Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)

(a) Design:

The design for our SDFS consists of a single leader and multiple worker nodes (the leader is also a worker node). The put, get, delete, and ls commands are sent from the client goroutine on any machine to the server goroutine on the current leader, and the leader determines relevant sources/destinations and sends that information to whichever machine(s) need to initiate a transfer of data. Our SDFS was written entirely with RPC functions, so reads are started by commanding another machine to write to you.

Our SDFS decides where to put new files and where to replicate files from failed machines by keeping track of the total number of bytes being stored on every machine. The leader will choose the machine(s) with the current fewest number of bytes to store new/replicated files in order to maintain an even distribution of data across all nodes. Our implementation also checks at all writing stages whether the source or destination is local so that those writes can happen locally rather than going over the network, which saves bandwidth and time.

(a.1) The replication level we decided to use was four replicas– at any given time the SDFS will have four replicas of each file, provided at least four machines are online. If less than four are online, then every machine will have a replica of each file. On worker node failure, the leader will consult the hashmaps it keeps locally to track files and servers to determine which files were on the node that failed, and it will replicate those files to other machines.

When the leader fails, we use a simplified version of the bully algorithm to elect a new leader– once the failure has been recorded by everyone, the machine with the highest number (1-10) proclaims itself the new leader. It then pings all other online nodes to get information on the files they are currently storing, which it uses to rebuild its snapshot of the SDFS. Once this process is complete, the leader tells online nodes that it is the new leader– we chose to do it this way so that client commands inputted after the old leader failed wait until the new leader is elected and then execute (they are not dropped).

(a.2) We avoided starvation for reads and writes by creating a single queue for all operations on a file by timestamp (when the operation request was received). This prioritizes consistency over availability. Read operations cost 1 and write operations cost 2; the total operation sum for a file at any given time must be <= 2, which means two reads queued back to back can happen concurrently, but if a write is occurring no other operations can be performed simultaneously. This avoids starvation by serving requests in the order they were received, so a newer request can never be processed before an older one.

When serving multiple read requests to the same file, the leader randomly picks the worker node with the file that will fulfill each read request so that multiple concurrent readers probabilistically get different source machines, which improves read times.

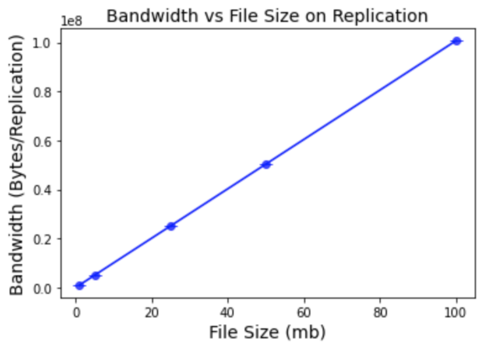
(b) Past MP Use:

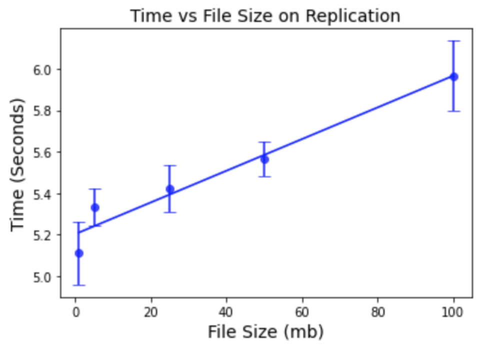
MP2's membership protocol is the backbone of our SDFS; we use it to know who is online at any given time and to detect the failure of nodes in the SDFS. MP1 was useful for debugging and for creating this report; we used it to look through the logs we created while calling multiread and multiwrite.

Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)

(c) Measurements:

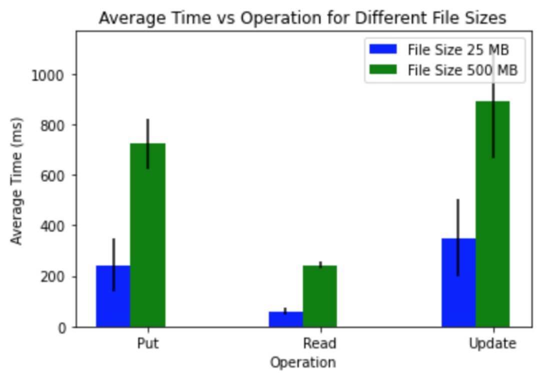
Overheads



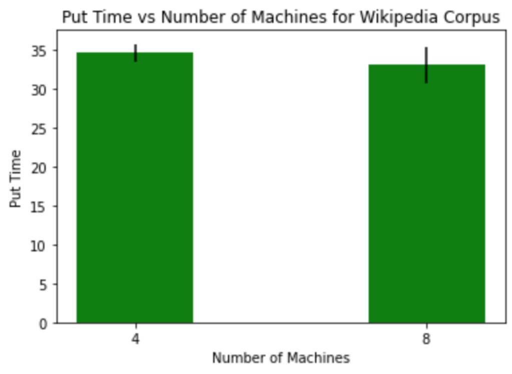


The Overheads plots show how on replication, bandwidth used and time taken are roughly linear to the size of the file being replicated. Also on the time plot, the time taken to replicate the file includes the time taken to detect the failure of the original machine. As the time vs file size plot shows, there is a high variance in the replication time. And throughout our entire testing process, we saw a large amount of variance in the time taken by different operations. This makes sense as there are many variables that affect the speed of the replication, including when failures are detected, and how quickly machines are able to write files to their storage system.

