Nache: Design and Implementation of a Caching Proxy for NFSv4

Ajay Gulati
Rice University
gulati@rice.edu

Manoj Naik
IBM Almaden Research Center
manoj@almaden.ibm.com

Renu Tewari
IBM Almaden Research Center
tewarir@us.ibm.com

Abstract

In this paper, we present Nache, a caching proxy for NFSv4 that enables a consistent cache of a remote NFS server to be maintained and shared across multiple local NFS clients. Nache leverages the features of NFSv4 to improve the performance of file accesses in a wide-area distributed setting by bringing the data closer to the client. Conceptually, Nache acts as an NFS server to the local clients and as an NFS client to the remote server. To provide cache consistency, Nache exploits the read and write delegations support in NFSv4. Nache enables the cache and the delegation to be shared among a set of local clients, thereby reducing conflicts and improving performance. We have implemented Nache in the Linux 2.6 kernel. Using Filebench workloads and other benchmarks, we present the evaluation of Nache and show that it can reduce the NFS operations at the server by 10-50%.

1 Introduction

Most medium to large enterprises store their unstructured data in filesystems spread across multiple file servers. With ever increasing network bandwidths, enterprises are moving toward distributed operations where sharing presentations and documents across office locations, multi-site collaborations and joint product development have become increasingly common. This requires sharing data in a uniform, secure, and consistent manner across the global enterprise with reasonably good performance.

Data and file sharing has long been achieved through traditional file transfer mechanisms such as FTP or distributed file sharing protocols such as NFS and CIFS. While the former are mostly adhoc, the latter tend to be "chatty" with multiple round-trips across the network for every access. Both NFS and CIFS were originally designed to work for a local area network involving low latency and high bandwidth access among the servers and clients and as such are not optimized for access over a wide area network. Other filesystem architectures such as AFS [17] and DCE/DFS[20] have attempted to solve the WAN file sharing problem through a distributed architecture that provides a shared namespace by uniting disparate file servers at remote locations into a single logical filesystem. However, these technologies with proprietary clients and protocols incur substantial deployment expense and have not been widely adopted for enterprise-wide file sharing. In more controlled environments, data sharing can also be facilitated by a clustered filesystem such as GPFS[31] or Lustre[1]. While these are designed for high performance and strong consistency, they are either expensive or difficult to deploy and administer or both.

Recently, a new market has emerged to primarily serve the file access requirements of enterprises with outsourced partnerships where knowledge workers are expected to interact across a number of locations across a WAN. Wide Area File Services (WAFS) is fast gaining momentum and recognition with leading storage and networking vendors integrating WAFS solutions into new product offerings[5][3]. File access is provided through standard NFS or CIFS protocols with no modifications required to the clients or the server. In order to compensate for high latency of WAN accesses, low bandwidth and lossy links, the WAFS offerings rely on custom devices at both the client and server with a custom protocol optimized for WAN access in between.

One approach often used to reduce WAN latency is to cache the data closer to the client. Another is to use a WAN-friendly access protocol. The version 4 of the NFS protocol added a number of features to make it more suitable for WAN access [29]. These include: batching multiple operations in a single RPC call to the server, enabling read and write file delegations for reducing cache consistency checks, and support for redirecting clients to other, possibly closer, servers. In this paper, we discuss the design and implementation of a caching file server proxy called Nache. Nache leverages the features

of NFSv4 to improve the performance of file serving in a wide-area distributed setting. Basically, the Nache proxy sits in between a local NFS client and a remote NFS server bringing the remote data closer to the client. Nache acts as an NFS server to the local client and as an NFS client to the remote server. To provide cache consistency, Nache exploits the read and write delegations support in NFSv4. Nache is ideally suited for environments where data is commonly shared across multiple clients. It provides a consistent view of the data by allowing multiple clients to share a delegation, thereby removing the overhead of a recall on a conflicting access. Sharing files across clients is common for read-only data front-ended by web servers, and is becoming widespread for presentations, videos, documents and collaborative projects across a distributed enterprise. Nache is beneficial even when the degree of sharing is small as it reduces both the response time of a WAN access and the overhead of recalls.

In this paper, we highlight our three main contributions. First, we explore the performance implications of read and write open delegations in NFSv4. Second, we detail the implementation of an NFSv4 proxy cache architecture in the Linux 2.6 kernel. Finally, we discuss how delegations are leveraged to provide consistent caching in the proxy. Using our testbed infrastructure, we demonstrate the performance benefits of Nache using the *Filebench* benchmark and other workloads. For these workloads, the Nache is shown to reduce the number of NFS operations seen at the server by 10-50%.

The rest of the paper is organized as follows. In the next section we provide a brief background of consistency support in various distributed filesystems. Section 3 analyzes the delegation overhead and benefits in NFSv4. Section 4 provides an overview of the Nache architecture. Its implementation is detailed in section 5 and evaluated in section 6 using different workloads. We discuss related work in section 7. Finally, section 8 presents our conclusions and describes future work.

2 Background: Cache Consistency

An important consideration in the design of any caching solution is cache consistency. Consistency models in caching systems have been studied in depth in various large distributed systems and databases [9]. In this section, we review the consistency characteristics of some widely deployed distributed filesystems.

- *Network File System (NFS):* Since perfect coherency among NFS clients is expensive to achieve, the NFS protocol falls back to a weaker model known as *close-to-open* consistency [12]. In this model, the NFS client checks for file existence and permissions on every open by sending a GETATTR or ACCESS operation to the server. If the file attributes are the same as those just after the previous close, the client will assume its data cache is still valid; otherwise, the cache is purged. On every close, the client writes back any pending changes to the file so that the changes are visible on the next open. NFSv3 introduced weak cache consistency [28] which provides a way, albeit imperfect, of checking a file's attributes before and after an operation to allow a client to identify changes that could have been made by other clients. NFS, however, never implemented distributed cache coherency or concurrent write management [29] to differentiate between updates from one client and those from multiple clients. Spritely NFS [35] and NQNFS [23] added stronger consistency semantics to NFS by adding server callbacks and leases but these never made it to the official protocol.
- *Andrew File System (AFS):* Compared to NFS, AFS is better suited for WAN accesses as it relies heavily on client-side caching for performance [18]. An AFS client does whole file caching (or large chunks in later versions). Unlike an NFS client that checks with the server on a file open, in AFS, the server establishes a callback to notify the client of other updates that may happen to the file. The changes made to a file are made visible at the server when the client closes the file. When there is a conflict with a concurrent close done by multiple clients, the file at the server reflects the data of the last client's close. DCE/DFS improves upon AFS caching by letting the client specify the type of file access (read, write) so that the callback is issued only when there is an open mode conflict.
- *Common Internet File System (CIFS):* CIFS enables stronger cache consistency [34] by using opportunistic locks (OpLocks) as a mechanism for cache coherence. The CIFS protocol allows a client to request three types of OpLocks at the time of file open: exclusive, batch and level II. An exclusive oplock enables it to do all file operations on the file without any communication with the server till the file is closed. In case another client requests access to the same file, the oplock is revoked and the client is required to flush all modified data back to the server. A batch oplock permits the client to hold on to the oplock even after a close if it plans to reopen the file very soon. Level II oplocks are shared and are used for read-only data caching where multiple clients can simultaneously read the locally cached version of the file.

