PADI-FS

Middleware for Distributed Internet Applications
Project - 2012-13

João Loff
Nº 56960

Alexandre Almeida
Nº 64712

Tiago Aguiar
Nº 64780

Group 06

Abstract

PADI-FS is a distributed storage system designed to manage a set of files. Developers that need to supply their applications a way to store their files in a persistent, replicated and highly available manner can use PADI-FS to fulfill their needs. These applications, the clients of the PADI-FS system, will make use of simple operations available - open, create, read, write, close and delete - to interact with the various data servers that are part of the PADI-FS architecture, and eventually with the files that are kept on them.

This paper will list the various details of the system architecture being developed, the various problems found in its design, the solutions for these problems, the reasons why these solutions were chosen and how they will be implemented.

1. Introduction

The file system is one of the cornerstone abstractions in which almost all computer systems rely on. On its basics, a file system is an abstraction to store, retrieve and update a set of files. Over the years, the storage needs for applications have increased and the typical vertical scalable approach wasn't getting it done. Adding to that fact, people started to want to collaborate over the same files and the usual approach wasn't fit for it. The solution was to implement a shared file repository, available to all the users at the same time. This approach became known as a **distributed file system**. In recent years, there has been an even greater deal of attention over this paradigm, with the introduction of cloud-based distributed file system such as DropBox, iCloud, Google Drive and Microsoft Skydrive. In this project we aim to implement a similar, though simpler, distributed file system, the PADI-FS.

The three core components of the PADI-FS system are: Metadata Servers, Data Servers and Client processes. The Client process makes requests to create, open, close, delete, read and write files. Files might be stored in several Data Servers, therefore Data Servers provide data replication, and can be placed on different places over the Internet. The Metadata Server is a server that it's reliably replicated and keeps metadata information about the files such as: filename, quorum size and Data Servers that have that file.

For testing and debugging purposes we will develop a new component, the *Puppet Master*. We won't be focusing on this component in this paper because it's irrelevant to the system itself. In the following sections we will explain the architecture of the system and how these components interact with each other, its implementation, the solutions and mechanisms found to address our concerns.

2. Architecture

Our architecture is composed by three Metadata Server (a master and two replicas), multiple Data Servers, both of them accessed by multiple Clients. The Master Metadata Server maintains all file system metadata, this includes: mapping from Client filenames to Data Servers filenames, current locations of files (and their replicas) and the quorum information. It also controls system-wide activities such as file placement on the various Data Servers and file migration between Data Servers - both part of its load-balancing duties.

The Clients issue requests on the file metadata to the Master Metadata Server, namely: open, close, create

and delete. Based on the information that is given by the Master, it will issue requests on the correct Data Servers to read or write the contents of the file.

The Data Servers main purpose is to store the file contents, and answering the Clients requests for reads and writes on those files. They periodically communicate with the Master Metadata Server, using heartbeat messages, piggybacking on them file usage statistics. The Master Metadata Server uses this heartbeat messages to decide on the system load balancing and file migration strategies.

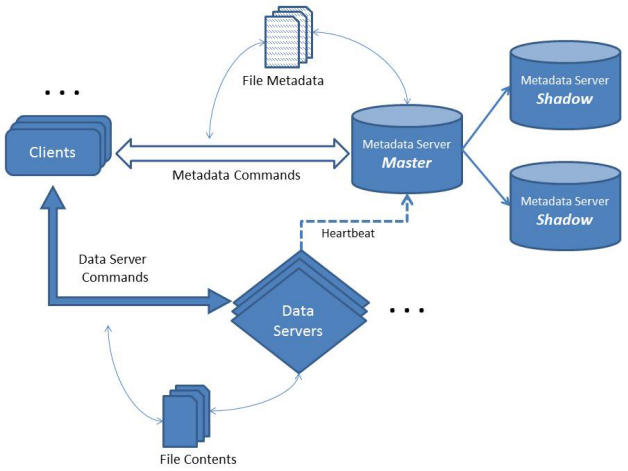

Figure 1. Proposed Architecture

3 Metadata Server

The Metadata Server serves as bridge between the Clients, that request operations on the files, and Data servers, that hold the files themselves. Each Client requests from the Master Metadata Server the metadata for that file, composed by: the original filename given by a Client upon create; the read and write quorum for that file; and the Data Servers where the file is stored, including the local filenames that the file has on those Data Servers. The metadata server is responsible for:

