LONG-TERM CACHING STRATEGIES FOR VERY LARGE DISTRIBUTED FILE SYSTEMS

Matt Blaze
Rafael Alonso

CS-TR-321-91

April 1991

Long-Term Caching Strategies for Very Large Distributed File Systems

Matt Blaze
Rafael Alonso

Princeton University
Department of Computer Science
mab@princeton.edu

ABSTRACT

This paper examines the feasibility of using long term (disk based) caches in very large distributed file systems (DFSs). We begin with an analysis of file access patterns in a distributed Unix workstation environment, and identify properties of use to the DFS designer. We then introduce long-term caching strategies that maintain consistency while dramatically reducing the load on file servers. We describe a number of algorithms for maintaining client caches, and present the results of a trace-driven simulation that shows how relatively small disk-based caches can be used to reduce server traffic by 60% to 90%. Finally, we outline possible mechanisms for dynamically organizing these caches into adaptive hierarchies to allow arbitrary scaling of the number of clients and the use of low-bandwidth communication networks. A small (2 or 3 level) hierarchy, coupled with smart caching techniques, has the potential to reduce traffic by an order of magnitude or more over a flat scheme.

1. Introduction and Motivation

Distributed File Systems (DFSs), such as Sun NFS [3], have gained wide acceptance as a mechanism for sharing files among small groups of computers. Although the advantages of a shared file system structure are well established, DFSs are seldom used with very large numbers of machines or over low-bandwidth, heterogeneous, unreliable, or geographically distant networks. While the advantages of a DFS could apply equally well to such large scale systems, the assumptions made by current DFS software do not "scale up". In particular, current systems require file server help for most client reads, so both the server and the network must be able to handle a load proportional to the amount of client activity. In order for a DFS to scale, most client activity must be hidden from the server. We seek a system in which, for example, all users (in the world) of a particular version of Unix could mount a single centrally maintained /bin directory, or in which the remote mount replaces mail and ftp as the method of choice for sharing files among machines spread across great distances.

In first-generation DFSs (such as NFS), the only time the file server does not have to intervene for a client request is when the data being read is already in the client cache. Because the caches are small (and fit in memory), only very recently accessed files and directory information are usually cached. Furthermore, cached data is flushed after a very short time (typically about 30 seconds) to minimize consistency problems. Although these caches do have an impact on client performance, they are not sufficiently large or persistent to enable servers to handle arbitrarily large numbers of clients.

A number of experimental file systems [2][8][9] aim at scalability, but no one system attempts to scale to arbitrarily large numbers of autonomous, heterogeneous clients on arbitrary networks.

1.1. Caching vs. Replication

Two methods of distributing data to client machines without requiring the server to serve every request are replication and caching. In the traditional sense, replication involves propagation of files to the remote sites as they change. When a file changes, the new data (or at least a message invalidating the old data) must be propagated to each replica. A replication strategy may be static, where the files (or file systems) to be replicated and the machines on which the copies are to be stored are determined in advance. This results in potentially large amounts of redundant traffic when frequently written-to files are propagated to clients who may never read them. Static replication strategies require careful planning and monitoring if they are to be used to maximum advantage. The administrative overhead of determining what files to replicate, plus the cost of replicating inappropriate files, makes it suitable mainly for widely used files known not to change very often (such as /vmunix). Several experimental systems make use of static replication.

Caching, on the other hand, can be viewed as a dynamic replication scheme ordinarily done to improve client performance by keeping copies of the most recently used data in memory. This helps only with files opened in the very recent past. Because of the short-term nature of most file system caches, consistency and update propagation are typically managed by simply limiting the maximum age of a file in the cache to a very short time.

Both static replication and caching can improve client performance, but for the purposes of designing a highly scalable system, this is not the right metric at which to look. A better measure is to look at how much load each client puts on its file server. A system can be said to be scalable only if the server can handle a very large number of clients without itself becoming a bottleneck. In a small DFS, it is reasonable (both in terms of performance and economy of hardware) for most requests to be served remotely; the success of NFS proves this. However, as systems grow beyond a few dozen machines on a local network, it becomes attractive to have local disk copies of the remote data. We seek a middle ground between static replication of entire file systems and short term memory-based caches.

2. Distributed File Access Patterns

Virtually all efforts to build scalable DFSs include some kind of replication or caching scheme. Caching helps to reduce traffic (or server load) only if files are accessed in a fairly consistent manner. This section attempts to identify the kinds of access patterns present in current distributed systems. Knowledge of such patterns is essential for designing and evaluating a potential cache algorithm.

The success of a caching strategy depends upon two factors: the ability to predict when a file will be next used, and the ability to predict when the data will become invalid by being overwritten or deleted. Fortunately, Unix files accesses tend to be surprisingly predictable in practice, and this regularity can be exploited in designing a DFS. Past studies [4][5][7] have found that Unix files are normally quite small (1k-10k) and that most accesses are of the entire file. This suggests that caching entire files is reasonable. It is also known that many files are changed or deleted within a few minutes of creation, which suggests that a replication scheme that blindly copies new files to all potential clients is not the best strategy.

