Z File System (ZFS) Specifications (v 00.01)

Arsalan Zaidi

11th August 2001

Abstract

This is the initial draft of the specification for a new distributed/network file system. The information presented here is **very** tentative and subject to change without notice. Although no knowledge of CMU's CODA is required, it may be helpful in understanding some of the ideas behind ZFS.

Introduction

Overview

ZFS is a **network of automatically replicating file servers**. The focus of the system is on scalability, security and usability. All data servers in ZFS hold (or have the ability to hold) identical copies of a master file system. For example, if a file named /home/azaidi/specs.txt exists on one data server, it can be accessed from a suitable authorised client from any other data server. The data servers don't nessasarily hold the entire file system, neither is it guaranteed that they will hold the most current version of the file in question, but they can access and retrieve the most current version of any file on the system automatically. Like CODA[1] (a file system developed at CMU and the source of many of the ideas behind ZFS), granularity is generally at the file level. Clients request files, cache them at their end and perform operations upon them. Any changes made to the file are local.

[1] The CODA home page. http://www.coda.cs.cmu.edu
Introductory Document. http://www.coda.cs.cmu.edu/ljpaper/lj.htm

Once the file is closed on the local machine, it is sent back to the data server for re-integration.

ZFS is *very* strong on security. Authentication is achieved using username/passwords (for the clients) and certificates (for the servers), and all data is *always* encrypted. SSL is used by every component in the system. In addition, ZFS tries to maximise bandwidth utilisation by both compressing data when ever it can and also by using an **rsync**[2] based protocol. Data compression can be turned off, if desired, to conserve CPU cycles[3]. Encryption is mandatory.

One of the aims of the project is to make ZFS as stable, secure and reliable as possible, both in it's design and implementation. In addition, the system has been designed from the ground up keeping scalability in mind. Ease of administration was also kept in mind and almost every aspect of a ZFS network can be administered from a central location.

The implementation will largely use off the shelf components (both hardware and software) and an effort as been made to make sure that a network can be set up quickly and cheaply. Though not required, it should be possible to increase performance if specialised hardware (like hardware load balancers/ switches) are used.

Applications

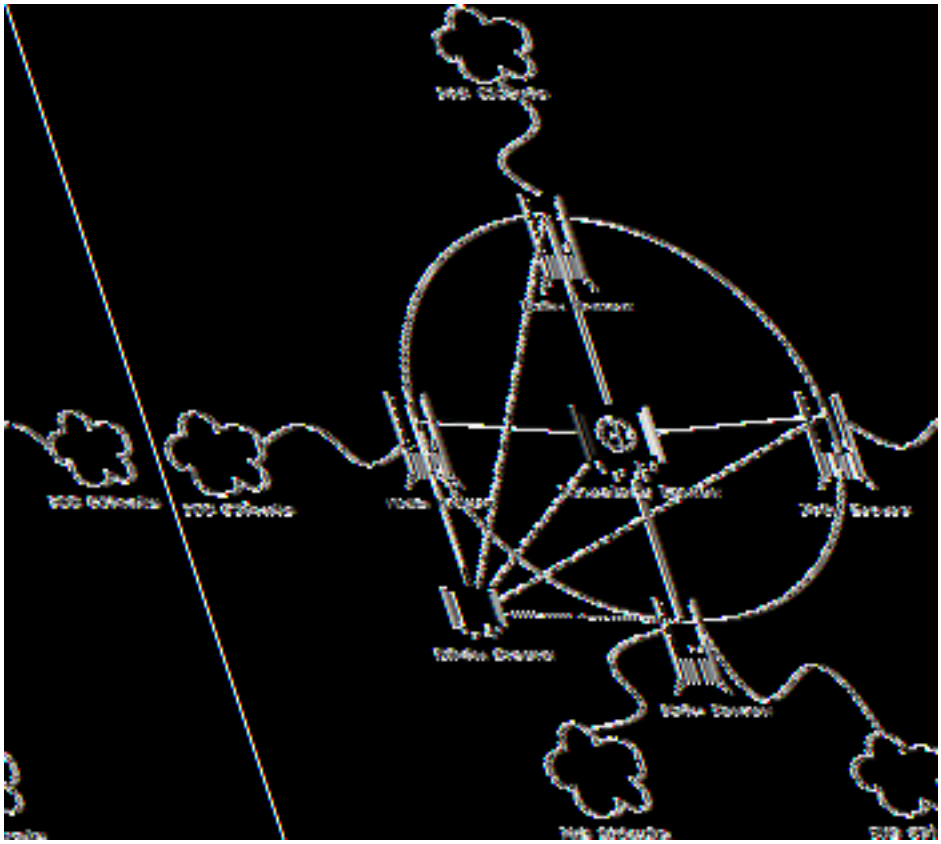
While conceptulising the design, some possible scenarios were kept in mind.

