Removing Bottlenecks in Distributed Filesystems: Coda & InterMezzo as examples

Peter J. Braam *Philip A. Nelson*

Carnegie Mellon University &
Western Washington University
braam@cs.cmu.edu
phil@cs.wwu.edu
http://www.coda.cs.cmu.edu/

ABSTRACT

Is it possible for a distributed filesystem to perform at the same speed as local disk filesystems, at least in important cases? This paper is trying to answer this question for traditional client-server distributed filesystems, not for distributed filesystems exploiting very fast networks and disk striping techniques. We claim the answer is "yes". Systems such as AFS, Sprite, Coda, Arla and DFS showed what can be achieved by eliminating much RPC traffic, while NFS showed how aggressive kernel optimizations can help. Performance analysis of Coda and NFS shows that in order to achieve local disk performance on read traffic the kernel needs more autonomy. In Coda this leads to satisfactory results, both in micro benchmarks and in a http server benchmark.

For read/write traffic performance is worse. Many operations lead to synchronous RPCs, but a new write-back caching model can permit a client to proceed without server interference and will eliminate most of the remaining RPC traffic. Coda can profit from this, but it does not solve the performance problems entirely. AFS and Coda maintain client caches with much the same functionality as a local disk filesystem, but these caches do not enjoy the superb performance and robustness of local filesystems like ext2. Secondly, working synchronously with a heavy weight cache manager costs Coda much in performance.

A journalling filesystem client that acts as a filter driver for a local filesystem and enjoys callbacks vis a vis the cache manager, can address both points. This design is being explored in the experimental InterMezzo file system.

1 Introduction

During[1] the last few years we have seen spectacular development in freely available distributed file systems. Good quality NFS comes with all free Unices; Samba is an almost fully featured alternative to the Windows NT SMB server; the ARLA project has built clients and servers compatible with AFS and Coda, with advanced features such as server replication and disconnected operation, is now available on several platforms. NetAtalk and a variety of software based on NCP offer file service over AppleTalk and IPX. Several other filesystems exist. Linux rapidly supported all these systems and, perhaps due to

[1]This research was supported by the Air Force Materiel Command (AFMC) under DARPA contract number F19628-96-C-0061. Additional support was provided by Intel and Novell. The views and conclusions contained in here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, Intel, Novell, Carnegie Mellon University, Western Washington University or the U.S. Government.

its supportive kernel development environment, Linux played a vital role in these projects.

Despite these generous offerings, many complaints are heard. The complaints concern every single aspect of these systems: security, stability, protocol quality, portability, usability, administrability and performance. Indeed, in comparison with a well engineered Linux distribution on a standalone workstation, network file systems function poorly.

This paper[2] will take a close look at client performance problems and combine desirable existing features and some new ideas. The thesis is that dramatic performance improvements are still possible. The design exploits obvious principles: limit the number of RPC's to a minimum, don't use cache managers synchronously, exploit the local disk file systems for a persistent cache. The question is how to put the pieces together and retain good semantics.

We will support our case partly with micro benchmarks and partly with analysis of existing systems. We are focusing on use of a *client* in a network file system as a software repository, a WWW page repository for a HTTP server running on the client and as a general purpose home directory. The picture which emerges is that a client must be able to proceed largely by itself, mostly in the kernel and exploit local filesystems.

Other uses will raise issues not discussed here, particularly when very large files or heavy write/write sharing is needed, such as for a databases.

[2]**Acknowledgements** The write-back caching protocol was designed jointly with Lily Mummert, who partially implemented it in Coda. Jan Harkes helped with performance evaluations. Michael Callahan, Jan Harkes and Mahadev Satyanarayanan generously shared time to discuss the approaches described here.

2 Overview of our conclusions

When looking at read-only use of a filesystem the crucial issues are to get good performance upon first access to the data, *and* to aggressively cache data in order to highly optimize subsequent access to the same data.