Although the results of past analysis of Unix file access patterns has been fairly consistent, less is known about the behavior of distributed file systems where the same files are available for use by many processors. We analyzed a week-long trace of file accesses conducted at DEC/SRC and kindly made available to us. This was augmented by a small trace conducted locally. The data consisted of a log of all system calls issued by about 120 Unix workstations in a computing research environment. We did not look at individual read and write calls; instead, to simplify our analysis, we looked only at open and unlink system calls. We consider an open for read system call to be a non-destructive "read" operation and an open for write or open for read/write or unlink system call to be a destructive "write" operation.

We identify three properties of file access from this trace data that, if generalized, make very efficient caching strategies feasible. The first property, locality, is analogous to the well-known property of memory-reference patterns, and says simply that files tend to be accessed from the same places from which that have been recently opened. The second, which we call "inertia", says that files tend to be accessed in the same manner as previous accesses. The third, "entropy", says that files become less "volatile" over time and tend to become read-only as they are read more often. We discuss these properties in detail below.

2.1. "Locality"

Intuition suggests that the most likely machine to open a file is one that has accessed it recently. We find this property to be very strong in the DEC trace data; about 90 percent of reads are on workstations that have read the file in the past.

Most reads and writes are of files that are used at only a few workstations, even including "system" files like /bin/sh and /vmunix. This is particularly true of writes.

Figure 1 plots opens for writing (and unlink) against the number of machines that have read the file since it was last written (or unlinked). Note the sharp decline in writes after just 2 machines have read the file.

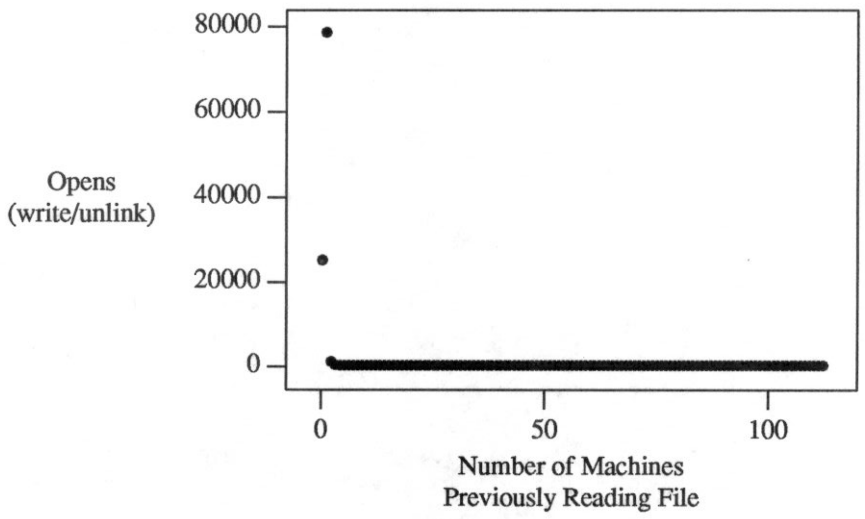

Figure 1 - Writes vs. Number of Machines

Figure 2 plots opens for read only against the number of machines that have previously opened the file since it was last written.

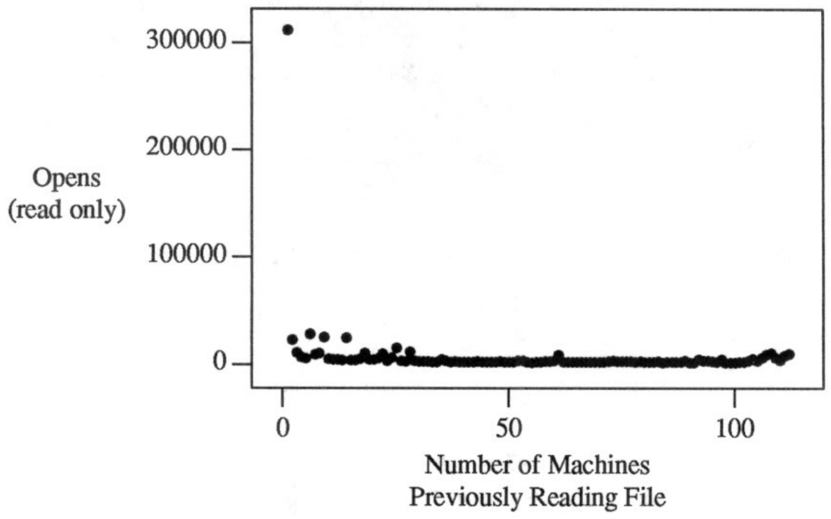

Figure 2 - Reads vs. Number of Machines

Locality is also quite strong in the sense that the previous machine to access a file is also very likely to be the next machine to do so. Almost 70% (429758 out of 623383) of file accesses are by the same machine that last opened the file.