1. A large global system of heavily accessed servers. E.g. Red Hat's distribution mirroring system.

2. A large University/Corporation.

[2] http://rsync.samba.org/

[3] An assumption of the design is that network bandwidth will always be more valuable than CPU cycles.



ZFS would be ideal for a large mirroring network. All one would have to do is update the files on one server, and it would automatically propagate itself to every other data server on the network that services a request for it. This means that the file will *only* be sent across the network to a data server if some client requests it from that server. Since files are only copied when requested, network bandwidth is conserved and so is data server disk space. At the same time, since you only have to upload a file to any one server, administration and server replication become trivial. The Data servers could be globally distributed, with the various subsidiary FTP servers logging in as ZFS clients.

In addition, ZFS would be a good solution for a university with a large number of computers. The filesystem may be mounted on a large number of thin client machines and users will be able to log in and use any machine as if it were their own. Additionally, it would be great for thin wireless clients, web tablets and PDA's. If the building has some kind of wireless setup, then thin clients with

very little storage can access vast amounts of information off their ZFS mounted filesystems. It would also be great for corporations which require their workers to use portable computers away from the office. One of ZFS's features is *automatic integration* and *disconnected operation*, which means that any files changed while the system is disconnected from the network will be automatically uploaded once the network connection is re-established. In addition, files can be *hoarded*[4] before disconnection so local cached copies exist. If proper care is taken, the user shouldn't notice any difference between connected and disconnected operation.

ZFS could theoretically replace FTP too. It's more secure by design and integrates the remote file system with the local one, making file transfers and management much more intuitive. Now all you need to do to update your website is simple **mount** the remote system and **cp** your files to it.

System Components

Client: A Client is any computer running the ZFS client module[5]. The client module will be transparent to the user. All one has to do is mount the appropriate ZFS server to a sub-directory and then cd to it to access it. There can be any number of clients, all of whom mount the remote partition under /mnt/zfs or some other suitable mount point.

Data Server: A data server is a computer/process that handles the actual storage of the different files on the ZFS file system. Clients talk to data servers and request files from them. There must be one or more data servers

Directory Server: The directory server handles the data look up issues. It keeps a track of the location of all the files on the system. Directory servers act like giant look up tables and are accessed by the data servers. There can be only one conceptual directory server, although the load may be balanced across several physical machines.

Admin Server: The admin server carries out various sundry tasks. It asks like a load balancer and allocates different data servers to different clients depending on various parameters like location and server load. It

[4] Automatic reintegration, disconnected operations and hoarding are concepts borrowed from CODA.

[5] The word 'client' throughout this document, refers to the software component running on the user's system and **not** the user himself.

also monitors data servers and remotely restarts them if they fail or if their config files need to be updated. In addition, it may also maintain logs and load statistics, a central repository for all authentication information, the time server which syncronises all the machines (for accurate timestamps) and software to handle the doling out of certificates. There is only one conceptual admin server, although the load may be balanced across several physical machines.

History

The author of this document started to think about implementing a new distributed file system while looking for ideas for a class assignment. Finding that people were widely dissatisfied with NFS, he looked around for alternatives and was pointed at AFS and CODA. Having read all he could find about CODA, he was rather impressed by it, but could immediately think of various ways to improve upon it.

ZFS was initially going to be 'CODA with Write Once, implemented cleanly', but one thing lead to another and ZFS and CODA are now quite distinct. The reader is encouraged to learn about CODA as ZFS borrows many concepts from it.

Why do we need another DFS ?

There are already several distributed file systems about[6]. However, none seem to be all that ZFS is. NFS is quite remarkable and completely transparent. However, it stateless nature means that scalability is an issue. CODA, with it's broadcast writes is not suitable in many situations. In addition, if the data distributed by the file system is going to travel over public networks, both security and bandwidth efficiency become issues. ZFS is an attempt to bring together the best out there and come up with a solution which better fits todays requirements and which can work reliably over the Internet.

