Term Project: Seattle U. File System (SUFS)

CPSC 4910/5910 Cloud Computing

Winter 2020, v1.0

Overview

In this project, your team will develop the Seattle University File System (SUFS). It is basically a clone of HDFS (the Hadoop Distributed File System) that runs inside AWS using EC2 instances to form the cluster. You will write the code and deploy it on EC2 instances.

You may implement this project in any programming language you choose (as long as it's not *too* obscure). You will have a lot of choice in the other tools and technologies that you use, although a couple specifics will be required (details below). That said, you may want to look at this link and see what languages have SDKs (software development kits) for AWS. An SDK makes it a *lot* easier to use the API (application program interface) to make calls that control the cloud from your code.

https://aws.amazon.com/tools/

Note that HDFS is an open source project that is implemented in Java. It is okay for this assignment if you want to *look* at the open source code to understand how it works, as long as you don't actually copy from that code – that includes copy-and-translate to another language. Between features in HDFS that you do not need to implement in SUFS (like logging and so forth) and the idiomatic way the code is written (in short, their code design would be way over-engineered for what you are building), it will be obvious if you do copy from it…

Understanding HDFS

The first thing you need to do is understand what HDFS is, what it does, and how it works. Read the following two web pages (it doesn't matter what order you read them in).

https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html

https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Feel free to explore the Hadoop / HDFS documentation further, or to google for more information as well.

What to Do

Your team will implement SUFS, the HDFS clone, including DataNodes, a NameNode, and simple client program. You will deploy the nodes on EC2 instances. (The client should, in theory, be able to run anywhere, but for demo & grading purposes, it's okay to also run it on an EC2 instance if you want to.) The easiest way to implement the client is probably to make it a command line tool. More specific details about what functionality you do and do not have to implement is later in these instructions.

You will also write a brief document describing your design. Drafts of this document will also be submitted with the project checkpoints. The official length is "as long as necessary", but I expect about 2-4 pages, *including figures*. (If you go to 5 pages that's probably fine, but yours is *significantly* longer than 4 pages, then you're probably either giving more specific detail than I'm expecting or repeating too much general information about HDFS that was covered in class or can be found on the HDFS documentation linked above.) Your document should describe the architecture of your SUFS, although feel free to reference the HDFS architecture for aspects that are the same (or to contrast things that are different), and should also include the following information:

- **Sequence Diagrams** showing the interaction between nodes (and the client) for all the major operations / event, such as writes, reads, DataNode failures, etc. Do not include diagrams for trivial events, however, such as just a single Heartbeat message.

- Each system component will need some **interface** that the other components can use to interact with it. This should be implemented as a **RESTful API** unless you get permission from the instructor to do something different (e.g., RPC or something else). Each type of component will need to implement it's own different API. Provide proper documentation for each of those APIs. RESTful APIs and how to document them will be covered in class. (If you get permission from the instructor to do something different from a RESTful API, then the instructor will discuss with you how to document that properly.)

- List of the **technologies or tools** that you are incorporating into your project and how they're being used (unless it's obvious how, like your programming language choice)
    - Make separate lists for development tools (things that are part of the *process*) and technologies incorporated into the SUFS design (things that are part of the *product*)

- What **system parameters** did you choose? i.e., What are the...
    - block size?
    - replication factor?
    - any other system parameters you had to choose a value for?

- Anything else I need to know to understand your design and how your system is going to work (if there is anything)
    - but don't repeat too much information about HDFS from the web pages above or that was covered in class – you may assume the reader already knows these basics; your report should instead be about the next level of detail in your design

What to Submit & Demo

You should submit your report to Canvas in either MS Word or PDF format. (Only make one submission per team.) Details for submitting your code are below. (Code will *not* be submitted to Canvas, only your report.)

For your demo, you will show your system running in AWS using EC2 instances. You will demonstrate normal behavior of creating/writing and reading files. Then you will demonstrate abnormal behaviors you have accounted for, such as DataNodes crashing (which you can simulate by terminating/stopping that EC2 instance). I will have a specific set of steps for you to go through at the demo (to ensure uniformity between teams' demos), and I will also ask each person on the team to walk me through a segment of the code and explain it.

