Project 3: Distributed File System

14-736 Spring 2020

Assigned: Monday, 16th March
Checkpoint due: Sunday, 29th March
Final Submission: Sunday, 5th April

Contents

1 Overview

In this project, you will implement a simple distributed file system. Files will be hosted remotely on one or more storage servers. Separately, a single naming server will index the files, indicating which one is stored where. When a client wishes to access a file, it first contacts the naming server to obtain the IP address and client port of the storage server hosting it. After that, it communicates directly with the storage server to complete the operation.

When completed, your file system will support file reading, writing, creation, deletion, and size queries. It will also support certain directory operations - listing, creation and deletion. It will be possible to lock files, and commonly accessed files will be replicated on multiple storage servers.

2 Logistics

You should work with a partner on this project. Mention the names and IDs of both partners in a **group.txt** file. Be sure to include all files necessary to compile and run the file system. Your archive (.zip) should contain everything needed to compile your code and test suite. Normally, your archive will contain a naming directory and a storage directory which contains your naming server implementation and storage server implementation and all the code we provide you for test. This project requires you to have an environment which has Java installed no earlier than Java 8 and OpenJDK 11 installed (our test uses java.net.http which is in OpenJDK 11). The final grading is done on unix andrew machine so please make sure your code is compilable on unix andrew machine.

3 Checkpoint

This project involves a large amount of work, and many features must be implemented before it is complete. To help you with this, we suggest a checkpoint, for which you should implement all file and directory operations except locking and replication. The checkpoint is a convenient intermediate goal for you to work towards. Your checkpoint submissions won't be graded but a penalty may be applied to your final grade if you submit your checkpoint late. Notice, you can't use any grace day for checkpoint.

4 Detailed Description

As the Figure 1 shows, the file system consists of several major components.

First, servers and clients need a way to identify files. Each file is identified by its path (string) in the distributed file system. The paths are transmitted through the various interfaces in the file system.

The primary function of storage servers is to provide clients with access to file data. Clients access storage servers in order to read and write files. Since storage servers store the data, they are also, in this design, the entities that report file sizes. Storage servers also must respond to certain commands from the naming server.

Clients do not normally have direct access to storage servers. Instead, their view of the file system is defined by the file system's single naming server, for which clients have the naming server's IP address and service port. The naming server tracks the file system's directory tree, and associates each file in the file system to a storage server. When a client wishes to perform an operation on a file, it first contacts the naming server to obtain the IP address and client port of the storage server hosting the file, and then performs the operation through the IP address and client port. Naming servers also provide a way for storage servers to register their presence.

4.1 Communications

First we need to understand how the communications happen among the naming server, storage servers, and clients. All the communications in this project are through HTTP-based RESTful APIs. Every component of our system provides a uniform interface, which is fundamental to the design of any RESTful system. It simplifies and decouples the architecture, which enables each part to evolve independently. Individual resources are identified in requests by using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. We choose to send all data in JSON format. The APIs you need to implement are specified in the handout folder *API/*.

4.2 Paths

As stated above, paths are represented by string, which are transmitted through all interfaces in the file system. Notionally, each path is a string of the form */directory/directory/directory-or-file*, but you may choose any internal representation you want - for example, you may represent a path internally as an array of path components. Paths are always taken to be relative to the root of the file system. Finally, the file system locking scheme requires paths to be comparable. See the section on locking for details.

4.3 Storage Servers

Storage servers provide two interfaces:

1. The client interface (document: *API/API_Storage_Storage.md*), through which clients perform file operations. It provides the client with three operations: file reading, file writing, and file size query.
2. The command interface (document: *API/API_Storage_Command.md*), through which the naming server may issue file management commands to the storage server. It allows the naming server to request that a file on the storage server be created, deleted, or copied from another storage server, as part of replication.

The job of the storage server is simply to respond to these requests. These requests may come in concurrently.

A question arises - where does the storage server actually store the data for the files it is hosting? In this design, the storage server is required to put all its files in a directory on the machine that it is running on - this will be referred to as the storage server's local or underlying file system. The structure within this directory should match the storage server's view of the structure of the entire distributed file system. For example, if a storage server is storing its files in the local directory */var/storage*, and it is hosting a file whose distributed file system path is */directory/README.txt*,

