CS425, Distributed Systems: Fall 2025
Machine Programming 3 – Hybrid Distributed File System

Released Date: October 14, 2025
Due Date (Hard Deadline): Sunday, November 9, 2025 (Code+Report due at 11.59 PM)
Demos on Monday, November 10, 2025

This is an intense and time-consuming MP! So please start early! Start now!

FakeCompany Inc. (MP2) just got acquired in a hostile takeover by the fictitious company ExSpace Inc., whose business model is to do couples therapy…but in space. Anyway, ExSpace loved your previous work from MP2, so they've re-commissioned you to build a Hybrid Distributed File System (**HyDFS**) for them. HyDFS will run on 10 machines in their ExSpace's spaceship as it orbits the Earth. HyDFS is a hybrid between HDFS (Hadoop Distributed File System) and Cassandra–you can look at HDFS/Cassandra docs and code, but you cannot reuse any code from there (we will check using Moss).

Only for this MP, you are allowed to use LLMs. If you do, please first come up with a design and use LLM as a sounding board to improve your design. Uploading the entire MP spec as an input is likely to generate unmanageable code that will be difficult to debug. Instead, once you have a design you could use LLMs to generate small pieces of code. Your use and experience with the LLM must be included in the Report (below); you must also cite its use and include your sessions as files with your submitted code. Given the increased use of LLMs to generate code in the industry, these relaxed rules for MP3 serve as an educational purpose for you. However, note that (as in your future job) you and only you are responsible for the correctness and efficiency of any code that you obtain from LLMs. If something fails in the demo, it is your responsibility, not the LLM's. Proceed cautiously and judiciously.

You must work in groups of two for this MP. Please stick with the groups you formed for MP2. (See end of document for expectations from group members.) Remember – Move Fast and Break Things! (i.e., build some pieces of code that run and incrementally grow them. Don't write big pieces of code and then compile them all and run them all!)

This MP requires you to use code from both MP1 and MP2.

HyDFS is intended to scale with the number of machines. HyDFS uses **consistent hashing** to map machines (aka nodes) and files to points on a ring. HyDFS uses a pre-determined hashing function to map nodes and files to the ring. A file is replicated on the first **n** successor nodes in the ring (just like in Cassandra). Each node uses a membership protocol to maintain/update the full membership list (use distributed group membership from MP2). Thus, given a filename, a node can, in O(1) time, route the request to one of the replicas that contains the file. Note that you should **not** store the list of all file-node mappings at each node. Just the membership list of nodes and their IDs is sufficient to route requests correctly.

HyDFS can tolerate up to two simultaneous machine failures. You must ensure that data is re-replicated after any machine failure. As in MP2, you can assume that any subsequent failure (after two machines have failed) can occur only after the entire system has stabilized. You must also ensure that when any new machine joins, data is re-balanced (just like in Chord/Cassandra). **Use only the minimum number of replicas of each file needed to meet this requirement. Don't over-replicate.** Additionally, during re-replication or re-balancing, you must ensure that the data is stored only in the first n successor nodes for a given file.

HyDFS is a flat file system, i.e., it has no concept of directories, although filenames are allowed to contain slashes. You will implement the following file operations:

1) `create localfilename HyDFSfilename` (create a file on HyDFS and copy the contents of localfilename from local dir; only the first create should succeed, subsequent ones should fail)
2) `get HyDFSfilename localfilename` (fetches the entire file from HyDFS to localfilename on local dir)
3) `append localfilename HyDFSfilename` (appends the contents of the localfilename to the end of the file on HyDFS; an append increases the file size by the size of localfilename). Append requires that the destination file already exists in the HyDFS.

All commands should be executable at any of the VMs, i.e., any VM should be able to act as both client and server.

Mandatory: HyDFS guarantees the following:

(i) **Per-client append ordering.** The results of two appends to a file from the same client should appear in the order that the client issued the appends, i.e., if a client completes append operation A and then issues append B to the same file, A must appear before B in the file. Thus, if a client completes an append of 'xxx' to DFSfile1 followed by an append of 'yyy', then DFSfile1's contents must be …xxx…yyy... Note that it could also be …xxxyyy… if there were no concurrent appends from other clients to DSfile1.

