High Availability Content Under Linux

Horms (Simon Horman)
horms@zipworld.com.au.

To be presented at
The 1998 Atlanta Linux Showcase
Atlanta, Georgia, USA
Friday 23rd - Saturday 24th October 1998

http://linux.zipworld.com.au./

"A distributed system is one in which I cannot get something done because a machine I've never heard of is down."
Leslie Lamport

I would like to acknowledge my employer Zip World www.zipworld.com.au for their assistance and patience that enabled this presentation to come together. I would also like to acknowledge the assistance of Computer Generation Incorporated www.compgen.com who are sponsoring my trip to the 1998 Atlanta Linux Showcase to present this paper. Finally this paper would not have been possible without the support of my friends. In particular Little Miss Fun, Mandrake, Raster, Mr. O'Brien, Gus, Miss Kim and K have helped to bring this work to completion.

Abstract

Redundancy of all forms is an important issue when administering servers. One aspect of this which is of particular importance is having highly available content. If your HTTP server dies your site shouldn't go down with it. This area is particularly challenging as solutions must provide for both redundancy and high performance.

One way to achieve high content availability is to house the content on a file server enabling you to hot-swap machines such as FTP servers using methods such as ARP spoofing. This solution poses interesting performance issues but the file server is still a single point of failure.

Regardless of whether the content is housed on a separate server or not, providing duplicate file servers is also important and synchronisation between servers can be implemented in various ways. Again performance is a major issue particularly during any synchronisation procedures.

Contents

1 Introduction

In a networked environment high availability of services is very important. Creating redundant content is a diverse topic and each application has its own nuances. This discussion will be made from the point of view of an Internet Services Provider where a typical application is to create a fault tolerant HTTP Server. This has different requirements to providing files for a group of workstations though much of the discussion will be applicable to this and other applications too.

There are two key issues involved in creating redundant content, replication and access.

1. Replication involves duplicating data by some means so that a failure on the box where files are housed will not mean inaccessibility.
2. Access involves providing a client with requested files whether this be of a local disk or across a network.

This discussion will cover freely available software solutions under Linux. While commercially available solutions such as the AFS and DFS[Tra] file system are interesting, I would much rather concentrate on options that anyone can run free of charge. I will also not be dealing with hardware solutions such as hardware bus sharing [HM98] or RAID disk arrays [1] as the software aspects of this issue are already numerous.

2 NFS

NFS is an access mechanism. Briefly, it enables files on one host to be mounted as part of the file system on another host. Although NFS itself is not redundant, it does bear some investigation due to its wide spread implementation and some new developments that increase NFS performance.

[1] RAID arrays can also be implemented in software but discussion of this is best left to a focused look at RAIDs.