3 Delegations and Caching in NFSv4

The design of the version 4 of the NFS protocol [29] includes a number of features to improve performance in a wide area network with high latency and low bandwidth links. Some of these new features are:

1. *COMPOUND RPC:* This enables a number of traditional NFS operations (LOOKUP, OPEN, READ, etc.) to be combined in a single RPC call to the server to carry out a complex operation in one network round-trip. COMPOUND RPCs provide lower overall network traffic and per command round trip delays.
2. *Client redirection:* The NFSv4 protocol provides a special return code (`NFS4ERR_MOVED`) and a filesystem attribute (`fs_locations`) to allow an NFS client to be redirected to another server at filesystem boundaries. Redirection can be used for building a wide-area distributed federation of file servers with a common namespace where data can be replicated and migrated among the various file servers.
3. *OPEN delegations:* File delegation support in NFSv4 is a performance optimization which eliminates the need for the client to periodically check with the server for cache consistency. Later in this section, we will investigate delegations in detail as they form an important component of Nache.

By granting a file delegation, the server voluntarily cedes control of operations on the file to a client for the duration of the client lease or until the delegation is recalled. When a file is delegated, all file access and modification requests can be handled locally by the client without sending any network requests to the server. Moreover, the client need not periodically validate the cache as is typically done in NFS as the server guarantees that there will be no other conflicting access to the file. In fact, the client need not flush modified data on a CLOSE as long as the server guarantees that it will have sufficient space to accept the WRITEs when they are done at a later time.

NFSv4 delegations are similar to CIFS oplocks but are not exactly the same [29]. Delegations in NFSv4 are purely a server driven optimization, and without them, the standard client-side caching rules apply. In CIFS, on the other hand, oplocks are requested by the client and are necessary for caching and coherency. If oplocks are not available, a CIFS client cannot use its cached data and has to send all operations to the server. In addition, NFSv4 delegations can be retained at the clients across file CLOSE as in CIFS batch oplocks.

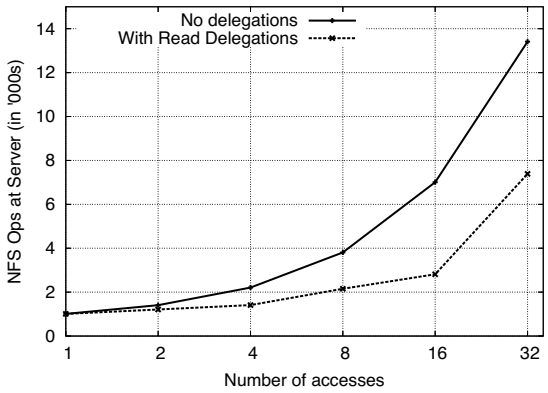


Figure 1: *Performance of Read Delegations: The graph shows the number of NFS ops (packets) sent to the server with and without read delegations. The X axis denotes the number of times a single NFSv4 client opens, reads and closes a file.*

3.1 Read Delegation

A read delegation is awarded by the server to a client on a file OPENed for reading (that does not deny read access to others). The decision to award a delegation is made by the server based on a set of conditions that take into account the recent history of the file [32]. In the Linux NFSv4 server, for example, the read delegation is awarded on the second OPEN by the same client (either after opening the same file or on opening another file) [13]. After the delegation is awarded, all READs can be handled locally without sending GETATTRs to check cache validity. As long as the delegation is active, OPEN, CLOSE and READ requests can be handled locally. All LOCK requests (including non exclusive ones) are still sent to the server. The delegation remains in effect for the lease duration and continues when the lease is renewed. Multiple read delegations to different clients can be outstanding at any time. A callback path is required before a delegation is awarded so that the server can use it to recall a delegation on a conflicting access to a file such as an OPEN for write, RENAME, and REMOVE. After a delegation has been recalled, the client falls back to traditional attribute checking before reading cached data.

Currently, the NFSv4 server hands out delegations at the granularity of a file. Directory delegations are being considered in future revisions of the protocol [21], but the design of Nache relies only on file level delegations.

To illustrate the benefit of read delegations, both in terms of server response time and message overhead, we measured the values for a single NFSv4 client, with multiple application processes, iterating over a OPEN-READ-CLOSE operation sequence on a file. Both the client and server were running Linux kernel version 2.6.17 [13]. Figure 1 shows the number of NFS oper-

ations processed by the server with and without delegations. As shown, the server load in terms of number of packets received is reduced by 50% with read delegations enabled. Further gains are achieved when the file is cached for a long time by saving on the additional GETATTRs sent to validate the cache after a timeout. We observe that if the file is cached beyond the attribute timeout (typically 30 seconds), the operations at the server with delegations reduced by another 8%.

3.2 Write Delegation

Similar to read delegations, write delegations are awarded by the server when a client opens a file for write (or read/write) access. While the delegation is outstanding, all OPEN, READ, WRITE, CLOSE, LOCK, GETATTR, SETATTR requests for the file can be handled locally by the client. Handling delegations for write, however, is far more complicated than that for reads. The server not only checks if a callback path exists to the client to revoke the delegation, it also limits the maximum size that the client can write to prevent ENOSPC errors since client has the option to flush data lazily. On a conflicting OPEN by another client, the server recalls the delegation which triggers the client to commit all dirty data and return the delegation. The conflicting OPEN is delayed until the delegation recall is complete. The failure semantics with write delegations are also complicated, given that the client can have dirty data that has not yet been committed at the server even after a CLOSE. To maintain the close-to-open cache consistency semantics, the client, even with a write delegation, may flush all dirty data back to the server on a CLOSE. Observe that, with write delegations, the consistency semantics are slightly tighter than a pure close-to-open model. The second client, on an OPEN after a delegation recall, sees the data written by the first client before the first client closes the file.

At the time of writing, the Linux server implementation (2.6.17) only hands out read delegations. We have a prototype implementation to enable write delegations which we discuss in detail in section 6. Figure 2 shows the NFS operations processed at the server with and without write delegations. The workload is the same as in the previous experiment with a single NFSv4 client iterating over a sequence of OPEN-WRITE/READ-CLOSE operations on a file with a balanced number of reads and writes. The server load, in terms of number of packets received, is reduced by 5 times with write delegations enabled. One would expect write delegations to provide significantly better performance than what is observed. The reason this is not the case is that write delegations are not completely implemented in our prototype. Although the delegation is granted, a client still sends WRITE requests to the server, whereas they need not once a delegation is obtained.

