Our design is based on a leader follower algorithm similar to the bully algorithm. In our design, the oldest node is elected the leader with the 2nd, 3rd, and 4th oldest nodes being sub leaders. Each leader maintains its own copy of the blocks held at each node in a map of file names to a 2d array of replicas storing a copy of each block. When a block is written to a node or deleted from a node, an ack is sent from the follower on which this change took place to the leader. The leader forwards this ack to the sub leaders and updates its own file name to replica mapping.

Once a PUT request is made, the client splits its file into blocks and randomly chooses which ips become replicas for the block. The previous MP is critical for this; it provides a list of ips to each node of the nodes connected to the network. Once blocks are sent over TCP and each follower/replica writes their block, they each send an ack to the leader, allowing them to update their file mappings. If a follower fails during this process while being written to, the client will see this through the TCP connection closing early, resulting in the client reselecting a replica.

For GET requests, the client queries the leader for the file to replica mapping of the file requested. From here, the client randomly selects a replica for each block, sending a request for the data, handing early failures accordingly. Once a request is sent, the follower responds stating how much data is in the block, and sends the block of data. The data is then appended to a local file in order of block indexes.

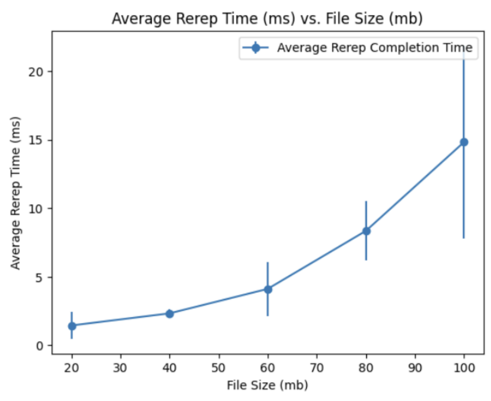
For DELETE requests, similar to GET, the client requests the leader for the file to replica mapping of the file requested. It then sends delete requests to all the replicas. Once a delete occurs, the replica sends an ack to the leader, letting them know to update the file mapping until it can be fully removed.

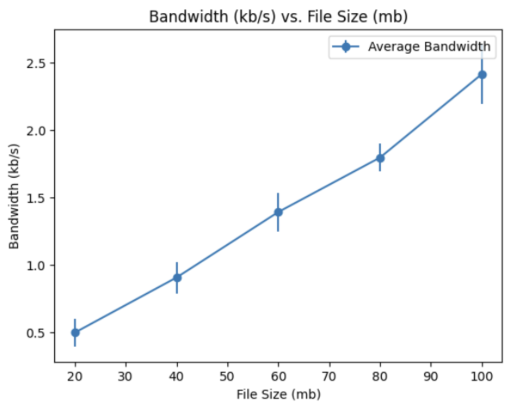
Re-replication is handled every time a node goes down. This is possible thanks to the code we wrote in the previous MP. When a node goes down, we go through the file to replica entries that contain that node and remove them. From here, we force another node to become a replica for the file, forcing a PUT request from a follower that contains the block we want to re-replicate to the new replica.

To avoid starvation, at each node, for operations to be fulfilled on the same file we store the read/write counter, a count of active readers, and a count of active writers. Reads increment the counter, writes decrement it. If a reading operation is inbound, if there are either no writer threads waiting, OR > 0 writer threads, and the read/write counter < 3, we allow the reader thread to continue. If a writing operation is inbound, if there are either no reader threads waiting, OR > 0 writer threads, and the read/write counter > -3, we allow the reader thread to continue.

We used the previous MP to keep track of the nodes alive for sending PUT requests, and whenever nodes died to handle re-replication.

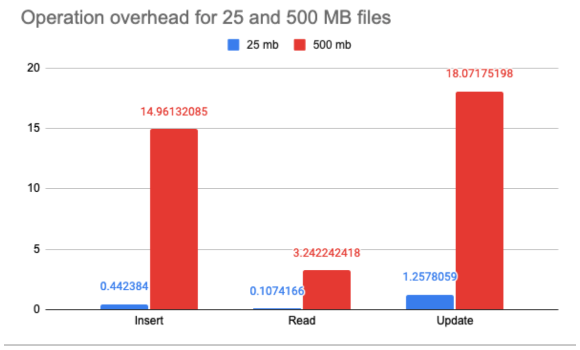
Part 1





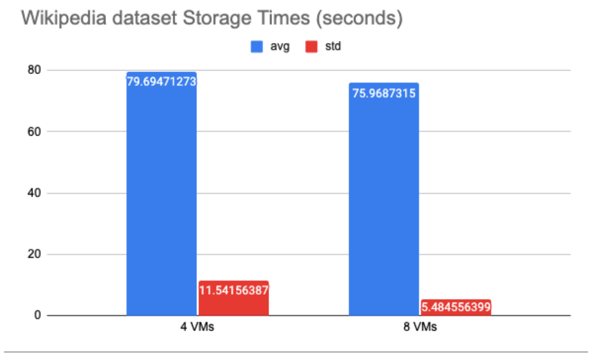
Bandwidth was measured by using nethogs to find the average kb/s being read through the master. These values were measured in a 6 node cluster. From the graph, we can see that both bandwidth and rerep time increase with file size. What's interesting is that rerep time increases exponentially while bandwidth increases linearly. This might be due to bandwidth from network connections being distributed across the network, while for re-replication, all the processing of determining who should get a block and waiting for an ack happens on one node (the master). Bandwidth increasing linearly makes sense as the average number of messages and blocks sent through the network is proportional to the file size.