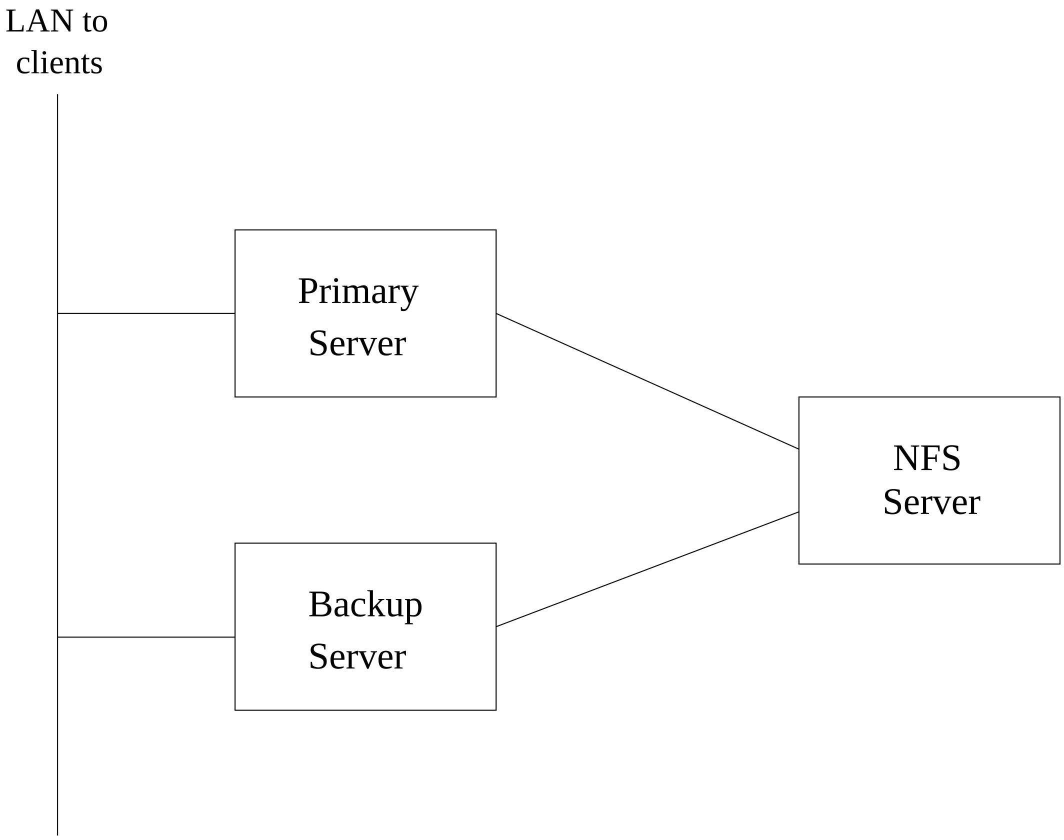

Figure 1: Replicated front end servers

An application of NFS which allows some form of redundancy without replication is shown in figure 1. The clients can be redirected to either the Primary or Backup server using mechanisms such as ARP spoofing [Hor98] or a layer 4 switch [Net98] which allows for session management of connections. This server can source its content from the NFS server. Hence the servers that clients see are redundant although the content itself is not replicated.

This is not an ideal situation as the NFS server is still a single point of failure and the current implementation of NFS in release kernels does have some performance problems. There is, however, hope as additional NFS servers can be maintained using a mechanism such as rsync for synchronisation as discussed in section 3. In addition, recent advancements in the NFS implementation have made significant inroads to the performance problem.

2.1 Caching NFS Client

The caching NFS client [GJB98] has been designed to address several performance issues relating to the implementation of NFS that is shipped with Linux 2.0 kernels. The most far reaching change is the implementation of a persistent cache for files retrieved from the server.

Blocks that are read from a NFS server are kept in a disk cache. As blocks of a file are read they are added to the cache for this file. Once the file is complete it is marked as being persistent and can survive client crashes. This is possible because once the whole file is cached kernel data structures are no longer necessary to gain information about which blocks of the file are present. If the cache becomes full, a process runs through the cache and removes persistent objects with preference for least recently used objects. Partially cached files are not eligible for cleaning as the additional complexity of updating the kernel data structures associated with these files during cache cleaning is tedious at best. As this mechanism favors files that are complete in the cache a background process runs to collect uncached blocks of partially cached files.

The client also supports an RPC lookup cache that holds recently requested information about file and directory attributes. These attribute requests actually contribute a large amount of the RPC traffic associated with NFS. This cache is however limited in its usefulness as the cache must expire after a time in the order of 100 ms to maintain NFS semantics. If the cache is held for longer then the client may no longer hold an accurate view of the attributes held on the server and there are no conflict resolution procedures in the NFS protocol to handle such a situation.

Finally the client supports asynchronous writing of files. Buffering of writes of a file to a server avoids traffic in the situation where a file is modified many times in succession. This is of limited importance in the HTTP server application where accesses are almost exclusively reads. Serving files to a group of workstations is a good example of an application where write performance is of importance. There is nothing more frustrating than a long compile being dragged out by poor NFS performance.

The lookup cache, asynchronous write buffer and persistent disk cache are all designed to reduce the number of RPC calls that a client makes and hence reduce latency and increase performance. The client runs under 2.0 series kernels and was designed to fill the gap between the poor performance of the current NFS implementation included with 2.0 kernels and the new implementation that is currently only available with 2.1 series development kernels. However at this time there is no maintainer for the project, and hence, deploying this in production is not strictly appropriate.

2.2 New Linux NFS Server And Client

The new NFS server and client that is now included in 2.1 kernels has many performance enhancements over the previous implementation [OK98]. The old server and client are still part of the 2.0 kernels and is hence in widespread use in production environments.