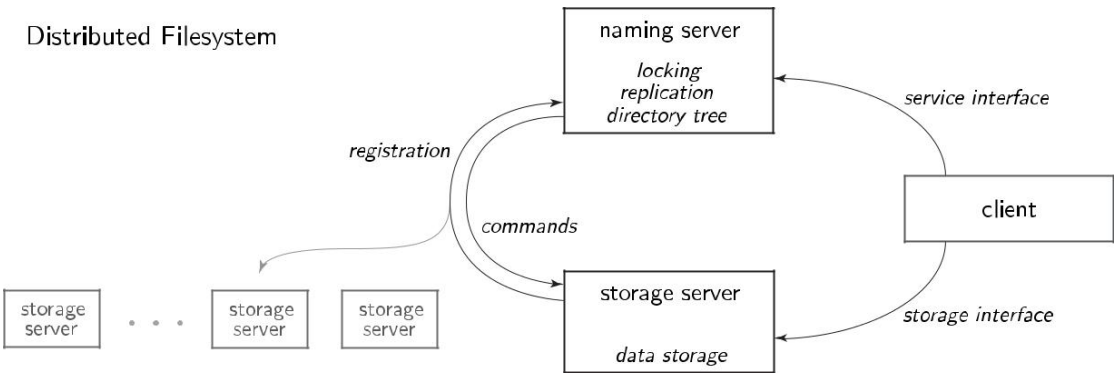

Figure 1: Distributed File system

then that file's full path on the local file system should be */var/storage/directory/README.txt*. If a storage server is not aware of the existence of a file in the file system (because it is hosted by another storage server), it need not store anything for the file. This scheme provides a convenient way to make data persist across storage server restarts.

4.4 Naming server

The naming server can be thought of as an object containing a data structure which represents the current state of the file system directory tree, and providing several operations on it. It also provide two interfaces:

1. The service interface (document: *API_Naming_Service.md*). For the optional checkpoint, the service interface allows a client to create, list, and delete directories, create and delete files, determine whether a path refers to a directory or a file (or neither), and obtain IP address and client port for storage servers. For the final version, the service interface also allows clients to lock and unlock files and directories.
2. The registration interface (document: *API_Naming_Registration.md*), which storage servers use to inform the naming server of their presence and join the file system. The registration interface is used once by each storage server on startup. When a storage server is started, it contacts the naming server and provides the naming server with its IP address and two port numbers: one for the storage server's client interface, which the naming server will later provide to clients, and one for the storage server's command interface, which the naming server will use to maintain the storage server's view of the file system in a consistent state. The storage server also lists all files that are present in its directory on its underlying file system. If any of those files are not yet listed in the distributed file system, they are added. The rest are considered duplicates, and the naming server will request that the storage server delete them.

In the final version of your file system, the naming server transparently performs replication of commonly accessed files, causing multiple storage servers to maintain copies of the same file. This is not under the direct control of the client. The details are given in the section on replication.

4.5 Coherence and Thread Safety

Each server in the file system must individually be thread-safe: attempting to perform operations concurrently on a single server should never cause that server's state to become inconsistent. The consistency requirements across the whole file system, however, are much more relaxed. The design is fairly fragile and depends strongly on having well-behaved clients. For the version up to the checkpoint (and therefore without locking), consistency also depends on luck.

At the checkpoint, without locking, clients maintain no special state over a file that would allow them to consider the file "open" or reserved for their own purposes. This means that, while a single read or write request should complete correctly once it arrives at the storage server, any other client may interfere between requests. Files currently being accessed by a client may be overwritten by another client, deleted, moved to another storage server, and re-created, all without the client noticing. The Final version of the project allows a client to lock a file in order to prevent other well behaved clients from performing any of these operations until the lock is released.

When implementing the storage and naming servers, you must decide when is the appropriate time for the naming server to command each storage server to create or delete files or directories, thus maintaining the servers' views of the file system in a consistent state. As much as possible, it is preferable to avoid having to rigidly and synchronously maintain all storage servers in the same state. However, the interfaces are highly simplified and do not provide good ways to implement complex schemes for lazy file creation or deletion, so code accordingly. As an example, a file

that the naming server has been successfully asked to delete should not remain accessible for subsequent requests to the storage server.

We ask that files that have been deleted from the storage server be deleted from the underlying file system (as opposed to merely unlinked from some internal data structure), and that the storage server eagerly remove underlying file system directories that have become empty. This allows us to test the behavior of your storage server in response to requests.

4.6 Locking

For the final version, you must implement a custom lock type and a particular locking scheme, which well-behaved clients can use to ensure consistency across multiple requests. Each file and directory may be locked for shared (reading) or exclusive (writing) access. Multiple clients may lock the same object (file or directory) for shared access at the same time, but when a client locks an object for exclusive access, no other client can lock the same object for any kind of access.

