RaftFS: a Distributed File System

Distributed Enabling Platforms Final Project

Luca Lovagnini
University of Pisa

April 8, 2015
A.Y. 2014/2015

Contents

1 Introduction

The FLP impossibility result [1] proved how in asynchronous system there is no consensus solution that can tolerate one or more crash/stop failures. The CAP theorem [2] main consequence is that a distributed system can guarantee at most two of the followings three properties across a write/read pair: Consistency, Availability, and Partition Tolerance. Figure 1 resumes this result and shows the choice made by the most famous distributed systems.

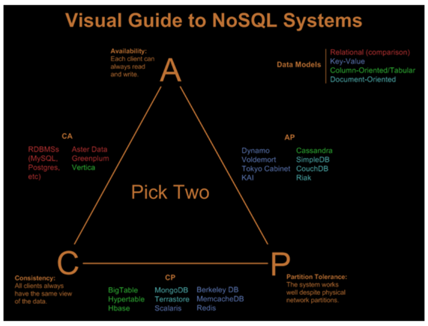

Figure 1: *a visual representation of the CAP theorem consequence*

Thinking a system which choose Availability and Partition Tolerance giving up any form of Consistency has no sense. However, we can introduce different kinds of consistence:

1. Strong consistency model: it guarantees that the distributed system behaviour (apparent order and visibility of updates) is equivalent to a single-machine system.

2. Weak consistency models: the previous guarantees are not true.

Even if weak consistency could seem useless, there are weak models which guarantees consistency with some condition (for example after some time or using data versioning), and they are widely used where some form of inconsistency is acceptable by the user/reader. The most famous system which uses a weak consistency model (and the first to implement it) is Dynamo [3].

However, if our goal is to implement a distributed file system, a weak consistency model may not be the right choice: weak consistency models are generally used where system designers can anticipate conflicts and devise appropriate resolution mechanisms. The author believes that strong consistency is a more suitable solution for a general-purpose application like a file system.

Now we introduce two models of strong consistency (which will be recalled later):

1. Linearizable consistency: in this model all the operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations. In other words, this means that we take *all* the events (both reads and writes) in the system, we order them (supposing that we can use some kind of total ordering) and they appear to be executed following exactly this order. Imprecisely, once a write completes, all later reads (where later is defined by wall-clock start time) should return the value of that write or the value of a later write. Once a read returns a particular value, all later reads should return that value or the value of a later write.

2. Sequential consistency: here all the operations appear to be executed in *some* order with the order that is consistent with the order seen at individual nodes and that is equal at all nodes. In other words, all the events on a particular node are consistent and there exists an order which have sense with the order of the others nodes. In this model there is no notion of real/global/absolute time: it's a weaker model than the linearizable one.

Laslie Lamport's Paxor[4] protocol is considered the first and most famous algorithm to solve the consensus problem guaranteeing strong consistency and partition tolerance. This means that it doesn't guarantee availability, which means that the consensus on a certain value (client operation) could never be reached.

Anyway, Paxos has two issues: it's tremendous complicate to understand and it's not built with the goal of an easy and clear implementation. As consequence, the most famous systems which implements some form of Paxos (such as Chubby[5] or Zookeeper[6]) have modified some aspects of the original algorithm or their implementation have not been published.

The Raft consensus algorithm [7] was defined with the main goal to solve the Paxos's problems described above:

- it guarantees strong consistency (in particular it follows a linearilizability model).
- it's fault tolerant until at least the majority of the nodes are up.
- like Paxos, it doesn't guarantee availability (again, for the CAP theorem).

- the paper where it's described shows hot to "easily" implement it.
- it has been statically showed how it's simpler than Paxos to understand.

RaftFS is a distributed file system which implements the Raft algorithm (you don't say?) in order to guarantee a strong consistency in the file system. The Hadoop Distributed File System (HDFS)[8] was used by the author as reference to design the RaftFS architecture. This report will not explain in details the algorithm and the framework cited above, assuming that the reader already knows them in detail. In section 4.3 the two systems will be compared on some aspects.

RaftFS was developed in Java and it contains roughly 2700 lines of codes without considering blank lines and comments. During its development a certain number of libraries and frameworks were used, such as Apache Maven [9], Apache FtpServer [10], Apache Log4j [11] and Vagrant [12].

The main goal of this project wasn't to build a file system where to uploads and manage big quantities of data (in fact FtpServer is not the best file system framework in terms of performances), but to implement a strong consistent algorithm (possibly different from the classic and complicate Paxos) and try to apply it to some kind of general-purpose application (like a file system indeed). More details about which kinds of files are suitable for RaftFS can be found in section 4.1.

The rest of this report is organised in the following sections: section 2 will illustrate the main implementation choices made by the author, system specifics and protocols implemented in RaftFS. Section 3 is an user manual containing detailed instruction about how to install and use RaftFS in different environments (local, pseudo-distributed and distributed versions, section 3.1) and how to run tests to verify its correctness (section 3.2). In section 4 the reader can find considerations and plots about RaftFS performance. Finally, section 5 consider possible improvements and future works about this system.

2 Implementation and Architecture

This section illustrates how RaftFS works, which aspects of the Raft algorithm were (and were not) implemented, which are the operation that a client can submit to the system and how they are executed, how the source code was organized and finally some technical consideration about the main project classes.

2.1 Raft Implementation

As already said in section 1, RaftFS implements the Raft algorithm to guarantee strong consistency, and in particular to guarantee linearizability.

The biggest lack in the Raft (and Paxos) algorithm is that it doesn't guarantee availability, and RaftFS isn't an exception: a command submitted by a client is immediately rejected by the system if there isn't a leader in that moment. This case isn't the only one where a client command is rejected by the RaftFS: as the reader (hopefully) already knows, a Leader can overwrite the entries in a node's log if they mismatch with the ones of the Leader. In RaftFS this scenario is managed sending a failed request to the client which submitted the overwritten command (entry).

As explained in Raft algorithm, some elements of each server state must be saved on the disk, otherwise a server would loose its entire history every time that it crashes. In order to implement this specific, each RaftFS server creates a directory which contains different files, each one explained in details in section 2.3.1. This directory and its files are created only once (the first time that the server joins the cluster) and they are read every time that the server rejoin the cluster after a crash. Each of these files are updated every time that one of their relative objects are modified at run-time (this may be not good in terms of performance, but it's a necessary passage to guarantee consistency).

One of the most difficult and important points of the RaftFS implementation was how to implements read-only operations without writing them on the log: the original Raft paper treats this argument in few rows of the last section, and it's not totally clear. A special thanks goes to Diego Ontaro (one of the two Raft's creators) to have reported the Raft dissertation [13] to the author of this project where this part is explained in details. There are two conditions in order to perform read-only operations without writing them in to the log:

1. The Leader has to know which index are committed when it assumes this role (in other words, the `commitIndex` parameter value). This process is done in Raft (and RaftFS too) exploiting the

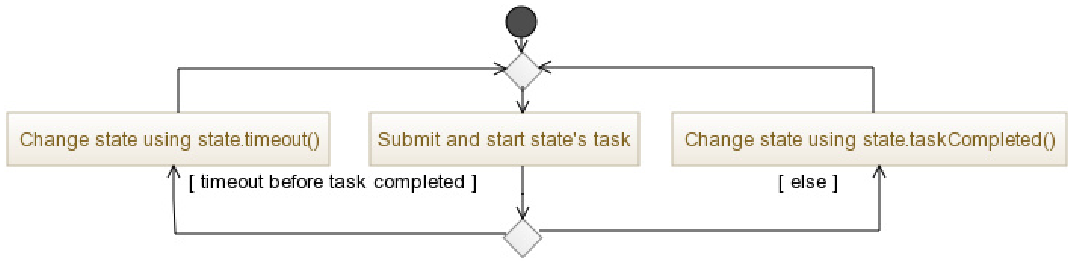
Change state using state.timeout()
Submit and start state's task
Change state using state.taskCompleted()
[timeout before task completed]
[else]

Figure 2: *The Executor's activity diagram*

notion of no-op entry (which is explained in project code and in the dissertation cited above). But why is so important to know the `commitIndex` value in order to process read-only operations? The answer is to ensure linearizability: when a client submit a query, it cannot be processed until the State Machine Applier have not applied all the entries committed before the query was received. Section 2.3.1 explains how this mechanism is implemented in RaftFS.