The new implementation implements NFSV3 [BC95] which possesses many improvements over the previous NFS specification [Mic98]. Of particular interest is the provision of more information in the return from RPC calls, allowing for more aggressive caching strategies.

The new server has been moved from user space to kernel space. The main advantage of this is that inode information no longer needs to be cached as the server has direct access to kernel data structures. This allows for multiple server threads to operate, as cache consistency is no longer an issue. In addition, running in kernel space reduces the number of memory copies and context switches needed to server data.

Empirical evidence suggests that when a block of a file is requested, subsequent blocks in the file are very likely to be requested in the near future. Rather than wait for each read request NFS clients typically employ a read-ahead mechanism to read the blocks of the file before they are requested. The NFS client has traditionally spawned copies of a daemon, nfsiod, to perform read-ahead. The new client has a single daemon that can handle multiple read-ahead requests simultaneously. This has the advantage of saving context switches as only a single process is involved in handling a read-ahead for a client. Additionally there has always been a restriction on the number of simultaneous read-ahead requests that can be handled, as running a large number of copies of nfsiod is not practical. The new single process approach removes this restriction.

While this NFS implementation is still in development and by no means addresses all of the performance issues, it is a significant advancement. NFS is not an ideal file system for redundant applications. However, it is a very mature technology that is supported across a variety of platforms. It is of particular note that read performance of NFS is quite good compared to the write performance for various reasons. This is to our advantage as the HTTP Server application that I alluded to earlier overwhelmingly involves reads from the server.

3 Rsync

Rsync provides a mechanism for replication. It is specifically designed for synchronisation of files over low bandwidth links [TM96]. The algorithm used can however extend to synchronisation over links of any speed.

The algorithm works by utilising a rolling checksum that is very inexpensive to calculate. The rolling checksum is taken on blocks of the foreign files and comparing this to the checksums for the local files. This checksum is designed to cheaply identify blocks that are likely to have changed. Ordinarily a strong MD5 checksum comparison is then made if a miss-match is found and only the changed blocks are transfered.

With a high speed link such as afforded by a LAN, the computational expense of the strong checksum can be avoided by sacrificing some bandwidth. An option to rsync allows for any differing files to be transfered whole rather than performing the strong checksum phase. This option is particularly attractive in a situation where we are using already loaded servers on a LAN and we are synchronising predominantly small files.

Rsync can be used to keep multiple front end servers or back end NFS servers in synchronisation with each other. In such a situation, one server should be set up as a master where all modifications are made, and rsync could be run periodically to propagate any changes to the other servers. Figure 2 illustrates a topology where our proverbial HTTP servers house their own data and this is synchornised using rsync.