[6] A list of some popular distributed file systems and information abut them may be found in a useful paper available online at http://www.extremelinux.org/activities/usenix99/docs/braam/braam.html

License

It is the authors intent to release his implementation of this specification under the Apache License[7].

The License for this document is based on the IETF license.

Basic Design

Some Design Decisions

Priorities

1. Scalability : This is of utmost concern. The system must be able to scale to accommodate a large number of users. It must be able to distribute load cleanly and in an optimum manner.

[7] Interested parties may visit www.opensource.org for the text of this and various other Open Source licenses.

2. Performance : The file system must be snappy and should use every trick in the book to increase file transfer speed.

3. Integrity : The system must perform reliably in less than ideal conditions and under heavy load. It must be able to withstand partial network outages and partitions.

4. Portability : The system should not discriminate on the basis of the client's OS. In addition, it should allow the implementors to run it on any platform they choose; within limits.

5. Adminsterability : The entire system must be easy to administer and setup.

Implementation Language

There were only three serious candidates.

1. C

2. C++

3. Java

The author agonised for weeks over this decision and finally choose Java.

Although both C and C++ are useful languages to work with, the clincher was the inherent portability of Java. Writing a large project in a non-OO language is harder than writing it in one that has inherent support for OOP. However, C++ lost out because writing portable multi-threaded code in C++ is a decidedly non-trivial task.[8] Java shines here, because it is easy to create multi-threaded applications in it. In addition, the language is inherently simpler than C++ and has less hidden pitfalls.

However, Java too has it's warts. It is much slower than C++ and harder to optimise. Despite the marketing hype, it *is* possible to run into portability snags. In addition, the language is under the complete control of one corporation and is still not entirely standardised. There aren't as many free tools available for Java, or

[8] Interested readers may read the C++ programming guidelines for Mozilla (www.mozilla.org) for further details on just how restricted programming becomes if one wants to stay portable.

as many free libraries available either. Besides, Java seems to attract non-system programmers and thus finding people to help with the project may be harder than if the implementation language were to be C++ or (even more so) C. Both these languages are widely understood and used by serious software hackers, with C being pretty much the *lingua franca* of the computing world.

Although the authors implementation will be written in Java, there is nothing about ZFS which is language specific. In fact, the client side software will have to be written in C because of its close interaction with the client operating system.

Plain TCP/IP vs RPC

Implementing a protocol using simple send() and recv() offers a lot of control. However, it is tedious and error prone. It would be far cleaner to use a well defined interface, which is what RPC gives us. There may be some loss in efficiency, but the gains are worth it.

CORBA vs RPC

It initially seemed that the CORBA ORB would fill the role of the Admin server quite nicely. Unfortunately, for the ORB to be useful, special non-standard extensions would have to be written and the time spent in extending a standard in a non-standard way would not be worth the effort. There would be no gain and thus the idea was abandoned.

XML-RPC vs Other implementations

XML-RPC is a simple RPC implementation. It uses XML to format the data and can be used from behind proxies and firewalls without any change to their configuration. Although heavier than many other forms of RPC, it is simple to use and it's ability to tunnel through proxies will prove invaluable for clients on corporate networks.

Component Philosophy

As far as possible in the design and implementation of ZFS, the author intends to use ready made libraries and systems. This is to speed up the process of de-

velopment and simplify it. In addition, it lowers the learning curve required to understand ZFS and makes is easier to maintain. Responsibilities are effectively farmed out. For example, SSL is used throughout for security and XML-RPC is used for communication. Another example is that ZFS does not define any file system to use on the Data Servers, but it is expected that some sort of Journaling File System[9] will be used.

A Walk Through of the System

One way to explain how ZFS works would be to provide a bit of a virtual walk, though a series of interactions with the system.

Mordred wishes to access his system and edit some files related to his upcoming project. He sites down on his home pc and boots it up into the Linux partition. He then mounts the ZFS volume under /mnt/zfs and gets to work. The ZFS client software is present on the system as a kernel module plus a user level manager. When the ZFS volume is mounted, it checks the local configuration files and retrieves the name of the Admin Server. It then connects to the Admin server and give it its location[10]. The admin server finds a data server in it's database which is closest to Mordred. In addition, it uses other statistics present with it to find the server which is the most lightly loaded. It returns the name of this data server to Mordred's ZFS client. The client now connects to the data server and logs in securely. It also sets up all encryption information using SSL.