2.2. "File Inertia"

Files in the trace had a strong tendency to be opened in the same mode as they were opened last, particularly for reading. Of 582302 read operations, 519204, or 89%, followed a read operation on that file. For writing, of the 104629 writes logged, 77267, or 73%, overwrote files that had never been read since they were last written. That is, 73% of write operations wrote data that were never read during the trace period. (Part of this is accounted for by writes to append-only log files and opens of write only devices such as /dev/tty).

2.3. "File Entropy"

As files are read more (and to a lesser extent, as they age) they become less likely to be written or deleted before they are read again. Put another way, as a file is read more, it becomes increasingly likely that the next operation on that file will be read rather than write or unlink. We call this property "entropy" and find it to be very strong in our trace data. Entropy takes effect very quickly; in the DEC trace data for any arbitrary file there is about a 10% chance that the next operation on that file will overwrite it, but once the file has been opened for reading at least once, this drops to less than 4%. After three reads, this becomes just 0.5%, and after 10 reads, less than .001%. In other words, files tend to become read-only once they have survived the first few reads. Surprisingly, this remains true (with slightly shifted curves) even when we restrict ourself to looking at system files (/usr, for example) or user's home directories. File entropy is a very useful property to exploit in a caching strategy, since files that are unlikely to change are the best candidates for replication. This property allows us to identify these files using a small history and without requiring the use or type of a file to be known in advance. It also suggests that the very files that change only infrequently also account for a large proportion of reads; in the trace data, over 66% of reads were of files that had been read at least 6 times in the past.

Figure 3 plots the number of previous opens for reading on a file for each open for reading and each open for writing. Figure 4 is the same plot but restricted to operations on files in users' home directories, which are more volatile. Note that these plots are cumulative; the number on the horizontal axis indicates that there have been that many *or more* reads of the file in the past.

It is worth noting that (over all files) reads outnumber writes by about 5 to 1, and that most reads are of files that have already been read several times before. Most writes, on the other hand, are of files that have been read zero or one times before.

Clearly, locality, entropy, and inertia plus the previously known properties of Unix files suggest that there is much to be gained by caching in distributed file systems. The next section of this paper develops and compares several simple strategies for maintaining client caches.

3. Trace-Driven Simulation of Caching Strategies

The properties described in the previous section suggest that most server traffic for reads can be avoided with a relatively simple client caching algorithm. This section describes a trace driven simulation of several caching schemes to determine whether, indeed, this is true, and to shed some light on the question of a reasonable cache size. We examined and evaluated fairly simple algorithms in order to avoid basing our conclusions on very specific "offline" properties of the trace data. In evaluating the various algorithms, we considered only the number of messages sent between client and server. We did not consider any (possibly non-trivial) local processing required at any CPU.

3.1. Simulation Caching Model

We used the DEC trace data to conduct our simulation. Again, we looked at only opens and unlinks and assumed that any write or unlink renders all cached copies of the file invalid. Each client can keep a cached copy of any file it reads but the file server can assign a (possibly infinite) expiration time to the

