PADI-FS

David Dias 65963, Pedro Barroso 65994, Rui Camacho 65998
Instituto Superior Técnico
Av. Prof. Doutor Aníbal Cavaco Silva
2744-016 Porto Salvo
{david.dias, pedro.barroso, rui.camacho}@ist.utl.pt

Abstract

PADI-FS is a distributed file system(DFS) which offers to it's users flexibility when they have to choose between availability vs consistency. Our solution provides users a way to tune up consistency and availability when they create their files, assuring that the requirements are met, through efficient File Allocation, Load Balancing algorithms and a Coordinated file access for mutations.

1. Introdution

PADI-FS is a DFS that enables users to store, access and edit their files remotely with a simple API (open, create, read, write, close and delete) that takes care of file replication, load balancing and Coordinated file mutation to avoid conflicts with two or more mutations to the same file simultaneously. In this document, we will present the goals of PADI-FS, the architecture developed to support this system, the Consistency model that gives the user the power to choose consistency vs availability for their files, how we process the load balancing through machines and explain how the various elements in the system interact in real usage scenarios. During all the document, we also present the challenges of building this types of systems and how we managed to overcome them.

2. Design Overview

2.1. Goals and Assumptions

Distributed Systems implementations are a case of study for the last decades, there isn't any "One size fits all" solution nowadays, instead we've several solutions that optimize for: performance, consistency, availability, fault tolerance, and partition tolerance. In PADI-FS we propose solutions to achieve:

Able to store different file sizes - Comparing to Database systems, FS are used to store files with different sizes, this can be from simple text documents of few KB to high quality video of hundreds of GB. Having this in consideration, file caching becomes a unfeasible solution, having in mind that we not foresee file fragmentation to add to this protocol.

Scalability - Prepare the system to cope with increase and decrease of the amount of files stored by using a efficient railing system of new data storage machine.

Enable the choice of Consistency vs Availability - Give the user the power to opt between Consistency and Availability by letting him choose how many times he wants his files to be replicated and choose the read and write quorums.

Load Balancing - Take advantage of the several data storage machines to assure that throughput and data store are used evenly to maximize the performance. This will mitigate slow file access by avoiding having machines overloaded.

Cascading failure mitigation - It's has been seen cascading failures of distributing systems, one example is the Gmail case in 2012 December, which by faulty load balancing logic, the load was not distributed uniformly causing every server failing sequentially. Our solution distributes meta servers load uniformly in case of a failure to mitigate this.

2.2. Arquitecture

Our architecture is designed to achieve a fully automanaged design, with a easy railing system for new data storage machines and able to cope with several clients simultaneously. PADI-FS divides into 4 components(Clients, *Metadata servers*, *Data servers* and File Handles) to work properly, we included one extra component, called Puppet Master, to be able to control every other machine for debugging and functionality showing purposes during development stage.

2.2.1 Client

The client is the system frond end, offers to the users an API to *Create, Open, Close, Delete, Read* and *Write* files onto the system, being the point of contact to *Metadata server* and *Data server*, assuring that the read/write quorums are correctly achieved using a 2-Phase Commit Protocol[1, 2, 3].

2.2.2 Metadata Server

Metadata servers store the metadata of each file: FileHandler[File name, NBData Servers, Read Quorum, Write Quorum, Data Servers[data servers, local-files], NBAccess]. We needed to decide how to organise the 3 available *Metadata servers*, here we present the studied solutions we looked for *Metadata server* organisations:

Primary Server & 2 Secondaries - This is a traditional approach in enterprise systems, creating a single point of contact, perfect to uniform and queue the transactions, however this solution focus in a single API endpoint for metadata reach at a given time, which can be overwhelming in moments of great system usage.

Random Access - In this approach Clients would try to contact a random *Metadata server* to serve their request, this create an abstraction to the Client by not having to know how *Metadata servers* are organised and leveraging the Distributed Locks on *Metadata server* Side, however, this raises the complexity of the system and creates unexpected cases of Distributed Deadlocks when accessing files

Contact all *Metadata servers* - This solution would assure that *Metadata servers* are almost updated at same time, however, since we can not guarantee the correct deliver of messages, a file mutation request from 2 different clients could result into a deadlock if not well coordinated, increasing the degree of complexity like Random Access option and not to forget that by contacting always all, we are wasting precious time/CPU resources to process 3 times each request.

