# CS425, Distributed Systems: Fall 2024
# Machine Programming 3 – Hybrid Distributed File System

Released Date: October 15, 2024
Due Date (Hard Deadline): Sunday, November 10, 2024 (Code+Report due at 11.59 PM)
*Demos on Monday, November 11, 2024*

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

FakeNews Inc. (MP2) just got acquired in a hostile and reluctant way by the fictitious company ExSpace Inc. (whose business model is to do couples therapy…but in space. They are based on a tweet by someone on social media that a lack of gravity increases the chances of reconciliation. And as we all know, all tweets are highly scientific.). 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).

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 be scalable as the number of servers increases. So, HyDFS does not store the file-server mappings in a central location like HDFS's Namenode. Instead, the HyDFS uses **consistent hashing** to map servers and files to points on a ring. All servers use a pre-determined hashing function to map servers and files to the ring. A file is replicated on the first **n** successor servers in the ring (just like in Cassandra). Each server uses a membership protocol to maintain/update the full membership list (use distributed group membership from MP2). Thus, given a filename, a server can, in O(1) time, route the request to one of the replicas that contain the file. Note that you should **not** be storing the list of all file-server mappings on each server. Just the list of servers that form the ring is sufficient to correctly route requests.

Data stored in HyDFS is tolerant of up to **two simultaneous machine failures** (as usual, like MP2, after the system converges, more failures are again allowed to occur, up to two failures simultaneously). After failure(s), you must ensure that

data is re-replicated quickly so that another (set of) failures that happens soon after is tolerated (you can assume enough time elapses for the system to quiesce before the next failure(s) occur(s)). **Use only the minimum number of replicas of each file needed to meet this feature. Don't over-replicate.** When picking replicas during re-replication, ensure the first n successor nodes of a file are the replicas. Here are some initial pointers. You can either use a push-based or a pull-based approach to re-replication. In push-based re-replication, the first successor of a file checks if a file has enough replicas and re-replicates if necessary. A pull-based re-replication might be bandwidth efficient instead. A node periodically checks if it has newly become one of the first n successors of a file range. If it finds so, it contacts the first successor of the file range to pull the necessary files.

HyDFS is a flat file system, i.e., it has no concept of directories, although filenames are allowed to contain slashes. The allowed file ops include:

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.

Mandatory: HyDFS ensures the following:

(i) Appends eventually are applied in the same order across the replicas of a file. This is required to ensure that replicas are eventually identical when there are no updates to the file or failures. In other words, the system must guarantee eventual consistency.

(ii) Two appends to a file from the same client should appear in the order the client issued the appends, i.e., if a client completes write operation A and then issues B, A must appear before B in the final order. For example, if a client completes an append of a local file with contents 'xxx' to DFSfile1 and then appends another file with contents 'yyy', DFSfile1's contents must be …xxx…yyy (Note that it could also be …xxxyyy… if there were no concurrent appends from other clients).

(iii) If a client performs a get, the get must return the file contents that reflect the latest appends that the same client performed. It need not necessarily reflect the appends performed by other clients.

Finally, you will implement

4) `merge HyDFSfilename`: once merge is complete, it ensures that all replicas of a file are identical while ensuring (i) and (ii) above. You can assume that there are no concurrent updates/failures during the merge

(though this is not true in the real world). Unlike Cassandra, you cannot use the latest timestamp policy to resolve conflicting file versions. This might result in appends getting lost. You need to merge two files in such a way that no client appends are lost and the above client ordering is preserved. Calling merge again after the first merge would complete immediately (i.e., if the files are identical, then merge is a no-op and should complete quickly).

Think carefully about what information each replica needs to maintain to check. 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 do the servers/clients need to maintain to satisfy all the above correctness guarantees?

