Volume 2, Issue 5, May 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Performance of Cache in Distributed Systems

Neha Pathak
Research Scholar in Dept. of CSE
S.V.I.T.S
Indore (M.P)

Prof. Dinesh Chandra Jain
Reader , Dept. of CSE
S.V.I.T.S
Indore (M.P)

ABSTRACT: **This paper introduces you to the concept of file caching, file caching policies and cache consistency and performance of cache for distributed computing environment. We will see that carefully constructed distributed concept can lead to lower server load and better overall system performance than in centralized concepts. In this paper we are considering the network file system, Andrew file system and sprite file system to study.**

Key words: Distributed File Caches, Network File System, Andrew File System, Sprite File System, Cache Consistency.

I. INTRODUCTION

File caching is the method that reduces the inherent speed difference between processor and disk. The disks are slow and memory is fast and file cache forms an intermediate storage between these two for centralized systems. In distributed system file access is based on a client-server computing model and have to propagate through various instances. This concept introduces opportunity for caching at various levels like the server, the client, or the network. We can employ caching to improve system performance. There are four places in a distributed system where we can hold data: on the server's disk, in a cache in the server's memory, in the client's memory, on the client's disk. The first two places are not an issue since any interface to the server can check the centralized cache. It is in the last two places that problem arises and we have to consider the issue of cache consistency. There are several approaches we have:

- ***Write-through***

All access would require checking with the server first (adds network congestion) or require the server to maintain the state on who has what files cached. Write-through also does not reduce the congestion on writes.

- ***Delayed writes***

Data can be buffered locally (where consistency suffers) but files can be updated periodically. A single bulk write is far more efficient than the lots of little writes every time any file contents are modified. Unfortunately the semantics become ambiguous.

- ***Write on close***

This is admitting that the file system uses session semantics.

- ***Centralized control***

Server keeps track of who has what opened in which mode. We would have to support a stateful system and deal with signalling traffic.

I. ARCHITECTURE of NETWORK FILE SYSTEM(NFS)[7][10]

In computing, a distributed file system or network file system[1] is any file system that allows access to files from multiple hosts sharing via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources. The client nodes do not have direct access to the underlying block storage but interact over the network using a protocol. This makes it possible to restrict access to the file system depending on access lists or capabilities on both the servers and the clients, depending on how the protocol is designed.

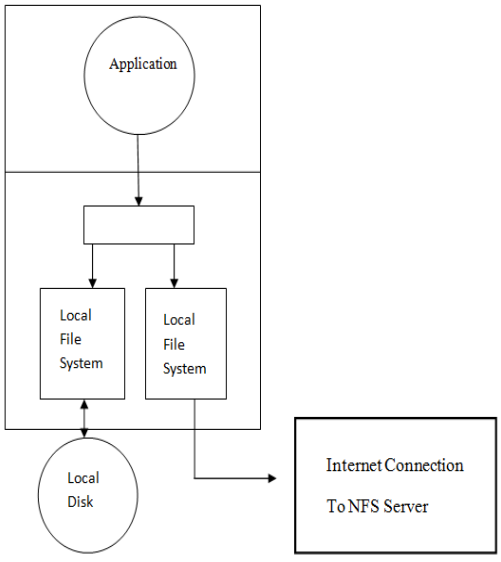

Fig1: NFS

When an application program executes, it calls the operating system to open a file, or to store and retrieve data in files. The file access mechanism accepts the request and automatically passes it to either the local file system software or to the Network File System client, depending on whether the file is on the local site or on a remote site. When it receives a request, the

client software uses the Network File System protocol to contact the appropriate server on a remote machine and perform the requested operation. When the remote server replies the client software returns the result to the application program.

II. CACHING POLICIES in NETWORK FILE SYSTEM(NFS)[1][2]

All caching models have a number of common features. Unless otherwise noted caches are assumed to operate at the file system block level, with a block size of 4-kBytes and a write-back policy with server driven invalidations. A cache replacement strategy of least recently used (LRU) is used in all cases. Although this strategy is not optimal, many studies have shown that it is closed to optimal. All models use fully associative caches which give good performance and only require minimal additional overhead in the case of file accesses, which are rather expensive operations already.