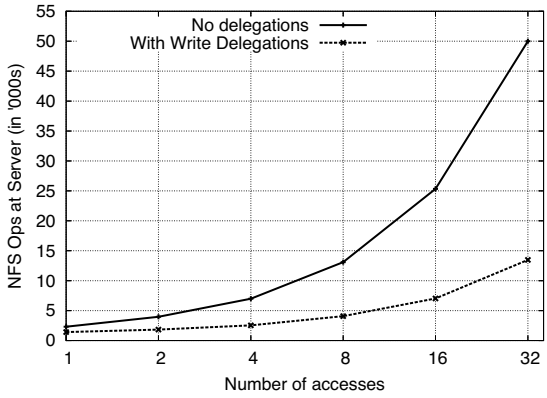


Figure 2: *Performance of Write Delegations: The graphs shows the number of ops (NFS packets) sent to the server with and without write delegations. The X axis denotes the number of times a single NFSv4 client opens, writes/reads and closes a file.*

As we have demonstrated, delegations can substantially improve cache performance and coherency guarantees for a single client that is frequently accessing a file. However, if read-write sharing is common (as is the case in joint software development), write delegations, if not granted intelligently, may make matters worse.

	LAN access	**WAN access**
No delegations	1.5-9 ms	150-500ms
Read delegation	1007 ms	1400 ms
Write delegation	1010 ms	1600 ms

Table 1: *Overhead of recalling a delegation. The table shows the time taken to complete a conflicting OPEN with delegation recall over a LAN and a WAN.*

Table 1 measures the overhead of recalling a delegation in terms of the delay observed by a conflicting OPEN (with read/write access) for both read and write delegations. The second OPEN is delayed until the delegation is recalled from the first client. With write delegations this also includes the time taken to flush all dirty data back to the server. We observe that the recall of a read delegation adds a one second delay to the second OPEN on Linux. This is because the NFSv4 client waits one second before retrying the OPEN on receiving a delay error from the server. The overhead with write delegations was similar to that with read delegations as the overhead of flushing data is low for a file smaller than 2MB. However, for large file writes the recall time is substantial. Figure 3 shows the client OPEN time with a write delegation recall when the size of the dirty data varies from 256KB to 32MB.

Clearly, the above experiments show that delegations

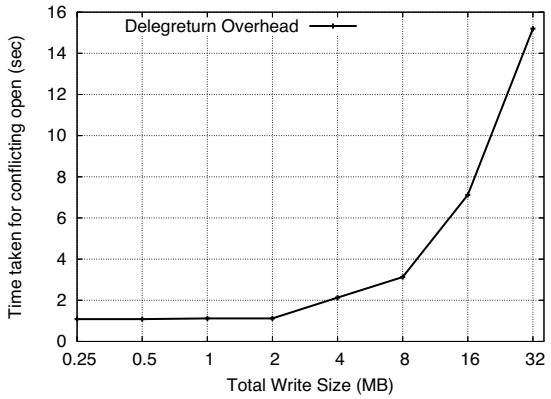


Figure 3: *Overhead of recalling a write delegation. The X-axis is the size of the data written to the file. The Y-axis is the time taken to complete a conflicting OPEN.*

are beneficial if a single client is exclusively accessing a file; and that any conflict can substantially affect the client response time. However, if it was possible for clients to *share* the delegations there would be no conflict. The underlying principle of Nache is to extend delegations and caching from a single client to a set of local clients that share both the cache and the delegation, thereby minimizing conflict. Even with read delegations that can be simultaneously awarded, Nache adds the benefit of a shared cache.

4 Nache Overview

The need for the Nache caching proxy arose from our earlier work on building a wide-area federated filesystem leveraging NFSv4 [16]. The goal of this system was to provide a uniform filesystem view across heterogeneous file servers interconnected by enterprise-wide or Internet-scale networks. Figure 4 shows the distribution of file servers and the common namespace view that they all export to the clients. The namespace was created by leveraging the client redirection feature of NFSv4. The client can mount the root of the common namespace tree from any server in the federation and see a uniform namespace.

4.1 Caching Proxy vs. Redirection

In such a federated system, with data on different geographically distributed physical locations, there are two models for data access: *client redirection and data shipping*. In the client redirection model, the client is referred to the server where the data actually resides for fetching it over the network. In the data shipping model, data from the remote server is cached on a server closer to the client to reduce frequent WAN accesses. Figure 5 shows the two access models.

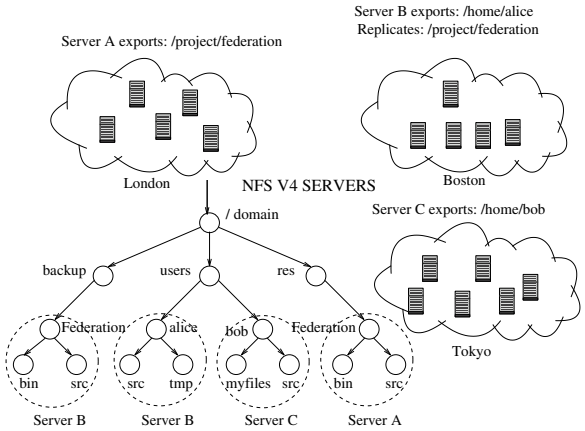
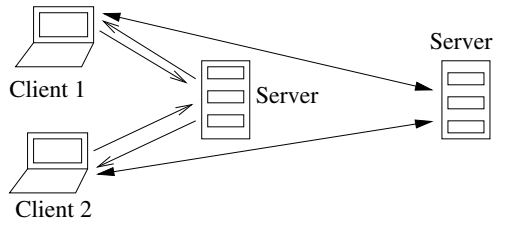
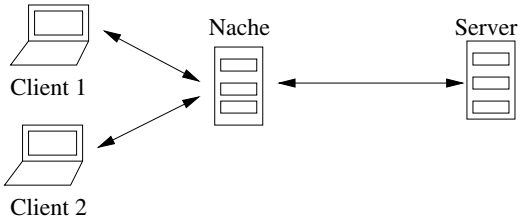


Figure 4: *Wide-area Federation of Filesystems*



(a) No Nache: Client redirection to remote server



(b) With Nache: Client access directly from proxy

Figure 5: *System Architecture (a) Client redirection without NFSv4 Proxy (b) With Nache*

The client redirection model relies on the standard NFSv4 client's ability to follow the protocol-specified referral. A client first mounts the root of the namespace from a local NFSv4 server. When it traverses a directory (filesystem) that happens to refer to a remote server location, the server initiates redirection and, when queried, returns an ordered list of remote server addresses and path names. The client then "sub-mounts" the directory (filesystem) from one of the remote servers in the list and continues its traversal.

