Ryan Kleck
kleck2
MP3 Report

Machine Problem 3

Design:

My design for MP3 is to build on top of MP2's synchronized membership lists. This was very useful for leader election. In MP3, the leader is elected using the synchronized membership lists. Each process looks at their membership list and elects the oldest active process as leader. This is a liveness property, if a leader is detected as failed then a new leader will be elected at each process that detects the failure. The membership lists are checked each time there is a file operation. As long as the process detects the failed master then it will eventually elect a new leader. Safety would ensure that two masters do not exist at any given time. My protocol cannot guarantee that, but the file operations will take the same actions no matter which process initiates them. Thus, safety is achieved as well.

My design uses sharded files to save file fragments on each server for writes. Writes are done using passive replication, where the master takes the file and shards it according to the shard size property provided by the user. The sharded file is then pushed to the process that will store the file. Each partial file is given a new DFS filename (something like 'localfilenamePART_i'). The DFS filename is hashed and all the file servers have a unique hash assigned to them. I used some ideas from the Chord system. After the filename is hashed I take the value, mod some large number, in order to place processes and files on a hash key ring. The process that will store the file is the server that has the next highest hash key value compared to the hash key value of the DFS filename. Also, there is a replication factor setting that will save a copy of the file on each successor node until the file has been replicated enough to satisfy the replication factor requirements. The replication is handled by the master process as well.

Reads are processed by getting the hash key of the DFS filename then going to the specified server and looking for the file. If it does not exist, then the node's successor will be contacted for the file until the file has been found. The master process is tasked with gluing the files back together and giving it back to the client. The client will then save the file locally in a folder called FromDFS. Reads could be cached simply by looking at the master servers own local files first, however, I did not have time to implement this.

When a failure is detected my group membership protocol passes a message to the file server protocol to rebalance the system (i.e. replicate files to ensure replication factor is upheld). The rebalance is done at each process. It looks through its local files for '_PARTi' files then it will look at the hash key of the filename. It will go through each successor and will only send the file to processes that don't already have the file. This is repeated until the process is sure the replication factor is upheld.

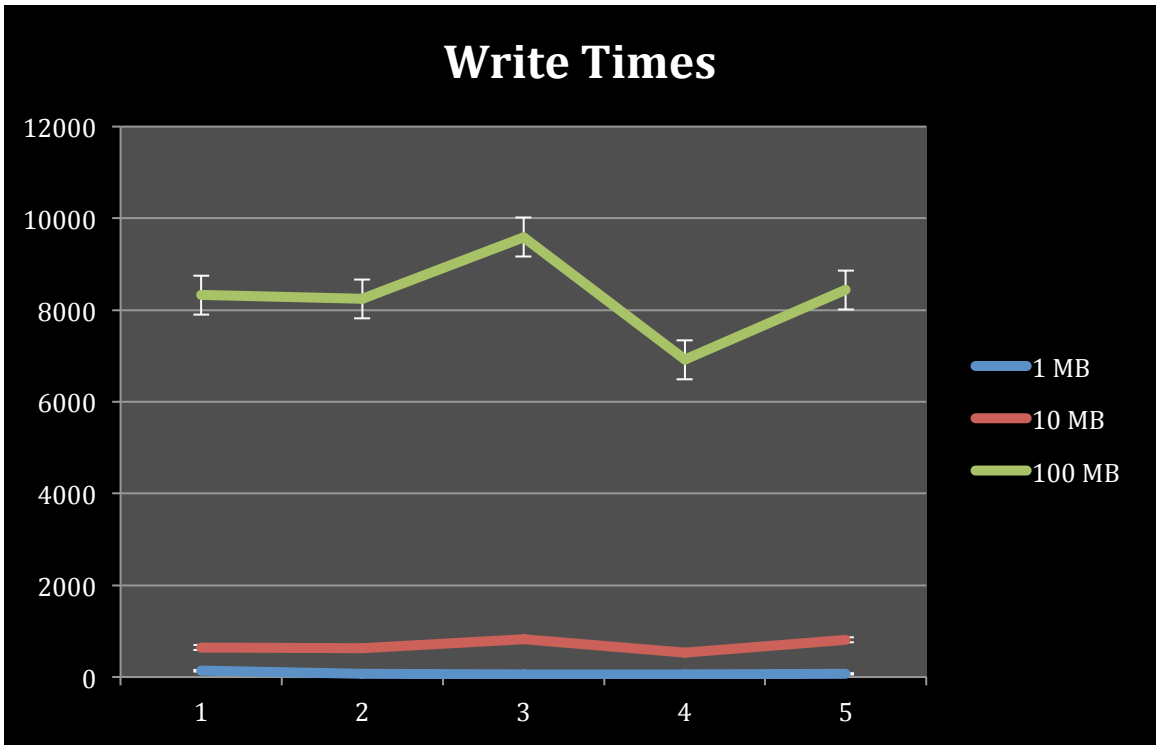
To pass the files and file operation messages between the servers I used a TCP port that the user sets in a property file. For the messages, I took an idea from Gnutella and set aside the first 64 bytes for the message header. The message header is parsed by the receiving server which then takes the action requested. There may be a file shard attached to this message. So the maximum size of a file operation message between servers is 64 bytes plus the shard size set by the user in the property file.