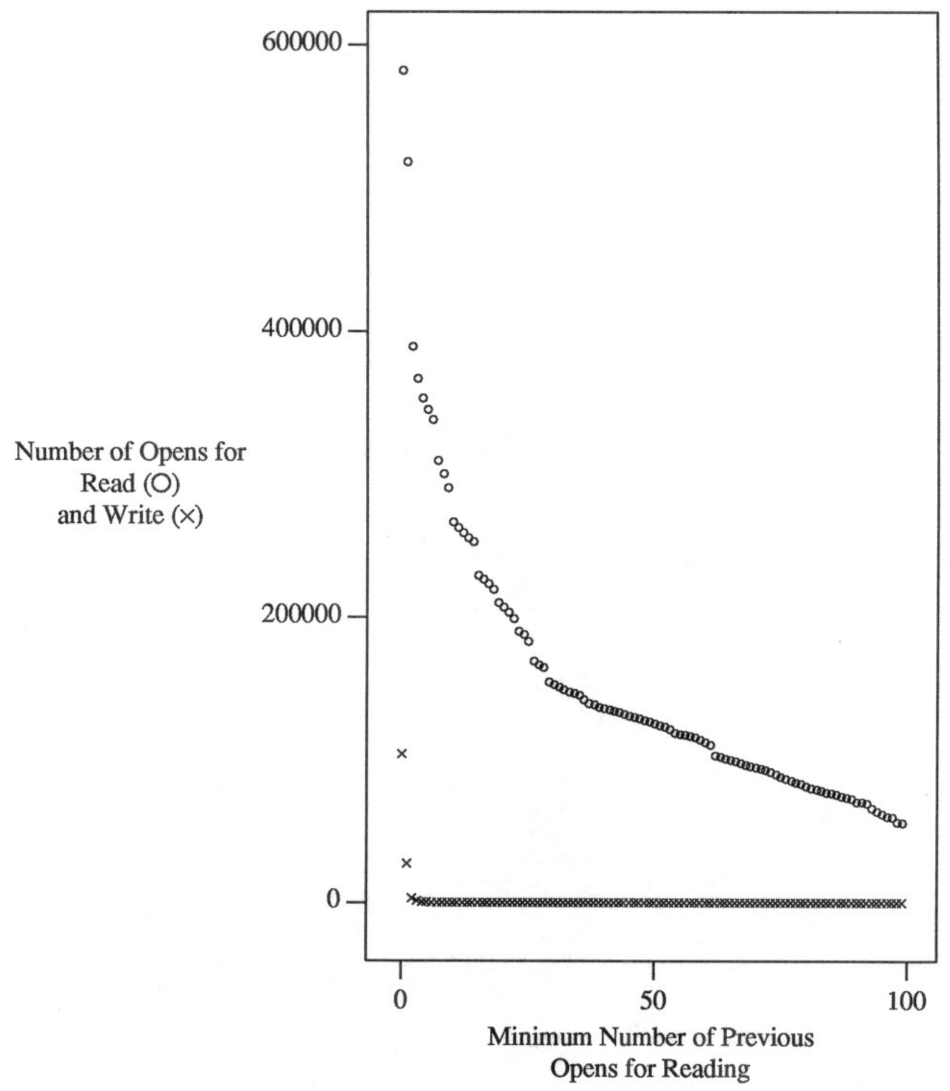

Figure 3 - Reads & Writes vs. Previous Reads (on same file) - All files

client copy at the time it is created. The "owner" of the file (the file server) maintains a list of clients with unexpired copies of each file. If the file is unlinked or written to, each client with an unexpired copy of the file must be notified (it need not be sent the new data, however). If a client reads a file that is in the local cache it need not communicate with the server. Clients may also delete files from their caches at any time without notifying the server. This is an extension of the "stashing" model [10]. Since we are looking only at open operations, we assume that each read is of the entire file; this is probably a reasonable assumption, based on what is known about most Unix file accesses.

3.2. Infinite-Size Caches

This is the simplest case to analyze, and it provides a useful upper bound on the impact of a cache on network traffic. In the infinite cache model, each client keeps a copy of every file it reads until it is told to delete it by the server when the file is written to or unlinked. The results of this trace were very encouraging and suggest that, in fact, caches can have a dramatic impact on network traffic. Of the 686991 operations traced over a five day period, 582302 were read operations, of which 523121 were of valid files in the clients' caches and hence would not require server intervention. So server reads are reduced by 89%. However, 55546 "cancel" messages had to be sent to notify clients of invalid data. If we assume (very

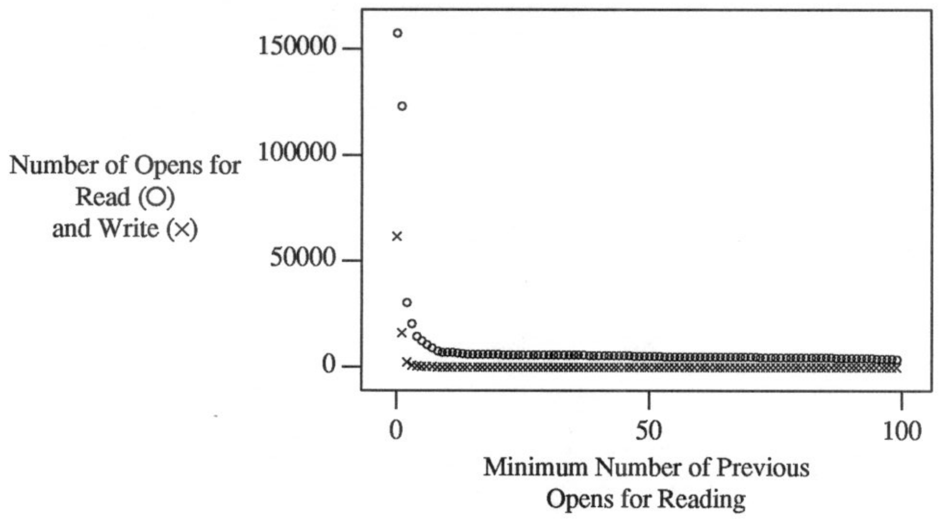

Figure 4 - Reads & Writes vs. Previous Reads (on same file) - /udir Only

conservatively) that sending a "cancel" message costs the same as sending the file, we avoid 80% of the server traffic in a scheme where the file server handles every read. This includes overhead caused by the simulation starting with each client with an empty cache. If the simulation is run for one day before collecting statistics, to allow the clients caches to approach a more "steady" state, the savings is even more dramatic: 93% of reads are served by the client cache.

Note that if clients are able to predict with 100% certainty when a file will be overwritten, the "cancel" messages can be avoided; clients can approximate this if they choose an accurate expire time for the file. This gives us an encouraging upper bound on the savings to be gained by caching in a DFS. With infinite caches and perfect ability to predict when files will change, our simulation yields a 93% client cache hit rate, after some startup overhead as empty caches are filled.

