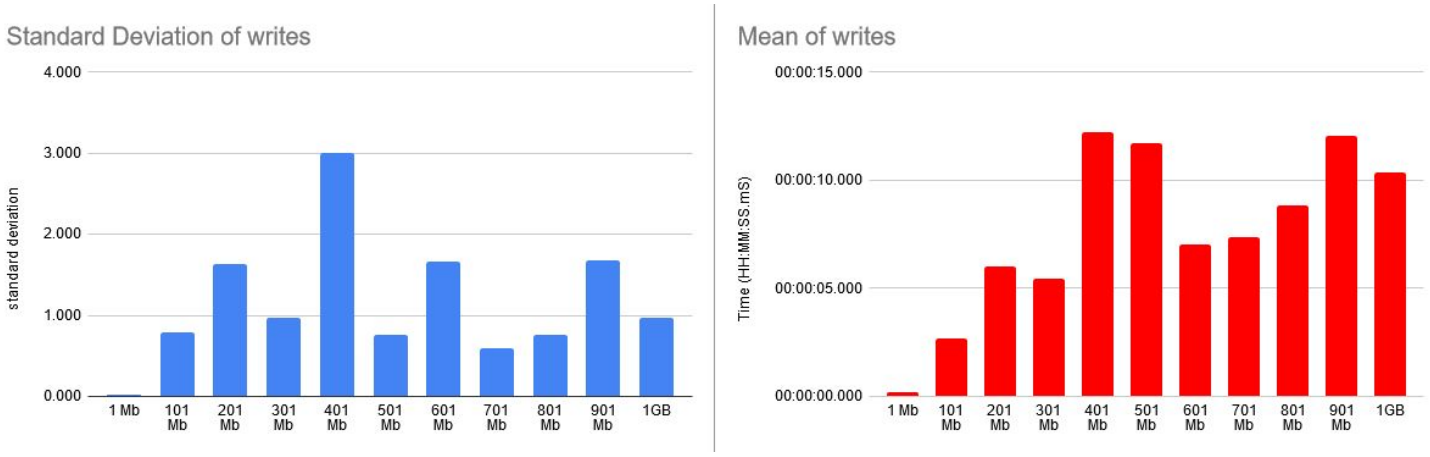
NetIDs: rkroko2, mmallon3 (Group 49)

To implement our SDFS, we split our project into a failure detection component (*fd)*, a SDFS component (*SDFS)*, and a third component called the node manager (*NM)*. The *fd* is almost exactly the same as the MP1 solution with some minor additions to fit the *SDFS*. The *SDFS* (either a slave or master) handles the file transfer, updates, etc. Lastly, the *NM* handle's the essential communication between the *SDFS* and the *fd,* and provides a text interface. The *SDFS* relies on its network members, so the *fd* must be started before the *SDFS*. We decided to separate the components this way because it allowed us to divide development easily, to make small changes easily, and to use MP1's solution without many changes.

Starting our SDFS system begins by executing a python command to set up the *NM* and initialize the *fd*. Then, the node is free to join the failure detection (via a command); after all the nodes have joined the network, the user can enter another command (from any VM's *NM*) to start the *SDFS*. This *NM* will then initialize a master node instance for its *SDFS* and send a message to all nodes in the network to initialize their *SDFS* as a slave. After this, the SDFS system is fully functional and can be used to store/retrieve files; the slaves send all queries/operations to the master, which tracks all file information and queues/handles all incoming operations from the slaves.