Part 2: Op times



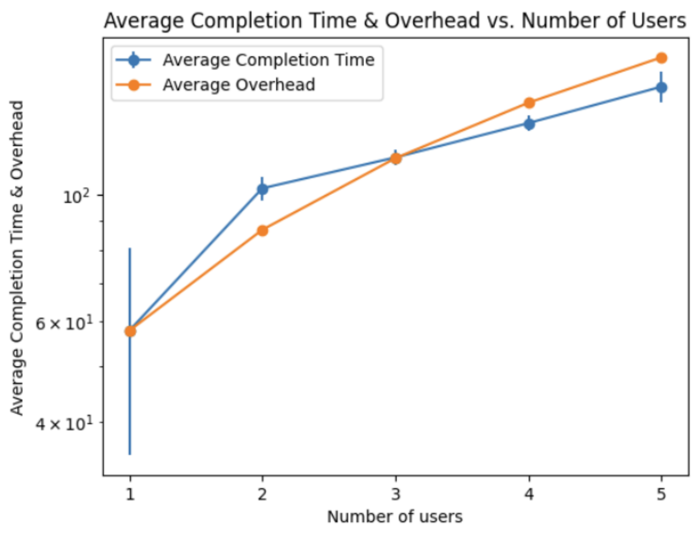
Our Op times were in seconds, for the time to put the 25 and 500 mb files in our SDFS. Inserts took substantially longer than reads, as inserts required sending 4 replicas of a single block to different machines. For reads, we simply had to open a connection to a random replica containing the block and read from it. As a result, the number of ops is 4 times more for insert than read. Additionally, update times took longer than reads, most likely due to us having to first delete the original file before writing the new one. This was done with a 5 KB block size.

Part 3



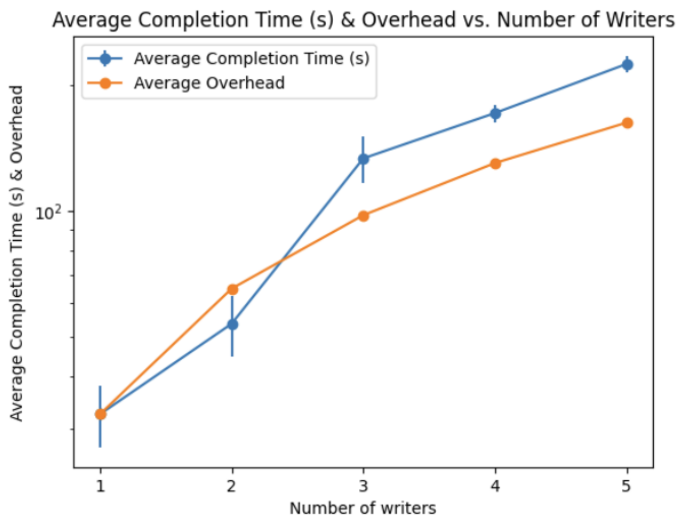
Our storage times are in terms of seconds. As we can see, the more VMs we have, the less time it takes to put a large file. This could be because since we have more machines, the average bandwidth and processing time at each machine decreases, leading to writes being processed faster on each machine. Done with 5 KB block size

Part 4



The file size is 5000mb and the typical read time is 58 seconds. As the number of users increases, we can see that the average completion time and overhead increases. Since only two reads from a file can occur at a time, this makes sense as the number of users that could be simultaneously processed is two. Additionally, the standard deviation with one user seems to be drastically greater than the standard deviation for multiple users. Relative to the overhead, for most of the time the completion time seems to be around the overhead (calculated by (m+2)/2 * 57.7887155, our file's initial read time). We use this formula due to the fact that we can read from two users at a time, thus our overhead increases by a half relative to each user joining. Because the standard deviation of measuring with only on user is so high, this could have contributed to the fact that the average completion time is lower at times than the overhead, since it normally should be higher.

Part 5



Our algorithm was tested in a six node cluster with a 1000mb file. With just one writer, the initial writing time is around 32 seconds. Since we only allow for one writer at a time, the overhead can be calculated as m * 32 where m is the number of writers. As we can see, the completion time initially starts below the overhead. This could be attributed to the high variance of the first sample set taken, which was used to determine our ideal initial writing time.