(ii) **Eventual consistency.** Appends from all clients to a given file are eventually applied in the same order across the replicas of that file. In other words, once the system quiesces and there are no outstanding updates to the file or process failures, all replicas will eventually contain an identical copy of the file.

(iii) **Read-my-writes.** If a client issues a get, the system must return file contents that reflect the most recent appends successfully completed by that same client. Note that it need not immediately reflect the appends performed by other clients; see (ii) above.

Finally, you will implement the following file operation.

4) `merge HyDFSfilename` Merge ensures that all replicas of a file are identical. You can assume that there are no new file updates or machine failures when the merge command is called (though this is not true in the real world). You need to merge file versions across replicas in such a way that no successfully completed appends from the past are lost and the above guarantees are preserved. Your design of HyDFS will include periodic background merging of file versions so as to satisfy the above guarantees. However, for the purpose of the demo, we ask you to implement this explicit merge command that lets us kick off a merge on a file (if one is not already in progress), and to wait for its successful completion. Once merge returns, the replicas of the file should be identical (assuming no new file updates or machine failures).

Think carefully about what information each replica needs to maintain. Also, files may be 100s of MBs; so how do you efficiently compare and merge files to guarantee eventual consistency and client ordering? One option is to store the file in smaller blocks, with each chunk storing the data corresponding to an append. What other information does HyDFS need to maintain to satisfy all the above correctness guarantees?

For demo purposes, you will also implement the following operations:

1) `ls HyDFSfilename` List all machine (VM) addresses (along with the VMs' IDs on the ring) where this file is currently being stored. Also prints the fileID of HyDFSfilename.
2) `liststore` (at any process/VM) List the set of file names (along with their fileIDs) that are replicated (stored) on HyDFS at this process/VM. This should NOT include files stored on the local file system. Also, print the process/VM's ID on the ring.
3) `getfromreplica VMaddress HyDFSfilename localfilename` Get the file from a particular replica. This should fetch the file and store it locally as `localfilename` but not perform any further actions on the file. We will use this on all replicas to check if files are identical after a merge.
4) `list_mem_ids` Augment list_mem from MP2 to also print the ID on the ring that each node in the membership list maps to. Please sort this by nodeID (useful for tests).
5) `multiappend(HyDFSfilename, VMi,…VMj, localfilenamei,… localfilenamej)` Launches appends from VMi,…VMj simultaneously to HyDFSfilename; VMi appends the contents of localfilenamei.

For each of the commands, please print terminal-level messages at the initiating client that show when the command has been completed. This should be shown when the operation completes in HyDFS. For creates and appends, also list the VMs to which the file was replicated. At every replica that receives a get or write (insert/append), print when the replica receives the request and print when the replica completes the operation (print terminal-level messages).

Handle failure scenarios carefully. Just like in MP2, we're implementing the crash/fail-stop model; when a machine rejoins, it must do so with a new node ID that is constructed based not only on IP and port but also a timestamp or version number that distinguishes successive versions of the same machine. If a node fails and rejoins, ensure that all file blocks stored in it get deleted before it can rejoin. Think through all failure scenarios carefully and ensure your system does not hang.

Feel free to explore various design possibilities: should you replicate an entire file or shard (split) it and replicate each shard individually? How does replication work – is it active replication or passive replication? How are reads processed? How exactly does your protocol leverage MP2's membership list? How do you select quorum sizes? Optimize for speed and correctness. Design first, then implement. Keep your design (and implementation) as simple as possible. Use the adage "KISS: Keep It Simple Si…". Otherwise, ExSpace may involuntarily send you to space without a partner or any counseling!

Create logs on each machine. You can make your logs as verbose as you want, but at the very least, you must log each time a file operation is processed (once when a node receives a request and once when the node completes the operation). We will request to see the log entries at the demo via the MP1 query mechanism. You should also use your MP1 solution for debugging MP3 (and mention how useful this was in the report).

Use your MP2 code to maintain membership lists across machines and detect failures. During the demo, we may need to invoke some MP2 commands e.g., to list membership.

We also recommend (but don't require) writing unit tests for the basic file operations. At the very least, ensure that your system works for a long series of file operations.

Machines: You will be using the CS VM Cluster machines. You will start by using all 10 VMs for the demo. The VMs do not have persistent storage, so you are required to use git to manage your code. To access git from the VMs, use the same instructions as MP1.