3.3. Finite Caches

Although infinite caches are useful for theoretical evaluation of the impact of caching on network traffic, real caches are, of course, finite. Several issues confront the designer of a practical DFS cache scheme, including what files to cache, how long to keep them in the cache, how to decide what files to discard when the cache is full, and, of course, how big to make the caches.

We assume that each client cache can hold a fixed number of files of arbitrary size each. Although this is not as useful as a simulation of caches of fixed block size, our trace data did not lend itself to this sort of analysis. The average file size of files in the trace was about 11k, so one could approximate the real cache size accordingly.

We considered three finite cache algorithms, and we simulated each using the DEC data. We ran each simulation twice: once gathering statistics for the entire period of the trace, and again where statistics were gathered after it was run for the first simulation day (the "steady state"). For each algorithm, we give the cache sizes simulated, the absolute number of reads not served by client caches and the percentage of total reads this represents, the number of "overhead" messages entailed by the algorithm, the sum of the file traffic and other traffic, and the percentage of total traffic this represents compared with an uncached scheme, assuming that the cost of an "overhead" message and sending a file are equal.

The first algorithm we examined is simple LRU, where, just as in the infinite cache case, the server maintains (along with the usual inode data) a list of all clients that have read the file since it was last written. Clients maintain a simple queue of files, putting the most recently read file at the head and deleting the least recently used files as the cache fills (without notifying the server). When a file is written or unlinked, each client which has made a copy is notified (whether or not the file is still in the client cache). This simple algorithm does fairly well, certainly compared with no caching at all, but still entails the overhead of sending "cancel" messages to clients who have long flushed the file from their caches. Table 1 is a summary of the LRU algorithm with various cache sizes; 1a is over the whole simulation, and 1b is the "steady state" after one day.

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	582302	100.0	55546	637848	109.5
32	208084	35.7	55546	263630	45.3
64	160492	27.6	55546	216038	37.1
128	124841	21.4	55546	180387	31.0
192	109646	18.8	55546	165192	28.4
256	100493	17.3	55546	156039	26.8
320	93006	16.0	55546	148552	25.5
384	87415	15.0	55546	142961	24.6
448	82378	14.1	55546	137924	23.7
512	78605	13.5	55546	134151	23.0
infinite	59181	10.2	55546	114727	19.7

1a - LRU (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	451029	100.0	40951	491980	109.1
32	155109	34.4	40951	196060	43.5
64	117350	26.0	40951	158301	35.1
128	88349	19.6	40951	129300	28.7
192	74935	16.6	40951	115886	25.7
256	66948	14.8	40951	107899	23.9
320	60064	13.3	40951	101015	22.4
384	54951	12.2	40951	95902	21.3
448	50218	11.1	40951	91169	20.2
512	46757	10.4	40951	87708	19.4
infinite	28227	6.3	40951	69178	15.3

1b - LRU (steady state)

Table 1 - LRU Simulation

The second algorithm we examined attempts to remedy the problems of sending expire messages to clients for files in which they are no longer interested and maintaining the server records for these "dormant" clients. We expire files just before we expect them to be overwritten, freeing the server from the obligation to notify the client of changes once the expire time has occurred. This is somewhat like a generalization of the "Lease" concept [6]. Clients first flush expired files from their caches when selecting a cached file for replacement; if no unexpired files exist, LRU is used.

This, of course, raises the question of how to calculate the expire time. We want to assign long expire times to files that are likely to be used again at the client, and short ones to files that are not likely to be read at the client before being overwritten. Recall that the likelihood of a file being overwritten decreases as the file is read more and more. After considerable experimentation, we found a reasonable expire time to be

$$t + \frac{d}{32} r^2$$

where t is is the current time, d is the length of time since the file was last written, and r is the number of times the file was read at the server since last written. This reduced the number of cancel messages dramatically (by a factor of almost 40), but also caused a modest decrease in the client cache hit rate (some files expired while still in the client cache and were later read by the client). Table 2 gives the simulation results for EXPIRE.

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	582302	100.0	1546	583848	100.3
32	219357	37.7	1546	220903	37.9
64	174292	29.9	1546	175838	30.2
128	140968	24.2	1546	142514	24.5
192	127072	21.8	1546	128618	22.1
256	119098	20.5	1546	120644	20.7
320	112884	19.4	1546	114430	19.7
384	106994	18.4	1546	108540	18.6
448	102360	17.6	1546	103906	17.8
512	99698	17.1	1546	101244	17.4
infinite	88779	15.2	1546	90325	15.5

2a - EXPIRE (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	451029	100.0	1071	452100	100.2
32	163835	36.3	1071	164906	36.6
64	128119	28.4	1071	129190	28.6
128	100374	22.3	1071	101445	22.5
192	88136	19.5	1071	89207	19.8
256	80943	17.9	1071	82014	18.2
320	75207	16.7	1071	76278	16.9
384	69568	15.4	1071	70639	15.7
448	65328	14.5	1071	66399	14.7
512	62809	13.9	1071	63880	14.2
infinite	52353	11.6	1071	53424	11.8

2b - EXPIRE (steady state)

Table 2 - EXPIRE Simulation

Our final algorithm is a simple adjustment to the EXPIRE algorithm. Clients do not flush expired files from their caches first; all replacement is done by strict LRU. When a cached but expired file is read, the client verifies with the server that the file is unchanged; if so, the rest of the read is processed locally and a new expire time is assigned to the file. This increases the number of locally processed reads to the level of LRU at the expense of these additional messages, while still maintaining the reduced cancel message overhead of EXPIRE. Table 3 gives the simulation results for EXP-LRU.