1. Fixed-Size Caches

The following reference models are used in the evaluation process. These models have been chosen to give some upper and lower bounds on specific distribution concepts. They are not intended to be sophisticated implementations of these concepts. The order of presentation is approximately an order of increased complexity. A more detailed description of the models can be found:

- ***Server Cache Only(SCO):***
  Server caching with cache less clients used as a reference point for comparing other policies. This model requires every client access to be forwarded to the server, resulting in substantial network and server load. It provides an upper bound on network traffic induced by the file system. The centralized design imposes strong limitations on scalability.
- ***Local Disk(LOD):***
  This model assumes each client to use a local disk for file storage and also assumes a memory cache to be operated by each client. The server's role is only that of a coordination instance, controlling the traffic flow between clients. Our interest is in the cache hit rates. As the client server traffic does not contribute to this measure it has been omitted. This model presents the most optimistic view of a completely distributed file service, where all accesses can be fulfilled locally. Any realistic implementation would also induce some client-server and client-client traffic.
- ***No Coherency(NOC):***
  With this model we assume a configuration with server and client caches. Compared to SCO the traffic on the network is reduced by the introduction of the additional caching level at the clients. Multi-client cache consistency is not modelled in this approach, thus reducing the network load to an absolute minimum. Only misses in the client cache and cache write-back operations generate traffic on the network. This approach presents an unrealistically optimistic network load. Realistic implementations would include higher network load due to coherency traffic.
- ***Write Through All(WTA):***
  An implementation of the NOC approach with added coherency traffic is presented with this model. WTA uses the easiest way to guarantee consistency in the system, which is a write-through caching scheme. All changed blocks are transferred from client to server as part of the write operation. This guarantees that the server is always in possession of the most recent version of every block and thus can service requests from other clients with up-to-date data. As many files are only used by one client, this protocol generates lots of unnecessary operations on the network and on the server. It is intended as a pessimistic model for guaranteeing global consistency on the block level. The amount of write traffic from clients to the server is the same as in the SCO model.
- ***Write Share Sequential(WSS):***
  Where as WTA writes back blocks which could be kept locally without degrading client cache coherency, WSS seeks to eliminate this additional traffic. Analyzing file access traffic reveals that most of the written files are not actually shared between clients. Only a small fraction of files are actively shared. By using different write policies for shared and non-shared files the excess coherency traffic can be eliminated. WSS uses a write-back policy for non-shared files, which is dynamically changed to write-through as soon as file sharing[6] is detected by the server. This guarantees a consistent view on the server. However, clients may still read old versions of blocks from their local caches. Although this drawback is acceptable for some applications. Although this drawback is acceptable for some application, it might not be desirable in general and can be eliminated by the next algorithm.
- ***Write Share Concurrent(WSC):***
  To overcome the coherency problem inherent in WSS, WSC uses a slight modification of the protocol. Instead of changing the write policy from write-through when a file is shared, the file caching policy is changed to be non-

cacheable on the clients. This forces the only version of the file to be kept on the server, which guarantees consistency under any circumstances. This approach loads the server with the burden of handling all shared file accesses. However, as long as the file sharing ratio is not too high this approach is acceptable.

2. Remote Memory Variable-Size Client Caches

Besides variations in caching policies as presented in the previous section, another orthogonal direction to explore is the usage of the network, i.e. remote memory accesses, to fulfil local cache misses. We investigate the use of remote memory by allowing each client to split its local cache into two distinct regions. One region is used to hold local cache contents, whereas the other region is exported to be used by other clients.

Splitting cache memory into two regions and exporting part of it to other clients reveals two questions. How much memory should be used locally, and which clients are allowed to use the exported regions. Considering the overall performance of the system as the target to be optimized, it can be proven that an optimal solution to this partitioning problem exists. The optimum, i.e. the minimum total number of misses in the whole system, is reached when the derivatives of all clients' miss functions with respect to their cache size are equal.

We have considered this strategy by making two sets of runs over the trace data; during the first set the optimal cache partitioning for various global cache sizes has been collected. The second set of runs uses these optimal cache partitions during its operation. In a production environment this two stage process needs to be replaced by a one stage process that uses an on-line cache partition predictionalgorithm.