Tools you may use

You might need to programmatically control EC2, S3, or other AWS services. This is possible using the AWS APIs. There are also convenient SDKs available in a number of different languages. Here is the general EC2 (first link) and S3 (second link) API documentation... and you already saw the link for the SDKs above.

https://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html

https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

There are a number of AWS Developer Tools that you may find useful when working in a group. Your team should use CodeCommit, a source control repository for git (similar to GitHub). You will give the instructor access to your CodeCommit repository, and instead of submitting your code, the instructor will simply pull your code from CodeCommit at the project deadline.

The other developer tools may be useful too, and you are encouraged to use them. However, CodeCommit is the only one that is required. Documentation about developer tools can be found at these links:

https://aws.amazon.com/products/developer-tools/

https://aws.amazon.com/tools/

The communication between nodes and with the client should be done via a RESTful API (unless you got permission from the instructor to do something else). There are a number of frameworks that you can use to implement a RESTful API. Your framework choice will partially depend on your language choice (or vice-versa), but some languages do have more than one framework available. Do some research and choose one, then do a little testing with it to be sure it works for what you want to do. Document your choice in your design doc, and also submit any test code with your checkpoint. The instructor will give you feedback on your framework choice (as well as any other tools or technologies you've chosen) – this will allow you to make changes if you need to before you're too invested in those decisions.

Design Details

In general, you should mimic the functionality and design of HDFS. Read and understand the HDFS documentation carefully to understand what those are. (Do pay attention to the "Assumptions and Goals" section!) All teams **DO** need to do the following:

- Implement a NameNode, DataNodes, and a client program.
- Creating & writing a file (as a single operation)
    - The client program should read the file data out of an S3 file and then write the data into SUFS.
    - You do need to divide the file into blocks that will be stored in SUFS; do not store files as a single chunk (unless they are smaller than the block size).
- Reading a file
    - The client program can take the file it read out of SUFS and write it to a file in the local filesystem.
- Client operations
    - *create* = create and write a new file in SUFS (the file data will come from an S3 object)
    - *read* = read a file from SUFS and store it on the client's local hard drive
    - *list* = given a filename, retrieve the list of blocks and DataNodes that store each block from the NameNode and display them
        - be sure to keep the *list* output from this somewhat neat - it could get long for files with many blocks - one suggestion is to output a separate line for each block and list the DataNodes for that block on that line
- DataNode fault tolerance
    - if up to N-1 data nodes fail concurrently, the system should continue to operate without data loss
    - new block replicas should be created when the number of block replicas falls below N – you may choose the N (replication factor) for your SUFS
    - you do ***not*** need to programmatically start up a new DataNode to replace a failed one – the system will replace the block replicas but not the DataNode itself, and the system will just continue to run fine with one less DataNode
    - you do ***not*** need to handle situations in which the number of DataNodes < N (the replication factor) – if that many DataNodes fail, then all bets are off!

- Use the following system values. (They do *not* need to be configurable; feel free to hard code them.)
    - block size = 128 MB
    - replication factor, N = 3
    - block report frequency = every 30 sec.
    - heartbeat = not needed; you may piggyback on the block report

Grad Teams (those registered for 5910) need to do the following, but undergrad teams (registered for 4910) do *not*:

- **Directories**
    - The directory structure is stored only on the NameNode, and you may store it in any way you want so long as it works.
    - Note that you do *not* have to support the concept of a "current directory" or relative paths; you may instead require entering a complete absolute path for every operation.
- **Deleting** files and directories
    - But you do *not* need to support undeleting or the /trash folder.
- The client should support the following additional operations:
    - *mkdir* = creates the specified directory
    - *rmdir* = removes the specified directory, if it is empty
    - *ls* = lists all the files and subdirectories of the specified directory
    - *delete* = deletes the specified file
- **Pipelining**
    - When creating a file, the client should only contact one DataNode for each block that it needs to write; that DataNode should then forward the block to the other DataNodes that needs replicas of it.
    - (undergrad teams, registered for 4910, may simply have the client contact each DataNode that needs a replica of the file, so DataNodes never need to talk to each other directly)

Here are some details that you ***do not*** need to worry about (applies to *all* teams):

- **NameNode failure** or Metadata Disk Failure – only worry about DataNodes failing
- **Public Access**
    - Your NameNode & DataNodes do not need to be accessible from outside AWS - so it's okay if your client only works when run from inside AWS (e.g., on another EC2 instance), and within the same Security Group.
        - this could happen, for example, if you use the "private IP" instead of "public IP" for the nodes, or if you have Security Group settings that block access from outside the Security Group
- **Authentication or Authorization**
    - All files stored in SUFS are accessible to anyone who can connect to the NameNode & DataNodes.