Two factors contribute to the overhead of handling a redirection: (i) the processing overhead of following the referral along with the new sub-mount at the client, and (ii) the network overhead of accessing data from a remote server, possibly over a lower bandwidth, higher latency link. Figure 6 captures the processing overhead of following the referral. It shows that the time taken for

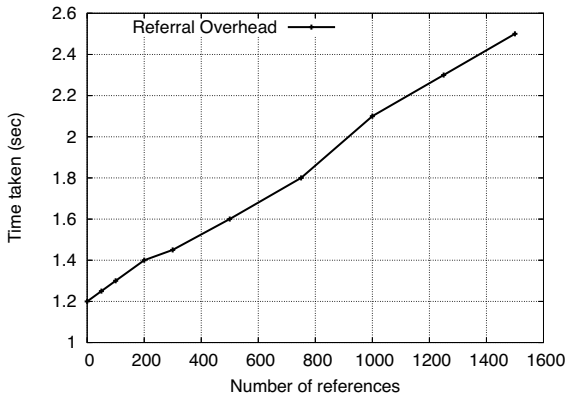


Figure 6: *Redirection overhead in traversing (ls -lR) a directory tree with 1500 directories. X-axis shows the number of redirections; Y-axis shows the response time of a tree traversal. All redirections point to the same path on the remote server.*

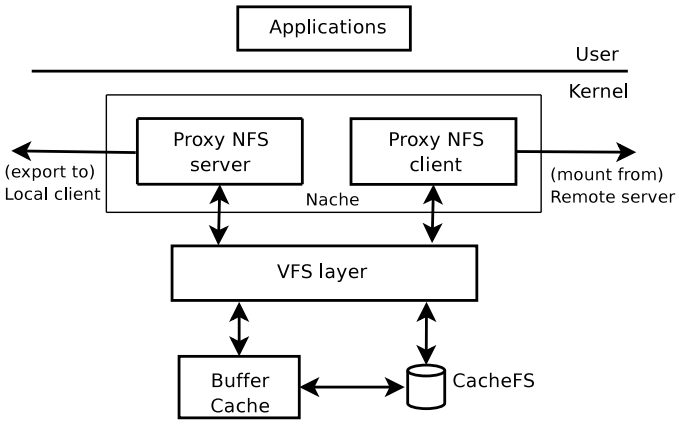


Figure 7: *Nache Architecture Block Diagram*

traversing an NFS mounted directory tree (with ls -lR) with 1500 directories and no re-directions is around 1.2 seconds. The time for a similar traversal where each directory is a referral to another server is 2.5 sec. In this example, all the referrals point to the same path on another server, thereby requiring only a single submount. As more servers get added to the federation, the time would further increase due to the additional submounts.

The network overhead of a redirection is, as expected, due to the latency and delay caused by a remote data transfer. For example, we measured the time taken to read a file of 8MB when the redirection was to a local (same LAN) server to be 0.7 secs while that when the redirection was to a remote server (over the WAN with the client in CA and the server in NY) was 63.6 secs.

When a number of co-located clients mount a filesystem from a local server they may each incur the redirection and remote access overhead. To reduce network latency, one obvious approach is to replicate the data at multiple servers and let the client select the closest replica location. However, replication is not feasible for all workloads as data may be constantly changing and what data needs to be replicated may not always be known.

Ideally, the data should be available locally on demand, kept consistent with respect to the remote server, and shared among all the local clients. The local server can act as a proxy by caching the remote data and forwarding requests that cannot be serviced locally to the remote server.

4.2 Nache Architecture

Conceptually, the Nache proxy is similar to any other proxy say a web caching proxy. However, a number of factors make it more challenging. First, NFSv4 is a stateful protocol with the server maintaining open state, clientids, lock owners, etc. Second, unlike web caches, data is both read and written and the close-to-open consistency guarantee has to be maintained. Third, other considerations of file handle management and security concerns make a file server proxy non-trivial.

Observe that the Nache proxy is different from a network layer-7 switch based routing proxies [4]. Such routing proxies typically terminate the TCP connection, parse the NFS packet information and route the packet to the appropriate server. They do not act as a fully functioning NFS server and client. The caching done in routing proxies is for read-only data where consistency support is not a concern. Moreover, the routing switch/appliance becomes a single bottleneck for all data to and from the client.

The Nache caching proxy relies on the delegation support in NFSv4 for improving cache performance. The delegation granted by the remote server to the proxy is shared among the local clients of the proxy. Thus, all operations on a file across multiple clients can be handled locally at the proxy. As we discussed in Section 3, in scenarios where a number of local clients are sharing a file for reads and writes, the overhead of a delegation recall is prohibitive. With the shared delegation model of Nache, a recall is avoided if all accesses are from local clients. If there is absolutely no sharing, however, it is better for clients to directly receive the delegations.

Figure 7 shows the block components of the Nache proxy architecture. It consists of the NFS server and client components that communicate via the VFS layer. CacheFS [19] is used to add persistence to the cache. Nache fits well in the NFSv4 model of a federated filesystem where each server exports the root of the common namespace and can act as a proxy for a remote server. On entering a directory that is not local, a server can either redirect the client to a local replica if data is replicated or act as a proxy to fetch the data locally.

5 Implementation

In essence, Nache acts as a bridge between NFSv4 clients and servers, handling client requests and forwarding them to target NFSv4 servers when there is a cache miss. It provides all the server functionality that NFS clients expect. Nache has been implemented on Linux by gluing together the client and server code paths for the various NFS operations. The implementation centers around three main areas: *Cascaded NFS mounts*, *NFS operation forwarding* and *Sub-operation RPC call handling*.

5.1 Cascaded Mounts

Nache enables an NFS client to access a remote NFSv4 filesystem locally by mounting it from the proxy. Internally, Nache mounts the remote filesystem and re-exports it so that local clients can access it over the NFS protocol. To provide such cascaded mounts, Nache must be able to export NFSv4 mounted filesystems. For this, we define *export operations* for the NFS client that allow client requests to be routed to remote servers.

