Design: Our SDFS is designed on top of MP2 failure detector. In our SDFS design, we will have total **4 replicas** for each file in the SDFS. This is done to tolerate the 3 simultaneous failures. We replicate entire file and not splitting into shards. We chose not to cache and We used TCP connections for file exchanges.

Read(R) and Write(W) Consistency levels: As we have 4 replicas (say N) , We will be using W+R>N i.e **W+R=5**, so that reads and writes will intersect in atleast one. To catch the write-write conflict W should be greater than N/2 so we fixed **W=3(since given in problem statement it should be less than N)** and To catch the read-write conflicts, W+R>N will suffice. So we set our **W=3 and R=2.**

Algorithm: We use Cassandra style mechanism for placing the replicas i.e we hash the file name and as per the hash value we decide the VMs to which the replica should be placed.

File Replication and Hashing: We pick the VM to which the file to be replicated based on the hash value of the file_name, and we wont be considering the version number of a file while hashing, which ensures all versions stays on same VMs. Hash value returns an integer ranging from 1 to 10, and then we pick the 4 active processes starting from that hash value in our ring structure.

File Reading: When a process wants to get a file, it hashes the file name, and then get lists of VMs that has the required file, it then pings those VMs and pick the 2 VMs with least ping time, i.e picks the 2 VMs which responds quickly. Then picks the latest file version among these 2 and downloads it.

File Versioning: Version number to a file is always assigned by the sequencer and we assume that if a file is uploaded to SDFS twice using the put command, even if the file's contents remain unchanged, it will be treated as a new version and have its version number increased.

Sequencer Functionality: Sequencer assigns the logical version number to a file. It maintains a counter for file versions and metadata about files. It updates its counter value of each file to latest version number by using the metadata.When it receives a version request from one of the process(say Process A) which is trying to put the file(either new/updated file) into SDFS. Process A sends file_name to Sequencer. Sequencer then set it's counter(file_name) to counter(file_name)+1 and send this new value as version number to Process A. In this way, modifications in the files will be in same version across the VMs **ensuring total ordering.**

Sequencer Failure case: If a sequencer fails and it gets detected by failure detector, then the next active process in the ring is chosen as the sequencer. So there is **no election**, we are just choosing the sequencer in a round robin fashion on the active processes in the ring.

Introducer Functionality: Introducer will still have the same functionality same as MP2 of assigning the process id (process name+timestamp) to joining process (say Process A) and shares the existing membership list to Process A and informs the rest of the system about process A. Along with this, Introducer will have the metadata which contains information about files with their locations and it will share this metadata to the newly joined process A.

Introducer DNS: We added the IP of the Introducer to the /etc/hosts file and mapped the IP to the string "introducer", everynode will use this DNS entry to connect to the Introducer.

Process Joins: For a new process to join the Distributed system, it will contact the introducer. The Introducer will send the membership list, along with the meta data of the files in the system to the newly joined process. Then the newly joined process, sets the process with the lowest id in the membership list as its sequencer and it checks the meta data for file names and calculate the hash of each file_name and according to the hash value it will download those files intended for itself.

Process Fails/Leaves: When a Process fails or leaves the system, then through failure detector, rest of the processes gets to know about this event and then each process checks its metadata that has information about the files of the failed/left process and then checks hash value of those file names, and then replicate those files accordingly.

Uploading a file to SDFS(put localfilename sdfsfilename): Whenever a process tries to put a file to sdfs, it first sends the file_name to the sequencer to get the logical version number. Once it gets the version number it appends that version number to the file name and then uploads the file to the respective VMs based on hashvalue of the file_name. When a replica is uploaded at a VM, its metadata will be updated and propagated to the entire system via everynode notifying its three neighbours. This will be active replication.

Downloading a sdfs file to local(get sdfsfilename localfilename): Whenever a process wants to download a latest version of sdfs file to its local directory, it hashes the file_name, and then the get list of VMs that has the required file, it then ping tests those VMs and picks the two VMs with least ping time, i.e picks the VMs which responds quickly and requests them for the latest version of that file.