Division of responsabilities - This is a solution inspired on DHT algorithms, that enables each *Metadata server* to hold responsibility for different set of files, being the perfect candidate to coordinate File Mutation Transactions, being a unique rendez-vous point for each set of files, assuring that the requests are treated sequentially, mitigating deadlocks on concurrent file mutations by several clients. This approach was our choice for PADI-FS and it is explained on detail on section 3.1

Metadata servers are also trusted to balance the load distribution between the several *Data servers* to avoid the mentioned problem of data overloading a server that may cause it to fail, this is explained further on section 3.2.2.

2.2.3 Data-Server

Data servers work as storage machines only, this means there is no algorithmic complexity running on them, they have no knowledge about the state of the global system, the position of other *Data servers* or the files that they hold. They server requests from Client and inform *Metadata servers* of their entrance to the system.

2.2.4 File-Handler

File-Handlers are data structures stored in *Metadata Server* and are passed to the Client so he can manipulate the Files stored in *Data Servers*, they contain the location of the File on the differents Data Servers and the name used to reference the File on those Data-Servers.

3. Consistency Model

3.1. Metadata

The metadata is replicated across 3 metadata servers, each of these servers is responsible to coordinate access to a specific set of Files and to update the other *Metadata servers* accordingly. To find the set of metadata that each server is responsible, we develop a solution inspired in DHT[4] algorithms like Chord[5] and Pastry[6], we calculate the Hash Value of the filename using SHA-1 hashing function, since we SHA-1 offers us a consistent hash value size (160 bit), we divide this 160 into 3 sets, allowing our system to assign each one of this sets to each *Metadata server*.

We opted for SHA-1 as our Hashing algorithm by it's ability to distribute the names uniformly, this can be seen in Figure 1, where SHA-1 was used for 300 000 filenames of music and video files. This empirical proof was provided by S. Rieche, H. Niedermayer, S. Gotz and K. Wehrle from the University of Tubingen.

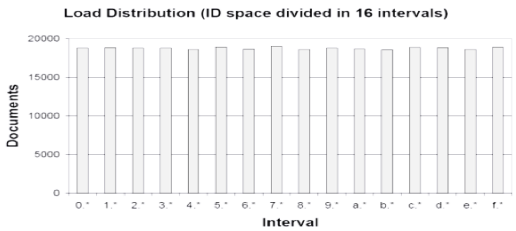

Figure 1. Distribution of file names after hashing with SHA-1 algorithm

Each *Metadata server* holds 3 tables, each one for the 3 sets of files and updates the one he is responsible to, informing in backgorund the other *Metadata servers* about

the update, creating a replicated state of present metadata in each *Metadata server*, maintaing consistency and creating the opportunity to have the perfect candidate for file mutation coordination, since each file has a unique rendez-vous point.

One other interesting studied approach making that the client didn't need to know what *Metadata server* he should contact and send the message to one randomly, then this *Metadata server* would evaluate the responsibility for that file and forward the request accordingly, however we didn't opt for this solution because because it would generate a 66% chance of generating an extra message in the network, we assume that *Metadata servers* Failures are punctual and span for short periods of time and by so being interesting and less complex that the Client knows whom should he contact next in case of one *Metadata server* failure, as explained in following section.

3.1.1 Meta Server Fault Tolerance

In our proposed solution, we assume there is no permanent failures of Machines, only out of reach periods of time. When a Client fails to contact the *Metadata server* responsible for a determined file, he has the chance to contact another *Metadata server*. To avoid Cascading Failure, by this we mean, throwing all the load from one *Metadata server* to another one, making him also fail and in the end all servers to fail, we developed an elegant solution to distribute the load. In order to choose the next *Metadata server* to contact, we divide the set of filenames that the failed Metadata server was responsible in half, attributing each half to 2 remaining servers. In our implementation to achieve this half division instead of dividing the SHA-1 namespace by 3, we divide it by 6, assigning 2 namespaces for each *Metadata server*, with this, we choose the next *Metadata server* to contact considering the namespace we were trying to reach, we can see on Figure 2 an example.

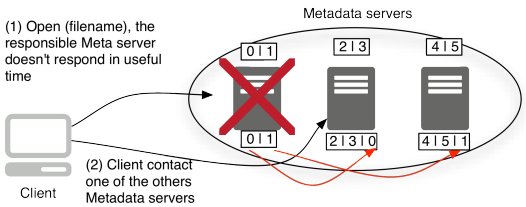

Figure 2. Metadata fault tolerance solution

When a *Metadata server* receives a request for a file he is not responsible, he will check the state of the primary for that file and result divides into two possibilities:

The primary *Metadata server* for that file is up - Which means probably Client got confused by a network delay in the message and try to contact another *Metadata server* to complete his request, in this case, he will inform the client to contact the primary *Metadata server* for that file again.