- **Configurable/Variable Block Sizes or Replication Factors**
    - You may choose a block size and a replication factor and hard code that into your system. You do *not* need to allow these to be configurable nor to allow them to be different for different files
- **Rack Awareness**
    - Don't worry about being "rack aware" - just treat all nodes as equal.
- **Using an EditLog on the NameNode**
    - You do not need to use the logging mechanism described for the NameNode.
    - You do not need to create files named EditLog and FsImage, but may store the NameNode data on the disk however you want and using whatever filenames (or structure) that you want.
- **Cluster Rebalancing**
    - Make a reasonably intelligent decision when you place file blocks initially, and then just leave them in that location from then on (unless necessary for fault recovery purposes).
- **Checksums**
    - You don't need to checksum blocks as described in the "Data Integrity" section of the HDFS documentation.

Testing

Since SUFS is intended to store large files, you can use the data files that you used in the MapReduce assignment. These are conveniently already in S3, so your client program can just pull them from there.

Be sure you test at least one really large file (> 1 GB) and at least one file small enough to fit in a single block. Also try at least one file that is an exact multiple of the block size and at least one file that isn't. Be sure the data you get from a read is *bit-for-bit* identical to the file that was initially written in. (You can use the *diff* command on the command line to compare two files.)

Note: *There is a fee for all data sent into and out of the AWS data center (which is small and normally not worth worrying about, but could add up quickly when we're talking about many GB of file data). However, there is no fee for moving data around within AWS... So, copying data from S3 to an EC2 instance avoids this fee, whereas downloading a file from S3 to your computer then uploading it from your computer to EC2 will hit you with an outgoing data fee* ***and*** *an incoming data fee. (Also, different 'regions' are considered different data centers, so fees apply between regions, but not when moving data around within a region. For this class, we'll stick to the us-west-2 Oregon region, so that should be okay.)*

Of course, be sure to test not only the normal operation and the expected failure states (i.e., the ones for which you designed fault tolerance) but also any other failure states and various edge cases! (In some cases, the client should report an error – what are those cases?)

Grading

This project is worth 30% of the course grade plus an additional 10% for progress updates, divided as follows:

- 2% for submitting the Team Charter and completing all peer/self-feedback forms
    - graded for completion and reasonable effort
- 4% each for submitting the two checkpoints
    - checkpoint grades are based on effort and providing the requested materials with sufficient details, not on successful functioning of the project
    - first checkpoint is a team grade only (individual feedback is for your information only)
    - second checkpoint is a team grade weighted by individual grade
- 30% final submission & demo
    - team grade weighted by individual grade
    - grade is based on successful functioning
    - see the rubric in Canvas for details

Individual Weighting

The first checkpoint will be a team grade only. (A peer/self-feedback will be done but will not affect grades – this is informative only and helps you address any concerns before it does affect your grade.) The second checkpoint and final submission will be weighted by an individual component.

Everyone on the team gets the same team grade out of 100% and each team member gets your own individual grade out of 100%. Your actual grade on the assignment is the two of these multiplied together, and rounded to the nearest 1% using natural rounding, for example:

- Team 90% and individual 100% = you get 90%
- Team 100% and individual 80% = you get 80%
- Team 90% and individual 90% = you get 81% (because 0.9 * 0.9 = 0.81)
- Team 90% and individual 95% = you get 86% (because 0.9 * 0.95 = 0.855, which rounds to 86%)

Individual grades will not be affected at all by how well your project does or does not work (that's what the team grade is for), they're only determined by how well you participate and contribute to your team.

Submitting the Team Charter will be a team grade only. Submitting the peer/self-feedback will be graded individually and only graded for completing the feedback surveys.