Mordred now starts up Emacs and opens up the file he's interested in; /mnt/zfs/home/mordred/project.txt. When the open() function is called, the ZFS client first checks the local database (nothing more than a glorified look up table) for the file. If the file exists, it then checks the status of the callbacks and see's if the callback on the file has expired. If it hasn't, that means that the file in the local data cache is the freshest. It opens up the file and returns a handle to Emacs.

If on the other hand, the file is not available or if the callback has expired (indicating that this copy is no longer the freshest), the client connects to the data server and requests the file. Files are identified using their full path and name. If the freshest version of project.txt is available on the data server (i.e. its callback

[9] For example, Reiser FS or EXT3 on Linux

[10] Optional and entered by the user in the appropriate .conf file. The software will use BGP to find the most optimum Data server for the client and the client locational information is just a hint. However, this is configureable.

promise has not been revoked by the directory server), then it returns the file (after compressing and encrypting it) to the client[11]. The client copies the file into the data cache (a large temporary work area on the local disk) after unencrypting and uncompressing it and returns a handle pointing to it to Emacs. A callback with the server is automatically set up. If there are any changes made to the file on the ZFS network, then the callback will expire and the client will know that the copy in its cache is no longer the freshest. It will then retrieve the freshest copy when the user demands it. In addition, an entry is made in the local database listing the file name and the timestamp on the file. The timestamp information comes from the server.

What if the file was **not** available on the data server? Or what if the data server knew the file it held was **not** the freshest copy? In that case, the ZFS data server would contact the Directory Server and give it the full path and name of the file the client requested (information about which directory server to access is retrieved from the admin servers on startup). The directory servers contain a large tree which lists *all* the file available of the ZFS network. It also lists the names of the data servers which contain the freshest copy of each file. The directory server will look up its file tree and return the names of a maximum of four servers which hold the freshest copy of the file. If there is no mention of the requested file in the Dir server, then it will return a 'file not found' error[12]. The data server now directly connects to the first of the data servers returned by the directory server and requests the file (after authentication and encryption). Once the retrieval is over, it starts a new thread to send the file over to the client and concurrently connects to the directory server saying it has the freshest copy of project.txt[13]. The directory server checks the timestamp (retrieved along with the file by the Data server[14] and sent by it to the directory server) and if it's identical to the ones already in the tree, it adds the server to it's list of servers which have the freshest copy of project.txt.

[11]The client can either ask for the entire file, or use the rsync algorithm to get only those parts of the file which have changed. The former method will be used when the file is being downloaded for the first time.

[12]At this point, if the Data server itself has a copy of the file available with it, it should add it to the Directory in the usual manner.

[13]The reason a query about a file doesn't automatically result in an addition of the querying server to the Directory as a carrier of the latest copy, is because a query about a file may not always result in it's download.

[14]In this situation, the Data server acts just like a normal client. It copies the timestamp from the source Data server and stores it in a database on it's system. The only time a Data server uses a timestamp from it's own clock is when it's notifying the Directory server about a new or modified file.

A callback is automatically established. Now if a new copy of project.txt appears on any other data server, the Directory server will have to be told about it. It will update its tree and break all its callbacks for that file. Our Data server will then automatically come to know that its copy cached of project.txt is no longer the freshest.

The reason both uploading the file to the client and updating the directory server proceed concurrently is as follows:-

If the file in question is a log file being updated rapidly, like once every second or so, then by the time the data server downloads it, a new copy of the file would have appeared on the network. When our data server then tries to establish a callback using the (now invalidated and expired) timestamp, it would be told it does not have the freshest copy. It would try once more to download the freshest copy and the cycle would continue. By sending the file over concurrently, the client gets a copy of the rapidly changing file as it was *at that moment*, when the data server tries to establish a callback, it is rejected. Now the *next time* someone tries to access the same file, the data server will re-fetch it. Notice that we're not going into a loop here like we would have otherwise.

So anyway, the data server retrieves the file and forwards it to the client. Mordred continues on his merry way, making whatever changes he desires. Once he'd done, he saves the file and closes it. On the close, the ZFS client takes the newly updated project.txt and (after compressing and encrypting it) places it in the *reintegration que*. Another thread scans the que and when it finds a file there, it tries to copy it back to the Data server. If it can't do this, because the network is down for example, it backs off for a bit, and then tries again later. In addition, it tries to optimise the que by making sure that there's only one version of a file there if Mordred opened and closed the file many times, making changes every time. In addition, the rsync algorithm is used to drastically speed up the upload of the file.