Op times



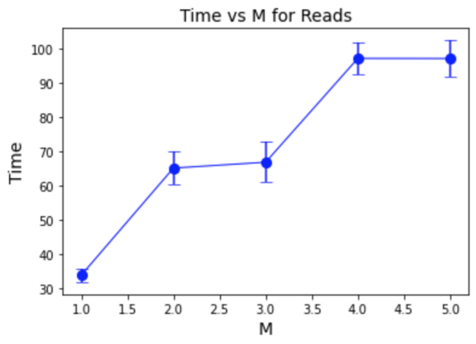
Large Dataset

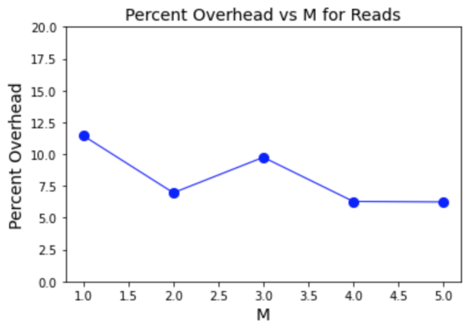


The Op times plots show that reading is much faster than writing (put/update). This makes sense because a read is a write from one source machine to one destination machine, while a write is four concurrent writes from one source machine to four destination machines. Although they are concurrent, we still found write operations to be slower than reads (maybe limited by outgoing bandwidth?). Update operations being slightly slower than normal puts makes sense, because the file has to be cleared before it can be written to. For the Large Dataset tests, it makes sense that the write times are essentially the same because the put time is not determined by the number of online machines as long as there are at least four machines to write to.

Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)

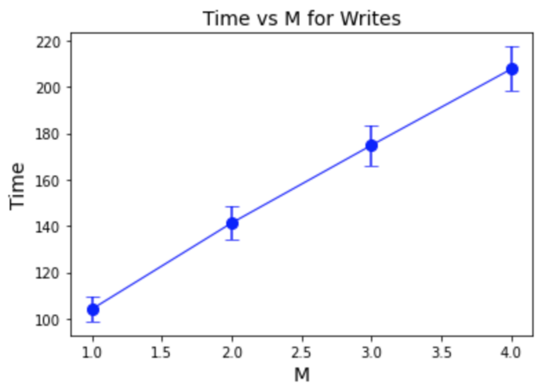
Read Wait

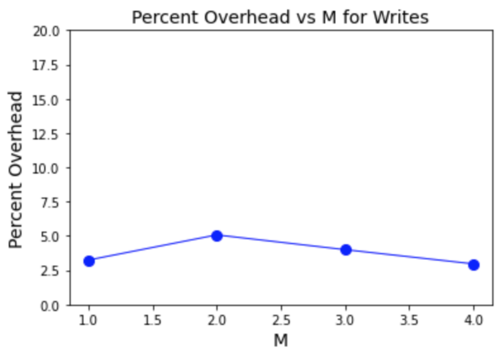




The minimum time for the Read-Wait operation is ceil((m + 1) / 2) * (time to read one file). The file we used was 4.39 gb and took 30.5 seconds to get. As the plot of Time vs M shows, our data follows that trend. However, there is some overhead involved in allowing multiple readers access to the file. This is shown in the plot on the right which has the percent overhead plotted against M. This overhead seems to decrease with an increase in M, which may be explained by the fact that any constant overhead we have is being spread out over more read operations, and the additional overhead from each additional read is not significant compared to that.

Write Read





The minimum time for the Write-Read operation is ((m + 2) * (time to read/write one file)), as there are m + 1 write operations which must complete before the 1 read operation can start. The file we used was 4.39 gb and took 34.1 seconds to put. The Time vs M for Writes on the left shows this; there is an essentially linear relationship between the value of m and the time taken to complete the overall operation. There does seem to be some overhead when having multiple writers queued up to write to one file, as shown by the Percent Overhead vs M for Writes graph on the right. However, it is interesting to note that this overhead seems to be a smaller percentage than the overhead for writing in the graphs above, which makes sense because we cannot have concurrent writes so the overhead from that concurrency is not present.