CS425, Distributed Systems: Fall 2018
Machine Programming 3 – Simple Distributed File System

Released Date: October 18, 2018
Due Date (Hard Deadline): Sunday, November 4, 2018 (Code due at 11.59 PM)
Demos on Monday November 5, 2018

FakeNews Inc. (MP2) just got acquired by the fictitious CallMeIshmael Inc. (whose business model is to look for underwater creatures and shoot them, with a camera that is. Go figure!). They loved your previous work, so they've re-commissioned you to build a Simple Distributed File System (**SDFS**) for them. SDFS is a simplified version of HDFS (Hadoop Distributed File System) – you can look at HDFS 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.

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

SDFS is intended to be scalable as the number of servers increases. Data stored in SDFS is tolerant up to **three** machine failures at a time. 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)). Store the minimum number of replicas of each file needed to meet this feature. Don't over-replicate.

SDFS is a flat file system, i.e., it has no concept of directories, although filenames are allowed to contain slashes.

Thus, the allowed file ops include: 1) `put localfilename sdfsfilename` (from local dir), 2) `get sdfsfilename localfilename` (fetches to local dir), 3) `delete sdfsfilename`. `Put` is used to both insert the original file and update it (updates are comprehensive, i.e., they send the entire file, not just a block of it).

For demo purposes you will need to add two more operations: 4) `ls sdfsfilename`: list all machine (VM) addresses where this file is currently being stored; 5) `store`: At any machine, list all files currently being stored at this machine. Here `sdfsfilename` is an arbitrary string while `localfilename` is the Unix-style local file system name.

Mandatory: SDFS is a **versioned** file system. This implies some design decisions that are mandatory and others that you are free to choose. Mandatory requirements include:

1. To make writes and reads fast, you should use consistency levels: writes ack-ed by (at least) W replicas, and reads by (at least) R replicas. Select W and R so that any two conflicting writes are intersect in at least one replica, and any write and read intersect in at least one replica. What are the values of W and R so that (W+R) is the least?
2. **Totally order** all updates to a given file.
3. A read returns the latest written (and acknowledged) value.
4. Implement the **new command** 6) `get-versions sdfsfilename num-versions localfilename` which gets all the last `num-versions` versions of the file into the `localfilename` (use delimiters to mark out versions).
5. You can assume `num-versions` is no more than 5.

If a node fails and rejoins, ensure that it wipes all file blocks/replicas it is storing before it rejoins. 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 the simplest possible to accomplish the goals, but also make it fast. Here is a first cut way to structure your design. One of the servers should be the master server. The master is responsible for deciding which files get stored where. All queries and operations can go through the master. If the master fails, a new master should be re-elected quickly. While there is no master, the system should not lose data or crash – however, file operations may not be possible while the master is down. Is this design enough to tolerate three simultaneous machine failures? No! Modify it so that it satisfies all the requirements!

Think about design possibilities: should you replicate an entire file, or shard (split) it and replicate each shard separately? How do you do election? How does replication work – is it active replication or passive replication? How are reads processed? Can you make reads/queries efficient by using caching? How exactly does your protocol leverage MP2's membership list? What does a quorum mean and how do you select it? Don't go overboard, but **please keep it simple.**

As usual, don't overdesign or overkill for the spec. CallMeIshmael Inc. is watching – if you have a complicated design that is an overkill, they will shoot you (with a camera and then put you on Instagram).

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 demo time, 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 actually work for a long series of file operations.

Machines: We will be using the CS VM Cluster machines. You will be using 7-10 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 7-10 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.

Language: Choose your favorite language! We recommend C/C++/Java/Go. 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 2 pages (12 pt font, typed only - no handwritten reports please!). Briefly describe your design (algorithm used and replication strategy), very briefly how useful MP1 was for debugging MP3, and the following measurements (not calculations!): (i) re-replication time and bandwidth upon a failure (you can measure for a 40 MB file); (ii) times to insert, read, and update, file of size 25 MB, 500 MB (6 total data points), under no failure; (iii) Plot the time to perform `get-versions` as a function of `num-versions`; and (iv) time to store the entire English Wikipedia corpus into SDFS with 4 machines and 8 machines (not counting the master): use the Wikipedia English (raw text) link at: http://www.cs.upc.edu/~nlp/wikicorpus/ . For each data point, run at least 5 trials and give the average and standard deviation bars. Discuss your plots, don't just put them on paper, i.e., discuss trends, 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 page limits.

Submission: There will be a demo of each group's project code. Submit your report (softcopy) as well as working code. Please include a README explaining how to compile and run your code. Submission instructions are similar to MP2.

When should I start? Start NOW. This 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 [40%], Report (including design and plots) [40%], Code readability and comments [20%].

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). CallMeIshmael Inc. is watching!

Happy Filing (from us and the fictitious CallMeIshmael Inc.)!