When the file is finally uploaded to the data server, it contacts the directory server like before and informs it that it has the freshest copy of project.txt. The timestamp on the project.txt file (added by the Data server) is the freshest, so all other data server names are purged from the list and their callbacks for this file are killed and the name of our data server is added in and a callback established. Since it is assured that our data server has the freshest copy of project.txt, it now ends the transaction by establishing a callback promise with the client.

Advantages

1. Faster than CODA on hardware which doesn't support broadcasts, because writes are one to one.
2. Able to work on networks which do not support hardware level broadcasts (except in the special case of the internal network when there are multiple Directory servers).
3. The clients are not relied upon to update all the data servers, just the one they're connected to.
4. Optimised for dial-up or slow links (Rsync updates/Aggressive compression).
5. Clients can work from behind restrictive Proxies and Firewalls.
6. Client caching and disconnected operation (data hoarding).
7. Strong security and authentication (both data streams and data in the cache are encrypted).
8. Fault tolerant.
9. Load Balanced.
10. Scalable.
11. Easy to administer.
12. Written in Java for portability.
13. Open Source.

Disadvantages

1. The implementation is complex
2. Untested and unproven compared to other products already available.
3. The centralised nature of the system, revolving around the Directory server makes it vulnerable to network partitions and attacks on the Directory server.

Overview

The Directory Server

The Directory Server is usually made up of a bank of machines. It keeps an up to date version of the file system tree in its memory and makes and breaks callbacks with the data servers. It is the hub of the system and the most sensitive and complicated part of the the entire network.

The reason for introducing the concept of a Directory server, even though it reduces scalability and increases complexity is to increase the speed of file retrieval, eliminate broadcast writes to servers and paradoxically, increase reliability. A file system which emulates the Unix paradigm cannot operate on the P2P model[15]. It is essential that the location of each file be known exactly.

The Data Server

The Data Server's job is relatively simple. All it has to do is service requests and send data over to the requesting client. It also has to keep in touch with the Directory and Admin servers.

The Admin Server

The Admin server handles various sundry tasks. It verifies certificates, redirects clients, holds the config files for Data servers, monitors the health of the data servers and hosts the network's NTP server.

The Client

The Client is made up of two components. There's the kernel module and a more portable user-level module. As much work as possible will be done in user space for reasons of safety and portability. The client's job is quite complex. This is because not only must is manage the sending and receiving of files, it must also incorporate various techniques to optimise the entire process.

[15] For an example of a peer to peer network, see www.freenet.org

Clients save data to an integration que, from where the files are trickled to the data servers for updation. Clients must detect network conditions and react accordingly.

Dealing with Network Outages and Partitions

The entire FS system must be able to deal gracefully with network partitions and outages. On a global network like the internet with an 20% downtime[16], you have to prepare for the inevitable. Here's how each component reacts to network partitions and partial reachability.

Admin Server

If the Admin server is unreachable, clients will be unable to discover the address of Data servers and thus will find it impossible to access the ZFS network. However, it is possible for a client to cache the address of the Data server used during its last session and use it again. That this will always work is not guaranteed. If the certificate server is also hosted on the Admin server, it will not be possible to authenticate any of the servers.

In addition, newly rebooted Data servers will not be able to go online because they depend on the Admin server to provide them with the address of the Directory server. A newly restarted ZFS system will find that its Directory servers cannot build up a file system snapshot because they do not know the addresses of the Data servers.

However, if the Admin server goes offline for a short time after the system is already setup, currently connected Clients/Data servers/Directory servers will not be affected.

Directory server

If the Directory server is unreachable, then Data servers and their clients will be directly affected. Data servers can react in two possible ways.

[16] ???Missing Reference???

1. They can continue to serve files cached on their disks, with no guarantee of freshness. Requests for files they do not hold will fail.
2. They can immediately stop serving files and respond with a failure code to every request.

The reaction of the Data servers is configurable.

Data server

If a Data server is unreachable, the Admin server will soon discover the fact and stop redirecting clients to it. Currently connected clients will automatically be disconnected and will (after several retries) contact the Admin server, which will redirect them to another Data server. File uploads and downloads in progress can be restarted on the new server.

Clients which had just finished uploading a file will proceed as explained on page 22 under Possible Problems.

Client

If a client is unreachable, the Data server simply closes its connection to it.

Detailed information about the different components

The Admin Server

The Admin server carries out various tasks.

1. It redirects clients to the nearest, most lightly loaded Data server. To do this, it uses BGP[17] and location information sent in by the client.
2. It monitors the health of the Data servers and remotely restarts them if they fail. In case that is not possible, it alert the administers.