This design scales well to large N because the lookup for the file is quick and the file shards help balance the system in case there are message drops.

Read And Write Times:

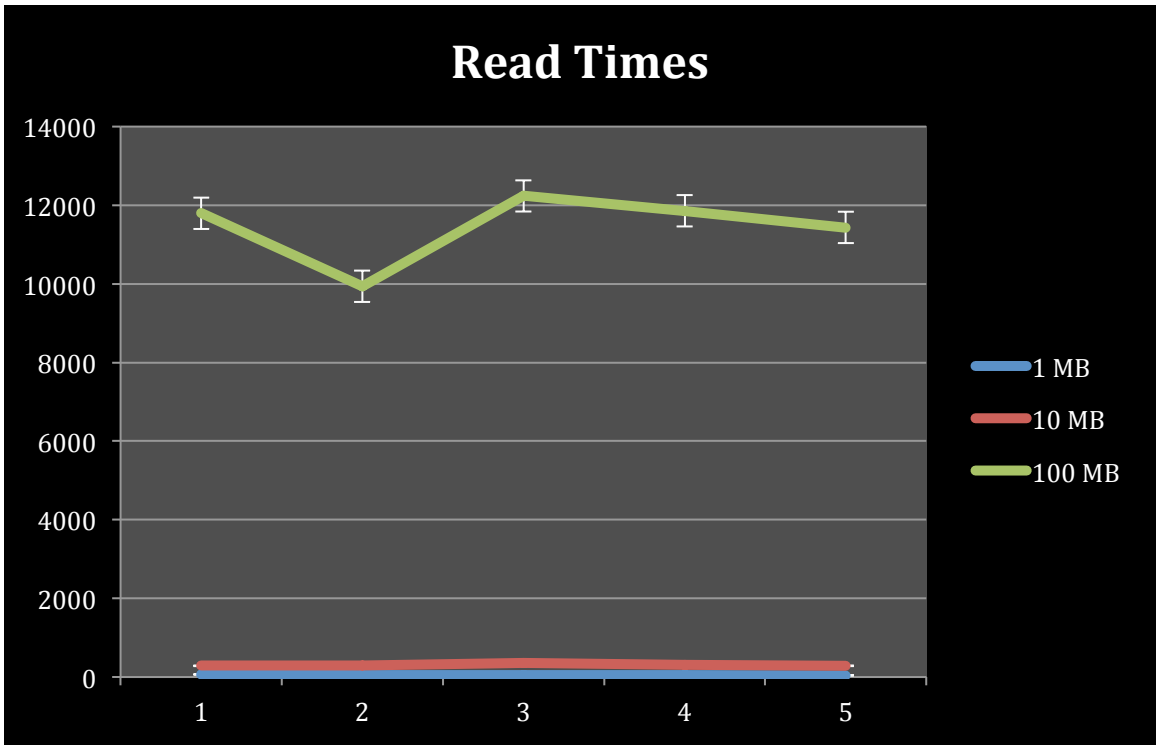
Write times are displayed below in a table. These include the time it takes the master to shard the file. In my experience that was the biggest area for improvement. I used a passive replication method and using an active replication method would save a lot more time. A small 1MB file that does not need sharded takes a very small amount of time however. Also, a major factor is how many connections must be made to move the small file fragments. A larger 100MB file takes longer since there are many more connections being made with the TCP methodology I used to transfer files. It has to make at least 100 connections with replication factor 2 and shard size of 1MB. The standard deviation of the write times for a 100MB file are somewhat compact compared to the 10MB file standard deviations. This probably means that the connection wait time is driving the write times and not the transfer of the smaller files.

Write Times (ms)			
Trial	1 MB	10 MB	100 MB
1	134	646	8326
2	65	634	8245
3	55	820	9593
4	60	532	6916
5	67	814	8438
Average	76	689	8304
Std. Dev.	33	125	950



Read times look slightly better than the write times, except for the 100MB file. The standard deviation is somewhat small again and it means that the wait for the TCP connection is probably driving the read times. Increasing the shard size or finding the optimal shard size could help reduce the read times.

