PADI File System

Abstract

Traditional standalone file systems aren't enough for current users demand because they need to store thousands of media files and access them everywhere at anytime. Recent file system must solve this problem being distributed and providing high-availability, replication and being reliable. We describe our PADI-FS design which provides a distributed file system interface based on three metadata servers and hundreds of data servers which are controlled by client applications. This system support load balance and its fault proof.

1. Introduction

Nowadays people rely on digital hardware to save photos, music, documents and business planing. File loss could result in large money loss so we have designed PADI-FS, a distributed files system with high available and reliability. It provides fault tolerance using three metadata serves and many data servers. PADI-FS share the same goals than Google File System [1] but it is designed for typical user data size, not for big data. Furthermore PADI-FS uses three metadata masters instead of a single master used by GFS. It provides load balancing, high-availability, consistency, parallel access and scalability.
We assume that components failures are the norm rather than the exception. System fault-tolerance on unreliable hardware will be archived though metadata and data servers replication. We also assume that overwriting operations are more common than append.
The rest of the paper provides the details of our approach. Section 2 overviews the system architecture. Section 3,4,5 describes the client, metadata and data server. Finally, section 6 presents our conclusions.

2. Architecture

Our architecture in composed by:

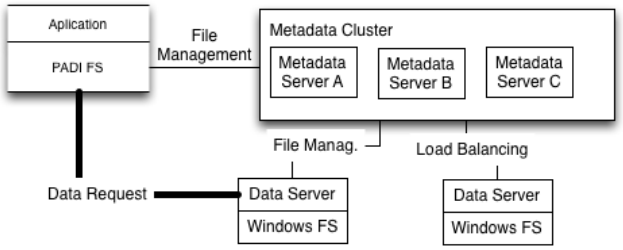

Figure 1. PADI-FS Architecture

- *Metadata Servers*: Three replicated metadata storage server
- *Data Servers*: File system storage
- *Clients*: Handle customer requests and forward them to PADI-FS servers.

Metadata servers keep all metadata replicated, high available and consistent. The client sends *create*, *open*, *close* and *delete* requests to **any metadata server**. The metadata server keep the file metadata and it coordinates the data server writes and file creation.
The reply is sent by the same server. All data transfers: *reads* and *writes* are made directly between client and data servers allowing high data transfer rates.

3 Client

3.1 Servers Interaction

We designed a solution where client application is able to connect to two kind of servers: Metadata and Data server. Metadata server handles clients requests for: *Open*, *Close*, *Create* and *Delete* file. Data server handle read and write operations without metadata server interference. The client knows the three metadata servers address by default but the data servers location are provided by metadata server at runtime when *open* or *create* request are made.

3.2 Load Balancing client requests

Our file sytem is designed to distribute client requests among all metadata servers. Since all metadata servers accept request, the client just need to choose one, send the request, receive the ack and wait. The client choose a random server and try it. If this metadata server its not available then client will forward the request to next server in a round robin model. This allows parallel requests and load balancing even in case of metadata servers failure. Each client saves and uses the last available meta server until it become unavailable.

3.3 Multithreading

Client application, which is managed by user, is multithreaded to allow simultaneous control over multiple files. Operations managed on same filename are serialized on metadata server but data servers access can be multithreaded.

4. Metadata Sever

Inconsistent metadata implies data loss and irrecoverable system so our algorithm choose reliability and availability as opposed to high performance. We implemented and tried 3 different solutions.
The first solution was a master-slave with passive replication, the second was based on reliable ordered multicast and the last was a hybrid solution.
The master-slave solution allowed any metadaserver to retrieve the request (behave as front-end), forward to current master, master serializes the requests, forward to its slaves and then reply to client. This is not fail proof. If master fails, all requests would be lost.
The Reliable multicast approach each server floods the request to others and then one of them serializes the requests and floods it. There is no master and all servers retrieve the message, then one server serialize it and then all servers deliver this message in order.
Our hybrid solution is based on chain-replication and namespace partitioning. It avoid master fail and flooding.

4.1 Churn

We tried two approaches for churn. For our master-slave approach, all servers must agree on which is the master. Our first server entrance and exit (churn) algorithm was based on *Bully* [2]. Each server had a state: Rowdy, Waiting, Electing and Online. When server starts, it become a new Rowdy and multicast its server ID. If listening server has a bigger ID, it send a "ShutUp" message and change to Electing mode broadcasting its ID. When biggest ID server is not "shutup", it changes to online and broadcast that he is the new leader.This algorithm was fail proof and very reliable to leader election.
Our second solution goal is not elect a leader but detect the other servers. Each server as 4 states: *New, Paused, Ready* and *Online*. A new server starts on new mode, broadcast to others its ID. All other online servers stop the requests **??** and change to pause mode checking if other servers are still available broadcasting its current *state vector* **??** and ID. If a server retrieve a older vector send it state or requests the state if newer. Since all messages requests are stop, this algorithm will converge because requests are always delivered **??**. If the server has the same state than all other paused server, then become ready. The new server can request a state update to any of ready servers (parallel requests would be possible) since all have the same state vector.
After all servers (including new) have same state, then servers change to online. Any failure is detected and servers restart the algorithm changing to pause again. This ensures that algorithm will converge if there is a stable view.