It is worth noting that whether EXPIRE or EXP-LRU is preferable depends upon the relative cost of sending entire files against the cost of expire and verify messages. If all messages cost about the same (as they would be where files are small and network packets are large), EXPIRE is the clear winner. Figure 5 compares (in the steady state) the total number of messages used by LRU, EXPIRE and EXP-LRU assuming that all messages are of equal cost. If, on the other hand, sending whole files costs considerably more than sending small, constant size, expire and verify messages, EXP-LRU becomes more attractive. Figure 6 compares the three algorithms where sending files is 10 times the cost of other kinds of messages. Of course, a better expire time calculation formula could make EXPIRE the better choice regardless of the

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Verify Messages	Total Messages	Total Traffic	Percent of Traffic
0	582302	100.0	1546	0	1546	583848	100.3
32	208084	35.7	1546	21770	23316	231400	39.7
64	160492	27.6	1546	23779	25325	185817	31.9
128	124841	21.4	1546	25434	26980	151821	26.1
192	109646	18.8	1546	26267	27813	137459	23.6
256	100493	17.3	1546	26774	28320	128813	22.1
320	93006	16.0	1546	27255	28801	121807	20.9
384	87415	15.0	1546	27579	29125	116540	20.0
448	82378	14.1	1546	27907	29453	111831	19.2
512	78605	13.5	1546	28147	29693	108298	18.6
infinite	59181	10.2	1546	29598	31144	90325	15.5

3a - EXP-LRU (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Verify Messages	Total Messages	Total Traffic	Percent of Traffic
0	451029	100.0	1071	0	1071	452100	100.2
32	155109	34.4	1071	17313	18384	173493	38.5
64	117350	26.0	1071	18865	19936	137286	30.4
128	88349	19.6	1071	20208	21279	109628	24.3
192	74935	16.6	1071	20956	22027	96962	21.5
256	66948	14.8	1071	21416	22487	89435	19.8
320	60064	13.3	1071	21866	22937	83001	18.4
384	54951	12.2	1071	22177	23248	78199	17.3
448	50218	11.1	1071	22485	23556	73774	16.4
512	46757	10.4	1071	22723	23794	70551	15.6
infinite	28227	6.3	1071	24126	25197	53424	11.8

3b - EXP-LRU (steady state)

Table 3 - EXP-LRU Simulation

relative cost of messages.

4. Consistency in Unreliable Networks

The above algorithms work well in a perfect network that remains fully connected and in which no hosts ever fail. Of course, this is even less true of the large networks for which such strategies are designed than it is of small, local networks. Clearly, consistency is an issue in such systems. Although it is well beyond the scope of this paper to conduct an in-depth analysis of consistency in distributed systems, it is worth discussing a few issues that naturally arise out of file replication schemes such as those we have described.

First, we must define what we mean by consistency. At one end of the spectrum, we have what is often called "Unix semantics": each update is guaranteed to be reflected by all other processes as soon as it completes. Clearly, strict Unix semantics are expensive to maintain in a DFS, since it requires clients to verify that their caches are up to date with the server for every read. This would all but eliminate the benefits of caching. Furthermore, if the connection to the server is down, the client read always fails, even when file is in the local cache. Few DFSs attempt to guarantee complete Unix semantics, however, and even some uniprocessor systems do not provide this degree of consistency. At the other end of the spectrum, we could guarantee nothing, and if a client looses its connection to the server, allow it to process