- Answering the commands that issued by the Clients: open, close, create and delete.
- Manage the entry and exit of Data Servers, performing load balancing operations to keep files correctly replicated.
- Deciding which Data Servers will be used to store each file, also considering Data Servers load.
- Perform garbage collection operations on the Data Servers

3.1 Master and Shadow Metadata Servers

We choose to adopt a passive replication system for the Metadata Servers, where we have a primary replica responsible to handle all the operations, and the backup replicas that receive state updates from the primary, and that can become the new primary in case of failure of the current one. Following the *Google File System* notation [3], we call the primary replica the **Master**, and the backup replica the **Shadows**.

This architectural decision vastly simplifies our design and enables us to perform better garbage collection and load-balancing management, since the Metadata state is inherently consistent. However, we are centralizing our architecture, therefore we need to deal with potential bottlenecks with requests and the state update to the Shadows.

- First possible bottleneck can occur in Clients requests, this is vastly reduced since Metadatas never deal with files themselves, and they just give the Client information in which Data Servers that file is located. This way Metadatas just deal with requests of a basic scale (especially in size).
- The second possible bottleneck can occur with the Master state update requests to the Shadows. All the state updates to the Shadows are made using a broadcast and non-blocking approach. This means that the Clients are not waiting on the Master to update the Shadows and are given the request response right away. We further expand this subject in the next topic.

The Master Election mechanism will be explained, after the Metadata Fault Tolerance due to their interconnection.

3.2 Metadata Fault Tolerance

First step to our fault tolerance architecture resides on the set up of a write-ahead log containing all Metadata operations. Following the write-ahead log philosophy used in the Database field, each Metadata needs to first write the operation that is going to do in the log, before realizing it.

After fulfill the client's request, it will try to update the Shadows in a non-blocking broadcast manner. If the Master detects that one of the Shadows is down, it will add a mark with its identification to the log. This mark indicates that on that point of the log, the identified Metadata failed. On recover, the failed Metadata, will request the Master to update its state. The Master checks its log for a mark for that Metadata, and answers back with the operation log since that mark.

Now, this broadcasting non-blocking architecture presents a glaring problem: we don't have guarantee that the messages sent from the Master to the Shadows will arrive in the order that arrived at the Master. This problem is similar to a known problem in distributed systems: total ordering. We adopt a very simplified version of a sequencer-type algorithm [1]. In our solution, and since we have the assurance that one Metadata is always available, we make the Master of our system, the sequencer. The master will then assign a sequencer value - in our implementation, a Lamport clock [5] - to each of the relevant operation. That sequence number is sent with each state update to the Shadows, and is also added with each operation added to the write-ahead log. On request arrival, the Shadows verify the sequence number attached to the request: if the request has the same value as its own clock, then proceeds to fulfill the request, otherwise stores that request temporarily until it can process it - until its own clock matches the request sequence.

3.3 Master Election

Another subset of the Metadata Fault Tolerance problem is the problem of the Master failure. When this happen the system needs to elect a new Master from the active Shadows. This situation is similar to the known problem in distributed systems called: leader choosing.

After experimenting with some of the most known approaches to leader election [2,4], we noticed that they were too complex for our simple and limited system. We then decided to roll our own leader choosing algorithm, taking some ideas implemented by those algorithms. We called it *Uprising*.