2. The Leader doesn't belong to a partition where there isn't the majority of the cluster nodes (in other words, that it's not a "fake" Leader). In order to do that, before processing and answering to a query the Leader has to make a so called "heartbeat round" and to be sure to receive an answer from the majority of the nodes of the cluster.

As last part of this subsection, it will be explained how each server state is managed and implemented. We can identify three identities which implements the Raft algorithm in RaftFS:

1. ServerRMI: it keeps the server state and implements its remote methods. This methods can forcefully change the ServerState executed by the Executor. Finally, it initializes the Apache FtpServer.
2. The ServerState: it's an abstract class which methods describes each server's behaviour: the task to execute during that state, the timeout value, what is the next state if the timeout elapses and finally what it the next state if the task is executed successfully (this last part is optional). For example, a Follower's task is simply to sleep until the timeout elapses, the timeout value is roughly 0.5/1.5 seconds (more details in section 4), it doesn't have a next state if it completes its task (it doesn't have one except sleeping!), and when the timeout elapses it becomes a candidate if either it

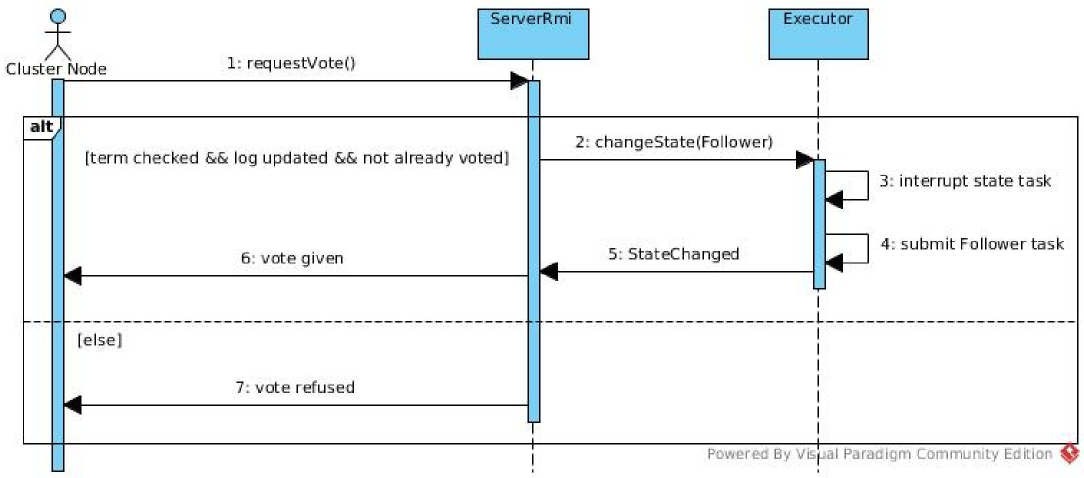

Figure 3: *The interaction between ServerRMI and the Executor during* `requestVote()`

`fsStrcutre`	
/	*directory, someEmptyDir, pad.pdf*
/directory	*image.jpg, anotherDir*
/directory/anotherDir	*book.epub, song.mp3*
/someEmptyDir	

Table 1: *An example of* `fsStructure`

has not received an heartbeat from the leader or voted for someone, otherwise it repeats the process (and so stay as Follower). Notice that this behaviour is a little bit different from the original Raft one: the timeout always elapses, but it doesn't mean that then the server changes its state.

3. The executor: its life-cycle is showed in Figure 2: if nobody forcefully change its state, it executes the ServerState task and starts the countdown. Finally, it changes the server state depending on if the timeout elapses or not. Notice that the next state could be the same as before.

An example of how these three entities interact between them is showed in Figure 3 (and it shows how the executor state can be forcefully changed).

2.2 File System implementation and Client Interaction

2.2.1 File System Implementation

RaftFS is a hierarchical file system. The domain of the legals file name is described in section 2.3.2. Each file system element can (eventually) be represented in two different ways: a logical representation and a physical one. Each file is replicated depending on the `replicationFactor` described in section 3, which defines on how many servers each file is replicated at most (by default each file is replicated on 2 machines). This doesn't mean that each file is replicated *exactly* on this number of servers, as explained in the next section. Each file system element is univocally identified by its absolute path (in both representations). The logical representation is used for both files and directories, while the physical one refers only to files.

The logical representation of the elements is realized through a map object called `fsStructure`: the key of each map element is the absolute path of a directory in the file system, while the associated value is the list of elements contained inside it. An example of this structure is showed in Table 1

In addiction, each file is contained inside another map object called `fileLocation`: each key is the absolute path of the file uploaded on the file system, while the associated value is a pair *(fileSize,servers)* where *servers* is a list of cluster nodes where the file is replicated. Table 2 is an instance of `fileLocation` (obviously the *fileSize* values are fictitious).

Combining `fsStructure` and `fileLocation` stored on each node of the cluster, each node in RaftFS knows both the file system structure and where each file is physically stored in the cluster (and the consensus algorithm guarantees that they are consistent).

During a cluster node setup, a directory `SererFS/*serverIdentifier*` is created. This directory represents the physical representation of each file stored on that particular node: regardless of their logical location, they are all physically stored inside the same directory. So how can we distinguish two files stored on different logical levels but with the same name if their physical representation are stored in the same directory? Each file name is equal to its relative absolute path in the logical representation. Unfortunately, we cannot save a file which name contains the "/" char, but it's sufficient to substitute it

`fileLocation`	
/pad.pdf	*(10, {Server0, Server3})*
/directory/image.jpg	*(93,{Server0, Server1})*
/directory/anotherDir/book.epub	*(63,{Server1,Server2,Server3})*
/directory/anotherDir/song.mp3	*(130,{Server2})*

Table 2: *A possible representation of* `fsStructure` *related to Table 1*

Physical file representation	
Server0	*@pad.pdf, @directory@image.jpg*
Server1	*@directory@image.jpg, @directory@anotherDir@book.epub*
Server2	*@directory@anotherDir@book.epub, @directory@anotherDir@song.mp3*
Server 3	*@pad.pdf, @directory@anotherDir@book.epub*

Table 3: *The physical representation of the files reported in Table 2*

with a special char not allowed in RaftFS (like "@"). Table 3 shows the content of each physical directory on each server of the cluster (with reference of the previous examples).