Shared access permits multiple well-behaved clients to perform operations such as reading files and listing directories simultaneously. Such operations do not interfere with each other: for example, two clients may safely read the same file at the same time. Exclusive access permits well-behaved clients to write to files and modify the directory tree.

When a client requests that any object be locked for any kind of access, all objects along the path to that object, including the root directory, must be locked for shared access. Otherwise, for example, a file that is locked for reading (shared access) can still be deleted when another client removes its parent directory, because the parent directory was not locked by the reading client. Be careful about the order in which you take these locks on parent directories. If locks are taken in haphazard order, it is possible to end up with a deadlock where two clients are each holding a lock, and both seek to also take the lock held by the other client in order to proceed.

The locking scheme has a further constraint. Some clients may need to lock multiple objects simultaneously. Doing this in arbitrary order on each client can also result in deadlock for the same reason. Therefore, we require that path objects be comparable. Clients must take locks on path objects in order from least path to greatest path, according to the results of comparison. This requirement interacts with the requirement to lock subdirectories: when an object is locked, it is not only the object itself whose lock is taken, but the lock on every object along the path to it. Great care must be taken to ensure that the order defined by the comparison does not lead to deadlocks due to this interaction. Be very careful about how you compare path objects. We ask that you describe your locking scheme and comparison scheme, and their interaction, in comments of your lock implementation.

In addition to all of the above, the locks must also provide some measure of fairness. It should not be the case that a client which is waiting for a lock is continuously denied it, and the lock is given to other clients that requested the lock later. This is especially important for clients requesting exclusive access. In the absence of fairness constraints, a large number of readers will make it impossible for any client to write to a file. The writing client will wait for the lock as new readers keep arriving and sharing the lock with current readers.

In order to avoid this, in this project, we require that you give the lock to clients on a first-come, first-serve basis. It must never be the case that a client which requests a lock later is granted access before a client which requested earlier, unless both clients are requesting the lock for shared access. In the latter case, however, if two clients are requesting the lock for shared access, and there are no intervening waiting clients, both clients must be able to take the lock at the same time.

4.7 Replication

The final version of the project must support replication according to the following simple policy: during a series of read requests, the file is replicated once for every 20 read requests, provided

there are enough storage servers connected to the naming server to maintain additional copies. At a write request, the naming server selects one storage server to keep a copy of the file, and all other copies are invalidated (removed) before the remaining copy is updated.

Since the naming server has no way of directly tracking read and write requests, or the amount of traffic associated with each file, it makes the simplifying assumption that taking a shared lock on a file is tantamount to a read request, and taking an exclusive lock is a write request.

Be careful about how replication interacts with locking. Well-behaved clients should not be able to interfere with the replication (copy) operation and cause the results to become inconsistent. Well-behaved clients should, however, be able to read from existing copies of a file, even as a new copy is being created.

5 APIs

In the API documents, many API will have this kind of JSON response:

{
"exception_type": "FileNotFoundException",
"exception_info": "File/path cannot be found."
}

In these cases, our tests expect your server to return EXACTLY the same "exception_type" field, no matter what programming languages you are using, so make sure you follow the specification for "exception_type". The HTTP return code and the "exception_info" field are not so important, we won't test whether you also have the same return code and "exception_info" and they are designed only to help you debugging.

In the handout, you could see a lot of Java helper classes under *jsonhelper/*. These are classes our test suite uses to parse the request data from JSON and generates responses to JSON. Our test suite uses gson-2.8.6.jar to help us convert Java Objects into JSON and back. In our API documents, we also explicitly tell you which helper class we use for each API. Do NOT modify anything in *jsonhelper/* ! If you choose to use Java to implement your server, you are free to use the same helper classes we provide, or you can write your own.

Because all the communication data is converted to JSON in our system, there is a potential problem for read() and write(), where we must send bytes array to the server. But JSON doesn't support byte[]. You may use base64 encoding/decoding to solve this problem. See the link for more information. Make sure that the programming language you choose has support for base64 encoding/decoding. You could use third-party libraries to achieve this.

6 Programming Languages

Although the test suite is written in Java, it strictly follows the RESTful APIs specified in *API/* to test your code, which means that you could use whatever programming languages you want to implement your naming server and storage server, as long as you also strictly follow the API specifications. Please use common programming languages so that we could help you better.

7 Implementation and Configuration for Servers

The handout includes all files requested to build the test suite. What you need to do is create 2 new directories, add your own naming server implementation under *naming/* and your own storage server implementation under *storage/*.

Please have a look at the file *test/port.config*. Your naming server should accept 2 arguments upon start.