One feature that you will implement in HyDFS is **client-side caching**. A client, when it reads a file, caches the file locally so that subsequent reads can go to the locally cached file, making it faster. File **writes** (i.e., creates/appends) do **not** go through the cache. To satisfy the above requirements of HyDFS (i - iii), in some cases, it is not safe to always use the client cache. For example, if a client performed an append, but the locally cached file copy doesn't reflect it, then you can't perform the read at the local cache. The client's memory is limited. You can't infinitely cache all files. So, you must restrict the size of the client cache (the cache only stores recently read files). Design HyDFS such that you **use cached data whenever possible** for performance **but still guarantee** the correctness requirements described above. Choose the quorum sizes correctly to provide maximum availability and performance while ensuring the above guarantees.

For demo purposes, you will need to add three more operations:

1) `ls HyDFSfilename`: list all machine (VM) addresses (along with the VMs' IDs on the ring) where this file is currently being stored.
2) `store`: At any machine, list all files (along with their IDs on the ring) currently being stored at this machine. Also, print the VM's ID on the ring.
3) `getfromreplica VMaddress HyDFSfilename localfilename`: variant of get to get a file from a particular replica indicated by VMaddress.
4) `list_mem_ids`: augment list_mem from MP2 to also print the ID on the ring each node in the membership list maps to.

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

Handle failure scenarios carefully. If a node fails and rejoins, ensure that it wipes all file blocks/replicas it is storing before it rejoins. **When a node rejoins, you have to move the correct files from other nodes to the newly joined node.** Think about all failure scenarios carefully and ensure your system does not hang. For instance, what if a node sends a write and then fails before the confirmation or after

receiving the confirmation notice but before responding? Work out these failure scenarios and ensure you handle them all.

Other parts of the design are open, and you are free to choose. Keep your design as simple as possible to accomplish the goals, but also make it fast. 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.

Think about design possibilities: should you replicate an entire file or shard (split) it and replicate each shard separately? 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, but also keep it simple. Don't go overboard, and **please keep it simple.**

Create logs at each machine. You can make your logs as verbose as you want them, but at the least, you must log each time a file operation is processed locally. We will request to see the log entries at the demo, perhaps via the MP1's querier.

Use your MP2 code to maintain membership lists across machines. You should also use your MP1 solution for debugging MP3 (and mention how useful this was in the report).

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

**Machines**: We 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 less 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, and c) **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 bandwidth upon a failure (you can measure for different filesizes ranging up to 100 MB size, 5 different sizes).

(ii) (**Merge performance**) Perform at least 1000 concurrent appends to a file (filesize at least 10MB). Measure the time it takes to perform 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 $VM_1$… $VM_j$ simultaneously to the filename. Plot this for two append sizes – 4KB and 40KB.

(iii) (**Cache Performance**) You will measure the performance of reads with and without client-side caching. Take a dataset containing many files (say 10000). The files are 4KB in size. Load the entire dataset into HyDFS. Generate a workload where a client reads files one after the other. The client should perform 2-3x the number of gets as the number of files. Measure the latency of file reads with and without caching for (a) when a client uniformly reads files at random and (b) when a client reads some files more frequently than others. (e.g., the requests follow a zipfian distribution https://en.wikipedia.org/wiki/Zipf%27s_law). Plot this for varying client cache size (where client cache size can hold only 10, 20, 50, and 100% of the data set).

(iv) (**Cache Performance with Appends**) Same experiment as above but with the workload consisting of 10% appends and 90% gets. Measure the latency of file reads with and without caching for uniform and zipfian distributions. Pick 50% cache size.

For each data point, take at least three to five readings each, 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**: (1) Submit your report to Gradescope BEFORE the deadline, and TAG your page(s) on Gradescope (there will be a penalty if you don't). Only one group member should submit – you can tag your partner.
(2) There will also be a demo of each group's project code. Signup sheets will be posted on Piazza.
(3) Submit your working code via gitlab sharing (including your group number in your project name). Please include a README explaining how to compile and run your code. Default submission is via gitlab sharing – please include your group number in your gitlab share names! Further submission instructions will be posted on Piazza. Other submission instructions are similar to previous MPs.

**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). FakeNews Inc. is 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)!