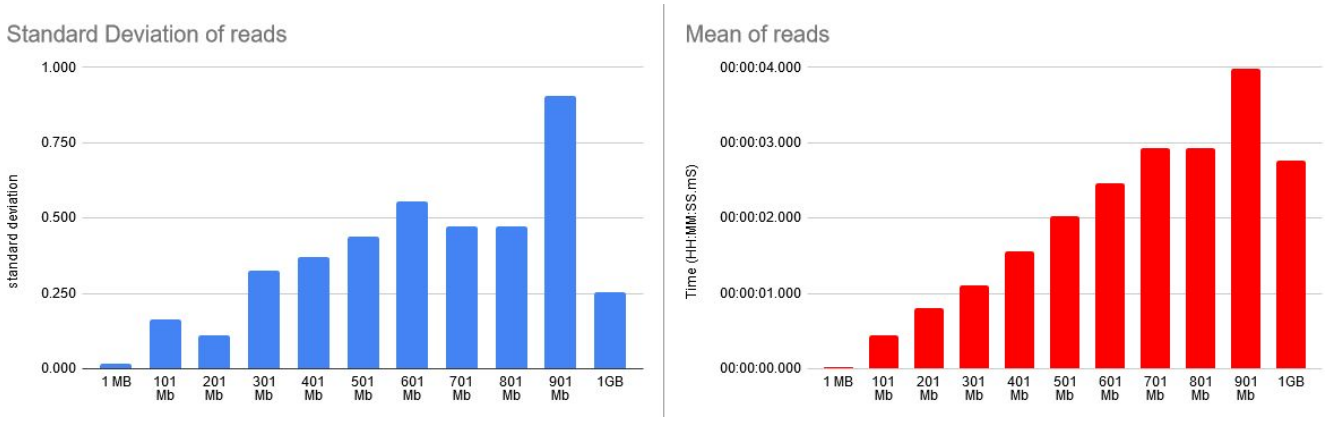
Since our SDFS was required to handle at most 3 simultaneous failures, we replicated the file 4 times ensuring that even with 3 simultaneous failures, the system is guaranteed to still have the files information. To choose the location of the replicated files, we designed an algorithm to choose the 4 machines with the least amount of files on them (if at any point a replica fails, that file is re-replicated until there are 4 again). However, if the master ever fails, the *fd* on each alive node will eventually detect this failure, and begin the election process to find the new master. We simply chose to elect the most recent joined node in the network to be the new master, and the *NM* on each machine updates its slave with this information; however, the *NM* of the new master shuts down the slave component, transfers any files on the *SDFS* stored locally to new slaves, and initializes a new master node instance with the former masters information. Thus, we have the master gossip backup information to all the nodes in the network whenever file/node information is changed (i.e. only when there is a write/delete/leave) so that it is available to the new master upon election.

Put Times vs. File Size



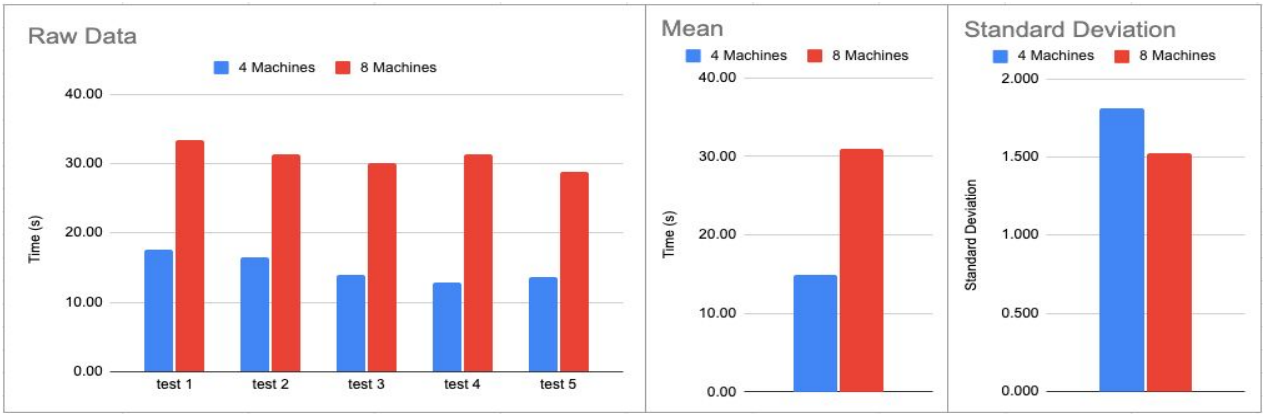
For this process, we wrote 10 files ranging from 100 MB to 1GB in size. Once the first write was put into the machine for each file, then each file had an assigned machine. For each subsequent write of the same file, the file would be written to the same machines. This is what causes the large variance, as machines with a weaker bandwidth could increase the file upload time by a large amount. One way to make this graph more linear is to take a machine's bandwidth into account when uploading files.

Get Times vs. File Size



This graph was much more consistent with smaller values than the write graph, as a machine only needs to communicate with a single file holder to read a file instead of four machines. Variance increases with filesize, as longer download times means more room for error.

Time to Store Wikipedia Corpus



For this graph, the most noticeable trend is the fact that writing to 8 machines takes twice as long as writing to 4 machines. This makes sense given our writing process, as the host machine writes to each of the destination machines one at a time. When its writing to twice the machines with similar bandwidth, then the total time should be roughly double