2.2.2 Client API and interaction

As already said in the introduction of this report, RaftFS uses Apache FtpServer in order to upload, remove, rename and download files from the client to the servers (and viceversa). These are the possible operations that a client can submit to RaftFS:

- put: upload a file on the cluster. The client can specify on how many servers this file can *eventually* be replicated (more details below).
- get: download a file on the cluster and save on the client machine.
- mkdir: create a directory on the file system. As the reader can imagine, this is only a logical operation, since no physical file is involved.
- ls: list all the elements contained in a particular directory.
- rm: remove a file or an empty directory.
- replicate: this is a special command, which utility will be more clear below. Anyway, it *eventually* replicates a file already uploaded on some servers of the cluster on other servers where the file is not stored yet.

As already said in the previous sections, the read-only operations are not written on the log (and so not replicated), so the ls and get operations are solved by the Leader without involving the other nodes.

Each file is both logical and physical represented. But how does RaftFS guarantee that these two representation are consistent? The next topic that is about how this problem is solved, and the description of each operation.

put: an example of the put operation process is showed in 4. These are the main steps:

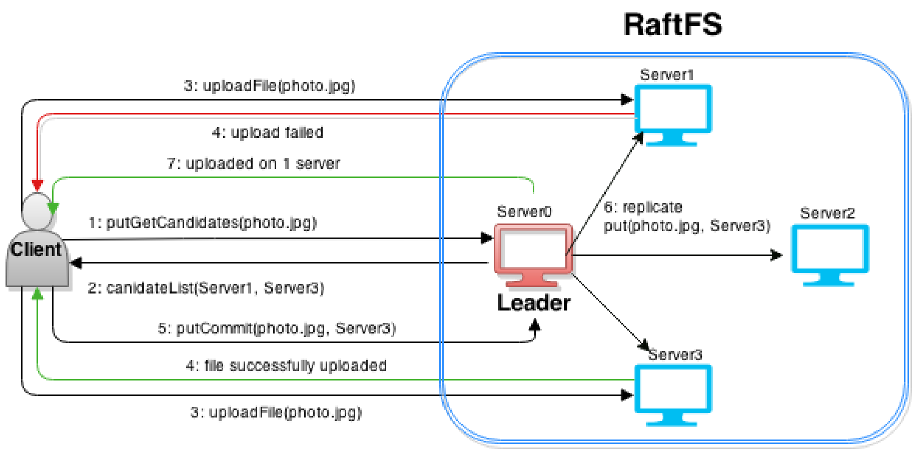

Figure 4: *Put operation step by step*

1. As first thing, the client needs a list of servers where to upload the file. So, it submits to the Leader a special read operation called `putGetCandidates` (which is not callable in other contest by the client) : if the file doesn't already exist in the cluster, then this operation returns a set of servers which are reachable and with enough free disk, otherwise an error message is returned. The size of this set is at most the minimum between the system replication factor and the one defined by the client put operation. But if there are a lot of feasible server, which ones are chosen? RaftFS implements a very simple load balancer to solve this problem: the top-k servers with most free disk space are chosen (where k is the minimum defined above). Notice that a server *must* be reachable because otherwise it could always be chosen as one of the candidates, even if any file will never be uploaded there!

2. Once the client knows where to upload the files, it uses the Apache FTPServer to physically upload the file on each candidate. The process is done in parallel and it can take at most 5 minutes to be completed, otherwise if the client doesn't completely remove the file on a candidate, it forcefully remove it. During the upload process, the file is temporally renamed with its original name followed by ".temp" (the reason is explained in the paragraph below which talks about the Garbage Collector).

3. Once that the physical upload process is completed, the client submit another special operation called `putCommit`: it's a write operation which creates the logic representation of the file (it creates the new entry in `fileLocation` relative to the file uploaded and updates `fsStructure` adding the file uploaded in the parent's directory entry).

4. Once that the entry relative to `putCommit` has been successfully replicated and executed, a success (or failed, as explained in the example below) message is sent back to the client.

Now, let's consider four particular cases:

1. A client successfully get a set of candidates, successfully upload the file on each of them, but then it crashes without committing the operation to the Leader.

2. A client dies during the upload operation.

3. A lot of clients submit a put operation at the same time, they get the same set of candidates (since the free disk space is updated only with the commit operation), they upload successfully the files on each candidate, but since the logical space of a server could be smaller than its physical one some of their commit operation fails for some server because there isn't enough logical space.

4. What happens if the entry relative to a `putCommit` operation is overwritten by a newer Leader?

The consequence of the examples above is that it could be possible that a file (or its temp version) is physically stored on a server but not logical stored, and so it is totally useless.

This problem is solved in RaftFS with the introduction of a Garbage Collector. It checks periodically (7 minutes by default) the set of files physically stored on a server. If there exists a file which isn't logically stored, it inserts this file on a so called "black list". If at the next Garbage Collector run the file is still not logically stored, it is physically deleted from the server machine. Notice that a partially uploaded file and the totally uploaded one are considered two different files by the Garbage Collector (because of the temp name explained in the protocol description above). The black list mechanism is necessary because otherwise a file could have been deleted wrongly (look at the `GarbageCollector` class description in the source code to see two examples).

The problem discussed above is the result of an "optimistic" estimation of the free disk space on each server. Notice that a "pessimistic" solution is not the right choice: logically store a file which isn't phisically stored, could lead to inconsistent situations (think to a get operation of a file which isn't phisically stored yet).

rm: the protocol of the remove operation is the inverse of the put one: in order to guarantee consistency, the file is first removed logically from RaftFS, and then the client physically removes it from each node where the file is stored. As before, if the client dies before physically remove the file, the garbage collector will do the work for him.

get: the read operation is quite simple: the client asks to the Leader the list of servers where the file is replicated. Then it tries sequentially to download it from each candidate until the operation is successful or the download fails from each candidate. A load balancing mechanism could be introduced in order to download the file from a particular server instead of another (for example the one which is using less its bandwidth), but such mechanism isn't implemented on RaftFS.

replicate: as explained in the put section, it could happen that a file is replicated in fewer server than the replication factor indicated by the client (which has sense if it was smaller than the `replicationFactor` of RaftFS). If a client isn't satisfied with the number of servers where the file was replicated, he can try to forcefully replicate it (again, under the `replicationFactor` constraint). The process is divided in four phases:

1. the client gets the list of servers where the file is replicated, exactly like the get operation. We'll refer to them as starting servers.
2. then he gets a set of possible candidates like the put operation, with the only exception that here the file already exists in the system and so the returned candidates must not contain the file already.
3. then each client tries to physically replicate the file from each starting server to each candidate, until the operation succeed or all the starting servers are down. The replicate operation is done exploiting dedicated ServerFTP methods.
4. finally the client submits a `putCommit` operation, considering that in this case the only operation done is to (eventually) add the new servers to the relative entry in `fileLocation`.

Notice that each consideration done for the Garbage Collector in the put operation is valid for the replicate one too.

mkdir: it's the most simple write operation: when the relative entry is successfully committed, a new entry is added to `fsStructure` (obviously if the new directory is legal).

ls: the list operation is trivial: a get operation is done on `fsStructure`.

2.3 Project Structure and Classes

This section describes the most important classes in RaftFS. The source code is already well documented and commented, so the reader is invited to read it for any detail. Some class was already been described (like the `Executor` or `Follower` classes, which have been already described in the previous sections), others will be described in the next sections (like `MainLocal`, `MainNetwork` and `MainClient`) and others are not so relevant to be described here.