[17] Super Sparrow. http://supersparrow.sourceforge.net/ss-0.0.0/index.html

3. It monitors the load on the Data servers and uses information returned by them to calculate which Data server should receive the next batch of clients.
4. It holds the Data server configuration files. The administrators need only make a change on the files held on the Admin server and the information will be automatically sent to the Data servers. In addition, the administrators can stop, start and restart servers, monitor their health, and change almost every variable that affects them from the Admin server console.
5. [Optional] It hosts a Certificate server which helps clients authenticate Data Servers.
6. [Optional] It hosts an NTP server which allows the networks servers to sync up their time.

Admin servers may be load balanced or backed up[18] in anyway the administrators think nessasary.

The Directory Server

The Directory server handles the central file system tree. It knows where each and every file on the system is stored. Data servers are in constant touch with the Directory server and query it for this information. They also update its tree by sending in information when a file is changed or added.

Setup:

The Directory Server is actually made up of a bank of several machines[19]. Each machine has a minimum of two network cards, with one network card on a network which supports hardware level broadcasts. All connections from the outside world come in through the other network card. These connections can be load balanced using a load balancing switch or a DNS server. The second network card is connected to an fast internal network which links up all the Directory servers. This should be something like a 100MBps or Gigabit Ethernet network[20]. Here's how the system works.

[18] UltraMonkey. http://ultramonkey.sourceforge.net

[19] Although it is possible to run a ZFS network with just one machine.

[20] The machines should be wired up using a bus topology. This is one place where the brittleness of such a network is actually useful. No matter where the network breaks, all the Directory servers will stop working; immediately signalling a problem. This reduces to some extent fragmentation due to netowrk partitions.

When ever there is an **update** request, the Directory server sends a broadcast (after conducting sanity checks) on the internal network informing others of the update. It itself picks up the data for the update from the wire; it doesn't add it directly[21]. Each update packet is numbered with a unique, incrementing counter. This counter is picked up from an external, shared resource[22]. The number helps in spotting missing updates.

Directory servers are required to keep track of at least the last 50 updates. Every time a new update arrives, the Directory server checks to see if the counter number is contiguous. If it's a duplicate update, it is silently dropped. If it is non-contiguous, but less than 5 increments away from the last update, it is stored and the Directory server waits for the rest to come in. If the counter is greater than 5 increments out of sync or the intermediate updates don't come in, the Directory server will contact the Boss[23] Directory server and ask for the last *n* updates and apply those to it's tree. If the Boss server itself lacks those updates or if it itself finds that it has missed some updates, a new election is called for where Directory servers without syncronised trees are not allowed to participate. The new Boss is then asked for the updates. In case all the directory servers are out of sync, manual intervention is required[24].

When a new Directory server is added to the pool, it asks the Boss server for the entire tree. While downloading the tree, it keeps track of updates and applies these to the tree once the download is over. It is now ready to honour updates.

Requests for information can be answered by any up to date Directory server. Requests require no traffic on the internal network.

Both requests and updates are load balanced by placing a software or hardware load balancing solution in front of the Directory servers. One simple way to load balance is to modulus the IP address as an integer with the number of the Directory server.

Note that implementors are free to add more network cards and stripe data across these.

[21] This is an implementation detail. By cutting functionality into two logically different units, the code will be more modular. Implementors are free to ignore this advice.

[22] A seperate counter dispensing server, the Boss server which is elected using standard election protocols. Or more simply, a file with a counter in it which is shared using NFS. Each Directory server locks the file, reads in the counter, increments it and then unlocks the file.

[23] The elected leader of the pack. In addition to managing the counter, it also monitors the health of the other servers.

[24] Most likely a complete reboot of the ZFS system.

Start Up:

When a new Directory server is added to the pool, it has to run through a series of steps to initialise itself.

1. The first thing the server must do is find out who the Boss is by broadcasting a request on the internal network. If there is a response, it should ask the Boss for the latest files system tree and keep collecting updates as the data comes in. It should then integrate the two. The server is now online.

2. If there is no response from the Boss, the server should try again for a maximum of three tries. If there is still no response, there is no Boss, so it should start the election process. Once the Boss is selected, it should ask it for the files system tree like in step 1.