We use Linux *bind* mounts to link the filesystem mounted from the remote server with a directory within the root of the exported pseudo filesystem at the proxy. Consider the case where a remote NFSv4 server `nfs4-server` is exporting a filesystem at `/export`, while `/nfs4` is the root of the pseudo filesystem exported by the proxy `nfs4-proxy`. In order for `nfs4-proxy` to re-export `/export`, we need to bind mount `/export` to a directory in the tree rooted at `/nfs4`, say at `/nfs4/export`. Here `/nfs4` is the root of the proxy's NFSv4 pseudo filesystem. This can be done at the proxy by the following sequence of commands:

```
mount -t nfs4 nfs4-server:/ /export
mount --bind /export /nfs4/export
```

The client can then access the exported filesystem from the proxy as:

```
mount -t nfs4 nfs4-proxy:/ /nfs
```

With cascaded mounts, the client can mount the remote server's filesystem from the proxy and access it locally at `/nfs/export`. We ensure that the proxy exports the remote filesystem using the appropriate options (`nohide, crossmnt`) that enable the NFSv4 client to view a filesystem mounted on another filesystem. Thus the modified code is able to export a remote filesystem that is mounted over NFS. The proxy is implemented by merging the functionality of the NFSv4 client and server kernel modules which communicate through an unmodified VFS layer. The interaction of the client and server components is shown in Figure 8. The NFSv4 client sends an RPC request to the proxy's server-side module (nfsd in Linux). The server-side module at the proxy forwards the call to the proxy's client-side module (nfs in Linux) using the VFS interface. Finally, the client-side module at the proxy forwards the call to the remote server if needed. The response from the remote server is stored at the client-side buffer cache and can be reused for later requests.

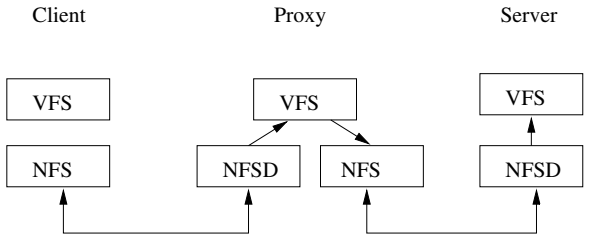


Figure 8: *Communication between kernel modules at client, proxy and server.*

The official Linux NFS kernel implementation does not permit re-exporting of NFS mounted filesystems (multi-hop NFS) because it is difficult to detect errors such as infinite mount loops. Moreover, there are concerns over host or network failures, access control and authentication issues along with the inefficiency of using an intermediate server between the NFS client and the file server in a LAN environment [36]. In a trusted enterprise environment with WAN access, however, multihop NFS can be used to reduce the high latency by caching locally at the proxy. NFSv4 also has better security, authentication and failure handling support that can be leveraged by the proxy. We discuss the security issues later in Section 5.5.

5.2 NFS Operation Forwarding

Once an NFS operation (such as LOOKUP or OPEN) is received at the Nache server-side module (Nache server), it is forwarded to the client-side module (Nache client) via the VFS interface. Thus the NFS request gets translated to a VFS call which then calls the corresponding operation in the underlying local filesystem. In the proxy case, this is the *NFS mounted* remote filesystem.

The translation between an NFS request to a VFS call and back to an NFS request works without much modification in most cases. However, operations that are "stateful" require special handling both at the Nache server and at the Nache client. Additionally, calls from the Nache server to the VFS layer need modifications to make them appear to have originated from a local process at the proxy. In the following discussion we will describe in detail some of the operations that need special attention.

- *OPEN:* The OPEN request processing required modifications due to the integrated handling of NFSv4 OPEN and LOOKUP operations in Linux. In the NFSv4 client, processing a file OPEN triggers a LOOKUP operation and the OPEN request is sent to the server as part of LOOKUP. The NFS client's

generic file open function (`nfs_open`) does not actually send an OPEN request to the server. This created a problem at the proxy during an OPEN request because the Nache server was invoking the generic filesystem open function through the VFS layer which translated to `nfs_open` for an NFS mounted filesystem. This function, however, would not send an OPEN request to the remote server.

To resolve this, we modified the open operation in the Nache server to emulate the way a local file open would have been seen by the Nache client. In Linux, this includes extracting the arguments of the open and calling the appropriate lookup function in the Nache client. The NFSv4 server stores state (`nfs4_stateid`) associated with a file OPEN request. To obtain this state at Nache, we modified the return path of the OPEN to extract the relevant state created at the Nache client and populate the stateid structure in the Nache server.

- *CREATE:* In the normal Linux NFS client, the create function (`nfs_create`) needs some data (`nameidata`) that is populated by the VFS layer. In Nache, when translating between the server-side and client-side functions, the information to do this initialization is not readily available. In the Nache server, on a create request, we extract the open flags and the create mode and use it to initialize `nameidata` before invoking the VFS function, `vfs_create()`. That, in turn, calls the Nache client's `nfs_create()` with the required data already initialized.
- *LOCK:* When a LOCK request is processed at the Nache server, it gets translated to the underlying POSIX function. The POSIX lock function, however, does not call the associated NFS locking (`nfs_lock`) function in the Nache client. This is because most regular filesystems do not specify their own lock operation. Hence, none of the lock requests (LOCK, LOCKU) would get forwarded to the remote server. We modified the lock function in the Nache server to check if the inode on which the lock is desired belongs to an NFS mounted filesystem and call the corresponding lock operation of the Nache client.
- *CLOSE:* This happens to be one of the most complicated operations to handle. The CLOSE operation depends on the state associated with the file at the client. A client with multiple OPENs for the same file only sends one CLOSE to the server (the last one). At the Nache server, there are two scenarios that could occur: (i) the multiple opens are all from the same client, (ii) the multiple opens are from different clients. Although the number of CLOSE requests received by the Nache server is equal to the number of distinct clients, the Nache client should only send one CLOSE to the remote server. To handle a CLOSE, we have to keep track of the state associated with a file and make sure that the file open counters are adjusted properly during the OPEN and CLOSE operations. The counters should be incremented only once per client and not for every OPEN request from the same client. This client state is difficult to maintain at the Nache server. In our current implementation, we experimented with various options of counter manipulation and eventually chose to simply count the number of open requests seen by the proxy across all clients. This implies that some CLOSE requests may not be sent to the remote server in cases where a client opens the same file multiple times. We expect to provide a better solution as we continue the development of Nache.

5.3 Sub-operation RPC Calls

Another issue that arose due to performance-related optimizations is that the NFS server while handling a request does low level inode operations which, in Nache, result in RPC calls sent to the remote server from the proxy. The NFS server is optimized for doing local filesystem operations and not for remote filesystem access. As a representative example, consider the OPEN request at the server. In processing the OPEN (`nfsd4_open`), the server calls other local operations for lookup (`do_open_lookup`) and permissions checking (`do_open_permissions`). In Nache, however these operations cannot be done locally and are sent to the remote server. Thus a single OPEN call leads to three RPC calls exchanged between Nache and the remote server. To avoid this, we reorder these checks in the usual path in the Nache server code as they would be eventually done in the OPEN processing at the remote server. Numerous such optimizations are possible to reduce the number of remote accesses.