2.3.1 ServerRMI

This is the most complicate and biggest class in RaftFS. In RaftFS each communication between servers or between server and client are implemented through Java RMI. The name of this class derives from this feature: it implements remote methods defined in the interface `ServerInterface`, which are `clientRequest` and the well known Raft methods `appendEntries` and `requestVote`. The two last methods are already well explained in the Raft algorithm, and the RaftFS code is full of documentation and comments about them, so they will be not treated here, while `clientRequest` is explained at the end of this section.

As already explained in section 2.1, the other ServerRMI task is to keep the server state. In particular, each persistent field is saved in a different file, which are:

- metadata: this file stores the server current term, the last applied entry to the state machine, and the identifier of the server which this server voted for. Since they are three simple objects (two int and one string), they are stored in the same file.
- log: it contains the entire log and its entries.
- cluster: as it will be explained later, each server has a reference to any other server which belongs to the cluster, and in particular its available disk space.
- fileLocation: since each servers has to know where the file is replicated, this crucial map object is written on disk.

- fsStructure: each time that a new directory is created or an element is added to a directory, this file has to be written on disk.
- scheduledOperation: as explained in section 2.3.3 this blocking queue contains all the committed entry which have to be applied to the state machine. This structure has to be written on the disk too, otherwise it could happen that a committed (but not applied) entry could be lost if the server crashes (and so lead to an inconsistent state).

Notice that the last three files (and relative fields) are managed by `MachineApplier` (section 2.3.3).

The Raft algorithm explains in details how to implements log compaction (or log snaphost) in order to easily manage the log size (in other words how to save log entries on disk and discard the entire log up to that point), but this isn't a feature implemented in RaftFS and it's considered a possible future improvement.

The list `cluster` is used to have a reference for each node in the cluster. Each cluster element contains fields to contact the relative node (IP address, node identifier and FTP port) and its free disk space. This informations are used by both the Raft algorithm and File System operations.

The Raft algorithm provides a mechanism to safely change the cluster membership, and so implements elasticity. Unfortunately, this feature is *really* complicate to implement, and it's considered a possible future work for RaftFS.

Finally, we explain how the `clientRequest` remote method works: as first thing, if this node is not the Leader, then the Leader reference are returned (identifier and IP address). Otherwise, if the `FSCommand` received is a write operation, then it is appended to the leader log. Otherwise, as already explained in section 2.1, two conditions have to be satisfied before the query can be processed: an entry from this term must be committed and an heartbeat round must be executed. If the first condition is not satisfied, then the RMI thread which is executing `clientRequest` waits that an entry is committed (which will be for certain at least from this term). The Leader task notifies all the threads which are waiting that an actual entry is committed (they are notified when the Leader steps down too). Then, a local variable `readIndex` is defined with the `commitIndex` value (it will be used later). As next step, the client starts an heartbeat round: it sends a special `appendEntry` where only the current term is checked and then a positive or negative answer is returned. If the majority of the nodes have answered successfully, then the thread will wait until the state machine applier will not have performed the write operation with position `commitIndex`. Finally, the read operation can be executed. Notice that if `commitIndex` is updated many times before `readIndex` is defined or others write operations are applied before the query is processed is not a problem, because linearilizability asks that the read operation should return the value of that write *or the value of a later write.*

2.3.2 FSCommand

This abstract class defines common aspect between all the file system commands. Each command defines if it is a write or read operation. A `ClientInterface` field is used as stub to callback the client once the operation is completed and finally it has an absolute path which represents the file system element where the command have to be executed. Each command path has to be an absolute path which cannot contain any special character, white spaces, and only the last token can contain a dot (for the file extension). This is the Java regular expression which describes the legals command path: $(/\backslash w+)*/\backslash w+(\backslash.\backslash w+)?|/$

In addiction, each file system command has to implement the abstract methods to execute a command and also to generate an answer to the client.

Subtyping polymorphism is used in order to treat each file system command in the same way.

2.3.3 MachineApplier

This class apply each committed entry to the state machine (and so execute each write command) and notifies the waiting readers when the command have been executed. In addiction, it keeps the logical representation of the entire file system: in fact `fsStrcuture` and `fileLocation` are created and managed in this class. Notice that `fsStructure` is initialized with the empty root directory "/".

Since these two fields are persistent (and so they have to be written on disk each time that they are updated), they can be accessed and modified only through apposite methods defined in this class. In this way, they cannot be modified without updating the relative file (and so RaftFS can be safely expanded with new operations).

There is another persistent field in this class: `scheduledOperation`, which is a blocking queue. When an entry is committed, it is appended to this queue. The `MachineApplier` class is a thread which wait for at least an element in `scheduledOperation`, execute it, and finally update the last applied field kept by `ServerRMI`. This life cycle is repeated until the server doesn't crash. Since `scheduledOperation` is a persistent field, even if the server crashes no operation to apply will be lost, and it will be executed as soon as the server will rejoin the cluster.

In order to guarantee that the read and write operations don't create inconsistent states given from concurrent execution, a `ReadWriteLock` object is used to guarantee thread safeness.

2.3.4 SingleClientRequest

In section 2.2.2 was described each command protocol. But how they are implemented? We can define a some kind of pattern about how this process is done:

1. The client chose which command to execute. It creates a `SingleClientRequest` object telling which command to execute. Since `SingleClientRequest` extends `Thread`, the client calls `start()` and the next step is executed. This means that in RaftFS each client request can be executed in parallel.

2. The `SingleClientRequest` task is to call `clientRequest` of ServerRMI. Since the Raft algorithm doesn't guarantee availability, we cannot know if the submitted command will be ever executed. As result, the client will receive an answer from the server as a callback. Once this process is started, there is no way for the client to stop it (unless he brutally kill the RMI registry of its machine).

3. If the contacted server wasn't the Leader, the client will receive a callback with the method `requestFailed` and he will submit the command again.

4. If for some reason the command fails, again the `requestFailed` callback method is invoked, the failure message is printed and the operation can be considered completed.

5. If the command is successfully executed, then the suitable `SingleClientRequest` callback is called while the abstract method `callAnswer()` in `FSCommand` is executed. For instance, when a `PutGetCandidates` command have been executed during a put operation, the callback method `putGetCandidatesResult()` is invoked in the `callAnswer()` implementation of `putGetCandidate`.

6. The callback called as answer from the server to the client execute different operations based on the command that it implements. For instance, `putGetCandidateResult()` implements the point 2 and 3 in the put protocol (it physically uploads the file on each candidate and then submit the write command `putCommit` to the server).

7. This process is repeated depending on the command itself.

This callbacks mechanism implemented on the client-side points out how a client cannot predict any kind of timeout and how an operation cannot be stopped once it is started. Anyway, since this behaviour doesn't create any kind of inconsistency, an eventually timeout mechanism on the client operations is considered a future work.

2.3.5 Candidate and Leader

The general behaviour of each role was already described in section 2.1 and in the Raft paper. The only important thing to say is that unlike the Raft algorithm, the `requestVote` and `appendEntries` calls are made sequentially and not in parallel. An initial version prooved how a parallel implementation would have added an overhead caused by the multi-threading management which would have decreased the system performance. However a better solution could be certainly found, but it's considered a future work.

There is another important thing to say: since RaftFS exploit the Java RMI, each server needs a reference to each server's RMI stub that it wants to reach in order to invoke its remote methods. An initial

RaftFS implementation provided to each `ClusterElement` a stub field too in order to contact any other server. Anyway, keep this field updated proved to be too much complicated, and so at each `appendEntries` or `requestVote` calling the server performes the `getRegistry()` and `lookup()` operations on each server. While the first is not so expensive, the `lookup()` method can take a lot of time. This method guarantee a totally reliable RMI system, but it's one of the main performance issue in RaftFS. A future version of this system could provide a better mechanism for sure.