3. It is assumed that Boss has been online for quite some time and has the latest file system tree. If this is not true and the Boss is newly elected and does **not** have the latest tree, then it will respond to the requests for the tree with a *wait*. The other Directory servers will poll for the tree at intervals. Until they get the tree, they will respond to Data server requests with an error code.

The Boss server will now start to gather the file system tree through the following steps.

1. It replies to requests from Data servers with an error code. It replies to Directory server requests with a *wait*.

2. It then gets the entire list of Data servers from the Admin server.

3. It now connects to the first Data server and calls a function within it asking it to refresh all its callbacks and to stop serving clients until it is done.

4. The Data server starts with the '/' directory and starts to ask the Directory server for callback promises.

5. The Directory server acts as normal and accepts all the files as the freshest (since he has nothing in his tree to contradict this position) and sets callback promises as required.

6. It then connects to the next Data server and repeats the process from step 3 to 5. The process of connecting to the Data servers and having them refresh their callbacks can take place concurrently.

7. If the files on the next Data server are fresher than the ones on the previous, the Directory server ***ques*** the callback cancels for the rest and sets a callback promise for the present one.

8. Once all the Data Servers are done, it sends off the qued callback cancels. The reason the calls to cancel the callbacks were qued, was because there is no guarantee than the freshest file in the tree while the file system tree is being coallated, will be the very latest one by the time the process is complete. The next Data server could hold a fresher copy. If we expire callbacks everytime, we simply load up the network and the servers for no real reason.

9. Along with the callback expiry messages, we also send a message to the servers telling them to start serving clients.

Care should be taken to make sure the entire process works even if there is only one server.

Note:

The reason this setup is a little complicated is to allow the transparent addition and removal of directory servers. The directory servers are the weakest link in the chain. If they fail, the entire network goes down hard. By allowing nodes to transparently fail and be restarted, the chances of catastrophic failure are reduced and the availability of the Directory system is enhanced. In addition, such a setup allows effective load-balancing.

The Data Servers

The Data servers answer requests for data from clients and from other Data servers.

Setup:

The Data servers are the machines on which the files actually reside. Clients connect to them and retrieve files from them. There is no guarantee that Data servers will hold all the files or that they have the freshest files, but they can

retrieve those files as and when requested and they then cache them for future use. The file timestamps and callbacks are maintained using a simple local database. One Data server can not export more than one ZFS mount point. To have more than one mount point on one physical machine, you would have to start another completely independent instance of ZFS.

Clients are authenticated using standard Unix usernames and passwords. This file can either be a part of the ZFS mount or the information may be stored on the Admin server. Once authenticated, clients may browse those files and directories for which they have permissions.

Data servers rely on the underlying file system for data integrity. It would be best if they were run on some sort of journalling file system.

Data servers may be load balanced in anyway the administrators think best, for example, behind a switch which maintains a per connection link between the clients and the Data servers. All the Admin server knows is that machine name server.somesite.com is a Data server.

The Client

The client is the machine which connects to the Data servers and requests files from them.

The main focus of the client is to try and reduce network utilisation and dependence as far as possible. Clients do this by caching as much as they can locally. In order to make sure that the files they have in the local cache are fresh, they are given callback promises by the Data servers. The Data servers expire these callbacks as and when the Directory servers expire them, indicating a change in the files contents or permissions.

When a client starts up, it immediately tries to contact a Data server (through the Admin server). It then tries to acquire callbacks on the files it has in its cache. It gets callbacks for those files that haven't changed and is denied callbacks for those that have. The client can then either request those files again[25] or remove them from its cache. This process is repeated if the client loses network connectivity, since it is possible the status of the callbacks changed in the time it was offline.

As and when the user requests a file (by trying to read it using some application), the client downloads the file from the Data server, gets a callback for it and stores

[25] This is something that it will only do if the file is in its *hoard list*

it in the cache. The file can then be used by the user. A strange situation may arise where the file in the cache is newer than the file on the Data servers. This may happen if the Data server to which the file had been uploaded crashed soon after, before the file could propagate to other Data servers. In such a case, the client will assume that the local copy is the freshest and continue as normal. The file will be placed in the reintegration que *even if no changes are made to it*.

When the file is closed, it is stored in the cache and added to the reintegration que if its contents have been altered. Note that the largest file that a client can use is limited by the size of the local data cache. If the file in question is larger than the cache, then the openFile call on it will fail. This could be a problem if the client is running on a platform with resource constraints (e.g. a PDA). One solution, which is a bit of a hack, is to mount an NFS directory and use that as the cache. This of course means the user will not be able to derive all the benefits of the ZFS system, like disconnected operations and the faster speeds that come from accessing local resources, however, the other benefits, like scalability, security, locational independence, ease of administration etc. can still be enjoyed.