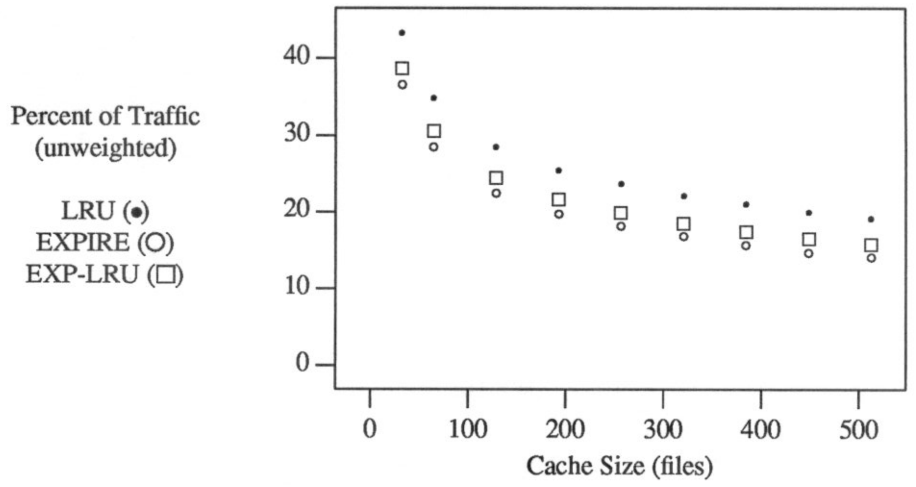

Figure 5 - Cache Algorithms Compared: File Cost == Message Cost

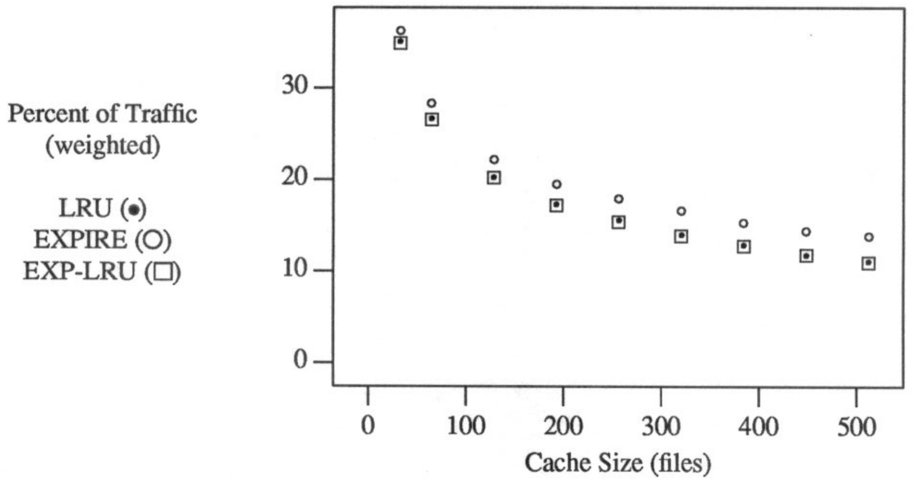

Figure 6 - Cache Algorithms Compared: File Cost == 10x Message Cost

requests with its cache for any files it chooses.

Again, we seek a middle ground between these two extremes in which clients can have some guarantees that the files they read are up to date, but do not have to verify each read with the server. The problem is for clients to determine that they have not lost contact with the file server (and missed invalidation messages) without having to contact the server for each read. We identify two consistency constraints that can be maintained without excessive server interaction. The first, which we call "bounded currency", guarantees that all reads from the cache reflect data out of date by no more than some real time bound. This is simple to implement with "keepalive" messages. Nothing is guaranteed about the self-consistency of the reads and writes, however. The second, "one-copy serializability", states that the global sequence of reads and writes must correspond to some ordering of events if all machines were connected to a single file system. Note that this says nothing about whether a read reflects a recent copy, but just that the set of reads and writes must be self-consistent.

4.1. Bounded Currency

Bounded currency guarantees that cached copies were up to date as of some known time in the past. This can be implemented by flushing the cache after some period of time (and in fact, this is how NFS provides the degree of consistency that it does). Less drastically, each client can keep track of the last time it communicated with each server from which it caches files. When a file in the local cache is read, the client first verifies that it has communicated with the server within the specified time bound. If it has not, it initiates a "keepalive" exchange with that server before completing the read. If the exchange fails, the read cannot be completed.

Clearly, this entails overhead of at most one exchange per client/server pair every time period, regardless of the the number of files each client has from each server. If the clients communicate with the server for other reasons, the number of overhead messages is lowered, of course.

4.2. One-Copy Serializability

This is a generalization of the well-known consistency metric from the field of distributed databases. Simply put, it requires that the global sequence of reads and writes must be self-consistent, in that they must correspond to some sequence of reads and writes on a single file system. This, by itself, does not require that a read actually return data that is in any sense recent. For example, if a client reads a file from a server, makes a cached copy, and becomes partitioned from the server, it might continue to use the old cached copy long after the file has been overwritten. However, we are guaranteed that any other files we read were not created based on data that is unavailable to us.

It is actually fairly easy to maintain one-copy serializability without excessive network traffic. It is sufficient to guarantee that each read from a client's cache reflects data that is current as of the last time that client wrote data read by remote machines. A simple (and fairly inexpensive) way to maintain this constraint is for a clients to verify that it can communicate with the file server any time it reads a file that was cached prior to the last time it wrote to a remotely readable file. If the client has already communicated with the server since it last wrote to a remote file, it need not initiate the keepalive exchange; it need only do so when it has not heard from the server since the last such write.

Observe that the two constraints are complementary; one provides temporal guarantees while the other provides logical guarantees. In practice, it is probably reasonable to provide a combination of bounded currency and one-copy serializability; note that the overhead keepalive messages required to maintain one constraint can reduce the number required by the other. It is also worth noting that not all clients actually require a high degree of consistency, and such parameters as the frequency of required keepalive messages could be a tunable option specified at either open or mount time.