3 User Manual and Tests

This sections show step-by step how to build a RaftFS cluster and how to interact with it. There are three different versions: local, pseudo-distributed and distributed. The first one was implemented for debugging and for testing, the second one to simulate RaftFS in a non-local host environment, the last one is the "real" RaftFS version.

The last two versions heavily use the Vagrant framework to build and manage virtual machines: with this tool we can use RaftFS without worrying about the host machine specifics (its operating system, its Java Virtual Machine version etc.). In conclusion, the system requirements are:

1. Local version: a system where is installed at least Java 7 and Maven.
2. Pseudo and distributed version: a system where Vagrant and Virtual box are installed, an internet connection.

Notice that you can always build a virtual machine used for the pseudo or distributed version and then run the local version!

Note: as first obvious operation, you'll have to create a directory and extract the content of *RaftFS.tar.gz* inside it. From now on this directory will be considered the starting point of each following manual.

3.1 User Manual

3.1.1 Local version

Project building and compiling: before you can run RaftFS in its local version, you need to build it with Maven. The following steps are made supposing that you are using a Linux system (again, use a RaftFS virtual machine if you doesn't have a Linux system!).

1. `mvn -f RaftFS/pom.xml package` to build the RaftFS project. If the process succeed, the directory *RaftFS/target* should have been created containing the jar file *RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar.*
2. Before the user runs RaftFS, be sure that the log4j file `log4j.properties` is in the same location where you lunch the project. The logger was used for testing purpose, but the author (as described in the project comments) left this feature for completeness.
3. `java -cp RaftFS/target/RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar PAD.RaftFS.Server.MainLocal 3` to run a local version of RaftFS with three servers. Each server identifier is `ServerI` where `I` goes from 0 to the value of the first parameter minus 1 (so in the case above `Server0,Server1,Server2` are created). Executing the command above without any parameter will print the local version usage.
4. Now that a local version of RaftFS is running, for each node a directory should have been created (containing the persistent fields files) and a directory *ServerFS* containing a directory for each server: it will contains each file uploaded on that server. Now, the user can interact with the system with a series of commands:
   - `timeout ServerName`: the state of `ServerName` will be forcefully changed in Candidate and so starting a new election with a new term.
   - `stop ServerName`: the server `ServerName` crashes (the executor is interrupted and the server not reachable by RMI).

- `resume ServerName`: reinitialize `ServerName` as a Follower. This operation is valid only if `stop ServerName` was previously executed.
- `lossOutputprobability ServerName OtherServer Probability`: this command is used to simulate the loss of packets from `ServerName` to `OtherServer` with probability `Probability` (which value must between 0 and 1).
- `lossInputProbability ServerName OtherServer Probability`: same as the command above but this command define the probability to loose the packets from `OtherServer` to `ServerName`.
- `createPartition ServersList`: it creates a network partition where each server in `ServersList` belongs to it. For instance `createPartition Server1 Server2` will totally isolate the two servers from the rest of the network (but they can still exchange message between them).
- `healPartition ServersList`: after being part of a network partition, each server in `ServersList` now can talk again with the rest of the network.
- `ls ServerName CommandPath`: ask to `ServerName` to return the files contained in `CommandPath`
- `get ServerName CommandPath RelativePath`: ask to `ServerName` to download the file `CommandPath` in the path (specifying the file name too) `RelativePath`.
- `rm ServerName CommandPath`: ask to `ServerName` to remove the file or empty directory `CommandPath`
- `mkdir ServerName CommandPath`: ask to `ServerName` to create the directory `CommandPath`
- `replicate ServerName CommandPath Quantity`: ask to `ServerName` to replicate the file `CommandPath` on at most other `Quantity` servers.
- `put ServerName CommandPath LocalFile Quantity`: ask to `ServerName` to upload the local file `LocalFile` on the file system with absolute path `CommandPath` on at most `Quantity` servers.
- `parallelClient Value`: as explained in section 2.3.4, each client request is made in parallel. With this command, if `Value` isn't equal to "true", then each client request is made sequentially. This command will be used in a test in section 3.2.
- `sleep Time`: make the main thread sleep for `Time` milliseconds. This command can be used if we want to submit two different commands with an interval of time.
- `logPrint ServerName`: print `ServerName` log.

The user can define a chain of commands to execute multiple commands in sequence. For example `timeout Server0 timeout Server1` will create a Leader election between two servers! This is helpful to test the system. Notice that if the command format is wrong it could be impossible to continue to interact with the system (the only solution is to stop the program and restart it).

3.1.2 Pseudo and Distributed version

Both Pseudo and Distributed versions are based on using a set of Vagrant virtual machines where RaftFS can be executed. Each RaftFS virtual machine is provisioned with the appropriate set of tools to run RaftFS, in particular Java Development Kit version 7 and Maven are installed on each machine. The folder synchronization mechanism given by Vagrant is disabled on each machine, except for the *ServerFS/ServerName* folder (so the user can see the uploaded file on the virtual machine on the host machine too) and *RaftFS* (so the virtual machine can access to the source code for any reason). Notice that the folder containing the server state files is not synchronized with the host machine (if for some reason you want to save it on the host machine you need to manually copy it on the host machine). When the machine is correctly provisioned then each machine compiles the RaftFS project using Maven and then cut the jar file `RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar` in */vagrant/* (which is not synchronized).

But how can we build a cluster of RaftFS virtual machines for the pseudo distributed version, or to build a single virtual machine for the distributed one? RaftFS uses a simple configuration file written in Yaml to define a cluster of RaftFS virtual machines. This Yaml file is equal for both pseudo and distributed version and is used both by Vagrant (to build a set of RaftFS virtual machines) and by each RaftFS server before join the rest of the cluster. When you run a RaftFS server the file `servers.yaml` has to be in the same location where you execute the jar file, but don't worry: this file is copied in

/vagrant/ when the server virtual machine is built (which is the default location to run each RaftFS server in pseudo or distributed version). An example of this file can be found in the directory *RaftFS*, and this this is a general pattern to correctly write it:

- A Yaml file is a list of pairs *(key,value)*. In `servers.yaml` the first key *can* be RaftArgs: this key isn't used by Vagrant to build a virtual machine, but it is used by RaftFS to set the project parameters at run-time. If this key (and its values) is not defined, the default values are used. These parameters are:
  - `rmiTimeout`: an upper bound to execute a remote method of a server. This parameter is crucial because it defines for how long a server waits an answer from a `requestVote` or `appendEntries` invocation, or a client to wait an answer from a `clientRequest` invocation. *Notice that this parameter doesn't stop the remote method execution, but simple doesn't wait for its answer!.* Default value: 100 ms.
  - `replicationFactor`: the system replication factor upper bound. Its default value is 2, which means that each file can be replicated at most on 2 servers.
  - `lowerBoundElectionTimeout`: as explained in the Raft paper, the election timeout has a deterministic component (which works as a lower bound) and a random one which is added to the deterministic one. This mechanism is used to reduce conflicts between candidates during the Leader election. The default value of this lower bound is 500 ms.
  - `randomElectionTimeout`: the random part explained in the previous parameter. Random value: 1000 ms.
  - `debugLvl`: this parameter defines how much the system is verbose. The possible values are `silent,normal,debug` and `crazy`. Default value: `normal`.
- The rest of the Yaml defines each RaftFS virtual machine that can be built by Vagrant. Each key defines the server identifier, and its parameter are:
  - box: the Vagrant box used to build the virtual machine.
  - ram: the size in MB of each virtual machine's memory.
  - ip: this is an important parameter: since both pseudo and distributed versions use public networks, if we assign a static IP to a virtual machine using Vagrant, it could happen that it's not feasible (for example the router already allocated it). The possible result is that a virtual machine is unreachable using static IP. A possible solution is to use DHCP to assign a safe IP to each virtual machine: this feature can be used leaving this parameter empty (as in the example file `servers.yaml`). Otherwise, you can try to use a static IP filling this field.
  - ftpPort: the port where the Apache FtpServer will listen to.
  - diskSpace: node's logical space in byte.

