15440 - Fall 2017

Project 2

Synchronization & Replication in FileStack

Out: October 04, 2017

Due: October 21, 2017

15440 - Fall 2017

Contents

1 Learning Objective

Project 2 applies the theory of two important aspects of distributed systems: *synchronization* and *replication*. The learning objective of this project is two-fold:

1. Examine and apply synchronization algorithms that enable correct and consistent sharing of common resources.
2. Apply intelligent replication strategies for load-balancing and performance.

2 Project Objective

FileStack (Project 1) was described as a distributed file system which allows clients to read or write files on remote storage servers. Two important aspects of such file systems are to:

1. **Ensure correctness when multiple clients read/write**: this problem is known as maintaining "consistency." You will now maintain locks for reader and writer clients. Since reads do not change the file, you can allow as many readers as possible to access the file at the same time. However, if a writer has to write to a file, you have to ensure that there are no other readers or writers who have locked the file. This concept is referred to as read-write locks. In addition to ensuring basic consistency, you will also solve the unfairness problem that manifests while ensuring consistency.
2. **Improve performance by smart-allocation of files across multiple servers**: we solve this problem by dynamic replication of files. Instead of all clients fetching a file from one storage server, we can now parallelize - and hence improve read/write times - by redirecting clients to different replicas. It also aids in load-balancing of storage servers: many clients who access a very popular file will now not go to a single storage server.

3 Conceptual Architecture

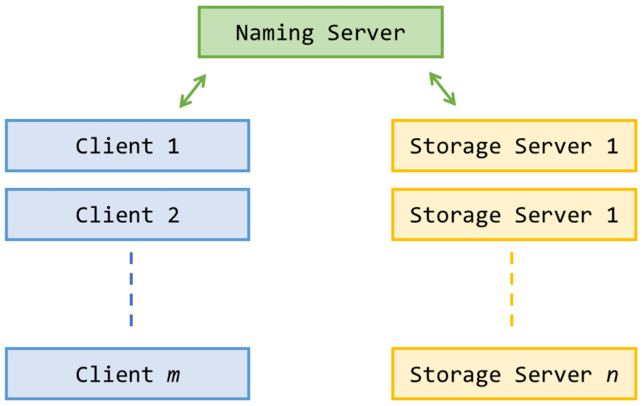

Figure 1: Architecture of *FileStack (Project 1)*

The conceptual architecture of the *FileStack* system, shown in Figure 1, remains the same as that of project 1:

1. The Clients will contact the naming server to access files.
2. The Naming Server will redirect the clients towards Storage Servers.
3. The Clients will read/write files on Storage Servers.

However, in this project, you will: **(1)** make sure that the files are *consistent*, and **(2)** implement a *replication* policy.

3.1 System Description

In this section, we describe the functionalities that you will implement and the nature of the protocol to ensure consistency and reliability. You will implement two main functionalities: *consistency* and *replication*.

3.2 Consistency

You will implement a scheme to ensure consistency in distributed systems called "read-write locks." The main concept is for reader and writer clients to grab locks before performing operations, and release them once these operations are done. If some other client already owns a lock, this means that the file is currently being accessed by that client. In such a case, it is possible for the file to become inconsistent if more than one client accesses the file at the same time. For example, if one writer is modifying a part of a file, and another writer modifies an overlapping part, then the file becomes inconsistent. The goal in this project is to avert the occurrence of such a scenario (and alike) via ensuring consistency when multiple clients are present. You will implement and design algorithms that achieve that goal.

In particular, you will implement *coarse-grained locks* (similar to Google Chubby). Here, the user will ask for an **exclusive lock** or a **non-exclusive lock** from the naming server for proceeding with a write or a read, respectively. As such, you have to provision two more methods in the Naming Server (in the Service interface).

3.2.1 Reader Operation

A reader will grab a **non-exclusive lock** to a file before reading, and will release the lock once it is done. While one reader is reading the file, you will allow other readers to concurrently read the file (i.e., other readers will also be able to grab non-exclusive lock to the file). This is because read operations do not modify files, and hence, cannot cause inconsistencies. Nonetheless, simultaneous reads to a file can decrease the time taken for reading the file by multiple readers (as readers do not wait for each other), and accordingly, improving performance.