5. Future Work - Cache Hierarchies

While the simulations described above suggest that small client caches can reduce server traffic by a factor of about five, this is still not enough to allow file systems on the massive scale we seek. As a distributed file system grows beyond a small set of machines on a single network, it is advantageous to organize clients into hierarchies so that they do not all contact the file server directly. In such a structure cached (or replicated) copies are used not only by the machine with the cache, but by "child" machines as well. A hierarchical structure offers potential performance benefits to both client and server. From the server's perspective, it can serve orders of magnitude more clients while communicating with only a few top-level nodes. At the same time, clients have access to potentially large caches that may be fewer network hops away.

Some experimental systems [2] permit such a organization to be specified statically as the system is set up. While such schemes are potentially very useful for large homogeneous systems belonging to a single organizational entity, we believe that a more dynamic scheme is required for very large scale file systems. In particular, static hierarchies are inefficient for files that are read infrequently (since they introduce extra layers between client and server) and may cause bottlenecks when the exact hierarchy in place does not reflect the actual traffic load (as when clients in different "leaves" share files). It is difficult to predict the access patterns of a network in advance, and it only becomes more difficult as the network grows and includes machines about which less is known. Our current work focuses on designing a file system that constructs an adaptive hierarchy for each file, so that heavily used files have several layers of caches between clients and servers, while all clients can go to the server for lightly used files. Clients should be able "share" caches on neighboring machines (especially when the server is very distant) and servers should be able off-load frequent file accesses to machines with surplus cache space. Such a scheme could, for example, be appropriate for very large networks such as the Internet, where no central administrative control exists and machines come and go at fairly random intervals.

We are presently addressing the question of how to best construct such a dynamic hierarchy. For example, each server could put a bound on the number of clients to which it is willing to provide cached copies. When the maximum number of clients is reached for a particular file, it could send the list of machines with copies to new clients requesting the file. The new client could select the "best" machine on the list (according to proximity or some other criteria) and route all future requests through that machine. This could be repeated recursively when the client machines themselves become overloaded, and there could be mechanisms for allowing machines to shift the caching load to one another.

6. Summary and Conclusions

A key requirement for successful large scale DFSs is the ability of clients to hide their file system activity from their servers. Caches, in the form of a modest amount of local persistent storage, provide a convenient and well-understood mechanism to do this. We have identified several properties (inertia, entropy and locality) present in a trace of workstation file accesses that suggest that caching can be used to considerable advantage in a DFS.

A trace driven simulation of simple caching schemes further supports the notion that server traffic can be reduced dramatically with fairly small caches. A trace of one such scheme showed that if clients can cache 200 files, 80% of network traffic is eliminated. As memory becomes less expensive, it will become practical to provide very fast file system caches of this size in memory.

Although the problem of consistency in replicated distributed systems is a complex one, two simple constraints (bounded currency and one-copy serializability) provide a fairly high degree of consistency at a reasonable cost in overhead. These can be combined and tuned to provide the degree of consistency required at any particular client.

In summary, we believe that, given small client caches and fairly simple caching algorithms such as those we have described, file servers (and the networks to which they are attached) can serve many more clients than possible with conventional methods.

7. Acknowledgments

We are extremely grateful to Andy Hisgen at DEC SRC for making the trace data available to us. The following people at DEC SRC contributed to the file system tracing facility: Susan Owicki, B. Kumar, Jim Gettys, Deborah Hwang, and Andy Hisgen. The actual trace data was gathered by Andy Hisgen. Brad Barber made helpful comments on drafts of this paper.

8. References

[1] Blaze, M. "Issues in Massively Distributed File Systems." *Proc. 2nd Princeton University SystemsFest,* April, 1990.

[2] Siegel, A., Birman, K., & Marzullo, K. "Deceit: A Flexible Distributed File System." *TR 89-1042,* Dept. Comp. Sci., Cornell University, Nov. 1989.

[3] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX,* Summer, 1985.

[4] Ousterhout J., et al. "A Trace-Driven Analysis of the Unix 4.2 BSD File System." *Proc. 10th ACM Symp. Op. Sys. Principles,* 1985.

[5] Staelin, C. "File Access Patterns" *CS-TR-179-88* Dept. Comp. Sci, Princeton U, 1988.

[6] Gray, C. & Cheriton, D. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proc, 12th ACM Symp. Op. Sys. Principles,* December, 1989.

[7] Floyd, R. "Short-Term File Reference Patterns in a UNIX Environment," *TR-177* Dept. Comp. Sci, U. of Rochester, 1986.

[8] Guy, R., et al. "Implementation of the Ficus Replicated File System," *Proc. Summer 1990 USENIX,* 1990.

[9] Kazar, M., et al. "DEcorum File System Architectural Overview," *Proc. Summer 1990 USENIX,* 1990.

[10] Alonso, R., Barbara, D., and Cova, L. "Using Stashing to Increase Node Autonomy in Distributed File Systems," *Proceedings of the Ninth Symposium on Reliable Distributed Systems,* October 9-11, 1990, Huntsville, Alabama.