Demo: Demos are usually scheduled on the Monday right after the MP is due. The demos will be on the CS VM Cluster machines. You must use all 10 VMs for your demo (details will be posted on Piazza closer to the demo date). Please make sure your code runs on the CS VM Cluster machines, especially if you've used your own machines/laptops to do most of your coding. Please make sure that any third-party code you use is installable on CS VM Cluster. Further demo details and a signup sheet will be made available closer to the date.
We expect both partners to contribute equivalent amounts of effort during the entire MP execution (not just in the demo).

Language: Choose your favorite language! We recommend C/C++/Java/Go/Rust. We will release "Best MPs" from the class in these languages only (so you can use them in subsequent MPs).

Report: Write a report of not more than three pages (12 pt font). Briefly describe the following (we recommend your report contain headings with bold keywords): a) **Design**: (no more than a page) algorithm and design used, and how you met the requirements – focus especially on your replication level, how you satisfied the consistency requirements, and how you implemented re-replication and merge, b) **Past MP Use**: (very briefly, 1-2 sentences) how MP2's membership protocol was used in MP3 and how useful MP1 was for debugging MP3, c) **LLM use**: If you used LLMs, describe in <100 words your usage and experience---especially how beneficial or not it was---and mention the filenames in your code submission that contain your chat sessions, and d) **Measurements** (measure real numbers, do not calculate by hand or calculator!): see questions below. For each plot/data, please use the terms in brackets so your report is easy to read.

(i) (**Overheads**) Re-replication time and system-wide bandwidth upon the failure of one node. You should measure for different file sizes ranging up to 1 MiB size, 5 different sizes. Preload the system with 100 files.

(ii) (**Rebalancing overheads**) Bandwidth to re-balance files after the addition of a new node. Preload the system with X files, each of 128KiB, and measure the bandwidth with different values of X (vary X from 10, 50, 100, 200).

(iii) (**Merge performance**) Perform a series of concurrent appends to a file (with an initial file size of at least 128KiB) and then call the merge command. Measure the time it takes to complete the merge. Plot this for a varying number of concurrent client appends (1, 2, 5, 10). To implement j concurrent appends, you would launch appends from $VM_1 \ldots VM_j$ simultaneously to the filename. Plot this for two append sizes – 4KiB and 32KiB.

(iv) (Subsequent-merge performance) The same as above, but you will call the merge command a second time after the initial merge. Compare the completion times of the first merge with the second merge.

For each data point, take at least three to five readings, and plot averages **and** standard deviations (and, if you can, confidence intervals). Discuss your plots, don't just put them on paper, i.e., discuss trends briefly and whether they are what you expect or not (why or why not). Measurement numbers don't lie, but we need to make sense of them! Stay within the page limit – for every line over the page limit, you will lose 1 point!

Submission: Similar to past MPs. More instructions on Piazza, but the checklist/workflow is the same. Do not miss the deadlines for signing up for the demos (usually the Friday before submission Sunday), or to submit the form and the Gradescope report (both due on Sunday).

When should I start? Start NOW. You already know all the necessary class material to do this MP. Each MP involves a significant amount of planning, design, and implementation/debugging/experimentation work. **Do not** leave all the work for the days before the deadline – **there will be no extensions**.

Evaluation Break-up: Demo [**50**%], Report (including design and plots) [**35**%], Code readability and comments [**15**%].

Academic Integrity: You cannot look at others' solutions, whether from this year or past years. We will run Moss to check for copying within and outside this class – first offense results in a zero grade on the MP, and second offense results in an F in the course. There are past examples of students penalized in both those ways, so just don't cheat. You can only discuss the MP spec and lecture concepts with the class students and forum, but not solutions, ideas, or code (if we see you posting code on the forum, that's a zero on the MP). FakeCompany Inc. and ExSpace Inc. are watching and will be very Sad!

We recommend you stick with the same group from one MP to the next (this helps keep the VM mapping sane on EngrIT's end), except for exceptional circumstances. We expect all group members to contribute about equivalently to the overall effort. If you believe your group members are not, please have "the talk" with them first, give them a second chance. If that doesn't work either, please approach Aishwarya/Indy.

Happy Filing (from us and the fictitious ExSpace)!