3.2.2 Writer Operation

A writer will grab an **exclusive lock** to a file before writing, and will release the lock once the operation is completed. In order to keep the file consistent, a writer can successfully grab the lock only when:

1. None of the readers are reading, or
2. None of the other writers are writing.

3.2.3 Locking a File/Directory in the Hierarchy of the Path

Consider writing to a file in the absolute path */home/users/student1/work/project2.txt*. While obtaining a write lock to the file, you have to make sure that:

1. No other reader or writer is operating on file *project2.txt*
2. No writer is modifying the hierarchy of directory. For example, consider a scenario where **client 1** is trying to rename sub-directory "work" to "play," and **client 2** is writing to *project2.txt*. Then **client 2** should write - after **client 1** is finished - to the new location */home/users/student1/play/project2.txt* (and not to */home/users/student1/work/project2.txt*). The likelihood of inconsistencies in DFSs renders increasingly high if whole directory paths are not locked. You will avoid inconsistencies in your *FileStack* system by:
   (a) Obtaining non-exclusive locks to all the directories in a given path, and
   (b) Obtaining an exclusive lock for the file to be altered (e.g., in the aboveexample).

Such a strategy will ensure that writers do not introduce inconsistencies in your file system. For readers, you will also follow almost the same strategy: you will lock all the directories and the file in the path by using non-exclusive locks. By applying this approach, you will not allow any writer (similar to **client 1** above) to rename or delete a directory when a reader is reading. Finally, be aware of deadlocks while locking a hierarchy of directories/files. You have to lock different directories and a file in a path. You might obtain locks to some and fail in getting locks to others. Design a mechanism that precludes deadlocks.

3.3 Replication

The second problem is to improve the performance through replication. The problem is motivated from the type of access-pattern to files that are seen in normal file systems and distributed file systems. Some files have a lot of requests (e.g., system log files), while others are very rarely accessed (e.g., some remote photos in a user's home directory). We refer to the former as *hot-files*, and the latter as *cold-files*. In distributed systems, if there are a lot of requests attempting to read/write to one *hot-file*, the networking and processing loads on the storage server that stores the *hot-file* is very high, while that of the storage servers that host only *cold-files* are rarely utilized. Consequently, the waiting queue for accessing the hot-files becomes very large, and might demonstrate a bottleneck. You will provide a solution to avoid such potential bottlenecks through replication.

3.3.1 Replication Policy

The naming server maintains a counter that keeps track of the number of requesters to a file. This information will be useful to measure the *hot-ness* of a file (e.g., the larger the number of requesters to a file; the higher the *hot-ness* of the file). To avoid stressing a replica as well as its hosting storage server, you will scale replicas linearly as the number of requesters increases. Specifically, you can set the number of replicas per a file as follows:

```
num_replicas = ALPHA * num_requesters
```

where `ALPHA` is a positive constant. By controlling `ALPHA`, you control the number of replicas per file. In addition, you would want to limit the replicas of a file. For that sake, you can maintain a `REPLICA_UPPER_BOUND`, and avoid replicating a file whose number of replicas has exceeded this threshold. In particular, you can now set the number of replicas per file as follows:

```
num_replicas = min(ALPHA * num_requesters, REPLICA_UPPER_BOUND)
```

This fine-grained control of replication will alter the number of replicas even upon a change of one or two requesters. This might add to an already large overhead of dynamic replication (what we are essentially implementing). To enhance the policy, you can rather apply a coarse-grained approach by rounding the number of replicas to the next integer that is a multiple of 20 (as an example). That is,

```
num_requesters_coarse = {N | N >= num_requesters & a multiple of 20}
```

Afterwards, you can compute the number of replicas as follows (*this is the final formula that you should implement*):

```
num_replicas = min(ALPHA * num_requesters_coarse, REPLICA_UPPER_BOUND)
```

Figure 2 shows an illustrative graph that demonstrates how the number of replicas for a file changes upon changing the number of requesters.

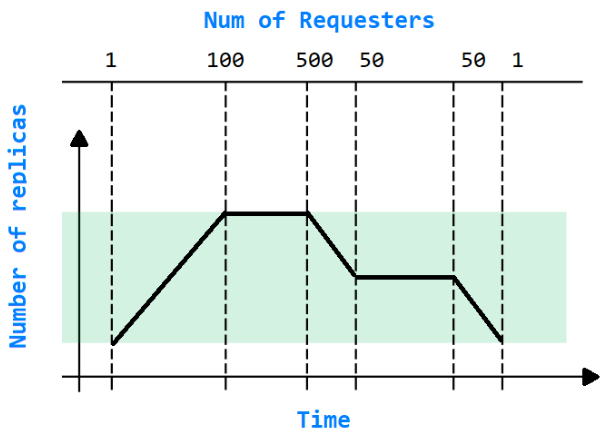

Figure 2: Linear Replication Policy with an upper-bound

3.3.2 Read-request Handling for Hot-files

A simple approach for the server to perform load-balancing is by randomly redirecting requests to replicas.