Network file systems have been implemented in a variety of ways. Some, like NFS and Sprite, have all client and all server code in the kernel. Samba is a good example of a user level file server, communicating with kernel based Windows or Unix SMB client code. AFS and Coda have a user level file servers and Coda relies on a user level cache manager on the client. See figure 1 for an overview.

There are different ways in which a network filesystem can do client side caching. Sprite and SMB have virtual memory caches, NFS has only minimal caching. AFS and Coda have a *persistent* client cache and focus on optimizing subsequent access. A call back scheme can be added to the cache, allowing subsequent access to be served from the cache, without contacting the server, until the server has broken the callback. We will show that the latter construction seems advantageous, but it doesn't solve all problems.

When evaluating Coda it appears that even with a warm cache the cache manager can be much slower than the local file system. We propose here to implement a second callback relationship between the kernel and the cache manager, so that the cache manager is generally not involved in processing, and calls are serviced from the kernel directly. We show that for read-only access we get local disk performance at no cost to the sharing semantics.

During *read-write* use we meet several bottlenecks. Clearly modifcations need to be made on the server as well as on cached data on the client. The network is involved, often in a synchronous

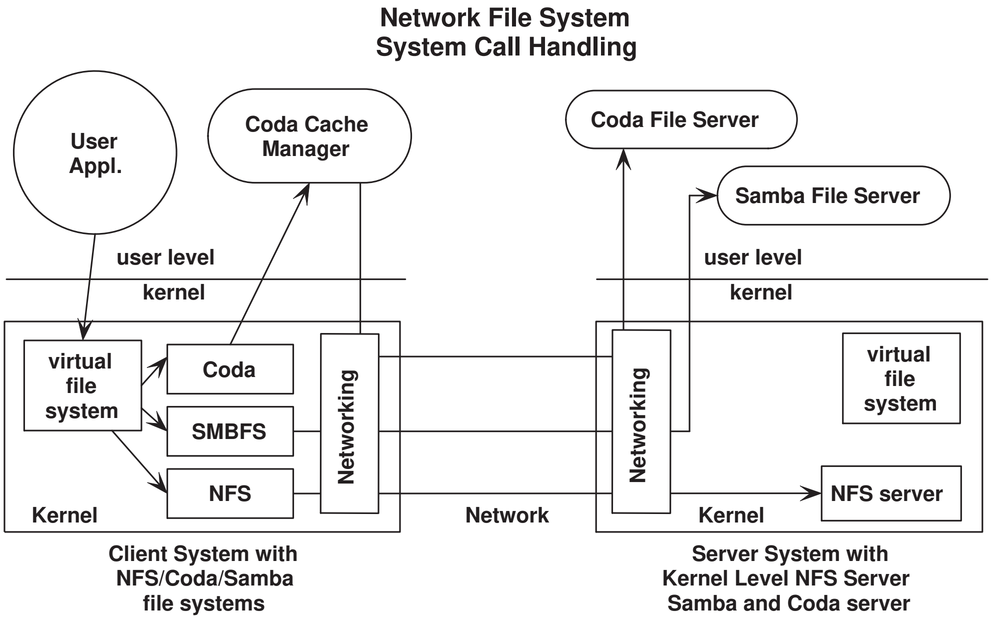

Figure 1: System Call Handling in Network File Systems

way which has disastrous consequences for performance. Here *write-back caching* is needed. When modifications begin a *write-back permit* is obtained by the client for a set of files. This guarantees that the client will - temporarily - be the only writer. The modifications are logged and shipped to the server asynchronously. We can demonstrate that Coda performs much better with write-back caching than without. However, these improvements still leave a very significant performance gap with the Ext2 local filesystem, which we proceed to analyze.

On a Coda client, the cache manager is involved, again often synchronous with application activity. Here again experiments show that one faces serious overhead. It also appears that the local filesystem can be much more efficient than the file systems implemented by cache managers.