Modification time is more recent,

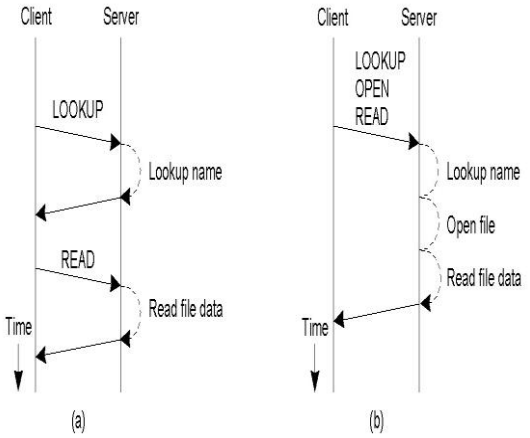

Fig2 Communication process in NFS

III. PERFORMANCE[5]

NFS performance was generally found to be slower than accessing local files because of the network overhead. To improve performance, reduce network congestion and reduce server load. File data is cached at the client. Entire pathnames are also cached at the client to improve performance for directory lookups.

- ***Server Caching***

Server caching is automatic at the server in that the same buffer cache is used as for all other files on the server. The differences for NFS-related writes in that they are all *write-through* to avoid unexpected data loss if the server dies.

- ***Client Caching***

The goal of client caching is to reduce the amount of remote operations. Three forms of information are cached at the client: file data, file attribute information and pathname bindings. We cache the result of read, readlink, getattr, lookup and readdir operations. The danger with caching is that inconsistencies may arise. NFS tries to avoid inconsistencies (and/or increase performance) with:

- Validation-if caching one or more blocks of a file, save a time stamp. When a file is opened or if the server is contacted for a new data block, compare the last modification time. If the remote

IV. ARCHITECTURE of ANDREW FILE SYSTEM(AFS)[5][8]

The Andrew File System (AFS) is a distributed networked file system which uses a set of trusted servers to present a homogeneous, location-transparent file name space to all the client workstations Invalidate the cache.

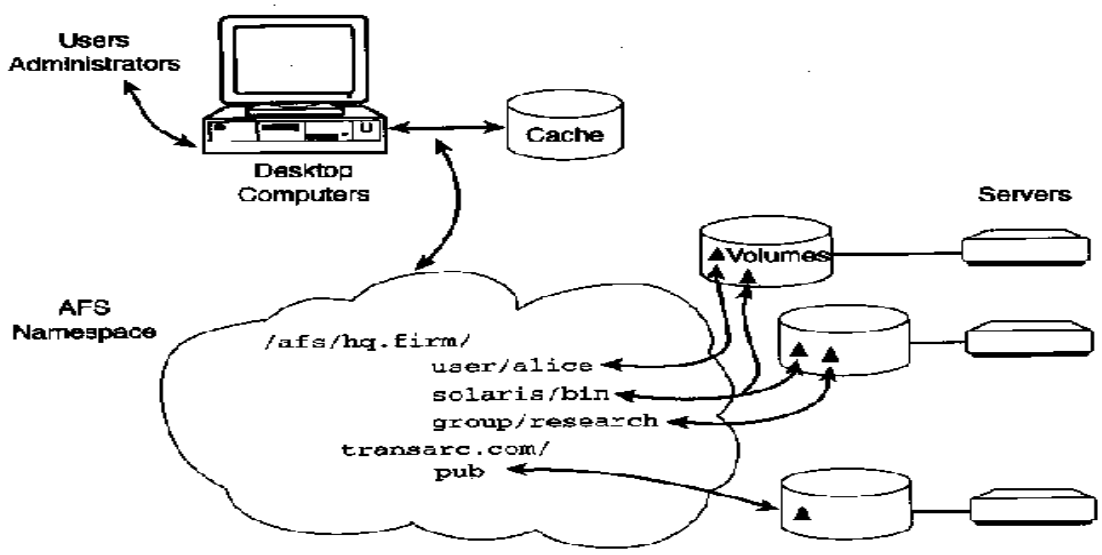

Fig. 3 AFS

- Validation is performed every three seconds on open files.
- Cached data blocks are assumed to be valid for three seconds.
- Cached directory blocks are assumed to be valid for thirty seconds.
- Whenever a page is modified, it is marked dirty and scheduled to be written (asynchronously). The page is flushed when the file is closed.

The file name space on an Andrew workstation is partitioned into a shared and local name space. The shared name space (usually mounted as /afs on the Unix filesystem) is identical on all workstations. The local name space is unique to each workstation. It only contains temporary files needed for workstation initialization and symbolic links to files in the shared name space.Clients may access files from any workstation using same name space. When file is open and closed only at that time client workstation interact with server.