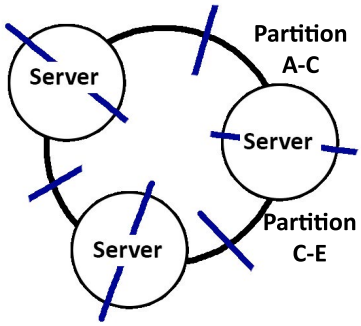

Figure 2. Metaserver Namespace Ring

4.2 Requests

Our hybrid solution is based on Chain Replication [3] and namespace partitioning concepts. Servers are organized in a circle which corresponds to a 16-character ASCII namespace. This namespace is split in 6 areas. Each area has a responsible server named *Serializer*. The *Serializer* is responsible for accept requests and set attach an *Order ID*. Since Order ID its mutual exclusive, there is not 2 requests with same id. This approach is simpler and faster

than an agreement protocol.This provides delivering order and a great load balancing through all servers.
If Serializer forward the request in parallel to other 2 servers. This was our first solution but Serializer became a single point of fail. The solution could be flooding: each server floods the request in parallel but this would cost 4 extra messages and process even without failures. Our current approach is simpler and more efficient. Each request contains a list of servers which already deliver it. While all servers of this view don't deliver this message, the server forwards the request to next. The last server to know this message on current view, deliver the request to client and ack to previous server on view. In worst case scenario without failures this algorithm provides a reply after 4 messages between servers.

4.3 Fault Tolerance

Our goal is to provide 2 servers failure tolerance without client retry. Our chain replication protocol supports *n-1* failures. When server retrieves the request, it is forward to circle until all servers deliver it. To avoid failures, each server keeps the request until next server ack (response sent to client). If next server fail, the algorithm pause the request, notify other servers to update them view (including a 2nd try to failed server) and then recalculate if all view servers know this message. This system is vulnerable if request retrival and next server fail without let 3 server to know that message. However, the filesystem its consistent and this difference will be solved by copy on boot protocol.

4.4 Thread Operation

Open and close operations are equivalent to append and remove a client ID on a file name, they are write operations. These operations are bound to a unique filename so we can perform this actions in parallel.
Metadata servers have 3 synchronization layers: Namespace, intra-namespace and internal. Since system use 6 namespace partitions, there are 6 different sequence numbers being used in parallel. Inside each namespace, the request is accepted if it has the next namespace partition sequence ID and this filename is not being used by other thread. For reliability and load balancing management reasons, each server core provides mutual exclusion on operations with same name. This design allow 6 parallel request stream, each stream doesn't look if filename it's not in use.

5. Data Servers

5.1 Handling entry and exit

All data server entry and exit are coordinated by metadata servers. Each data server must be registered at metadata servers before replying to clients. The registering request includes information regarding the data server and also the list of its current localfiles. This is done like a common metadata server transaction where metadata server register an uniquely ID generated by dataserver. This process avoid conflicts caused by delayed or lost requests preventing 2 registries for same server. The request is done in a round robin way with a predefined timeout until the data server gets registered. The metadata server adds the new server to the server pool, commit the changes to the other servers and then send an acknowledgment back to data server.
When a data server wants to shut down properly it sends an unregister request to the metadata servers. The metadata server is then responsible for hiding this data server reference from all the files that point to this data server. In this way the metadata server will reply only with available servers avoiding timeouts caused in the clients when accessing the data servers.

5.2 Failure and Recovery

As all the files are persistent on disk, when a data server recovers from failure it will send its localfiles attached to the register request. If the register request was successful the metadata server verifies the references, if there are deleted files it will reply with a list of them to the data server. This is useful to avoid space leaks.

