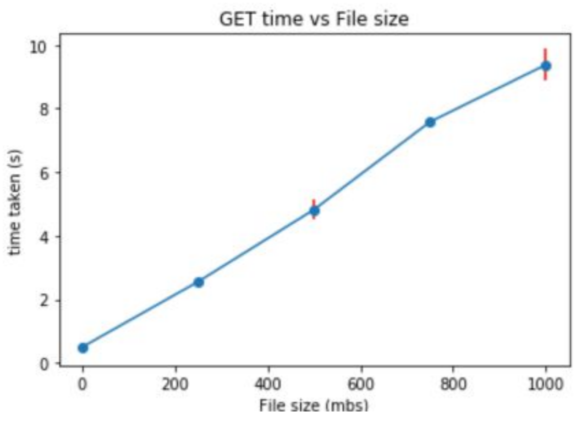
CS 425 MP2 - Simple distributed file system
Ruiqi Peng ruiqip2
Qingyi Huang qingyih2

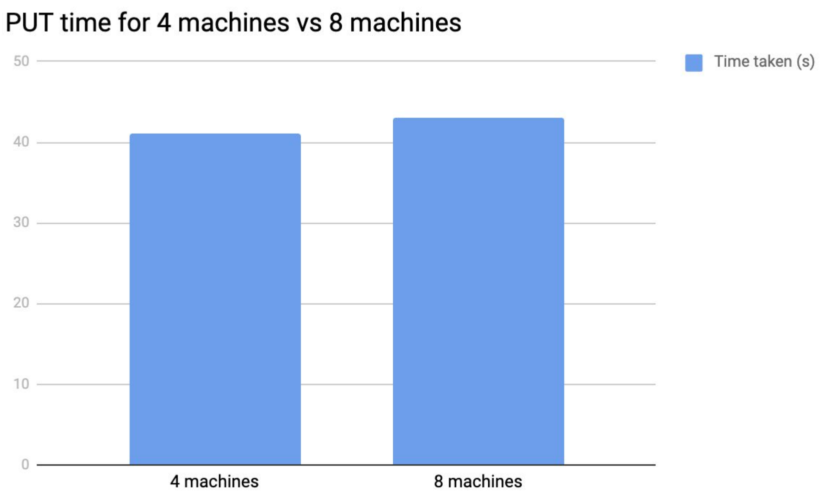
Design Detail

The design for our sdfs is simple. We run the file detector and the sdfs services on different ports, so that we handle the heartbeats from failure detectors and messages from sdfs separately.
We appoint one node to become the introducer node (for failure detection) and the master node (for file system). The master nodes will be storing file directory infos. It is a data structure that maps the file name to a list of hostnames which store one of the replicas of the file.
We let every file system related operation (PUT, GET...etc) initiated from any node to send a message to the master first. The master node will deal with these requests and perform corresponding operations. For instance, if we hope to upload a file from any node. The node will first send a PUT message to the master, then the master will determine the destinations of replicas and send REPLICATE messages (containing the hostname of the source) to those destinations. Finally, the destination will send REPLICATE_COMPLETE upon finishing replicating the file (using scp command to copy files from the source) as an acknowledgement to the master, and the master will update its directory accordingly.
Our system's replication strategy is simple. For every file, we compute its hash value mod 10, let the value be the target node to store the file. In addition, we have 3 nodes that have ids near the target node to store the replicas.
Our system handles read-write conflict and failures as well. For read-write conflicts, for instance, a node tries to GET a file that is being updated. Since the node that is still updating the file hasn't sent back a REPLICATE_COMPLETE to the master, the master hasn't updated its file directory such that GET should have returned nothing. However, we instead place the GET request into a data structure that maps a filename to a queue of requests and return a GET_WAIT to let the client wait for a bit. And once the master receives REPLICATE_COMPLETE, it will clear the queue of requests that correspond to the filename in the REPLICATE_COMPLETE message and deal with all of the requests. For failures, we let the failure detector send a failure message to the master node, and the master node will first, traverse and find the files that need to be re-replicated and for each file it counts the amount of new replicas needed; Seconds, it randomly select some other nodes and let them to store the replicas.





The first graph is showing the relationship between the average time taken to PUT a file vs the size of the file. We could clearly see a linear trend such that the time taken increases as the file size increases. This is totally expected. The linearity is also expected because we distributed replicating the file to different servers. The second graph shows GET time vs File size. It has a similar trend that is totally expected. However, we can observe that GET takes a lot less time than a PUT. This is because our design defines a successful PUT once all of the replicas have been successfully transferred whereas for GET we would only need to initiate one connection.



The PUT time taken to store the English Wikipedia corpus (1.3gb) is roughly the same for the system with 4 machines and 8 machines. This is totally expected because we will make 4 replicas anyways.