Downloading a sdfs file version(get-versions sdfsfilename numversions localfilename): When a process wants to download "n" latest versions of sdfs file to its local directory, then it hashes the file name, and then get list of VMs that has the required files, before fetching each version a ping test will be done and the fastest responder will be chosen for that version to get the file. If that VM has more than n versions, then it will send the latest n versions, if not it will send all the versions it has.

Deleting a SDFS file(delete sdfsfilename): Lets say for an sdfs file "sample.log", the four replicas are at VMs 1-4. Now the delete command (delete sample.log) is given at some (say VM6), then process at VM6 will hash the file_name and get to know about VMs 1-4 and it will send delete messages to processes at VMs 1-4. Then they remove all the versions of the file from their sdfs directory. VMs 1-4 update their metadata and propagate this update to entire system.

Listing VMs that has the file(ls filename): Process hashes the file_name to get the list of VMs on which this file is present and this list will be displayed to the user.

Show the files at this VM(store): By checking the metadata at the process that this command is executed, it lists all the files presents in its sdfs directory.

Usage of MP2 & MP1in MP3: As stated above, MP3 leverages MP2s failure detector, inorder to update the metadata which contains information about files. Membership list designed in MP2 will help to maintain the information about sequencer & metadata uptodate. We logged the errors of MP3 programs into a file. We then used the distributed grep of MP1to debug which saved us a lot of time.

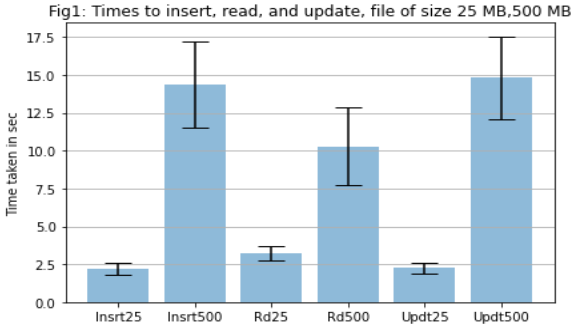
Measurements: We took the measurements five times for each data point and then took the mean and standard deviation.

(i) **Re-replication time and bandwidth upon a failure for a 40MB file**

We measured bandwidth usage using iftop command in linux. Re-replication time makes sense as it includes both failure detection time and replication time.

	Mean	Standard deviation
Re-replication time (sec)	4.08	0.1215
Bandwidth upon a failure (MBps)	46.1729	5.0638

Fig1: Times to insert, read, and update, file of size 25 MB,500 MB



(ii) Time to insert, read and update for files of size 25MB and 500MB. **(Plot in Fig1 (above))**

	Insert		Read		Update	
File size	25MB	500MB	25MB	500MB	25MB	500MB
Mean (sec)	2.1914	14.3839	3.2372	10.2834	2.2396	14.8293
Std dev(sec)	0.3671	2.82526	0.4613	2.55815	0.3253	2.7392

It took more time to read than to insert the small file(25MB), this is due to our design decision to ping test all those VMs (which has the file) and get the latest version of file from that VM which responded quickly(network aware) So here extra latency for ping test is getting added. However for larger files (like 500MB), the impact of this ping test will be outweighed by the fast network connection to read this file.

(iii) Time for get_versions as function of num_versions, done with 25MB file **(Plot in Fig2)**. For each version of the file, we will ping test and get the file from the VM that responds quickly, so this latency added for each version & hence time taken is linearly increased with num_versions.

	num_versions=1	num_versions=2	num_versions=3	num_versions=4	num_versions=5
Mean (sec)	**2.23**	**4.14**	**5.73**	**7.43**	**9.18**
Std dev(sec)	**0.3671**	**0.416**	**0.516**	**0.58**	**0.62**

(iv) Time to store the entire English Wikipedia corpus into SDFS with 4 and 8 VMs is given below. Both take the same time because, for same file_name, the hash value will be same and so it will be replicated to same 4 VMs even if 4 VMs or 8VMs present in the SDFS. **(Plot in Fig3)**

VMs	Mean(sec)	Std dev(sec)
4	135.38	4.9180
8	133.45	4.32749