Who calls *Uprising*? Only Metadatas are allowed to perform a new election. A new election vote can occur when either:

- A new Metadata enter the system. In this situation the entering Metadata calls *Uprising* on any of the other Metadatas.
- A Metadata recovers from a crash. In this situation the recovering Metadata calls *Uprising* on any of the other Metadatas.
- When Clients or Data Servers contact a Metadata that they think it's the Master, but it isn't. In this situation the Metadata, suspicious on why a Client or Data Server is contacting it, will call *Uprising* on itself to check if there is a new Master. If at the end of *Uprising*, the contacted Metadata is not the Master, it will return an error, and piggybacked on it the new Master identification.

3.3.1 Uprising

The Metadata were *Uprising* was called on, first checks if itself is the Master. If it is, it immediately answers back to the requester with its own id as the new Master.

If it isn't, then it will send his vote - composed of its own id and clock (the one used as sequencer) - to all the remaining Metadatas. Each Metadata, receives that vote, and compares it with its own vote, choosing the one with the highest clock (received more operations from clients) and as a tie-breaker, the one with the lowest Metadata id. The chosen vote is then sent back to the Metadata that requested it.

There, the Metadata will then choose, applying the same rules as before, from all the votes received the best one. Finally, it informs all the Metadatas who the new Master is, and answers back to the requester of *Uprising* with the new Master id.

3.4 Garbage Collection

Considering that Data Servers are highly unstable, we took the decision to not delete files from Data Servers in case of a delete operation on the Master. Instead, we took the garbage collection approach. We leverage our garbage collection system on the already existent heartbeat mechanism between Data Servers and the Master.

Since in each heartbeat there is information from each file that's on the Data Server (needed for load balancing purposes), performing garbage collection becomes trivial. On each heartbeat, the Master compares the list of files that it has, with its own list of files that should have. Then, it just sends back a list of files that the Data Server should delete.

3.5 Load Balancing

We can separate load balancing in two distinct aspects:

- First one is what we called preventive load balancing. In this type, the Master decides where a certain file will be placed based on the load of a Data Server
- The second type, we call, reactive load balancing. In this type, the Master proactively migrates files between Data Servers, so the load of the system is evenly distributed between all the Data Servers.

For the load of the system we analyze the load of each file, and consequently the load of a Data Server. A load of a file is composed by its number of reads and writes. The load of the Data Server is an average of all

its files loads. And finally the load of the system is the average of all its Data Server's loads. The choice of the number of reads and writes for each file as the file load was made based on the simplicity of the system. Many more factors could be taken into account, but adding more factors for such a simple system would just be hindrance to the load balancing performance.

3.5.1 Preventive Load Balancing

The chosen preventive load balancing strategy was Dynamic Round-Robin [6]. Dynamic Round Robin is a variation of the known Round Robin scheduling algorithm, but instead of blindly assigning files to Data Servers, it assigns files to Data Serves based on a load factor that we call **weight**. Each Data Server has a weight associated with it, which is updated on each heartbeat. When the Master has to place a file on a Data Server, it just traverses it's upwardly ordered Data Server list, assigning the file to the requested number of Data Servers.

3.5.2 Reactive Load Balancing

Reactive Load Balancing is a service that is running on a set time interval that traverses all of the files that the system has. For each file it checks for mainly two conditions (there are others minor tuning conditions that are not relevant to describe in here but are referenced in section 6):

- If there is a Data Server were that files is a *better fit* than the current one, i.e., when a file weight plus the Data Server weight don't go over the average system load (plus a threshold).
- If a file is free, i.e., if there are no clients that have that file currently open.

If both conditions are true, we first lock that file and then we proceed to migrating the file between Data Servers in an atomic manner. It is interesting to refer that we do not perform a delete operation on the Data Server, instead, we leave that job for the garbage collector.

Reactive load balancing moves just one file for each data server, in each call. It gradually balances the system, instead of trying to balance in just one step. This way we avoid peaks in file migrations that could happen when there are peaks in Data Server load.