- port number for its service interface
- port number for its registration interface.

Your storage server should accept 4 arguments upon start.

- port number for its client interface
- port number for its command interface
- port number of the naming server's registration interface (it need to know this to register itself to the naming server upon start)
- the local directory in which the storage server is to locate the files it is to serve. (your storage server should create this directory upon start if it does not exist).

After you finish your naming server and storage server's implementation, there is one more thing to do: modify the current Makefile so that when we run '*make*', it will compile all tests (we have already done this) as well as your naming server and storage server.

The final thing to do: please have a look at the file *test/Config.java*. It specifies the command to start the naming server and 2 storage servers (Yes! We need to start 2 storage servers to test your storage replication implementation!). Our test will use these 3 lines to start your servers as background processes. If you are using Java to implement your server, then perhaps you don't need to change this. But if you are using other programming languages, you need to modify these 3 lines to make sure that we can use them to start your servers under the root directory of this project.

8 Grading (100 points)

8.1 Testing (90 points)

When testing your code, the test suite will start your naming server and multiple storage servers as background processes, which means that you couldn't see your server's standard output or standard error through the terminal. You could redirect your server's output and error to file to assist your debugging (for Java, you can have a look at System.setOut and System.setErr).

We test your naming server implementation and storage server implementation separately, which means that you could even pass all naming server's tests without implementing your storage server and vice versa. This is because we have 'fake' server implementations in the test suite to isolate yours. The bug in your naming server won't affect your storage server, and the bug in your storage server won't affect your naming server.

Before implementing your own naming server, you can have a look at the file *test/storage/TestNamingServer.java*. This is a very simple naming server implementation in Java which only implements the registration interface for testing purposes. The register() implementation doesn't meet the real naming server's requirement. But we hope you could understand the logic of the naming server, for example, the naming server should have 2 built-in HTTP servers to listen on different ports which have different handlers to handle different incoming URIs. The file *TestStorageServer.java* under *test/naming* might also be helpful.

After you modify the Makefile and *test/Config.java*. Now it's time to test!
Under the root directory of this project:
To run tests for checkpoint: make checkpoint
To run tests for final: make test

8.2 Coding Style (10 points)

Please maintain a good and consistent coding style for all the code you write. You may use a different coding style from the starting code, but please keep it readable and consistent in all the files you modify or produce. To get full points for coding style, your code needs to meet the following requirements.

1. File header for each file you create
2. Function header for each function
3. Good variable naming
4. No dead code, no obsolete comment
5. Good modularity
6. Brief comment for function logic, variable usage

9 Important notes and tips

1. We recommend you first read this writeup, then read the API documents under API/. After this, you will have a basic understanding of the duty of the naming server and the storage server.
2. Be careful about what local directory you start the storage server in. Storage servers register with a naming server on startup. Every file in the storage server's directory on the underlying file system which is already present in the distributed file system will then be deleted to eliminate duplicates. This means if you start a storage server in a directory, a large number of files could suddenly disappear. Make sure this is not an important directory. The test cases only start storage servers in temporary directories.
3. When implementing locks for the final version of your file system, you must not simply make every method of the naming server synchronized, and then take locks. This will decrease concurrency. Instead, you are expected to rely on the per-object locks to ensure thread safety as much as possible. Two clients should be able to traverse the same directories simultaneously within the naming server, provided they are locking the objects along the way for shared access.
4. The storage's copy interface, which is used for replication, may be used to replicate very large files. Such files cannot be stored in the virtual machine's memory all at once. To simplify this, you may assume that we won't test to read/write/copy very large files.
5. The section on replication is very short, because replication is easy to specify. It is, however, not trivial to implement. Replication is a potentially long-running process that runs concurrently with many other processes, including accesses to existing copies of the le being replicated. Allow yourself adequate time to think through, design, and test the replication mechanism.
6. If you are using Java. Be careful about the semantics of standard Java methods. The documentation is often terse and assumes a certain understanding (or willingness to test), and may act in somewhat unexpected ways in corner cases. For example, File.mkdirs, which might be useful for your storage server to manage the underlying file system, will
7. return false if the directory it is asked to create already exists. This could lead to an annoying corner case if you always blindly call this method, taking it to mean "ensure that the directory exists.
8. If you use random number generators for deciding which server should store a new file, or for picking servers during replication or read accesses, be aware that Java's standard random number generator is not thread-safe.

References

File management (File object is really more like a path)
http://docs.oracle.com/javase/6/docs/api/java/io/File.html

Java synchronization
http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

START EARLY!