5.4 CacheFS and Persistence

To add persistence to the cache, increase cache capacity, and survive reboots, Nache relies on CacheFS [19]. CacheFS is a caching filesystem layer that can be used to enhance the performance of a distributed filesystem such as NFS or a slow device such as a CD-ROM. It is designed as a layered filesystem which means that it can cache a *back* filesystem (such as NFS) on the *front* filesystem (the local filesystem). CacheFS is not a standalone filesystem; instead it is meant to work with the front and back filesystems. CacheFS was originally used for AFS caching but is now available for any filesystem and is supported on many platforms including Linux. With CacheFS, the system administrator can set aside a partition on a block device for file caching which is then

mounted locally with an interface for any filesystem to use. When mounting a remote NFS filesystem, the admin specifies the local mount point of the CacheFS device. In Nache, CacheFS is used primarily to act as an on-disk extension of the buffer cache.

CacheFS does not maintain the directory structure of the source filesystem. Instead, it stores cache data in the form of a database for easy searching. The administrator can manually force files out of the cache by simply deleting them from the mounted filesystem. Caching granularity can range from whole files to file pages. CacheFS does not guarantee that files will be available in the cache and can implement its own cache replacement policies. The filesystem using CacheFS should be able to continue operation even when the CacheFS device is not available.

5.5 Discussion

Security Issues In a proxy setting, security becomes an important concern. The proxy can provide clients access to read and modify data without the server knowing or verifying their authority to do so. The current implementation of Nache does not include any additional security support beyond what exists with vanilla delegations. Presently, the server returns the list of access control entries when a delegation is awarded. However, the user ID space at the proxy and server maybe different. In such cases the proxy resorts to sending an ACCESS request to the server to verify the access permissions for that client for every OPEN. In case Nache is deployed as part of a managed federation, the client access control can be centrally managed.

Delegation Policy Another issue that needs attention is the policy used by the server to decide when to award a delegation. For example, the current implementation in Linux awards a delegation on the second OPEN by the same client. This policy may be too liberal in giving out delegations which must be recalled if there is a conflicting access. It may not be feasible to implement a complex policy based on access history of each file within the kernel. As part of the Nache implementation, we are exploring different policies that try to maintain additional access information about the file to better grant delegations.

Protocol Translation One interesting use of Nache is as a protocol translator between NFS versions. Nache behaves exactly like a v4 server to v4 clients and as a v3 server to v3 clients. This is useful when leveraging client features in NFSv4 such as redirection and volatile file handles and the WAN-friendly protocol features. The file delegation happens between the proxy acting as a v4 client and the back-end v4 server, hence the consistency semantics are maintained for both v3 and v4 clients. Efforts to integrate with an NFSv3 client are ongoing in Nache.

Failure Handling Any component of the system — client, proxy, or server can fail during an NFS operation. In general the failure of the server can be masked somewhat by the proxy as it can respond to reads from the cache (although these can no longer be backed by delegations). However, the failure of the proxy will affect the clients in two ways. First, the clients will not be able to access the data on the server even when the server is operational. Second, the writes done at the proxy that have not been flushed back to the server may be lost. Some of the problems of write delegations to a client without a proxy are further complicated with the use of a proxy as dirty data could have been read by multiple clients. If the writes are flushed periodically to the server the lag between the proxy and server state can be reduced.

6 Experimental Evaluation

The Nache implementation is based on CITI, University of Michigan's NFSv4 patches [13] applied to Linux kernel version 2.6.17. User-space tools required on Linux (such as nfs-utils) also have CITI patches applied for better NFSv4 capabilities. We used IBM xSeries 335 servers with Intel Pentium III (1133MHz) processor, 1GB of RAM, 40GB IDE disk with the ext3 filesystem for our experimental testbed. For the WAN access experiments, we used machines over a wide area network between IBM Almaden (California) and IBM Watson (New York) that had a round-trip ping delay of about 75 msec. One of the local machines was set up as the Nache proxy that runs the kernel with *nfs* and *nfsd* kernel modules suitably modified for proxy implementation. The remote server machine's *nfsd* module is also modified to enable write delegations and provide a fix for the COMMIT operation on a delegated file, as we discuss later in this section.

The evaluation is divided into four categories. First, we evaluate the delegation support currently in Linux. Next, we experiment with write delegations and their performance with certain workloads. We argue that write delegations should be awarded more carefully. Third, we test Nache with a set of workloads for gains achieved in terms of the total NFS operations sent to the server and the time taken to complete certain file operations. Some of the workloads are based on the different profiles available in Sun's filesystem benchmark Filebench [24]. The setup consists of two or more clients, a Nache proxy and one server. We compute the benefits with Nache in scenarios where clients show overlap in their data access. Finally, we measure the overhead of deploying Nache especially in scenarios where there is no sharing among clients, thereby, limiting the benefits of a shared cache.

6.1 Effect of Delegations on NFSv4 Operations

We discussed the advantages and shortcomings of the NFSv4 delegation support in the Linux kernel in Section 3. Here we further investigate how individual file operations are affected when a delegation is granted. Recall that a delegation is awarded to a client when a callback channel has been successfully established between the server and the client for a recall; and a client opens a file twice (not necessarily the same file). As mentioned in Section 3, presently the Linux server only hands out read delegations, while we added a prototype implementation for awarding write delegations. Our prototype uses the same policy as used by read delegations, namely a write delegation is granted on the second OPEN for write from the same client. With our changes, we observed that the server did not recall delegations correctly on a conflicting OPEN and was not handling a COMMIT from a client with write delegations properly. The server tried to recall the delegation on a COMMIT which the client would not return until the COMMIT succeeded. This deadlock caused new OPENs to see long delays. Our prototype fixes the delegation recall handling on a conflicting OPEN and the COMMIT on a delegated file. We expect these issues to be correctly resolved when write delegations are officially supported in the Linux kernel.

NFS Op.	Delegation ON	Delegation OFF
OPEN	101	800
CLOSE	101	800
ACCESS	101	1
GETATTR	1	700
LOCK	101	800
LOCKU	1	800
READ	100	100

Table 2: *The number of NFSv4 operations seen by the server with and without read delegations.*

To assess activity locally and at the server in the presence of delegations, we performed a sequence of OPEN-READ-CLOSE operations over 100 files and repeated it 8 times. Table 2 shows the operations with read delegations enabled. Observe that there are 101 OPEN (and CLOSE) operations that are sent to the server with read delegations. This is because the first OPEN from the client does not get a delegation as per the server decision policy and is sent again to the server on the second run. All other files only require one OPEN to be sent to the server. However, as per the protocol, the client must check caller's access rights to the file even when delegations are available (after the attribute timeout has expired). We detect one such timeout during our run, hence observe 101 ACCESS calls in the presence of delegations. Note also that no GETATTR requests need to be sent to the server for revalidation when read delegations have been obtained. Also the number of reads sent to the server are the same with and without delegations as the file data is cached after the first OPEN in both cases. Similarly, we introduced the LOCK-ULOCK pair in the sequence to determine the LOCK operation behavior and observe that unlocks can be serviced locally when delegations are available.