Now that we have written our cluster configuration file, we are ready to build a set or a single RaftFS virtual machine (respectively for the pseudo and distributed versions). *The rest of this manual suppose that you are using the default `servers.yaml` in RaftFS.*

The classic command `vagrant up` doesn't work in RaftFS. In fact we have to specify which virtual machine we want to build. You can do that in three ways:

- `MACHINE=ServerName`: where `ServerName` is a key contained in `servers.yaml`. This command is used to refer only one server virtual machine, and it's usually executed for the distributed version. For example `MACHINE=hal9000 vagrant up` will create only the RaftFS server machine called `hal9000` in `servers.yaml`. If we want to destroy it, execute `MACHINE=hal9000 vagrant destroy` and so on and so forth.

- `MACHINE=client`: this is a special virtual machine. Unlike the servers machine, the client machine is synchronized with the project root table, so the client can use the `dns.yaml` file (explained below) and access to the *DataSet* directory to uploads file on the cluster. Execute `MACHINE=client vagrant up` in the distributed version if you want to build the client machine.

- `MACHINE=all`: this is the default command used for the pseudo distributed version. It takes all the virtual machines defined in `servers.yaml` and the `client` virtual machine and apply the next Vagrant command to each one of them. For example, executing `MACHINE=all vagrant up` will build three RaftFS servers (`jarvis`,`hal9000` and `jarvis`) and the client machine `client`.

Important note: since Vagrant need a reference to a network interface in order to implement bridging for a public network, if the host machine has more than one network interface, you'll have to specify which one to use (usually in a Linux system it's called `wlan0`).

Now that we have built our cluster, we have to run each server machine:

1. To access to a specific machine called `ServerName` execute `MACHINE=ServerName vagrant ssh`.
2. Execute `cd /vagrant` to reach the directory where RaftFS can be executed. If you execute the `ls` command these should be the elements returned:

   `log4j.properties RaftFS RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar ServersFS servers.yaml`

   . There is only one problem: if we have built some machine using the DHCP feature, we have not updated the `servers.yaml` with the correct IP addresses (check it with `cat servers.yaml` and see that the `ip:` field is empty).
3. To know the ip address of the guest machine: `ifconfig`.
4. Test that the virtual machine is reachable: execute the `ping` command from the host machine (or another one if you're testing the distributed version) using the IP address returned by `ifconfig`. If the ping command fails, then try to reload the machine with `MACHINE=ServerName vagrant reload`: it should get a new IP address for the virtual machine. Otherwise as last resort try with a static IP address.
5. Update `servers.yaml` in *RaftFS* inserting each virtual machine IP address.
6. Opern `dns.yaml` on the root project folder on the host machine: it should contain the name of each server contained in `servers.yaml`, but with the IP address empty. Fill it with the value that you used in `servers.yaml` too. This file will be used later.
7. On the guest machine execute `sudo cp RaftFS/servers.yaml .` to overwrite the old version with the new one (check it with `cat servers.yaml`.
8. `sudo java -cp RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar PAD.RaftFS.Server.MainNetwork`

   to run the server!
9. Now you can interact with it using the `timeout,lossInputProbability,lossOutputProbability` and `logPrint` exactly like the local version.
10. To reset the server state execute (after having stopped it) `sudo rm -rf ServerName ServerFS`

The cluster now is running! The last thing to do is running RaftFS on a client virtual machine:

1. `MACHINE=client vagrant ssh` to access to client machine.
2. `cd /vagrant` to reach the directory where RaftFS can be executed. If you execute the `ls` the element returned should be the same of the folder where you extracted the RaftFS project.
3. Now check that `dns.yaml` is updated with the values used during the cluster setup. Since each client has to know the name of each server and its IP address, this file is used by the client at run-time as reference to each cluster node (or a subset of them).
4. Find out the client's IP address with `ifconfig`

5. `sudo java -cp RaftFS/target/RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar PAD.RaftFS.Client.MainClient IPaddress`

   where `IPaddress` is the IP obtained from the previous step. Notice that this parameter is optional, but you'll have to specify it at run-time anyway.

Now you can interact with the cluster from the client with the same client commands that you used in the local version (`put,get,rm,ls,replicate` and `mkdir`.

Important note: the command `help` print the client usage. Notice that in this version the default replication factor value for `put` and `replicate` is 2. So you can execute for example `put jarvis /file.jpg DataSet/cloud_atlas.jpg` and the file will be uploaded on at most two servers (w.r.t the system `replicationFactor`)

3.2 Tests

This section describe some test to verify the RaftFS consistency and correctness. The local version was implemented for debug purpose, but also to simulate situations which could be difficult to recreate in the distributed version.

Important note: between one test and another be sure to execute `rm -rf Server*` in order to reinitialize the cluster's state every time.

3.2.1 Overwritten entry test

Test descrption: This test shows a complete case in appendEntries execution. In particular, how an entry can be overwritten by a newer Leader. Moreover, in this test is showed the "back rolling" process made by a Leader during the replication phase.
Network size: at least 3 nodes
Test step-by-step:

1. Node N becomes Leader.
2. N receives two client requests.
3. N replicates only the first entry, while it crashes before the second one is replciated.
4. Node M becomes a leader.
5. M riceives at least three client requests. Now, M's log has size 4 (since only the first entry received by N was replicated).
6. Before that M replicates any entry, it makes elapses its timeout, becoming leader again.
7. Now the user can see the "back-rolling" process made by M during the replication phase, in other words how `nextIndex` (which value is 3 if three requests are received) decrements until it becomes 0, and how at every appendEntries a new entry is appended to each node which is up.
8. N resumes and the user can see how its second (not replicate) entry is overwritten.

Test instructions:

1. Supposing that Server0 becomes Leader (execute timeout Server0 otherwise)
2. parallelClient false mkdir Server0 /dir
3. lossOutputProbability Server0 1 mkdir Server0 /dir1 stop Server0
4. Supposing that Server1 becomes Leader (timeout Server1 otherwise)
5. lossOutputProbability Server1 1 mkdir Server1 /dir2 mkdir Server1 /dir3 mkdir Server1 /dir4 timeout Server1 lossOutputProbability Server1 0
6. ls Server1 / should return: `List result: dir1 dir2 dir3 dir4` (dir1 doesn't exists because Server0 didn't replicate it).
7. resume Server0

3.2.2 Total crash and committing test

This test shows the replication mechanism. In particular how not committed entries aren't lost even if the entire cluster crashes, and how they are replicated as soon as the system goes up again. Initially this test was designed to show also an entry with a term smaller than the actual one cannot be committed. But since the read-only mechanism was implemented in RaftFS, the no-op entry ensure that as soon as a Leader is elected the no-op entry is committed (and so committing an actual entry).
Network size: 2
Test step-by-step:

1. Node N becomes Leader
2. Node M crashes
3. N receives a client request which entry is E. E cannot be committed since it's not replicated on the majority of the cluster nodes.
4. N crashes. Now the system is totally down.
5. N and M resume.
6. N becomes Leader (M cannot become the Leader since it doesn't have the log updated!)
7. N replicates E on M as soon as the no-op entry is committed.