5.3 Read Requests

We design a simple data server. This data server just keep a simple tuple: version number and file content. Our system provide two consistence models: *Default* and *monotonic*.
Using *Default* mode the client just reads the quorum specified by metadata server. Since quorum is established by the client, there is no warranty that the number of writes (W) is less than number of reads (R) and W+R>N, with N being the Number of replicas. It means that a client can read a version older than the last one.
The client application will make a parallel read quorum that was specified in the file's creation. The read process only return when is obtained the correspondent

number of reads from different data servers.
After obtaining all the file replicas, the client application verifies which is the more recent replica and reply it to the user. In the case that some data server doesn't reply until a pre-determined time (Time Out) the correspondent thread will try the next replica of the list in a Round Robin way but excluding the servers that are been used by other threads. In the case that all the possible servers were tried without sucess N times, it will be created a new thread that will contact the metadata server trying to obtain an updated version of the data server's list. This process is important in the case when there are servers that recovered from failure and the clients list is not updated. This is a problem that may cause starvation.
Monotonic semantics ensures that clients must always read a version equal or higher than the version that they have read before.
The client application will always store the last version of the read file. In this way all quorum reads will need to be at least equal to the last version. To achieve this, every time that a thread reads a lower version it will choose the next server from the list trying to find an updated versio in a Round Robin model.

5.4 Write Requests

In a way to guarantee that concurrent writes made by different users doesn't cause inconsistent reads and lost of data caused by overwrites, the metadata servers are responsible to coordinate the write requests order. When the client application wants to write, it makes a request to the metadata server asking for excluively access to the data server i.e. asking for the locking of the file until the client finishes the write. Before allowing the client to write, the metadata server will check the version that the client sent and see if it is more recent that the one it have stored. If it's equal or older, the metadata server reply a message to the client with the indication that the client should read again the file in a way of obtaining a more recent version. It's a responsability of the client application to decide if it wants to overwrite, losing data, or to make a monotonic read in an order to have the last file's version before writing the new version. This assures that every writes has read last version before writing a new version. The concorrent writes are also solved with this approach.

5.5 Load Balancing

Our system provide 2 kinds of load balancing: metaserver requests and dataserver load.
Since all metaservers servers know the view members, each server just use a deterministic algorithm to decide which server its the partition Serializer. Using 6 partitions with 3 servers allows 2 partitions for each server. If one server fails, then 3 partitions each server. The number of partitions allows perfect load balancing. We assume that all metadata is accessed with same frequency and the namespace is well distributed. Metadataservers don't provide a use based load balancing. All metasevers servers can retrieve requests and reply to client so it's well balanced.
Each file has a different read/write so the load on each dataserver depends on files. We propose 2 load balancing solutions: **File Migration** and **Quorum Change**. On each view, a single metaserver is responsible for monitoring and leverage the best data server load balancing. To do it, the metaserver requests each dataserver file statics (read/write rate and traffic). This file statics allow engine to decide which dataservers are overloaded. If file read rate is to high, then metadata increase number of servers and the write quorum (**Quorum Change**). If one server is overloaded and other is not, then it can move the file.
This changes are made when file is not open. This requests are exchange as client requests through all metaservers. The internal synchronization mechanisms lock the load balancing request thread until file is not open by any client.

6. Measurements

In this section we present a few micro-benchmarks to illustrate the bottleneck inherent in the PADI File System architecture and implementation.

6.1 Micro-benchmarks

We measured performance on a PADI-FS cluster consisting of 3 metadata servers, 2 dataservers and 2 clients. All machines where configured with Intel Dual Core E8400 3.00 GHz, 2GB of Memory and 1 Gbps ethernet connection (ping latency < 1 ms). We split the following discussion into two parts: throughput and latency of requests. We used log4net log framework to log all messages and requests and Visual Studio Performance Analysis tool.

6.2 Throughput

30 clients read simultaneously from the file system. Each client reads 3MB text string from its own file. This is repeated 100 times. The read throughput is 120Mbit/s from a single server. The writing throughput is 70.6 Mbit/s. We tried 54MB file transfer to

Operation	Latency (ms)
Create	215
Open	183
Close	50
Delete	50
Read	159
Write	64

Table 1. Latency times using cluster

Operation	Latency (ms)
Create	152
Open	91
Close	71
Delete	60
Read	80
Write	60

Table 2. Latency times using single machine

one dataserver. The data was transfer in 1589ms, equivelent to 274.08 Mbit/s. These data rate are very good because this is a gigabyte ethernet and client had exclusive access to data server. I were not able to check the data server maximum throughput since hard-drive slowed down and crash the program.
We used 30 clients with 100 metadata operations to explore our solution limit. Our system (described at beggin) replied to 3 000 request in 6 minutes: 120 ms each request. Each client make requests on single filename.

6.3 Latency of requests

Based on algorithm structure we predict an average of 5 messages between the 3 Meta Servers + message from client and response to client. Experimental results proved this result values.
We performed a first evaluation on a single machine. Using a single client, single data server and 3 metadata servers, each one running as single process. We considered this results as base performance 6.3. We considered this case as the best case scenario.
The results in our 7 machines cluster is different.