V. CACHING POLICIES in ANDREW FILE SYSTEM(AFS)[6]

- ***Cache Consistency***

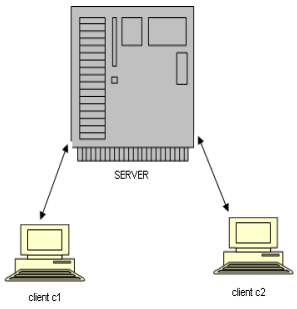

Fig. 4 Cache Consistency

Because of callbacks and whole-file caching, the cache consistency model provided by AFS is easy to describe and understand. When a client (C1) opens a file, it will fetch it from the server. Any updates it makes to the file are entirely local, and thus only visible to other applications on that same client (C1); if an application on another client (C2) opens the file at this point, it will just get the version that is stored at the server which does not yet reflect the changes being made at C1. When the application at C1 finishes updating the file, it calls close() which flushes the entire file to the server. At that point, any clients caching the file (such as C2) would be informed that their callbacks are broken and thus they should not use cached versions of the file because the server has a newer version. In the rare case that two clients are modifying a file at the same time, AFS naturally employs what is known as a **last writer wins** approach. Specifically, whichever client calls close() last will update the entire file on the server last and thus will be the winning file, i.e., the file that remains on the server for others to see. The result is a file that is either one client's or the other client's. Note the difference from a block-based protocol like NFS[12]: in such a block-based protocol, writes of individual blocks may be flushed out to the server as each client is updating the file, and thus the final file on the server could end up as a mix of updates from both

clients; in many cases, such a mixed file output would not make much sense (i.e., imagine a JPEGx image getting modified by two clients in pieces; the resulting mix of writes would hardly make much sense).

VI. PERFORMANCE[11]

One of the reason AFS is popular in large complex environment is because it allows centralized file and back up without the whole system falling apart when loaded. The cost of copying a file over the net can vary widely, based on network load, workstation load, server I/O load. These variables can make a difference in the performance of the application, but over time, they even out. Usually we assume that AFS file are as fast as non AFS and for the vast majority of file, this is true.

VII. ARCHITECTURE of SPRITE FILE SYSTEM[12]

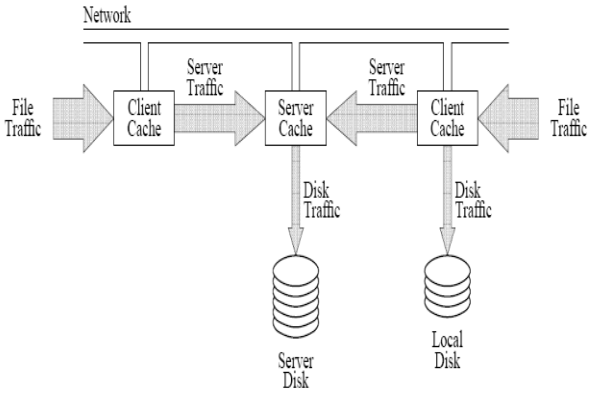

Fig. 5 SFS

When a process makes a file access, it is presented first to the cache of the process's workstation (''file traffic''). If not satisfied there, the request is passed either to the local disk, if the file is stored there (''disk traffic''), or to the server where the file is stored (''server traffic''). Servers also maintain caches in order to reduce their disk traffic.

VIII. CACHING POLICIES in SPRITE FILE SYSTEM[9]

The policy used to write dirty blocks back to the server or disk has a critical effect on the system's performance and reliability. The simplest policy is to **write data through** to disk as soon as it is placed in any cache. The advantage of write-through is its reliability: little information is lost when a client or server crashes. However, this policy requires each write access to wait until the information is written to disk, which results in poor write performance.

An alternate write policy is to **delay write-backs**: blocks are initially written only to the cache and then written through to the disk or server some time later. This policy has two advantages over write-through. First, since writes are to the cache, write accesses complete much more quickly. Second, data may be deleted before it is written back, in which case it need not be written at all. For Sprite, we chose a delayed-write policy. This policy avoids delays when writing files and permits modest reductions in disk/server traffic, while limiting the damage that can occur in a crash.