To get past these problems we propose that the client kernel module for a network file system acts as a *filter driver* for a local filesystem. The filter intercepts cache misses and journals modifications. It communicates with cache manager when needed, that is, when a cache miss needs to be serviced or when a page of the journal is ready for post processing by the cache manager. The filter driver's journal of operations is transferred to the cache manager page by page, asynchronous with the modifying application.

Coda cannot easily exploit the local filesystem and act as a filter driver, hence we are building a new experimental filesystem named InterMezzo to demonstrate the performance gain from filtering and journalling at the kernel level. A summary of InterMezzo's design appears here, as well as some preliminary evaluations of its performance.

3 Methods to keep data consistent

The inherent uncertainty in a network file system is that a file may change on the server. Repeated access to the same data is very frequent and several paths have been explored to allow this to happen efficiently. The first one is to guess: if access to data falls within a short period of the previous access, the client assumes that the data is still valid. This is used by NFS and leads to remarkably good results or failures depending on the perspective. Sprite and SMB avoid repeated fetching by granting a client an *oplock* or *callback* for data cached in VM. (Callbacks were first introduced in AFS.) However, upon reboots or disconnections from the net, all cache data is lost without a persistent cache.

AFS, DFS and Coda have a persistent cache. (Microsoft's IntelliMirror also has a persistent cache, but the details of its functioning are not known to the author.) A crucial difference between these AFS based caches and the ones used in NFS, SMB and Sprite are that entire directories are cached on the clients, thereby eliminating the need of *lookup* remote procedure calls. During connected operation, the server grants the client a *callback*, *lease* or *oplock* when it first caches data. In this case the server will notify the client before the data is changed on the server, thereby invalidating the cached data on the client. The client can re-use the data without contacting the server until it receives a callback revocation, and is guaranteed that it will never see stale data. If the callback rpc from server to client fails, the client is declared disconnected.

The price of the callback scheme is that the server must maintain state in *virtual memory* about what clients hold what callbacks. AFS/ARLA, DFS, Coda, Sprite, SMB and numerous other systems have a callback scheme.

To maintain a persistent cache valid across reconnections/reboots, a second mechanism named *validation* is needed. Data is kept on the client with a precise *version stamp* and only the version stamp needs to be checked with the server to validate the data; hence the version stamp uniquely defines the data, and it is changed upon every close of a file after writes and after modifying directory operations. Upon reconnecting a client will validate the contents of its cache through comparing version stamps held on both servers and clients. In Coda the first comparison is done at the level of *volumes* or *filesets*, if those succeed, all cached files in the volume are consistent with those on the server. If they fail the version stamp of individual files in the cache must be compared with those on the server. This mechanism is present in AFS.

Version stamps are extremely desirable metadata for network file systems, but must be maintained as persistent objects on client and servers. Presently they are not part of the metadata of disk filesystems, and this - as well as other considerations involving e.g. volumes and access control lists - have led systems like AFS and Coda to use their own file system formats to incorporate this metadata.

Coda with replicated servers faces one further problem: clients must keep the version vector identical across multiple servers. If the RPC connections do not fail nothing new is needed. However, to cope with failures it is very advantageous to add a version array to the version stamps. A version array consists of an integer for every server in a replication group. Upon receiving a modifying operation from the client, server *i* increases the i-th component of the version array. The client finally informs server *i* of the increases in the version array reported by the other servers, thereby allowing all version arrays to be come equal. In the case of server failures the version arrays allow for *comparison* of versions and the system can automatically select the latest version of a file in most cases.

4 Kernel Level Callbacks

The purpose of this section is to show how a filesystem client can be implemented to deliver *read-only access to files* at almost the same speed as when a local disk filesystem is used, after the files have been fetched once. One could call this a *warm client cache*, but care has to be taken, since the client cache can have a persistent component (such as for AFS and Coda) as well as a kernel based volatile cache. Hence caches can be "warm" in more than one way.