6.4 Performance of barriers

We tested successfully our client random access distribution: using just 2 servers as front-end makes the 3rd server to overload and break the system with (30 clients) Then we change the system to 3 front-end servers and the results were better: 43 clients within same time.
We used Gigabyte ethernet but network connection was not a limitation. Data-servers generate a average traffic of 5Mbps.
The main limitation was the remote calls. Since these requests are not CPU intensive, the remote calls that system needs to agree on values to provide fault-tolerance are the main CPU consumption cause. Our suggestion it's to change this system to 2 servers and separate the namespace partitions, providing just 1 server fault but better performance.
Our protocol could be improved if we forward the request from front-end strait to serializer, reducing the message average to 3 messages.

6.5 Load Balancing

We measured the CPU and Memory usage on each metadata server 3. The load between metadata servers is well balance: CPU and memory usage is similar. These were the expected results since metadata perform the same tasks when filename space and clients are well distributed.
Our load balancing algorithm performs well. However it doesn't balance the hard drive load. The size of file is considered on traffic but not managed. However this was not our goal. Changing the quorum allowed to reduce clients delay, reduce the server load and decrease the network traffic. Our algorithm performs after each close with minimum timespan of 5 minutes.

7. Conclusions and Future Work

PADI-FS is a distributed file storage intended for high availability, reliable and large-scale data storage. Its design is based on well-known Google File System and uses concepts as Chained Replication Protocol, serialization node and monotonic read/writes.
We suggest as future work an improvement on load balancing algorithm to take into account the recent past actions and leverage when it should start. Chained Replication is simple but hard to implement in a resilient mode. We suggest an approach with 2 phase differential copy: first copy all data while servers are running and then copy just the updates.
Our name space approach allows parallel requests. Our replicated metadata cluster it's resilient against server faults. This provides file system high-availability and reliable. We move the complexity from data servers to client application to improve system scalability. The solution design will archive the goals: high-availability,

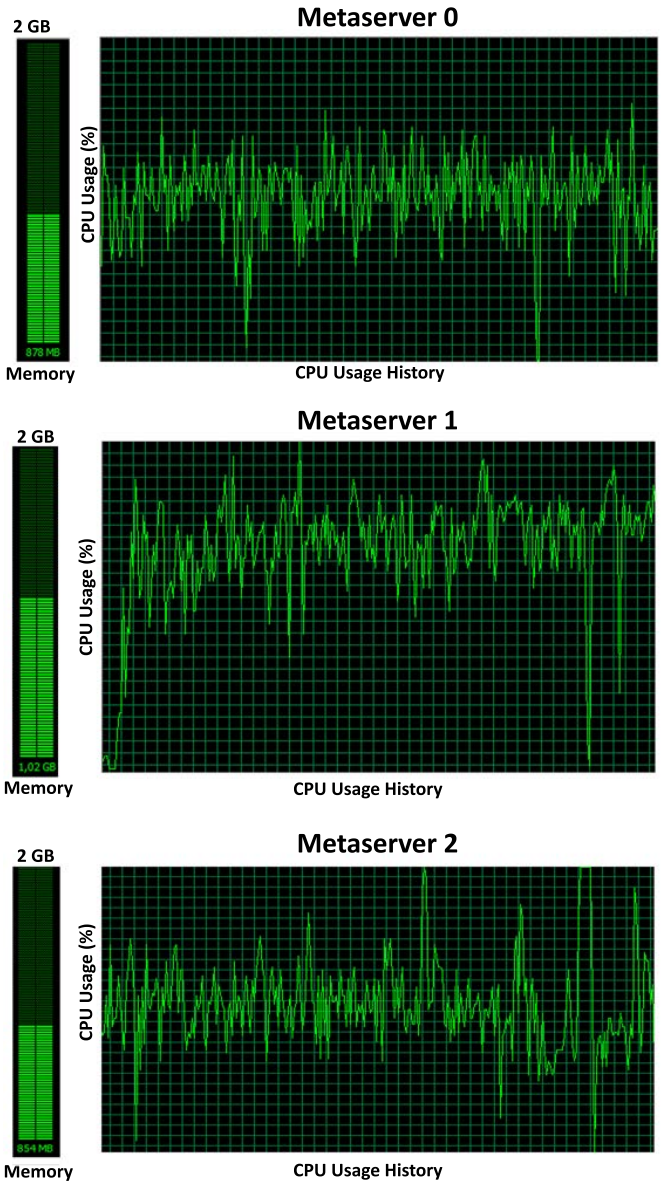

Figure 3. Metadataserver Usage

reliability and a user oriented (not big data) file system optimized for reads and overwrites.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[2] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[3] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. 6:7–7, 2004.