Project 1: Distributed File System (v 1.2)

Starter repository on GitHub: https://classroom.github.com/g/Rb8Lz-hz

In this project, you will build your own distributed file system (DFS) based on the technologies we've studied from Amazon, Google, and others. Your DFS will support multiple *storage nodes* responsible for managing data. Key features include:

- **Probabilistic Routing**: to enable lookups without requiring excessive RAM, client requests will be routed probabilistically to relevant storage nodes via a collection of bloom filters.
- **Entropy-Driven Compression**: file entropy is analyzed before storage; low-entropy files will be compressed before storage to save disk space.
- **Parallel retrievals**: large files will be split into multiple *chunks*. Client applications retrieve these chunks in parallel using threads.
- **Interoperability**: the DFS will use Google Protocol Buffers to serialize messages. *Do not use Java serialization*. This allows other applications to easily implement your wire format.
- **Fault tolerance**: your system must be able to detect and withstand two concurrent storage node failures and continue operating normally. It will also be able to recover corrupted files.

Your implementation must be done in Java, and we will test it using the *orion* cluster here in the CS department. Communication between components must be implemented via sockets (*not* RMI, RPC or similar technologies) and you may not use any external libraries. The Java Development Kit has everything you need to complete this assignment.

Since this is a graduate-level class, you have leeway on how you design and implement your system. However, you should be able to explain your design decisions. Additionally, you must include the following components:

- Controller
- Storage Node
- Client

Controller

The Controller is responsible for managing resources in the system, somewhat like an HDFS NameNode. When a new storage node joins your DFS, the first thing it does is contact the Controller. At a minimum, the Controller contains the following data structures:

- A list of active storage nodes

- A **routing table** (set of bloom filters for probabilistic lookups)

When clients wish to store a new file, they will send a *storage request* to the controller, and it will reply with a list of destination storage nodes (plus replica locations) to send the chunks to. The Controller itself should **never** see any of the actual files, only their metadata.

To maintain the routing table, you will implement a bloom filter of file names for each storage node. When the controller receives a file retrieval request from a client, it will query the bloom filter of each storage node with the file name and return a list of matching nodes (due to the nature of bloom filters, this may include false positives).

The Controller is also responsible for detecting storage node failures and ensuring the system *replication level* is maintained. In your DFS, every chunk will be replicated twice for a total of 3 duplicate chunks. This means if a system goes down, you can re-route retrievals to a backup copy. You'll also maintain the replication level by creating more copies in the event of a failure. You will need to design an algorithm for determining replica placement.

Storage Node

Storage nodes are responsible for storing and retrieving file chunks. When a chunk is stored, it will be checksummed so on-disk corruption can be detected. When a corrupted file is retrieved, it should be repaired by requesting a replica before fulfilling the client request.

Some messages that your storage node could accept (although you are certainly free to design your own):

- Store chunk [File name, Chunk Number, Chunk Data]
- Get number of chunks [File name]
- Get chunk location [File name, Chunk Number]
- Retrieve chunk [File name, Chunk Number]
- List chunks and file names [No input]

Metadata (checksums, chunk numbers, etc.) should be stored alongside the files on disk.

After receiving a storage request, storage nodes should calculate the Shannon Entropy of the files. If their maximum compression is greater than `0.6` (`1 - (entropy bits / 8)`), then the chunk should be compressed before it is written to disk. You are free to choose the compression algorithm, but be prepared to justify your choice.

The storage nodes will send a **heartbeat** to the controller periodically to let it know that they are still alive. Every 5 seconds is a good interval for sending these. The heartbeat contains the free space available at the node and the total number of requests processed (storage, retrievals, etc.).

On startup: provide a storage directory path and the hostname/IP of the controller. Any old files present in the storage directory should be removed.

Client

The client's main functions include:

- Breaking files into chunks, asking the controller where to store them, and then sending them to the appropriate storage node(s).
  - **Note**: Once the first chunk has been transferred to its destination storage node, that node will pass replicas along in a pipeline fashion. The client should not send each chunk 3 times.
  - If a file already exists, replace it with the new file. If the new file is smaller than the old, you are not required to remove old chunks (but file retrieval should provide the correct data).
- Retrieving files in parallel. Each chunk in the file being retrieved will be requested and transferred on a separate thread. Once the chunks are retrieved, the file is reconstructed on the client machine.

The client will also be able to print out a list of active nodes (retrieved from the controller), the total disk space available in the cluster (in GB), and number of requests handled by each node.

NOTE: Your client must either accept command line arguments or provide its own text-based command entry interface. Recompiling your client to execute different actions is not allowed and will incur a **5** point deduction.

Tips and Resources

- Use a logging framework to track events in your system (either use the built-in Java logging framework or an external library). For example, if a StorageNode goes down, the controller should probably print a message acknowledging so. This will be extremely helpful when debugging your system.
- Use the orion cluster (orion01 – orion12) to test your code in a distributed setting.
  - These nodes have the Protocol Buffers library installed as well as the `protoc` compiler. However, you may want to simply bundle the latest version of the library (as a fat jar) with your code instead.
  - To store your chunk data, use `/bigdata/$(whoami)`, where `$(whoami)` expands to your user name.

Project Deliverables

This project will be worth 20 points. The deliverables include:

- **[5 pts]**: Controller
  - **[2]** Routing table (Bloom Filter implementation and lookup functionality)
  - **[1]** Node failure detection
  - **[2]** Coordinating replica maintenance
- **[6 pts]**: Storage node implementation:
  - **[1]** Storing chunks and checksums on local disks
  - **[1]** Detecting (and recovering from) file corruption
  - **[1]** Automatically compressing low-entropy files
  - **[2]** Coordinating replica maintenance
  - **[1]** Heartbeat messages
- **[4 pts]**: Client implementation:
  - **[1]** Storing files (chunk creation, determining appropriate servers)
  - **[2]** Retrieving files in parallel
  - **[1]** Viewing the node list, available disk space, and requests per node.
- **[3 pts]**: Interactive tests. You earn all 3 points unless:
  - You have to restart components of your system during testing
  - The client crashes during storage, retrieval, etc.
  - A file is returned corrupted but then retrieves correctly on the second try
  - Depending on severity, these issues carry a **1 point** deduction per instance.
- **[2 pts]**: Design document and retrospective (due **after** code submission). You may use UML diagrams, Vizio, OmniGraffle, etc. This is more to benefit you later when you want to refer back to the project or explain it in interviews etc. It outlines:
  - Components of your DFS (this includes the components outlined above but might include other items that you think you'll need)
  - Design decisions (how big the chunks should be, how you will place replicas, etc…)
  - Messages the components will use to communicate

- Answers to retrospective questions (will be supplied later)

Note: your system must be able to support **at least 12** active storage nodes, i.e., the entire orion cluster.

Grading

We'll schedule a demo and code review to grade your assignment. You will demonstrate the required functionality and walk through your design.

I will deduct points if you violate any of the requirements listed in this document — for example, using an unauthorized external library. I may also deduct points for poor design and/or formatting; please use good development practices, break your code into separate classes based on functionality, and include comments in your source where appropriate.

Changelog

- *9/10*: Initial version 1.0 posted
- *10/4*: Updated entropy details
- *10/8*: Grading rubric and a few more clarifications