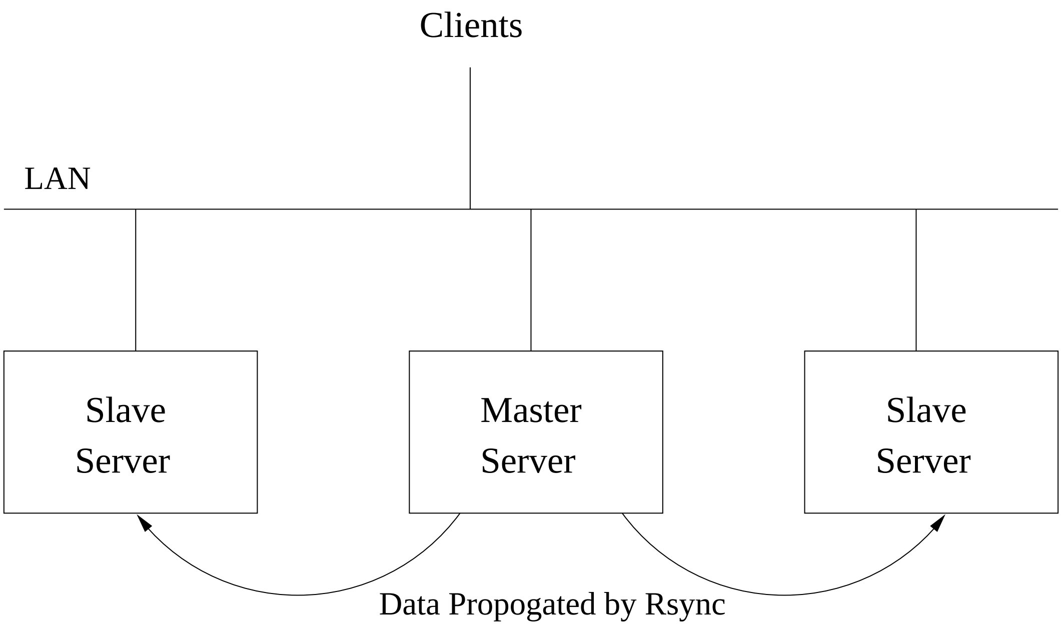

Figure 2: Servers Synchronised Using Rsync

The main problem with this solution is the high load that is borne during the synchronisation process. Additionally this solution is not particularly scalable as there is no mechanism for broadcasting the synchronisation, hence a separate synchronisation process needs to be run for each server that data needs to be propagated to. This solution is, however, advantageous in the absence of any other replication mechanism, as it does allow for easy data recovery in the event of disk failure on one of the servers.

4 Coda

Coda [Bra] is based on AFS2 [mpb] and has been specifically developed to work in high latency environments such as disconnected mobile computing. The replicated volume and persistent client cache facilities provided by Coda are of particular interest in creating high availability content.

4.1 Replicated Volumes

In a Coda file system a Volume is a logical grouping of files which is typically smaller than a partition and larger than a single directory. Volumes can be non-replicated, replicated read-only or replicated read-write. It is the latter two that are of particular interest to us.

Each replicated volume has a Volume Server Group or VSG which all contains all the servers that carry the volume. The Available Volume Server Group or AVSG contains the servers in the VSG that are currently accessible. When a client wants to retrieve an object from a volume, information about the object is retrieved from all servers in the AVSG. If this information is inconsistent then resolution procedures are invoked. This mechanism is beyond the scope of this paper. Once that client has the object information it works on a read from one, write to many basis. The cost of the write to many semantic is minimised by the use of broadcast RPC calls.

4.2 Client Cache

The cache in coda is designed to be useful in disconnected situations as opposed to the caching NFS client which is designed to reduce synchronous RPC calls. When a request is made for a file the Coda client daemon Venus checks the client cache and if the file requested is present then no server interaction is required. Failing this Venus requests the entire file from the Coda server daemon Vice running on a member of the AVSG. For writes, dirty files are only flushed back to the servers in the AVSG once open for write is closed. If file inconsistencies occur on write back to the servers then resolution procedures are invoked.