4 Data Servers

The Data Servers are responsible for storing the contents of the files. Each file includes a version and its contents (represented by a byte array). The Data Servers receive the following requests from the Clients:

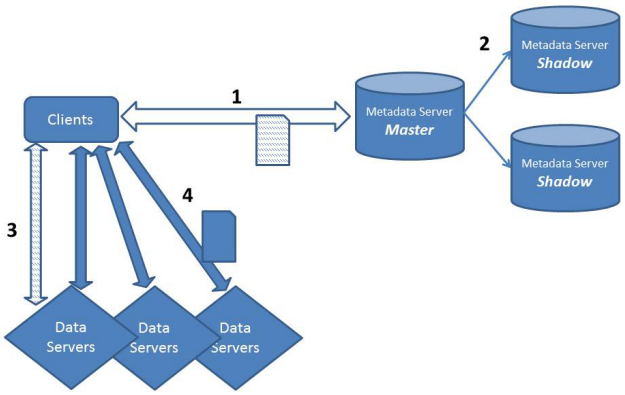

Figure 2. Write Request Lifecycle

- Read, that reads the most recent contents from a specific file from the local file system
- Write, that (tries) to write new contents on a file. We say *" tries to write* because there is no guarantee that file version a client is trying to write is the most recent one. If that is the case the Data Server simply ignores that version.
- A special third operation, that is used by the other two main ones, is available, we call it Version. The Version, returns to the client the current version stored in the Data Server.

The usage of the Version request, as well as the coordination of read and write operation, is explained in the Clients section.

The Data Server also has to timely perform heartbeats to the Master Metadata Server, piggybacking load statistics for each file and for itself (the average of all its file weights).

5 Clients

The client communicates with the Master Metadata issuing the following requests over file metadata: create, delete, open and close. In the Data Servers, clients can issue: reads and writes on the file contents. Let's go over a write request lifecycle.

1. Client requests the Master to open a certain file. The Master answers back with the metadata for that file, specially the list of Data Servers that contain that file.

2. Meanwhile, and after replying to the Client, the Master will write to its own log, and update the Shadows.
3. The Client will perform a **version operation** on all the Data Servers, reading - accordingly to the quorum - the most recent version from that file. After reading the most recent version, it increments that version and writes the new contents to that file.
4. It now broadcasts the new file contents (and the new version) to all the Data Servers that have that file, waiting on the write quorum. There is no guarantee that the version the Client wants to write will in fact be written in the Data Server.

In case of insufficient quorum on the read version or write operations, the Client blocks until it can fulfill that quorum. While it's blocked, the client re-sends the requests to the Data Servers and requests the Master Metadata Server possible new Data Servers were that file is located, trying to fulfill the quorum and unblock itself.

5.1 Coordination of Read and Write operations

When entering a context where multiple clients are performing concurrent reads and writes to the same set of Data Servers, we need to keep a consistent view of the file contents through the operations. It turns out, the solution for versioning suggested in the project statement is insufficient. We decided to change the versioning system, to a more collaborative one, between the Clients and the Data Servers.

- Each file has file version consists in a pair: the **Client id** responsible for the last write, and a the value of the file's **Lamport Clock** [5] that marks the last write operation. The initial value of this clock is set as 0 by the Data Server, upon file creation (the first read or write operation). We assume that Client ids are unique.
- For both read and write operations, the Client starts by making a version request, that returns the current file version of that file. If the request is a write, the client will set the file version client id to its own and will increment the file's clock.
- The Data Server, upon arrival of a write request, will compare the file version it has locally stored with the one that arrived in the request, choosing the one with the highest clock. If both of the files have the same clock value, it will use as tie-breaker the lowest Client id.

6. Implementation Evaluation

6.1 Clients to Master requests

Recalling our architecture for Clients to Master requests: the Master was instantaneously answering back to the Client, meanwhile sends state updates to the Shadows in a non-blocking broadcast manner. In a more traditional type of passive replication system, the primary only answers to the client after updating its replicas. So, the latency for a Client request's answer is: $RTT_{Client-Master} + N_{Shadows} * RTT_{Master-Shadows}$. Meanwhile using our architecture the same time is reduced to just: $RTT_{Client-Master}$.