Read Times (ms)			
Trial	1 MB	10 MB	100 MB
1	65	287	11796
2	44	302	9937
3	51	364	12240
4	55	317	11853
5	38	286	11430
Average	51	311	11451
Std. Dev.	10	32	894

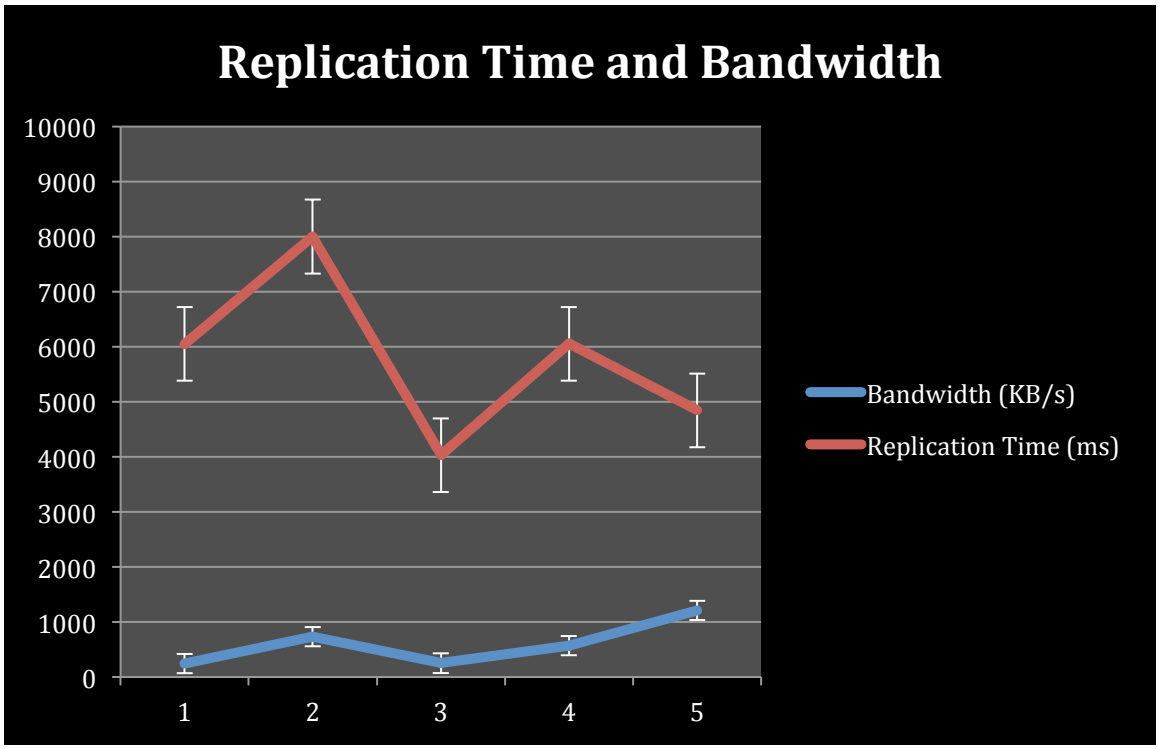


Bandwidth:

With no failures the gossip membership protocol uses about 9KB/sec bandwidth per machine. Total bandwidth of the system for 4 servers is about 36KB/sec at rest.

The bandwidth when a process fails varies depending on how many files are stored on the failed process (i.e. how many files need replicated). For my tests, I made the master fail and recorded the replication time and the bandwidth when there was only one 100MB file stored on the DFS. If the distribution of files are skewed on the DFS then closing the server with more files will impair bandwidth and replication times. The trials were separate (i.e. the bandwidth is not the same trial as the replication time measurement in the table).

Trial	Bandwidth (KB/s)	Replication Time (ms)
1	245	6052
2	734	8001
3	251	4031
4	571	6054
5	1211	4844
Average	603	5796
Std. Dev.	400	1501



The replication time was measured as the max time at any one process between when file replication started and ended. The standard deviation is quite large indicating that some of these processes may have contained a lot more files than other processes.

The bandwidth was measured similarly as replication time and shows the same thing. There is a high standard deviation, which shows some skew in the distribution of files on the system. The bandwidth of the whole system is at worst three times the value in the table above (since 1 server dropped out of 4). This bandwidth is much higher than when the system is idly gossiping, about 30x.

I had difficulty trying to time when a process stops or crashes and when another process detects it. Theoretically it should be 2 * timeFail. The protocol will start replicating files if it detects no updates from that process in 2 * timeFail.

4th Credit:

Master election in this protocol will take at most 2 * timeFail as discussed above as well. This protocol uses the group membership list to lookup the oldest process. Since each process has a synchronized list they can all do one lookup and find the master process. I discussed this feature at length in the Design section at the top of the page the first paragraph, and also used master fail times for the data in the chart in the Bandwidth section.