A local cache which is full will not represent a problem, because files which have not been accessed for some time will be removed to make room for new files.

If the computer is connected to the network, the client will attempt to upload the file to the server using a protocol based on the rsync algorithm. Multiple saves of the file will be represented by only one entry in the que. If the computer is not connected to the network, reintegration will be postponed until such time as it is. The que data is stored on permanent storage to guard against data loss.

In order to conserve bandwidth, an rsync type protocol is used both when uploading and downloading files from the data server. In addition, files are compressed if they will benefit from it.

All data streams to and from the Data servers are encrypted. In addition, the files stored on the local disk can also be encrypted. The former cannot be turned off, however, the encryption of locally stored data can be switched off.

Another job of the client, is to *hoard* data. There are some files which are very important and the user would like a fresh copy of these files to be maintained in the cache at all times. They are not allowed to be overwritten either. The user will use some user-friendly software to inform the client about the files he wishes to hoard. In addition, another application will allow the user to monitor a session and figure out which files have been accessed by the user. This information can then be used to build up the hoard list.

The client itself is made up of two different, but complementary components. There is the user level code (Phobos) which is written in portable C and carries out the bulk of the activities. The kernel level component (Deimos) is specific to the OS kernel and seamlessly integrates the user level component into the system[26]. The reason for splitting up the software into two is to increase portability.

Minimum requirements.

Possible Problems

1. Information may be stored on only one server. If that goes down, the user *may* lose that data. The situation can be detected if the client attempts to re-read the file while using another Data server. The file will either be inaccessible (because the Data server which held it is gone) or an older version may be found on the network. The client in this case assumes that the file in its cache is the freshest and passes the user a handle to it. The file is placed in the reintegration que even of no changes are made to it. However, if the file has been deleted from the cache, there is no way to retrieve it.

Security

Authentication

Authenticaition will be carried out using SSL and Certificates for the servers and username/passwords over an encrypted SSL link for the clients. Clients are allowed three tries and if they can't authenticate by then, the connection is closed. If there are a lot of failed attempts from a single IP, it is automatically banned.

Encryption

Encryption will be carried out using SSL and the various protocols it supports. The SSL connection is computationally expensive to set up and so clients will

[26] Obeying tradition, the two parts of the client system are named after heavenly bodies.

be required to *keep-alive* the connection for as long a they are connected. If a client attempts to make a new SSL connection for each request, it will be banned. This is to prevent a DOS[27] attack where the client can tie up the server by forcing to it continiously make new (and expensive) connections. In addtion, certain IP addresses can be banned as well.

SSL sessions can be maintained across connections. One session may be used over several connections sequentially or simultaneosly.

Possible Attacks

< <FIXME> >

Additional Services Required

Network Time Server running on the Admin server (or elsewhere) for syncronization.

A Certificate server running on the Admin server (or elsewhere) for authentication.

< <FIXME> >

Network Setup

100MBps/GigaBit Ethernet bus network for the Directory servers.

Software/Hardware Load balancer before the Directory servers.

< <FIXME> >

[27] Denial Of Service.

The ZFS API

The Admin Server

Exposed

getDataServers()
authenticate(username, password)

Used

The Directory Server

Exposed

getServersForFile(filenamewithpath)
setFreshestFile(filenamewithpath, timestamp)
authenticate(username,password)
expireDataServersList()
getFileSysTree()
whoIsBoss()

Used

The Data Servers

Exposed

readFile(filenamewithpath)
writeFile(filenamewithpath)
writeFileBlocks(filenamewithpath, blocksize, checksums)
openFile(filenamewithpath)

closeFile(filenamewithpath)

deleteFile(filenamewithpath)

renameFile(filenamewithpath, newfilenamewithpath)

linkFile(???)

symLinkFile(???)

makeDir(dirnamewithpath)

removeDir(dirnamewithpath)

setFileLock(filenamewithpath)

releaseFileLock(filenamewithpath)

expireAllCallbacks()

expireCallback(filenamewithpath)

setCallback(filenamewithpath, timestamp)

setBulkCallback(filenamewithpath[], timestamp[])

getLoad()

authenticate(username, password)

Used

The Client

Exposed

expireCallback(flenamewithpath)

read(flenamewithpath, offset, size)

Used