We will compare having a scheme with callbacks and without (respectively Coda and NFS), and compare each to the disk file system (Ext2). The test below shows the result of `ls -lR dir > /dev/null` / where `dir` contains a tree of approximately 1,500 files and 300 directories. We compared NFS, Coda and Ext2. [3] Table 1 presents the results. In the first row, all caches are cold: this means that the server has not seen disk access to this data (other than through the boot sequence) and for ext2 no disk access to the files happened since boot. The Coda persistent cache is also empty on the first run. On the second run all caches are warm, eliminating all network traffic for Coda and all disk traffic for ext2 and Coda.

Clearly callbacks can do a lot of good in this situation. Coda makes no remote procedure calls to the server on subsequent runs (assuming that other clients do not modify the tree, causing callbacks to be *broken* by the server). However, it is still troublesome that Coda is running at half the speed of of Ext2. What is going on?

First of all when `ls -lR` is running the `ls` program engages basically in the following operations. It *opens* directories, does *getdents* to get the entries and calls *stat* to get the detailed file attributes. In Coda the *getdents* operations never leave the kernel, and with the *dcache* - the directory name cache - in Linux 2.2 the *stat* calls can be serviced from the cache, in almost precisely the same way for Coda as for ext2. The trouble lies with the *opendir* calls and the associated closes.

	NFS	Coda	Ext2
First run	7.9sec	9.71sec	4.0sec
Subsequent runs	10.4sec	0.5sec	0.26sec

Table 1: Comparing Ext2, Coda and NFS on `ls -lR`. Client was P266/80MB ram/IDE disk, server a PII 266/128MB ram/fast SCSI disks. All caches were cold in first run, all caches were warm in subsequent runs. (We do not understand why NFS was consistently slower on the second run.)

Coda (like Arla) utilizes a user level cache manager. From an engineering perspective this is important, since it dramatically enhances portability. This design implies that an application which accesses files in a Coda filesystem through system calls will be serviced by the kernel in an unusual way. The kernel communicates the request to the cache manager and awaits a reply from the cache manager before servicing the application. This does not apply to all system calls, for example *read, getdents* and *stat* are (mostly) exempt, but as of Coda 5.0 *open, close* (as well as all modifying operations except *write*) *always* go to the cache manager. Servicing a system call gives rise to a context switch when the cache manager needs to run.

It is disconcerting that Coda with a warm cache spends twice as long as the local file system. Further profiling reveals that 50% of Coda's `ls -lR` execution is spent on the 300 *open* and 300 *close* calls. Since this makes for 600 out of approximately 4,500 system calls, this is a typical bottleneck. A separate analy-

[3] We are using a kernel level Linux NFS server.

sis of *open, close* calls reveals that they take 50 times longer for files in Coda than in ext2, each with a warm cache.

The way forward is clear. The kernel must be enabled to open and close files without interaction with the cache manager. To ensure consistency we use a callback mechanism once again: the kernel data structures needed for opening and closing the file (for Coda this is the device and inode number of the cached copy of the file) need to remain available and will be invalidated by the cache manager in two cases. First a server may send a call back revocation if the file or directory changed on the server, secondly, the client cache may be getting full and the file may need to be purged from the cache. In each case the cache manager invalidates the kernel cache.

Coda can easily be modified to honour this protocol with the cache manager and a preliminary implementation shows no loss of performance over the local ext2 filesystem. The experimental InterMezzo Filesystem (to be discussed later) was engineered from the ground up to maintain this relation with its cache manager and similarly shows local disk performance.

Many other things can be deduced from Table 1. First there appears to be no significant advantage of the kernel level NFS server over the user level Coda server. However, the NFS and Coda protocols are very different, so this comparison may not be very useful. Secondly while the cache manager is expensive when all caches are warm, its impact on performance appears to be much smaller than that of having to take data off the disk or over the network.