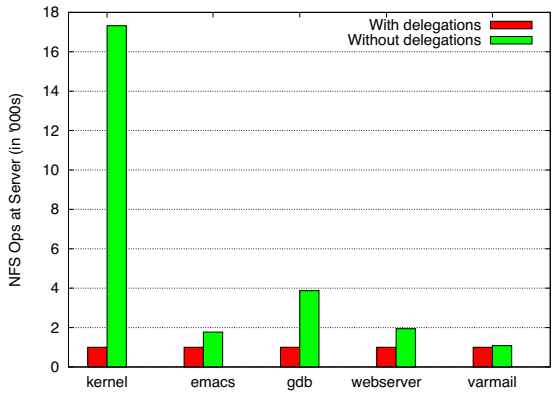


Figure 9: *Performance of Read and Write delegations (total ops at server): the Y-axis shows the server ops with and without delegations.*

We repeated the experiments with write delegations and found that all the writes are flushed to the server on a CLOSE although they can be kept dirty if delegations are in place. The inconsistent results are due in part to the fact that the Linux server's delegation implementation is incomplete.

To further study the benefits of delegations, we used different workloads to evaluate the performance. One set of workloads consist of compilations of different source code packages, namely the Linux kernel, Emacs and GDB, where the source tree is mounted over NFS from a remote server. Another set includes webserver and varmail profiles from the *filebench* benchmark. Figure 9 shows the server operations for different workloads with read and write delegations enabled. Here the number of operations at the server are 16 to 1.2 times lower for the different workloads when delegations are granted. Notice the massive benefits in the compile of the Linux kernel compared to those of Emacs and GDB. This is because the kernel is relatively self-contained and does not reference many files outside the source tree. On the other hand, the compiles of Emacs and GDB use standard include files and libraries during the build which are accessed from the local filesystem. Figure 10 shows the benefits in terms client response time and throughput for the same workloads.

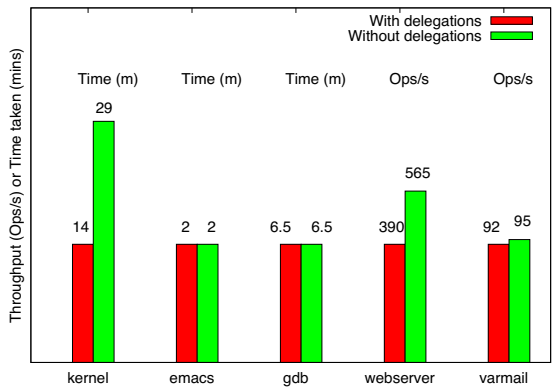


Figure 10: *Performance of Read and Write delegation (total throughput): the Y-axis shows the server throughput (ops/sec) or response time with and without delegations. The number at the top of the bar is the actual value.*

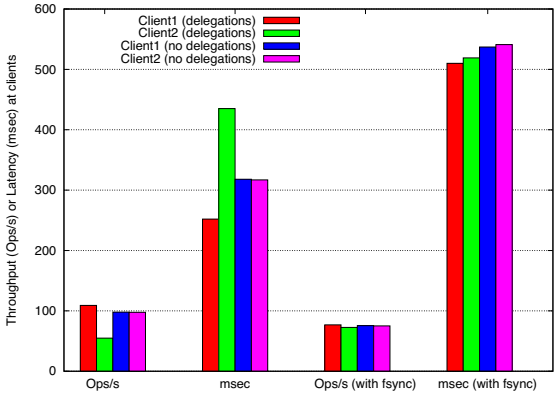


Figure 11: *Effect of WRITE delegation on a workload with write sharing among clients. The 4 bars in each set show the performance of Client1 with write delegation, Client2 with a conflicting write and Client1 and Client2 with delegations disabled respectively.*

6.2 Impact of Write Delegation Policy

In Section 3, we showed that the overhead of a delegation recall can be substantial enough to negate its benefits. Here, we further investigate the impact for delegation recalls in workloads with concurrent writes. To simulate such a workload, we modified Filebench to use multiple threads each doing a sequence of open-read-append-close operations on a set of files directly from the server. Because of write delegations being awarded and recalled, the performance varies by a large degree depending on the number of clients. With two clients doing concurrent writes, one client's throughput was observed to be 109 ops/sec while the other client's was half that at 54 ops/sec. Similarly, the average latencies observed by the clients were 252 ms and 435 ms, respectively. With write delegations disabled, both the clients had similar performance. The first two bars show these results in Figure 11.

We had also shown in Section 3 that the time taken for a conflicting OPEN increases substantially with the amount of dirty data that needs to be flushed on a delegation recall. However, if the dirty data is synced at regular intervals, the time taken to respond to a conflicting OPEN should be significantly reduced. To verify this we added a `fsync()` operation before the `close()` in the Filebench workload for each file.

This improved the throughput and latency for the second client and both achieved 75 ops/sec with average latencies of 531ms and 548ms respectively. These results are shown in the last two bars of Figure 11. Keeping the dirty data size less than 1MB seems to be a good heuristic as the overhead remains fairly constant when the unwritten data ranges from a few KBs to 1 MB. Note that latencies are worse with `fsync()` because data is written more frequently.

Similarly, Figure 12 shows the average latency for

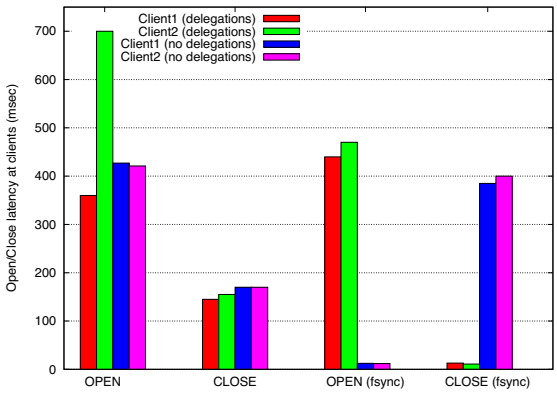


Figure 12: *Effect of WRITE delegation on latencies of OPEN and CLOSE operations.*

OPEN and CLOSE operations for each of the two clients in the presence of write delegations and with the optional periodic `fsync()`. We observe that the second client's OPEN takes twice the amount of time as the first client when write delegations are handed out. This reenforces the claim that the delegation policy can have an adverse affect on the performance if there are conflicting accesses.

6.3 Performance Benefits of Nache

In this section we evaluate the performance benefits of Nache using different workloads. In the experimental testbed, a group of client machines access a remote file server via the same proxy server. For results demonstrating LAN access, all machines are on the same 100 Mbps LAN. For experiments with WAN access, the access is over an enterprise network between California and New York.