3.3.3 Write-request Handling for Hot-files

You will implement an *invalidation-based* policy. Specifically, during a write call, you will invalidate all-but-one replicas, and subsequently write on the remaining replica (the remaining file). Keep in mind that if you are to write to a replicated file, you have to wait for every reader or a writer (*if any*) to release their/its lock(s).

4 Design and Implementation Guidelines

In this project, you will improve your Project 1 to add concurrency and replication. Hence, it is vital that basic components of project 1 (RMI, storage and naming) are working well.

4.1 File Concurrency and Synchronization

The first important part in implementing correct synchronization is to provide each file/directory with a capability to lock. But where do you implement the lock? Do you implement it at the naming server? Or do you implement it at the storage server where the file resides? How would you ensure that locks work with replicated files (second part of this project)? First, think about these questions.

The second design decision is regarding queues. Recall that there are readers and writers waiting to access a file. A writer can lock only when there are no readers: what happens when multiple readers request the file - one after another? Will the writer ever get a chance to lock the file for writing?

The third decision is ensuring the ordering of reads/writes. A simple way is *FIFO*: the first request (read/write) occurs before the second, second before the third, and so on. However, note that there is no ordering between reads and writes. This compromises the exact ordering, but helps improve the performance of distributed systems: all reads can go on together even if there is one write in-between. For some systems (Google search, for instance), this works perfectly fine (Read the section 7.3.1 "Eventual consistency" in the *Tanenbaum* textbook). All reads should be in *FIFO*, and all writes should be in *FIFO*. How do you ensure the order of reads and writes? How are the requests queued?

4.2 Replication

Replication introduces whole new range of challenges. The exact logic of replication is not too complicated. However, remember that you have to copy files from storage servers to others. This might require ensuring that directory hierarchy for a file is exactly created (but do not overwrite the files that already exists in sub-trees) as in the original storage server.

The second issue is to re-think file functionalities. What will happen if you read a file? How do you write a file, and update all replicas at different storage servers? How do you handle file/directory deletions?

5 Design and Implementation Guidelines

Please use the code in **P2_StarterCode.zip**.

The starter code is an extension of the starter code for Project 1. The starter code includes complete replacements for the apps/, test/, and conformance/ directories, as these are all entirely the course staff's responsibility, and are affected by the changes going from Project 1 from Project 2.

The main features of the starter code are as follows:

- The build/ and client/ directories are unaffected, and are included just for completeness.
- The naming/ and storage/ directories contain only the new *.java* files for the updated *Command* and *Service* interfaces. These files are meant to replace the existing ones that you have. You should copy your implementations of the interfaces from Project 1, and edit them to conform to the new interfaces. There are only a few new methods - **lock** and **unlock** - in Service, and **copy** in Command.
- The unit/ directory is not included. You should copy your own, if you want to use the unit tests that you already have.
- The rmi/ directory is not included. It is completely unaffected, and you should copy your own rmi code into this directory.
- The common/ directory is not included, because there is only one file in there. It does change, but most of it stays the same as in Project 1. You should copy your Project 1 version, and then make the **Path class implement *Comparable*** for Project 2. The purpose of this is to allow applications to pick a locking order when taking multiple locks - the order on the paths will help to prevent deadlocks. A file "NOTE" is included with the starter code to give you a starting point in doing this, but if you choose to use it, you should copy the method skeleton in "NOTE" into your own copy of Path.java.

6 Test Suite

We have provided test code for Project 2 as well. The test cases test if your code is conforming to the above design guidelines, and to check if the implementation is correct. **Please note that this is a service offered to help you design and test faster. You are solely responsible to make sure that your code works perfectly**. During grading, we will also use other test cases to make sure that your project is working as expected.

We have also provided "apps" that will let you use your distributed file system using command-line.

7 Q&A

We use Piazza as a platform for asking questions and receiving answers. Posting your questions on Piazza will help the whole class benefit and will certainly avoid redundancy. Find our Piazza page at:

https://piazza.com/qatar.cmu/fall2017/15440/home.

8 Deliverable

As a final deliverable, you should submit an archive containing the source code for the RMI library, naming server, storage server, and test cases in separate directories.

9 Submission

Submit your code using AFS (Andrew File System):
`/afs/qatar.cmu.edu/usr10/mhhammou/www/15440-f17/handin/p2/`*`userid`*`/`,
where *`userid`* is your andrew ID.

10 Late Policy

- If you hand in on time, there is no penalty.
- 0-24 hours late = 25% penalty.
- 24-48 hours late = 50% penalty.
- More than 48 hours late = you lose all the points for this project.

NOTE: You can use your grace-days quota. For details about the quota, please refer to the syllabus.