In the context of a non-persistent cache, this protocol has already been explored before by VaxClusters [12] Sprite [8], Spritely-NFS [7] and SMB [6]. In AFS-3 [2] and DFS/DCE the entire cache manager was moved into the kernel, so one may expect that these systems to have good performance. However, experiments cannot confirm this. A warm cache comparison of the `ls -lR` test on NetBSD's FFS and AFS running on NetBSD 1.3 reveals that AFS lags a factor 4-8 over the local file system. (The NetBSD FFS appears to be 10 times slower on this test than Linux Ext2. Likely this is due to the integrated buffer cache and VM system in Linux, or synchronous updating of atimes in NetBSD.)

An application that requires read-only performance is a WWW server. Storing a WWW site on a Coda client provides automatic mirroring of the site, thus performance of web servers is an issue. We ran the WebStone-2.0[4] WWW server benchmark in two tests. First, having the WWW site stored on a local Ext2 file system and then with the files on a Coda file system. Tests were made with 1 to 10 simulteanous clients, each getting files as fast as the server would deliver the pages. Each test ran for 10 minutes with the majority of files being retreived from the WWW server in the size range of 500 bytes to 50K bytes. A total of 5 different files were used in the test and thus testing repetitive reading by multiple processes. We used Apache for the WWW server. It was configured to have a maximum of 20 processes all reading pages as requested by the clients. Clients and severs all ran on the same machine eliminating traffic on the network for the http requests. Table 2 gives a summary of the tests in connections per second and by total megabytes per second served by each test, and shows that excellent read performance can be achieved.

[4]WebStone-2.0 was distributed by Silicon Graphics, Inc.

Number of clients	Local file system Conn/s	Throughput Mbytes/s	Coda file system Conn/s	Throughput Mbytes/s
1	127.25	19.07	116.72	18.41
2	235.90	35.74	218.21	34.66
3	299.27	45.40	269.12	40.04
4	340.83	52.22	298.18	43.40
5	365.58	56.66	316.00	46.74
6	389.04	59.40	311.77	47.44
7	398.81	58.99	337.48	51.93
8	395.69	60.20	340.21	51.21
9	392.88	60.93	334.52	52.23
10	404.57	59.52	330.24	52.08

Figure 2: Comparing WWW Server performance with files on Coda and Ext2.

5 Write-back caching in the cache manager

We will now turn our attention to modifying operations. The Unix I/O models describe the semantics of simultaneous reading from and writing to a single file. The BSD FFS similarly shaped the thinking about writing data to stable store to guarantee recovery of filesystems after unexpected crashes of the system. Network filesystems took these two as an initial target for distributed systems.

In the Unix I/O model, write/write sharing states that two write operations are mutually atomic and that the results of writing are immediately visible to readers, i.e. the results are similar to the use of shared memory. Much of the thinking in distributed file systems has aimed at having such semantics in the distributed case too. AFS and Coda pioneered different semantics, namely that the visibility to other workstations is at the granularity of *closes* of the file, instead of the granularity of individual writes. This means that a modified version of a file becomes visible to other clients after the file is closed[5]. For many situations this more suitable than write/write sharing, but when using databases in a distributed filesystem write/write sharing is a must.

The BSD FFS tradition for recoverability had a similar impact on distributed file systems. FFS guarantees that modifications to metadata are flushed to the disk before the system call returns which made the changes. Distributed filesystems once more tried to follow this. Of course the overhead of making a remote procedure call and a disk flush for every modifying operation is enormous. NFS v3 allows a server to postpone the flush. AFS and Coda during normal operation make RPC's to the server for every modifying operation. Figure 3 shows the relative performance between NFS, Coda and Ext2. We see that write back caching dramatically improves Coda's performance, bringing it close to NFS. However, ext2 is still well ahead.

Coda also allows for disconnected operation.