- ***Cache Consistency***

Sprite guarantees that whenever a process reads data from a file, it receives the most recently written data, regardless of when and where the data was last written. We did this in order to make the user view of the file system as clean and simple as possible, and to encourage use of the file system as a shared system-wide store for exchanging information between different processes on different machines. We hope that shared files will be used to simplify the implementation of system services such as print spoolers and mailers. Of course, we still expect that concurrent write-sharing will be infrequent, so the consistency algorithm is optimized for the case where there is no sharing.

IX. PERFORMANCE[11]

The high performance attainable with client caches casts doubts on the need for local disks on client workstations. For users considering the purchase of a local disk, our advice is to spend the same amount of money on additional memory instead. We believe that this would improve the performance of the workstation more than the addition of a
local disk: it would not only improve file system performance by allowing a larger cache, but it would also improve virtual memory performance.

X. COMPARISON[12]

system	cache location	cache size	writing policy	consistency guarantees	cache validation
NFS	Memory	Fixed	On close or 30 sec. delay	Sequential	Ask server on open
Andrew	Disk	Fixed	On close	Sequential	Server calls client when modified
Sprite	Memory	Variable	30 sec. delay	Sequential, Concurrent	Ask server on open

in NFS or Andrew is open simultaneously on several clients and one of them modifies it, the other clients will not see the changes immediately; users are warned not

to attempt concurrent write sharing. But SFS permits both concurrent and sequential write sharing. SFS uses high-performance kernel-to-kernel RPC mechanism, delayed writes, and kernel implementation but AFS uses user-level implementation. There is a "disk-full" condition arise in SFS, we can ''write'' system call if the disk is full in which each client is given a number of blocks from which it can allocate disk space. If the client uses up its limit, it requests more blocks from the server. This remove the above "disk-full" condition.

XII. FUTURE WORK

As we have seen various approaches to file caching, performance and to assure cache consistency in NFS, AFS and SFS. We can implement those caching policies using trace driven simulation with trace data available from a measured, real workload.

ACKNOWLEDGMENT

Neha Pathak pursuing Master Of Engineering in Computer science branch from SVITS indore. She has done research work in the area of distributed systems.

I would like to thanks to Prof. Dinesh Chandra Jain for supporting in this research paper.

REFERENCES

- [1] Klauser Arty, Posch Reinhard : "*Distributed Caching In Networked File System*", Institute of applied information processing and communications Graz University of technology,Austria,june 1995.
- [2] The NFS Distributed File Service: NFS white paper, Sun Microsystems, March 1995 http://www.sun.com/software/white-paper/wp-nfs/
- [3] [Baker, Hartman, et al.1991] Baker, M.G., Hartman, J.H. Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K.: "*Measurements of a Distributed File System*"; Technical report, University of California at Berkeley, Computer Science Division, July 1991, also appeared in Proceedings of the 13th Symposium on Operating Systems Principles, Oct. 1991.
- [4] [Biswas et al.1994] Biswas, P., Ramakrishnan, K.K., Towsley, D., Krishna, C.M.: "*Performance Benefits of Non-Volatile Caches in Distributed File Systems*"; Concurrency-Practice and Experience, 6,4(1994), 289-323.
- [5] "*Distributed File System Design*" Paul Krazyzanowski, rutgeers University-CS 417:Distributed System.
- [6] M. L. Kazar. *Synchronization and caching issues in the Andrew File System*. In Proceedings of the USENIX Winter Technical Conference, February 1988.
- [7] D. Walsh, B. Lyon, G. Sagar, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. *Overview of the Sun Network File System.* In Proceedings of the 1985 Winter Usenix Technical Conference, January 1985.
- [8] IBM's Transarc division provides the commercial AFS products. Some overview info on AFS can be found at http://www.transarc.ibm.com/Product/EFS/index.html
- [9] Nelson, Welch, Ousterhout 1988] Nelson, M. N., Welch, B. B., Ousterhout, J. K.:"*Caching in the Sprite Network File System*"; ACM Transactions on Computer Systems, 6, 1(1988), 134-154.
- [10][SAND85] Sandberg, R. et al. ''*Design and Implementation of the Sun Network Filesystem.*'' *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.
- [11][HOWA87] Howard, J.H., et al. ''*Scale and Performance in a Distributed File System.*'' *ACM Transactions on Computer Systems*, to appear.
- [12] Michael N. Nelson, Brent B. Welch, John K. Ousterhout ''*Caching in the Sprite Network File System*'', Computer Science Division (EECS), University of California, Berkeley, CA 94720.

