DISTRIBUTED FILE SYSTEMS

Florin Isaila

Karlsruhe University – Germany

Distributed file systems (DFS) are file systems, which manage the storage capacity of several computing nodes, connected by a networking technology and offer to clients a file system interface.

Topics

File server. File service.
Usually the central part of a DFS implementation is the *file server*. A file server is a process, which manages a pool of storage resources and offers a file service to remote or local clients. A *file service* is an interface though which the clients request services (e.g. read, write, seek) to a file server.

File servers can be *stateful* or *stateless*. Stateful servers are keeping state information about their clients, whereas the stateless don't. Stateful servers have the big disadvantage that if the server crashes all the state information is lost. They are not very scalable due to the space overhead. Their big advantages are: shorter messages can be used and better performance. Stateless server are more fault-tolerant and don't waste space for table. A detailed discussion about file servers can be found in [8].

RAID (Redundant Arrays of Inexpensive Disks)
One of the design goals of the DFS is to efficiently use the storage resources network-wide. Due to their slow access times disks are very often system bottlenecks. *RAID* is one method of scalable increasing the disk bandwidth by accessing them in parallel: data is striped on the available disks redundantly; redundancy is used for recovery and availability if one of the disks fails. DFS can implement a software RAID to take advantage of all the disks available in a high performance network.

Location transparency

One of the major goals of the DFS implementation is *location transparency*. This means that the interaction between client and server must be invisible for the users of the system. The users should see all the storage resources in the system and their abstractions (files) as if they would be local. Looking at the path of a file, one should not be able to tell if the file is remote or local.

Unfortunately hiding the locality doesn't hide the difference between local and remote access times. In order to mitigate this problem two techniques are widely used: *caching* and *prefetching*.

Caching

In the one-machine systems *caching* is used to improve local disk access times, providing copies of the low-speed disks in the faster memory. Supplementary DFS caches have the role of providing local copies of remote resources. Caching improves the performance of the applications which exhibit temporal locality of access, i.e. in a program, once a block has been accessed, it is highly probable that it will be accessed again in the near future. Performance measurements show that this is the case with most applications.

In DFS, if we assume the most frequent client-server design, several caching levels of the server disks can be identified from the perspective of a client:

- client memory cache
- server memory cache
- other clients' memory caches
- client disk cache

If a high performance network is used, under the conditions of the actual technologies, the access time will increase from the first to the forth-aforementioned caching levels.

Cooperative caching

The caching levels can be used independently from each other or in cooperation. *Cooperative caching* allows requests not satisfied by the local cache (first level) to be satisfied by another caching level and only lastly by the original resource. Some examples of cooperative caching implementations will be given in the next section. Several cooperative caching algorithms and results of their simulations can be found in [3].

Cache coherence

In the case of reading access the only limitation of caching is the size of the caches. In turn when a cache is written, additional care must be taken to insure *cache coherency*, i.e. if a process writes to any cache location a subsequent read of any process must see that modification.

The above definition is exactly what Unix semantics guarantees. This is easy to implement in the one-machine systems, because they usually have a centralized file system cache which is shared between processes. In the DFS several caching entities can contain the very same copy, and the modification of one copy must trigger either an update or an invalidation of the others, which incurs a considerable overhead. An alternative to this approach, which eliminates the need for a coherency protocol, is to consider all caches in the distributed system as a single large cache and not to allow replication. However the drawback of this approach is that it would reduce access locality.

In order to reduce the overhead of a Unix semantics implementation, relaxed semantics have been proposed. In the session semantics all the modifications made by a process to a file after opening it, will be made visible to the other processes only after the process closes the file. The same idea but at a smaller granularity was used in the database file systems: all the modifications made between control instructions begin-transaction and end-transaction will be visible to the other processes only after execution of the last instruction finishes.

Prefetching

Prefetching means reading ahead from disk into cache data blocks very probable to be accessed in the near future. The applications with predictable access patterns can mostly benefit from prefetching.

In a DFS parallel prefetching can be employed to read ahead in parallel from the available disks. For instance a software RAID can use the disks in a network in a balanced way. Aggressive prefetching can be used to bring the data into caches very early, but this can lead to bad cache replacement choices, which may actually increase the number of accesses. To put it in another way, data being prefetched too early increases the chance that blocks, which are still needed, are evicted from the cache and if the data is fetched too late, the accessing process must wait for I/O to complete. Several algorithms have been proposed for finding the optimal tradeoff between caching and prefetching policies. A good theoretical study of them can be found in [6]. Unfortunately because of the lack of cooperation between caches in the DFS they have not been widely implemented and tested yet.