[5]In Coda, if two clients hold open a file simultaneously for writing, the last close doesn't overwrite the first, but creates a *conflict*, which allows the user to select one of the versions or perform a merge.

	NFS	Coda	Coda with WB caching	Ext2
create/close/unlink	1.68sec	89.6sec	6.0sec	0.25sec
mkdir/rmdir	2.4sec	70.2sec	4.5sec	0.27sec

Figure 3: Comparing Coda/NFS and Ext2 on 1000 elementary modifying operations - (create x ; close x ; unlink x) and (mkdir x ; rmdir x)

During disconnected operation Coda serves data from its cache to the extent it is available. It makes modifications to its cache and writes a modification log. Upon reconnection, the version stamps of objects are verified and the log is shipped to the server for re-integration. Coda also has special behavior when the connection to the servers is *weak*, i.e. of low bandwidth. In that case it serves cache misses by fetching data from the server, but still makes modifications locally while maintaining the client modification log, called the *CML*. When the CML reaches a specified age or size, it is transferred to the server. This process is called trickle reintegration [Mummert, 1996]. An important aspect of the CML is that it can be *optimized*. Many files live a short life and are then removed, others are saved multiple times in a short interval. This can be detected by scanning the CML and many operations never need to be sent to the server.

This mechanism can form the basis of write-back caching. The only problem is that other clients no longer have a guarantee of seeing consistent data. However, this stumbling block can be overcome.

Coda's write-back caching scheme builds on the trickle re-integration. When the client makes a modifying operation it will request a *write-back permit* from the server. Before granting this permit, the server breaks all callbacks at other clients for the fileset in question. Further modifying operations can now be written in the CML which is transferred to the server asynchronously. Before another client, *say* client2, can get access to the data the server needs to revoke the permit and ask the client which is writing back to reintegrate all its changes as fast as possible. Client2 will validate the files it still has in its cache and fetch the modified objects.

It is clear that this scheme will not function well in all situations. For example, if there are very many clients sharing a volume, and if the modifications to be made just affect a few files, it will be disastrous to grant a write-back permit, since the overhead of breaking all the callbacks and subsequent validation will be enormous. However, for a users home directory to be shared by a few machines, this might work really well. Sprite's delayed writing mechanisms were implemented much earlier and bear great similarity to Coda's, but are VM based.

This scheme has partially been implemented in Coda at present and we can see some of the performance gains we achieve by logging the modifications, vs writing them to the server. Figures 3 and 4 give some of the results. Note that Coda with write back caching is still far behind ext2. The difference with Ext2 is what we will address next.

It is interesting to note that the comparison between Coda with WB caching and NetBSD's FFS is much more favorable for Coda: it merely lags a factor of 2, vs. 25 over ext2. Clearly the synchronous behavior of FFS comes into play.

Upon further investigation, is not so hard to understand why Coda is so slow when working connected. It appears that at present the servers perform many more RVM transactions than is necessary, and it is not unrealistic to expect up to one order of magnitude of improvements by

	NFS	Coda	Coda with WB caching	Disk FS	AFS
Linux	88 sec	320sec	47.1sec	1.55sec	
NetBSD		312sec	43.0sec	54sec	132sec

Figure 4: Unpacking tar archive. 1500 files, 300 directories. Client P400/128MB ram/ide disks. Servers as in table 1.

relatively simple code cleanup.

6 Write-back caching in the kernel: InterMezzo

How can we make modifications to a network filesystem at approximately the same speed as they can be made in the local filesystem? Looking at Table 3 we see that a Coda client running with write-back caching performs *much worse* than the local ext2 filesystem. Two possibilities for this big performance difference exist: first the synchronous operation with the cache manager during modifications is worrisome, since we saw its overhead already during the read-only discussion. Secondly it can be confirmed with easy experiments that Coda's Data structures cache is much slower in its implementation of directory operations than the local filesystem. This can be seen by comparing it with InterMezzo which we do below.