The primary *Metadata server* for that file is down - He will take care of the request for the client and update that file metadata accordingly and updating as well the other left metadata Server, when the primary get's back up, we will be updated with the changes not coordinated by him and continue doing is job.

3.2. File data

Data servers don't have any information about the state of the network, having only one purpose, which is to store and retrieve data. To achieve coordinated file mutations operations, like write, we implemented a 2 Phase Commit solution used by the Client and Coordinated among clients by the *Metadata servers* to avoid conflicts, this will be described in section 3.2.1. Also, to optimize the the usage of *Data servers* in terms of throughput, we developed a load balancing algorithm to distributed more requested files through machines, described in section 3.2.2.

3.2.1 Access method

We defined two models of file access: File Mutation Operation(Create, Delete Write) and Non File Mutation Operations(Open, Close Read). On the first ones the client will ask *Metadata server* permission to do the operation, this will create a lock over that file and inform other metadata Servers, mitigating the possibility of having two clients changing the same file at same time, this process of locking access and giving the client permission to access the file is made atomically. We achieve sequentially file mutations by having only one coordinator for each file thanks to the division of responsibilities. Once the Client receives the permission to manipulate the file, he will execute a Quorum based 2PC on the *Data servers* holding this file, when the Quorum is achieved, the operation is committed and reported to *Metadata server* to unlock the access to the file, so other Clients in queue can access it, the updated metadata is piggybacked in this operation.

3.2.2 Load Balancing and Migration Strategy

We looked into the solutions adopted n Google File System[7] and the P2P Structured Overlay Networks such as Chord[5] or Pastry[6] which tend to use file striping and/or a DHT[4] to divide the data equally among the Data nodes of the system , this gives the system an even distribution, however, file striping raises a lot of complexity problems when we talk about load balancing, since Meta Data for each file gets huge revamp and DHT don't have a concern about balancing the throughput load in each machine.

Our approach is inspired by the balancing scheme proposed by Huffman, based on Greedy Algorithm, which takes advantage os the actual state to choose the next better solution. We sort metadata by File-Heat, a unit measured calculated with the number of access to a file taking in mind it's size:

File-Heat = (n° of access/n° of total access) + log(size of file)

This tell us the files most used in our system and those who should not share same machines, to make sure we don't have machines that overloaded with heated files. The second measure we take into account is Machine-Heat, this is calculated by:

Machine-Heat = Sum(File-Heat of each file on this Machine)

This gives us the notion of what machines are more used and those who aren't, we this measure can know where to write/migrate files. This solution has to goals:

File Allocation - We use the Machine-Heat measure to calculate where to put files in a Create action, this measure is calculated periodically and stored.

File Redistribution - This is the core of Load balancing, the File-Heat and Machine-Heat are calculated periodically and used to calculate next file distribution, we developed an algorithm to achieve the desired even distribution:

1 Calculate File Heat

2 Calculate Machine Heat and sort a Machines by heat ascending order

3 Match the high half of the Machine-Heat list with the lower half and using Thermal Dissipation concept to even those machines, passing heat files to Cold machine so they can get even as shown in Figure 3.

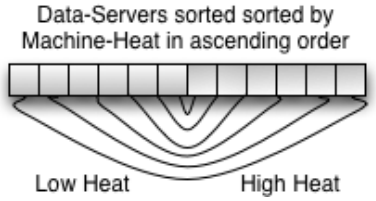

Figure 3. Migrating heat files to reach even state

4 Calculate the Average machine heat and verify if the Max Machine-Heat is over the Threshold imposed (by default 15%)

Repeat the steps 2,3 and 4 a number of times defined by a imposed threshold, a lower threshold leads to less file migrations but a less balanced system while more iterations leads to more balanced system but also more file migrations that could be very expensive with big files. This operations are done over metadata, so they are extremely fast to converge since we don't have to transmit any file until we know the final organization of files. If the number meta-files reaches astronomic numbers, we may need to use more computing power than it's available by the *Metadata servers*, so a solution as MapReduce[8] may be used for this purpose.

4. System interactions

4.1. File Mutations

The operations that can be executed over the PADI-FS files are create, delete, open, close, read and write.