4.3 A Simple Coda Network

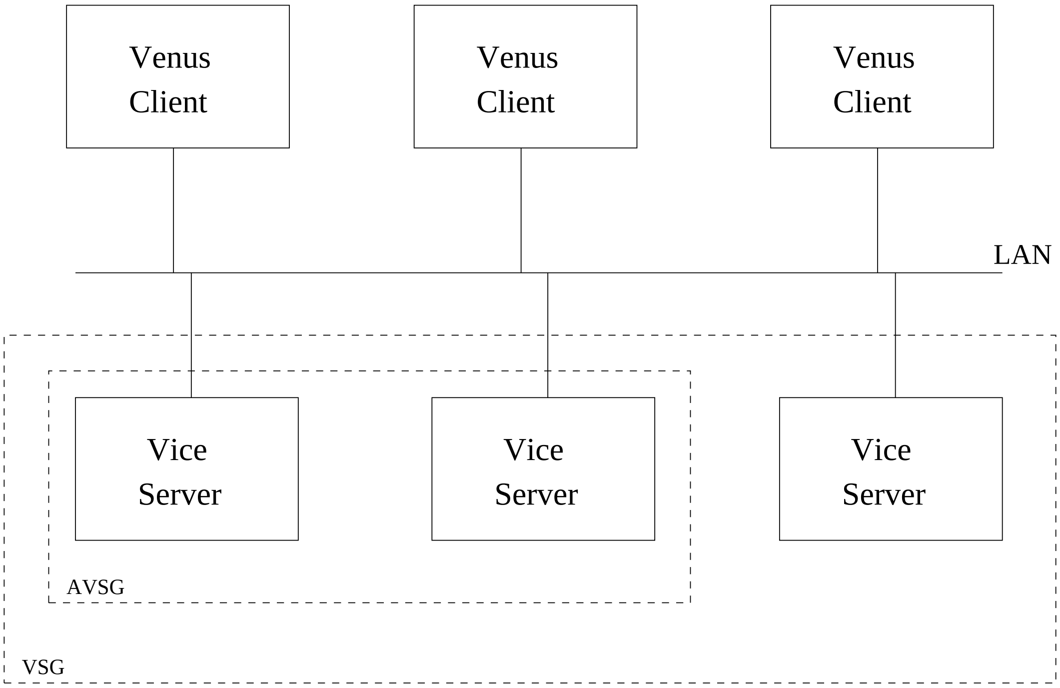

Figure 3: A Simple Coda Network

A simple coda network is illustrated in figure 3. For the sake of argument, the "Vice Server" on the right is currently down for maintenance and hence, has not been included in the AVSG. In this instance the "Venus Clients" will be reading uncached files from either of the two remaining servers in the AVSG. Any writes the clients need to make would be propagated to both servers, assuming that the volume accessed has been set up as replicated read-write. We can imagine that the Venus Clients could be anything, from a group of workstations to a HTTP server farm.

Coda is still in development and is not widely considered to be mature enough for production use. The development of Coda is however, very active and this system should enjoy wide spread use in the future. Unlike other solutions presented, this addresses both the access and replication problem and hence, is the most complete solution.

5 Discussion

Creating redundant content under Linux is a field that is still in a state of development. The most mature technology for providing distributed access to files is NFS. Performance issues relating to the current NFS implementation have lead to various projects including a caching client that works with current release kernels. This, unfortunately, is not at this time well supported, though it is reported to be stable. The main stream NFS implementation has also undergone a major reworking and the fruits of this are, at the time of writing, available in development kernels. Once this implementation comes into main-stream use, NFS under Linux for distributed operations will become more attractive.

Replication of files can be achieved asynchronously using a tool such as rsync. This can be used in conjunction with an access mechanism such as NFS or on servers that house their own content. This type of solution does however suffer from scalability problems and the asynchronous nature of this makes it inappropriate for many applications. Particularly the usefulness of this for serving workstations is extremely limited.

Coda is a solution that provides both access and replication mechanisms. The use of broadcast RPC for writes and a read from one host scheme makes this a more scalable solution. Coda is however still under development and I would not recommend this in a production environment.

The provisioning of redundant content under Linux is an area where many advances still need to be made before a satisfactory solution gains wide spread acceptance. At the moment Coda looks like a strong candidate for this, although ad hoc solutions based on systems such as NFS and Rsync will have their place for some time to come.

References

[BC95] P. Staubach B. Callaghan, B. Pawlowski. Rfc 1813: Nfs version 3 protocol specification, June 1995.

[Bra] Peter J. Braam. The coda distributed file system. http://coda.cs.cmu.edu/ljpaper/lj.html.

[GJB98] gjb@cs.washington.edu Greg J. Badros. An enhanced disk-caching nfs implementation for linux. http://www.cs.washington.edu/homes/gjb/doc/enhanced-linux-nfs-client/index.html, April 1998.

[HM98] hm@seneca.muc.de Harald Milz. Linux high availability howto. http://www.muc.de./~hm/linux/HA/High-Availability-HOWTO.html, http://sunsite.unc.edu./pub/Linux/ALPHA/linux-ha/High-Availability-HOWTO.html, February 1998.

[Hor98] horms@zipworld.com.au Horms. Creating redundant linux servers. http://linux.zipworld.com.au./, May 1998.

[Mic98] Sun Microsystems. Rfc 1094: Nfs: Network file system protocol specification, March 1998.

[mpb] mpb@acm.org. Afs frequently asked questions. http://www.angelfire.com/hi/plutonic/afs-faq.html, ftp://ftp.transarc.com/pub/afs-contrib/doc/faq/afs-faq.html.

[Net98] Alteon Networks. User's guide: Aceswitch software, February 1998.

[OK98] okir@monad.swb.de Olaf Kirch. Linux nfs performance. ftp://ftp.mathematik.th-darmstadt.de/pub/linux/okir/expo97-paper.tar.gz, August 1998.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. ftp://samba.anu.edu.au/pub/rsync/tech_report.ps, June 1996.

[Tra] Transarc. Afs - dfs overview. http://www.transarc.com/dfs/public/www/htdocs/.hosts/external/Product/EFS/Brochure/index.html.