Log-structured file systems

The design of the *log-structured file systems* was guided by two major assumptions: the caches absorb most of the reads and the disk traffic is dominated by small writes. As a consequence a large time of disk time was spent seeking for the right sector. Log-structured file systems addressed these two major issues by gathering all the writes in a memory segment called log and writing it to disk in a single operation when it became full. This approach improved the average disk time with one order of magnitude for small writes.

Log-structured file system used a checkpointing strategy for recovery. In case of failure the last checkpoint is loaded and the available log is played.

Distributed file systems took over this idea and implemented it efficiently in combination with a software RAID. See the next section for two examples (Zebra and XFS).

DFS implementations

NFS (Network File System)

NFS is the most popular DFS. The basic entities of NFS architecture are servers and clients. The servers are stateless and their main task is to export a local file system. The clients access remote directories by mounting them. Location transparency is guaranteed.

Implementation is based on UNIX virtual file system (VFS) interface, which is used in this case to hide the locality/remoteness of access. When a client accesses a file, a call to the proper VFS function is made. If the file is local the request is serviced by the local file system. Otherwise the server is contacted to fulfill the request.

NFS uses a limited form of cooperative caching. Both servers and clients have caches. If the accessed block cannot be found in one client's cache, it is looked for in the server cache and only then brought from disk. Unfortunately the block is not searched in other clients' caches which would be faster than bringing it from the disk.

A major drawback of NFS is that the caches can become incoherent. When a client modifies its cache, the modification can be sent to the server as late as after 3 seconds for the data blocks and 30 seconds for directory blocks. Therefore the other clients will not see the modification until then. This choice was based on the assumption that file sharing is very rare in a DFS.

NFS servers have also been criticized for not being scalable. When the number of clients increases, they saturate and become a bottleneck for the system. Servers and clients are also not fault-tolerant. If one of them fails it must be manually restarted and the modified cache contents may be lost.

NFS uses a simple prefetching policy based on spatial locality (when a block is accessed it is very probable that the next contiguous block will be needed in the near future). The client usually reads ahead next contiguous block of a file after it has got the currently accessed block.

Petal/Frangipani

Petal is a distributed logical disk. It is designed as a collection of storage servers that cooperate to manage a pool of physical disks. Petal provides a kernel driver interface, which hides the locality/remoteness of storage resources. Therefore all existing file systems can be run unmodified on top of it.

Petal can tolerate and recover transparently from any component failure: server, disk and network. It is also scalable, new storage can be added/removed transparently to/from the system.

Frangipani[9] is a DFS running on top of Petal. Several independent file servers share the Petal distributed disk and synchronize using a distributed lock service. They are using non-cooperatively the UNIX buffer caches. The system is scalable, file server can be added/removed transparently to/from the system, without performance degradation.

Zebra

Zebra[5] is a DFS, which combined for the first time two ideas: log-structured file systems and RAID. Each client writes always into its own log. When the log is full it is striped and the stripes are written to different storage managers, which in turn can write them to disk in parallel.

The servers are responsible only for the administrative information (metadata), including pointers to data, which is stored and managed by storage managers. The servers are relieved of data transfer duties. Therefore they can become performance bottlenecks only in the case of frequent access of small files.

Zebra can tolerate and recover from single storage manager failures. It also uses a checkpointing strategy as the log-structured file systems for recovering from system crashes. The file server keeps its metadata on the storage managers and in case of crash it can recover it from there.

XFS

XFS[1] proposed an innovative design namely serverless network file system and was the first one to implement cooperative caching. The system consists of workstations closely cooperating to provide all file system services in a scalable manner.

Like Zebra, XFS uses a combination of log-structured file systems and RAID ideas in order to improve the write performance and reliability. Unlike Zebra it distributes the control information across system at file granularity and uses cooperative caching to improve access performance. Each time a block is not found in the local cache it is looked for in other clients' caches and only as a last solution is brought from disk. Locality is encouraged by trying to keep the block in the cache of the machine where it is more likely to be accessed. In the cache replacement policy blocks that have multiple copies have priority to be replaced over the non-replicated blocks. XFS uses a token-based cache consistency scheme, which guarantees UNIX semantics to the applications.