Clearly it would be useful if the local disk file system could be used as a persistent cache for a network file system. This is heavily inspired by so called *Filter Drivers* used in Windows NT [11], but goes back much further to so called *stackable file systems*, developed at SunSoft [9] and UCLA [10]. The InterMezzo Filesystem is an experiment to see if we can exploit the local ext2 filesystem as a cache, in conjunction with a cache manager. We envisage that InterMezzo will have version stamps, callbacks and write-back permits and a file protocol similar to Coda. However, the client is implemented very differently and has a different kernel cache manager protocol, which cannot be easily added to Coda.

InterMezzo's cache manager is called *lento* and the kernel filesystem module *presto*. Lento maintains all metadata which cannot be retrieved from ext2, such as version stamps, volume information and access control lists. All other metadata is part of an unmodified ext2 filesystem. Presto *filters* the operations on the ext2 filesystem: both before and after the ext2 file system operations run, presto executes its own routines which update volatile kernel state associated with the operation. Let's look at a few examples.

During read-only operation lookup operations, to find a name in a directory, and open operations on files and directories are among the most important to intercept. When a lookup operation reaches presto, it checks if it holds a kernel level callback on the ext2 directory. If it does, the operation proceeds by doing an ext2 lookup. If not, presto contacts lento synchronously, and lento either grants a callback because it knows that the cache is up to date, or it fetches fresh object from the server. A similar construction takes place upon opening files and directories. When files are opened their data is fetched. When a file changes on the server the callback is broken and lento can invalidate the data.

Even during read-only use post-processing operations are needed. When Lento installs a fresh directory it is important that the after it is installed it is flagged with a callback, and here we see the post processing. Presto makes a dis-

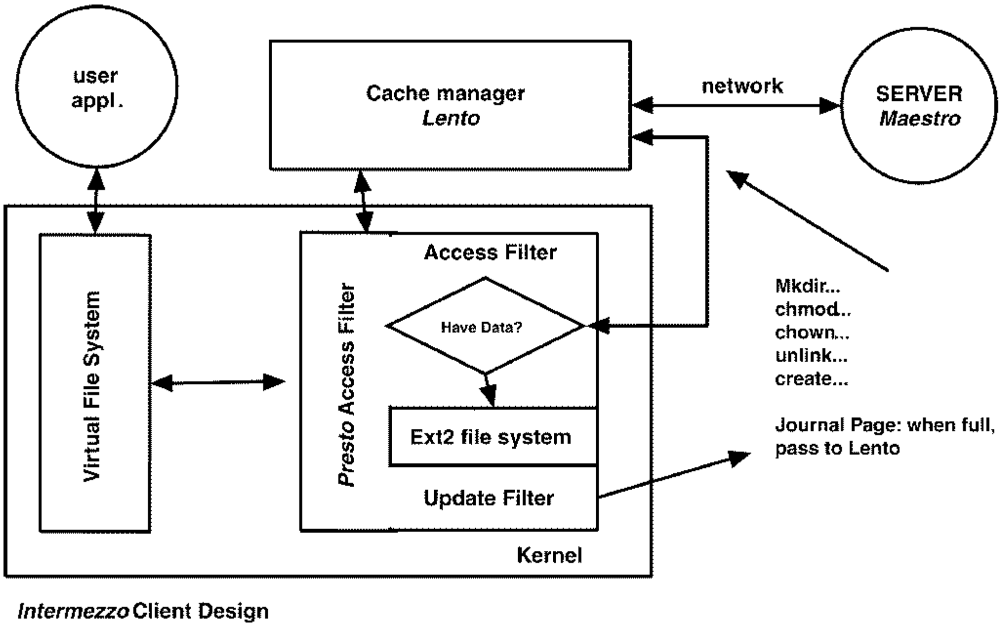

Figure 5: InterMezzo Client Design

tinction between calls from lento and from other applications, and lento's calls to presto module are filtered differently. [6].