6.2 Master failure and Uprising

Consider the situation that the Master failed - when it didn't the execution is trivial and just answers with it's id to *Uprising* requester. We can state that the time for *Uprising* to finish is: $RTT_{Metadata-Metadata} + 2 * N_{Metadata} * RTT_{Metadata-Metadata}$. Remember that there is a first round to choose the votes from all Metadatas, and a second round to inform all the Metadatas who the new Master is, hence the $2 * N_{Metadata}$. Consequently, the number messages exchanged between Metadatas is also $2 * N_{Metadata}$.

Remember that in this scenario - Master failed - *Uprising* was triggered from one of the Shadows that received a request from a Client. What is missing is the update of the remaining system processes (Clients and Data Servers). This operation is not needed as explained before, because if a process calls an operation on a Shadow when the Master is still alive, the Shadow will return an error message, and piggybacked on it, the new Master id. This avoids message flooding from the Metadatas to the processes.

So, if on one side we have a somewhat a relatively high number of messages between Metadatas to choose the new Master, between the Metadatas and the remaining system processes we have very few. This is a tradeoff that we gladly take, knowing that we are most likely to have more Clients and Data Servers than Metadatas.

We are aware that *Uprising* might lack in performance for more complex systems, but recall that it was design for this very specific (simple) system. Still, we are confident that this algorithm could be improved and scaled to larger systems with minor changes to it.

6.3 Shadow recovery

Shadow recovery performance is irredeemably connected to our log performance. Recall that we mark

the log when a Shadow failed, and when it recovers we send the log of all the operations that it should execute after it. Therefore the performance of a Shadow recovery is **linear** to the number of requests is has to redo.

6.4 Read and Write

Let's focus this analysis on the decision to first request the last file version from the Data Server, before executing either the read or the write requests.

- Looking at read requests, we need to retrieve the version to check the read quorum. It would be a waste to retrieve all the file contents, while only checking for the read quorum. Even worse, imagine that the Client is locked in a quorum voting, constantly reading files from the Data Servers to check for new versions, there is no need to also constantly retrieve the file contents.
- In write requests since we have to perform a read operation to get the last version file, why retrieve all the file contents when we just need the version? Even worse, if we fall into the same situation of an insufficient quorum, explained in the read operation.

With this optimization we easily decrease the latency of read and write requests, as well as bandwidth used by the system. As a tradeoff, the number of messages needed to perform both operations increases.

6.5 Load Balancing

For our load balancing implementation we implemented a black list of situations where the load balancer should skip the balancing for a specific file or Data Server. Here is the complete list of skip situations:

- If the weight of file's Data Server is inside the average system load plus a threshold
- If a migration to the new Data Server already occurred in that iteration
- If the Data Server is known to be down. A Data Server is known to be down, if it failed the last 3 heartbeats.
- If the potential Data Server migration target is inside the average system load plus a threshold
- The weight of the file plus the weight of the target Data Server is above the average system load plus a threshold
- If the file is currently open by any client

If it fails all the skips cases, then file migration will occur. By analysis of the conditions, one can notice that the load balancing is targeted at systems with medium-to-high load average. Another interesting conclusion is that our load balancing occurs over time, avoiding spikes in file migrations if there are spikes in system load.

7. Conclusion

Overall architecturing, designing and implementing this project was a very gratifying experience, tough two aspects stand out:

- It was with great joy that we saw our initial architecture almost fully implemented without any core changes. It showed us that we could not only implement a system but also architecture one from the ground up with very few deep flaws
- The second aspect was the opportunity that this project gave us to implement some of the algorithms that were theoretically given to us, and see *in loco* some of the hidden aspects of their implementations that can escape at a first glance.

References

[1] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[2] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Computers*, 31(1):48–59, 1982.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[4] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, November 1980.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[6] Der-Chiang Li and Fengming M. Chang. An in-out combined dynamic weighted round-robin method for network load balancing. *Comput. J.*, 50(5):555–566, September 2007.