Test instructions:

1. Supposing that Server0 becomes Leader (timeout Server0 otherwise)
2. stop Server1
3. mkdir Server0 /dir
4. stop Server0
5. resume Server0 resume Server1

3.2.3 Leader election test

In this test the user can see how a node which hasn't an updated log cannot become the Leader, even if it's only candidate for that term.
Network size: at least 3 nodes
Test step-by-step:

1. Node N crashes (it doesn't matter if either it's the Leader or not).
2. Node M becomes Leader
3. M receives a client request which entry is E. E is replicated among the other nodes.
4. Node N goes up again and becomes a candidate with the highest term.
5. Nobody give its vote to N, but everyone becomes a Follower again (even M) since they have found someone with an higher term.
6. Node Z (!=N) becomes Leader and replicates E on N.

Test instructions:

1. stop Server0
2. Supposing that Server1 becomes Leader (timeout Server1 otherwise)
3. mkdir Server1 /dir
4. lossInputProbability Server0 1 lossOutputProbability Server0 1 resume Server0 timeout Server0 timeout Server0 lossInputProbability Server0 0 lossOutputProbability Server0 0.

3.2.4 First read-only condition test

Test to check the first read-only condition: a query cannot be processed until the Leader didn't commit an actual entry. In order to to perform this test it is needed a network big enough so that the no-op committing is so long that the read operation thread goes to sleep. Generally, 25 nodes should be enough.
Network size: ˜25 nodes.
Test step-by-step:

1. Node N becomes Leader
2. Node N makes it timeout elapses. After some moment N receives a read command.
3. Since the no-op entry was not committed yet, the read operation thread goes to sleep.
4. When the no-op entry is committed, the read operation thread wakes up and the query is executed.

Test instructions:

1. Supposing that Server0 becomes Leader (timeout Server0 otherwise)
2. timeout Server0 ls Server0 /

3.2.5 Second read-only condition test

This test check the second read-only operation: the Leader must receives at least the majority of the answers from an heartbeat round. This test shows how a read operation fails for this condition.
Network size: at least 3 nodes.
Test step-by-step:

1. Node N becomes Leader
2. N becomes part of a network partition where the majority of the nodes DOESN'T belong to it (for example a network partition which is composed only by N).
3. N receive a client request, but even if the first condition is verified, the second isn't since the Leader cannot reach the majority of the nodes.
4. A failure message is sent to the client

Test instructions:

1. Supposing that Server0 becomes Leader (timeout Server0 otherwise)
2. createPartition Server0 ls Server0 /

3.2.6 File System operations test

This test shows each File System operation (except the replicate command) and show their correctness.
Network size: at least 2 nodes.
Test instructions:

1. mkdir Server0 /dir
2. put Server0 /file.jpg DataSet/cloud_atlas.jpg 2
3. ls Server0 /
4. put Server0 /file/file anything 1 (errore: file into a file)
5. put Server0 /dir/file1.jpg DataSet/cloud_atlas1.jpg 2
6. ls Server0 /dir
7. rm Server0 /dir (errore: directory not empty)

8. rm Server0 /dir/file2 (error: file doesn't exist)
9. rm Server0 /dir/file1.jpg
10. ls Server0 /dir
11. rm Server0 /dir
12. ls Server0 /dir (errore: directory doesn't exist)
13. ls Server0 /file.jpg (errore: file listing)
14. put Server0 /file.jpg DataSet/cloud_atlas1.jpg 2 (errore: adding a file with the same path)
15. mkdir Server0 /file.jpg (errore: like 13)

3.2.7 Multiple put and Garbage Collector test

In this last test it is showed what could happen if two files are uploaded at the same time on RaftFS. In particular, how a file can be physically uploaded on a server but then not logically stored. The configuration for this test is done based on the file `cloud_atlas.jpg` in the *DataSet* folder. The test instructions include the command to run RaftFS with particular arguments for this test.
Network size: 3
Test step-by-step:

1. A client will put the file `cloud_atlas.jpg` on the cluster. Two servers will store it, and so (since the particular cluster configuration) they will have half of their disk space available.
2. Two clients uploads the same file at the same time: the first put phase consists in the execution of `putGetCandidates`, where the servers with more free space are chosen. The load balancer will return the same two servers for both `putGetCandidates` operations (because they are executed at the same time since they are read only-operations).
3. The clients upload at the same time the file on the two servers.
4. The clients invoke `putCommit` at the same time. Since it's a write operation, the two commands are executed sequentially (and not in parallel).
5. The first `putCommit` will be successful: now two servers will have half of their spaces free, while the third one will be full.
6. The second `putCommit` will be partially successful: one of the candidates hasn't free space, while the other one will have half of its space available. So, the file will be logically stored only on one server.
7. The Garbage Collector of the node where the file wasn't logically stored will add it to the black list.
8. At the next run, the Garbage Collector will delete it. If you want, you can stop this process with a replication command.

Test instruction:

1. Run a local version of RaftFS with the following command:
   ```
   java -cp RaftFS/target/RaftFS-1.0-SNAPSHOT-jar-with-dependencies.jar 3 normal 2
   1101198
   ```
   (which means 3 servers, normal output, replication factor 2 and 1101198 byte of logical disk space per node).
2. put Server0 /file.jpg DataSet/cloud_atlas.jpg 2
3. put Server0 /file1.jpg DataSet/cloud_atlas.jpg 2 put Server0 /file2.jpg DataSet/cloud_atlas.jpg 2
4. (optional) replicate Server0 /file2.jpg

4 Performance

In this section it will be explained which kinds of files are suitable for this distributed file system, some performance evaluation in RaftFS, and finally a comparison between a well-known distributed file system like the HDFS and RaftFS.

4.1 Which kinds of files are suitable for RaftFS?

As it has already been treated during the introduction of this report, RaftFS isn't suitable for big quantities of data for two reasons:

- Apache FtpServer isn't the state of art in term of performance: there are surely others remote file system frameworks with better performances.
- In order to avoid inconsistent situations, the Garbage Collector has to know the total how long at most the file upload would take. For this reason we have to introduce a timeout for each file uploading: as explained in section 2.2.2 in the put subsection, the client has at most 5 minutes to load in parallel the file on each node. This means that bigger is the file, less can be the number of nodes where the file is *initially* replicated (a workaround could be to put the file only on one node and then try a replicate command on others server).

Anyway, it's not easy to define a reasonable upper bound on the size of the files which an be uploaded on RaftFS, because it obviously depends on the network bandwidth. However, the author believes that with a replication factor of three servers and an average network, the user can use RaftFS to upload files in the order of some megabytes. This is perfect to use RaftFS as image and music storage, which are widely used today for distributed storage.

Did you notice that the *DataSet* folder is almost empty? Well, the reason is that the files contained in it are the only ones needed to perform the tests described in this report, and in addiction because you can use *any* file which size isn't bigger than some megabytes, like a mp3 or a jpg file. Enjoy it!