InterMezzo will initially operate at the open/close granularity of Coda. Before a modifying operation is started, a write-back permit will be obtained by presto, which asks lento, which in turn obtains it from the server. Presto will now allocate a page of memory and start logging all modifying operations, i.e. *close, setattr, create, mkdir, link, symlink, delete, and rmdir* in it. When the page is full, a new one is allocated, and the full page will be given to lento *asynchronously*. When the server revokes the write-back permit, lento can request all outstanding changes from the kernel, perform log optimizations on the operations and send them to the server.

[6]In order to achieve full consistency we also install methods for the *dcache* to intercept and validate data that would have been given out by the kernel without asking the ext2 filesystem.

We evaluated the performance of InterMezzo for the sequence of rmdir/mkdir operations used above. The results are displayed in Table 5.

The factor of two lag over ext2 is easly explained, given that both the mkdir/rmdir process as well as the cache manager are contending for the CPU. Clearly a lot of engineering would be needed to turn this experiment into a viable filesystem, but initial results are promising.

7 Summary

In this paper we have evaluated the performance of a client in a distributed filesystem. We analyzed cache consistency and write-back caching for performance enhancements. We showed that local disk performance can be achieved in Coda for read-only traffic, using kernel level callbacks. On read-write traffic two levels of improvements are needed. First cache managers should use write-back caching vis a vis the file

NFS	Coda	Coda with WB caching	InterMezzo	Ext2	NetBSD
2.5sec	75sec	4.5sec	0.5sec	0.25sec	10sec

Figure 6: Mkdir/Rmdir Micro Benchmarks. All tests on Linux except last one. Servers as in table 1.

servers, and we demonstrated the benefits of this in Coda. These improvements still leave a large gap with local disk file systems and we suggest a mechanism for bridging this gap. We combine a filter driver working with ext2 with a write back cache for the kernel vis a vis the cache manager.

8 References

1. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E Okasaki, E.H. Siegel, D.C. Steere, Coda: a highly available file system for a distributed workstation environment, IEEE Transactions on Computers, Vol. 39, No. 4, April 1990.

2. Howard, J.H., An Overview of the Andrew File System, Proceedings of the USENIX Winter Technical Conference Feb. 1988, Dallas, TX.

3. Mummert, L.B., Ebling, M.R., Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access , M. Proceedings of the 15th ACM Symposium on Operating Systems Principles, Dec. 1995, Copper Mountain Resort, CO.

4. Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems Feb. 1992, Vol. 10, No. 1, pp. 3-25.

5. Kumar, P., Satyanarayanan, M., Log-Based Directory Resolution in the Coda File System. Proceedings of the Second International Conference on Parallel and Distributed Information Systems Jan. 1993, San Diego, CA, pp. 202-213.

6. John D. Blair, Samba: Integrating Unix and Windows, Specialized Systems Consultants Inc, Seattle, 1998.

7. V. Srinivasan, Jeffrey C. Mogul, Spritely NFS: Implementation and Performance of Cache-Consistency Protocols, Compaq Western Research Laboratory, Research Report 89/5, May 1989, http://www.research.digital.com/wrl/techreports/abstracts/89.5.html.

8. Michael N. Nelson, Brent B. Welch, John B. Ousterhout, Caching in the Sprite Network File System, Transactions on Computer Systems, 6 (1): 134-154, February 1988.

9. Rosenthal, D.S.H., Evolving the Vnode Interface, Proceedings of the Summer 1990 USENIX Technical Conference, June 1990, pp. 107-118.

10. Heidemann, J.S., Popek, G.J., File-System Development with Stackable Layers, ACM Transactions on Computer Systems, Vol 12, No 1, Feb 1994, pp. 58-89.

11. Nagar, Rajeev, Windows NT File System Internals: A developers guide, O'Reilly and Associates, 1997.

12. Kirby McCoy, Vms File System Internals, Digital Press, 1990.