4.1.1 Create and Delete

Client sends a request to *Metadata server* to get permission to create that file on *Data Servers*, the *Metadata server* verifies the existence of a file and if it doesn't exist selects the data servers where to store the file's replicas, creates a *File-Handler* and stores that information and the one provided by client. The file handler is sent to client piggybacks permission to create the file and a lock is made over that file in the Metadata servers. Then the client does a create operation based on fully 2PC on Data servers. Client when executes Delete operation asks permission to metadata Server, after permission granted (which means he will be the only one mutating the File) he performs a fully 2PC to delete the file from all the *Data servers*, after commit, informs *Metadata server* of result.

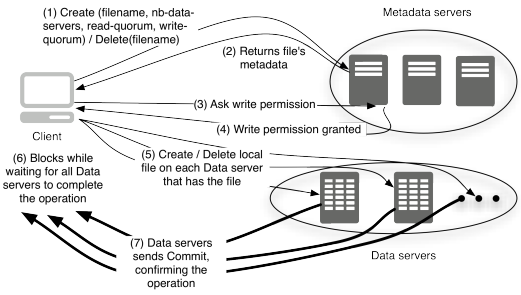

Figure 4. Create and Delete Operations

4.1.2 Open and Close

Client makes open request to *Metadata server* and it receives a *File-Handler* with the information about that file, such as Data Servers where file exists. On Close, client informs *Metadata server* that he will stop using that file and discards the *File Handler*.

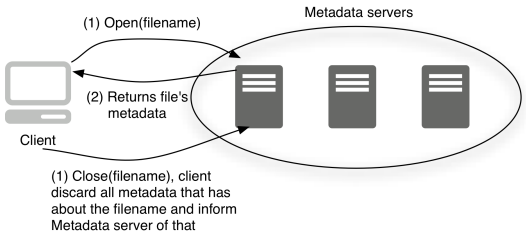

Figure 5. Open and Close Operations

4.1.3 Read

Client uses a previously obtained file handler to contact directly the Data servers where that file is stored, it performs the Quorum read operation and blocks itself waiting for a number of responses from servers to reach Quorum. The client then selects the response with most recent version number in *Default* semantic case, or in *Monotonic* semantic it only select the most recent response if it is more recent than the one it already have.

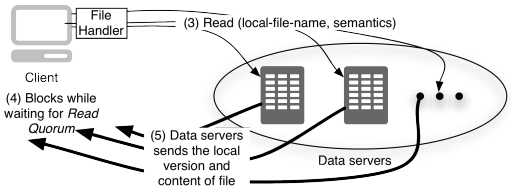

Figure 6. Read Operation

4.1.4 Write

The client asks to *Metadata server* permission to write, the *Metadata server* locks the file and gives permission to the client to write. Client uses a Quorum 2PC over the *Data servers* where file is allocated, once the determined quorum is achieved the client commits the operation and inform *Metadata server* of the successful operation.

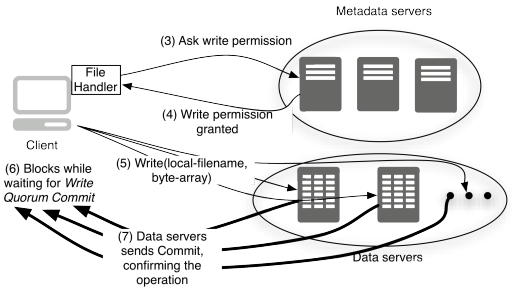

Figure 7. Write Operation

4.2. New Data server Join

When a new data server joins the system informs the *Meta-Data servers* that is alive and available to receive files.

4.3. Data server Failure

Data server failure is considered a period during which the server won't reply to any request, since we assume that file corruptions are impossible to happen, we didn't develop a system to verify the file state

5. Performance Analysis

5.1. Efficient Coordination System

The efficient coordination system implemented, dividing the file responsibility amongst *Meta-Data servers* has proven to be robust, from all the tests executed, there was zero file corruption/collision between different clients trying to change same file on the Data-Servers, this was due to the single *Meta-Data Server* coordinator base for each file, which confirms that our request balancing at *Meta-Data servers* level, can coordinate file access in an effective way, as discussed in section 3.1.1.

5.2. Load Balancing Analysis

Our Load Balancing tests have confirmed our predictions, the user of PADI-FS is able to control the precision and effectiveness of it's Load Balancing algorithm, achieving a better balanced load in function of time. As we can see in figure 8, as we raise the number of cycles (iterations) of our thermal dissipation algorithm, the load balance process increases linearly, this is an important decision to make due to it's great impact on the performance of PADI-FS, since files that are being "balanced" are locked to avoid wrong reads or writes by the clients. It's necessary to minimise the time they are in this state, so client operations don't get blocked. Considering that is possible to do a Write on 84 milliseconds under the test environment conditions, changing from 10 iterations to 70, can result in a extra time wait time of 3 seconds, which means that lot's of operations will have to wait.