4.2 RaftFS performance

As we have already discussed in section 2.3.5, one of the main performance issues in RaftFS is that at each remote method invocation, the Leader looks up for an updated and reliable stub to communicate with any other server. This problem have influenced the election timeout of both Follower and Candidates, and in general they are strongly correlated with how long the `lookup()` operation takes. On the network of three servers where the distributed version was tested, the average time to perform this operation is something between 5ms and 50ms, but a discrete numbers of time it could take even 100ms. For this reason the default timeout to lookup a server's stub is 100ms. As consequence (as suggested in Raft algorithm), since the election timeout should be one order of magnitude bigger than the average time to perform each remote method, the default election timeout in RaftFS is randomly chosen between 500ms and 1500ms.

The Raft original paper focuses its performance evaluation on how long the system is without a Leader. Why? As we have already said several times, the weak point of Raft is that it cannot guarantee availability, that can be traduced with "how long the system has no Leader" (in fact when the system has no Leader, the system will refuse any client request).

RaftFS was tested to figure out for how long the system is without a Leader. In order to do that, a special class `ServerKiller` was implemented: it's a daemon thread that after some time (1 second) that a Leader was elected , it forcefully crashes the actual Leader. With this mechanism a Leader election is periodically repeated. Obviously, this mechanism was disabled in the RaftFS final version, but the code was left for completeness (without running the daemon, of course). Each line that will be shown in the following plots refers roughly to 1000 Leader elections. The next two sub sections show two different kinds of test about Leader election.

4.2.1 Leader Election and Network Size

The first kind of test wants to show how the time needed to elect a Leader changes w.r.t. the network size. Since the networks tested are up to 100 servers, it's obvious that the only feasible version is the

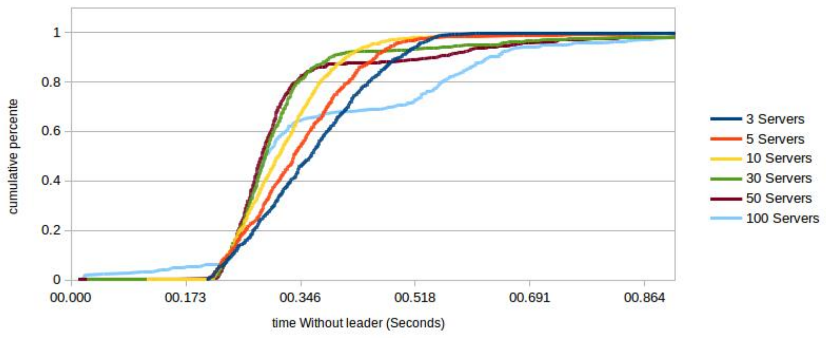

Figure 5: *Leader election in local version using different networks.*

local one. Figure 5 shows the cumulative percentage for the Leader election for networks from 3 to 1000 servers. The most incredible result is that up to 30 servers, the Leader Election average time decreases instead of increase. This is strange because a bigger amount of servers would mean more Candidates at the same term, and so a more probable election without winners (and so a system without a Leader). So how this is possible? The more reasonable explanation is that with a bigger amount of servers the probability that a server generate a low election timeout is higher than with few servers. Anyway, from 50 servers it can be seen that the number of times where the Leader Election requires a lot of time is more frequent with the increment of the network size. In conclusion, we can say that until 30 servers RaftFS increases its performance about Leader Election, while from 50 servers the Leader Election is more "conflicted": many time it reaches the best result seen in this test, but many time the worst too!

4.2.2 Leader Election in local, pseudo and distributed versions.

The second kind of test wants to show if there is some kind of difference between the time needed to elect a new Leader in the local, pseudo or distributed versions. RaftFS distributed version was tested on a network of three servers (and that's the reason because the test in the previous section starts with the same amount of servers). Figure 6 shows that the performance reached by the three versions are almost equals. As conclusion we can deduce that the results obtained in the previous test can be extended to the pseudo and (more important) distributed versions.

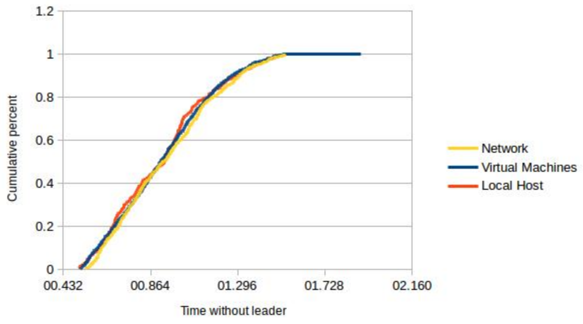

Figure 6: *Leader election in a cluster of three nodes in local, pseudo and distributed versions.*

4.3 RaftFS vs HDFS

The HDFS main issue is that the NameNode is a single point of failure: if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine. Since Hadoop 2.0.0, High Availability feature addresses the above problem by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby.

However, in HDFS the consensus algorithm has to be executed only between the DataNodes where the file is replicated and the NameNode, while in RaftFS the consensus has to be found between all the nodes in the cluster, which means that the performance could be worse in RaftFS than HDFS. In addiction, HDFS implements a series of optimization algorithms that RaftFS doesn't: for instance, in HDFS when a file is uploaded the client organize a pipeline with the chosen DataNodes to efficently upload the file, while in RaftFS the file is uploaded in parallel.

In addiction, HDFS is a distributed file system designed for big data (in fact each block is 64MB, reducing the number of blocks needed to represent a big file), while RaftFS is designed to store small files (as we have already seen in 4.1).

In conclusion, HDFS introduce a series of mechanism that we could expect one of the most famous distributed file system, but we can say that RaftFS is for sure a distributed system is incredibly more failure tolerant since each node can assume any kind of role, while in HDFS a DataNode cannot become a NameNode and viceversa.

5 Conclusions and Future Works

The final version of RaftFS reaches the main project goals: a strongly consistent file distributed system with an high failure tolerance and even with some scalability features. In fact we can easily deduce that with quantity of files and frequent client read requests RaftFS would reach better performance than a single-machine remote file system since the single-machine would be incredibly stressed, while in RaftFS the work is distributed between different machines. Anyway, this aspects was not tested since simulate such a situation would have been too much complicated (but it's considered a future work for sure).

Anyway there are a lot of aspects which can be improved in RaftFS, some of them was cited during this report:

- Introduce some kind of reliable timeout for client requests.
- Use some kind of metric to download the file from the best machine (for example the most idle one).
- Allow to upload big files.
- Improve the uploading process (maybe with a HDFS pipeline strategy).
- Implements the replicate and remove operation in parallel (like the put one)
- Implements a more efficient mechanism to refer each server's stub (avoiding the `lookup()` operation every time).
- Calling remote methods in parallel (like in the Raft algorithm).
- Implements snapshots for log compaction.
- Implements some elasticity mechanism to modify the cluster configuration.
- Introduce some kind of user policy and session.

References

[1] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 1985.

[2] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2002.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. 2007.

[4] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems 16*, 1988.

[5] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. *Proc ATC10, USENIX Annual Technical Conference*, 2010.

[7] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *Annual Technical Conference*, 2014.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies (MSST)*, 2010.

[9] Apache. Maven. http://www.maven.apache.org.

[10] Apache. Ftpserver. http://mina.apache.org/ftpserver-project/.

[11] Apache. log4j. http://logging.apache.org/log4j/2.x/.

[12] Hashicorp. Vagrant. https://www.vagrantup.com/.

[13] Diego Ongaro. *Consensus: Bridgin Theory and Practice*. Stanford University, 2014.