Hot topics

Network attached storage

One of the limitations of the classical server-client design of the file systems is that server machine can quickly become a bottleneck. One of the solutions proposed was to separate the storage from the host and to attach it to a high-performance network. Servers are relieved from data transfer duties, while the smart storage system (having a dedicated processor) is responsible for data management, including transfer and optimal placement.

Network Attached Secure Disks (NASD) project[4] file system design aims at separating file management from file storage. File server responsibilities are reduced to access policies and decisions. Therefore when a client contacts the server for opening a file it receives an authorization token, which it can subsequently use for accessing the disks bypassing the servers.

Mobility

The increasing development of mobile computing and the frequent poor connectivity have motivated the need for weakly connected services. The clients should be able to continue working in case of disconnection or weak connectivity and update themselves and the system after reintegration. Coda is a DFS, which exploits weak connectivity for mobile file access[7]. Aggressive prefetching (hoarding) is employed for collecting data in anticipation of disconnection. If the wrong data is hoarded, progress can be hindered in case of disconnection. Another drawback is that cache coherence problems are more likely to occur and they may require user intervention. Unfortunately both aforementioned drawbacks can’t be solved by system design, but by providing connectivity, which is a task of the future.

Extensibility

Many DFS implementations assume the most common application access patterns and hardware configurations and implement general mechanisms and policies, which have to be used by everyone. This results in performance penalties for the applications, which are not running under the implementation assumptions. Giving applications the possibility to enforce their own policy or making the replacement of policies easy would increase the performance of the system. Exokernels and microkernels are only two proposals which allow implementers to easily tailor policies to application needs by moving resource management in user space and offering a relative easy augmentation of system functionality compared with monolithic kernels. In the DFS case locality, caching and prefetching policies could mostly benefit from an implementation, which takes into account the application needs.

References

[1] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli and R.Y. Wang, Serverless Network File Systems, In. Proc. SOSP 1995.

[2] R.Buyya, High Performance Cluster Computing Architecture and Systems, Prentice Hall, 1999.

[3] M. Dahlin, R.Wang, T. Anderson and D. Patterson, Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In. Proc. of the First Symp. on OSDI, 267-280, 1994.

[4] G.A. Gibson, D.F. Nagle, W. Courtright, N. Lanza, P. Mazaitis, M. Unangst, J. Zelenka, NASD Scalable Storage Systems, In Proc. of USENIX Extreme LINUX Workshop, 1999

[5] J.H. Hartman and J. Ousterhout, The Zebra Striped Network File System, In Proc. of ACM Transactions of Computer Systems, 1995.

[6] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G.A. Gibson, A.R. Karlin and K. Li, A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching, In. Proc. of the 2nd USENIX Symposium on OSDI, 19-34, 1996.

[7] L.B. Mummert, M.R. Ebling and M. Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access, In. Proc. of the 13th Symp. on Operating System Principles (SOSP), 1995.

[8] A.S. Tannenbaum, Distributed Operating Systems, Prentice Hall, 1995.

[9] T.A. Thekkath, T. Mann and E.K. Lee, Frangipani: A Scalable Distributed File System. In. Proc. SOSP, 1997.

Cross Reference:

Cache coherency *see* Distributed File Systems.

Caching *see* Distributed File Systems.

Cooperative caching *see* Distributed File Systems.

Distributed logical disk *see* Distributed File Systems.

Extensibility *see* Distributed File Systems.

File server *see* Distributed File Systems.

File service *see* Distributed File Systems.

Location transparency *see* Distributed File Systems.

Log-structured file system *see* Distributed File Systems.

Network attached storage *see* Distributed File Systems.

Network File System *see* Distributed File Systems.

Prefetching *see* Distributed File Systems.

Redundant Arrays of Inexpensive Disks *see* Distributed File Systems.

Serverless network file system *see* Distributed File Systems.

Spatial locality *see* Distributed File Systems.

Temporal locality *see* Distributed File Systems.

Dictionary Terms:

Caching

Replicating the content of a resource for a quicker access.

Cooperative caching

Coordinating the file caches of many machines on a network in order to achieve better performance.

Distributed File System

File systems, which manage the storage capacity of several computing nodes, connected by a networking technology and offer to clients a file system interface.

Distributed Logical Disk

A software layer, which provides the image of a single disk for a pool of storage resources in a network.

File server

A process, which manages a pool of storage resources and offers a file service to the clients.

File service

An interface used by the clients to request services from a file server.

Location transparency

Hiding the location of a resource from its user.

Prefetching

Reading ahead data from a resource in anticipation of usage.