The criteria used for the decision of how many cycles we must use in the Load Balancing algorithm, is based on the goal intended by the system administrator to ensure that the balance is even. We call this "threshold", the Load Balancing algorithm stops executing as soon as we get to the threshold, for example, the machine with most heat has only 15% more heat than the average, this way we can ensure some acceptable load distribution without entering into a infinite loop until until threshold goals are met.

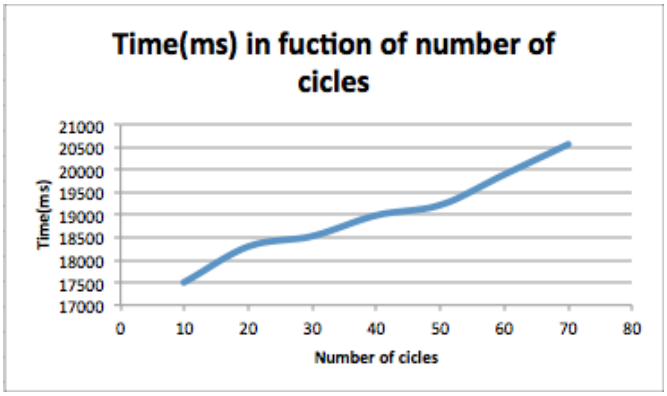

Figure 8. Load Balancing - Time/Cicles

Figure 9 demonstrates how PADI-FS performs while reaching an even distribution regarding the number of cycles.

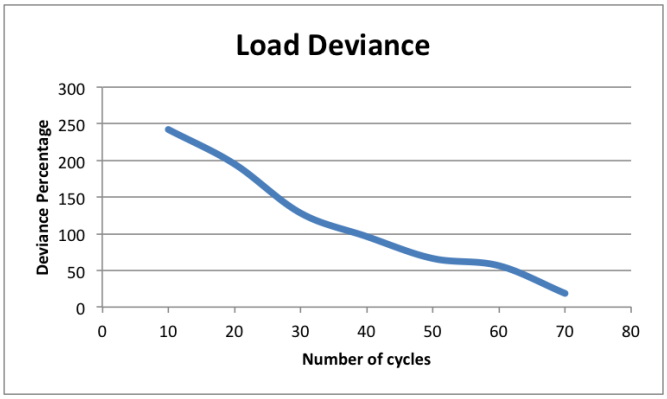

Figure 9. Load Balancing - Deviance/Cicles

The more cycles we execute, the more even is the distribution, however as seen in Figure 8, this brings a cost associated with time, in this sense, it's up to the system administrator of PADI-FS to consider how system is accessed and choose the best relation cycles/time.

6. Future Work

To increase the performance and the consistency of PADI-FS, we propose to realise Wide-Area Network tests so it's possible to test the feasibility of the solution implemented facing network problems. There is also space for creating a better service quality by assuring the replication level for when a *Data-Server* fails, so a client doesn't need to wait for the same to come back and reach it's Read/Write Quorum.

Taking in consideration the learnings from Cloud Computing Data Centers, where we can see a tendency during the day for burst of huge demand of computing and the long tails that follow, it's possible to build an adaptive Load Balancing algorithm that takes advantage of learning the system's demand, and increase the Load Balancing cycles when the request is lower, and reducing when the demand is high, making it, most of the time, evenly distributed in the long term.

7. Conclusions

Our proposed solution for PADI-FS was developed to correspond the requirements for this system, of course that some features could be added to improve the system, for instance the assurance of the replication level when a Data server fails or the investment of some intelligence in *Data server* in order to make them capable of file version synchronisation between themselves in background, but that extra complexity would turn the project not feasible in time, besides isn't required to achieve the requirements of the system.

References

[1] Y. Raz, "The dynamic two phase commitment (d2pc) protocol,"

[2] A. Wilson, "Distributed transactions and two-phase commit,"

[3] B. C. DESAI and B. S. Boutros, "A two-phase commit protocol and its performance,"

[4] S. G. S. Rieche, H. Niedermayer and K. Wehrle, "Distributed hash tables,"

[5] D. K. M. F. K. Ion Stoica, Robert Morris and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications,"

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems,"

[7] H. G. Sanjay Ghemawat and S.-T. Leung, "The google file system,"

[8] J. Dean and S. Ghenmawat, "Mapreduce: Simplified data processing on large clusters,"