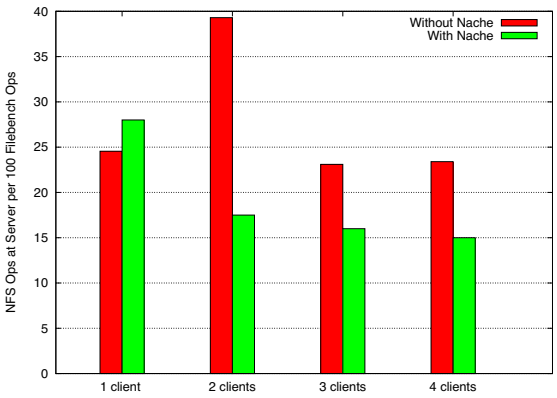


Figure 13: *Benefits of Nache: Filebench Web server workload.*

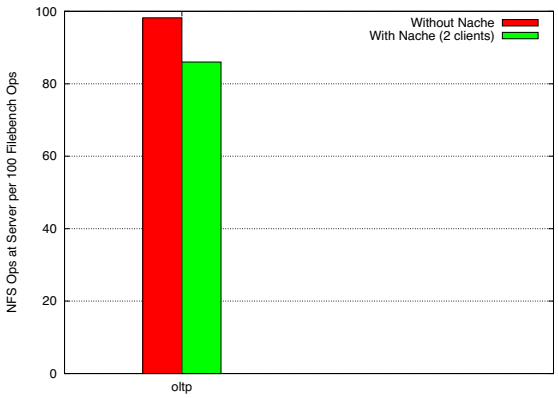


Figure 14: *Benefits of Nache: Filebench OLTP workload.*

6.3.1 Filebench Workloads

The Filebench benchmark contains a number of profiles that are representative of different types of workloads. Some of them are: a webserver with read-only data, an OLTP transaction processing system, a *varmail* workload (similar to the Postmark benchmark), and a web proxy workload (*webserver*). We provide results for the webserver and OLTP workloads.

Webserver: This workload generates a dataset with a specified number of directories and files using a gamma distribution to determine the number of sub-directories and files. It then spawns a specified number of threads where each thread performs a sequence of open, read entire file and close operations over a chosen number of files, outputting resulting data to a logfile. As part of the runs, we observed that Filebench was accessing all files uniformly with no skew which we believe is not representative of a typical webserver access pattern [11]. We modified the access pattern function to select files based on a Zipf distribution [11]. We obtained results for both the uniform and Zipf file access pattern using 500 files, 10 threads and a run time of 100 seconds. Figure 13 shows the total number of operations sent to the server normalized with respect to the total number of Filebench operations. The normalization was done to remove the effect of any variance in the total number of operations generated by filebench as the number of clients varied. We observe that Nache reduces the number of operations seen at the server by 38% with four clients.

OLTP: The OLTP workload is a database emulator using an I/O model from Oracle 9i. This workload tests for the performance of small random reads and writes. In our experiments we use 20 reader processes, 5 processes for asynchronous writing, and a log writer. Since Filebench was originally written for Solaris and modified to work on Linux, we found that it was quite unstable in running the OLTP profile, possibly due to the asynchronous I/O requests. We could not reliably run the OLTP workload for more than two clients. Figure 14 shows the total number of operations sent to the server normalized with respect to the total number of filebench operations. Observe that with Nache, the server operations are reduces by 12.5% for two clients.

6.3.2 Software Builds

In this experiment we consider the scenario of a joint software development project where clients build individual versions of a large software package for local testing. This involves using a common source tree at a remote server and building it with locally modified files. We performed the build of three software packages: Linux kernel (version 2.6.17), GDB (version 6.5), and Emacs (version 21.3). In each case, the directory containing the source was NFS mounted at the proxy and the object and executable files generated during the compilation are written locally at the client.

Figures 15, 16, 17 show the number of NFS operations sent to the server for each of the three workloads with varying number of clients sharing the proxy cache. Based on these experiments we observe that: (i) with Nache, the operations at the server decrease or stay flat as the number of clients increase (i.e., there is more sharing), (ii) without Nache, the operations at the server linearly increase with the number of clients (as do the operations at the proxy with Nache), (iii) with Nache, the time taken over a LAN for various builds stays constant as the number of clients increase (iv) with Nache, the time taken for various builds decreases over WAN (as shown in Figure 18). For example, for the kernel build in Figure 15, the server operations are reduced by more than 50% with Nache and 2 clients. Interestingly, with Nache the number of operations at the server with multiple clients is sometimes less than that for a single client. This is because some operations such as CLOSE will not be sent to the server if multiple clients have opened the file via Nache. Furthermore, the number of granted

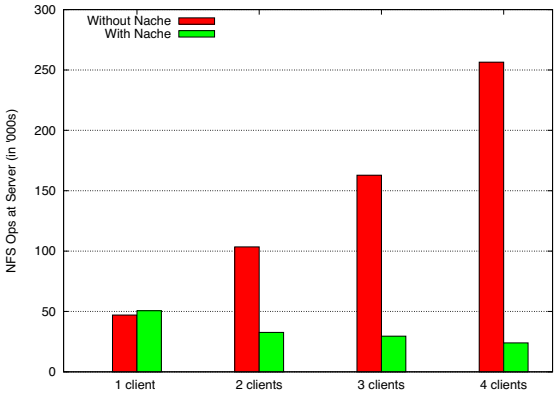


Figure 15: *Benefits of Nache: Compile of Linux kernel.*

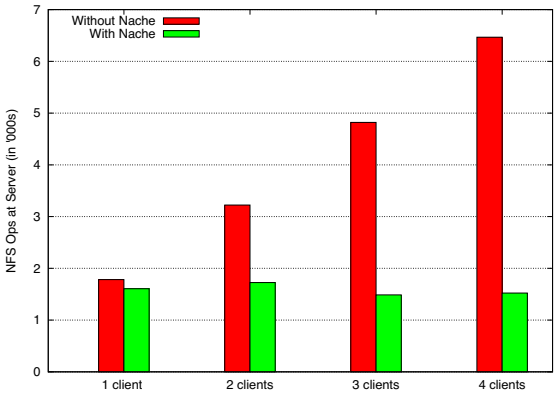


Figure 17: *Benefits of Nache: Compile of Emacs.*

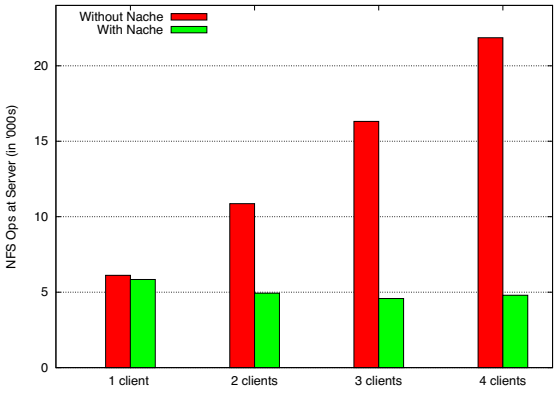


Figure 16: *Benefits of Nache: Compile of GDB.*

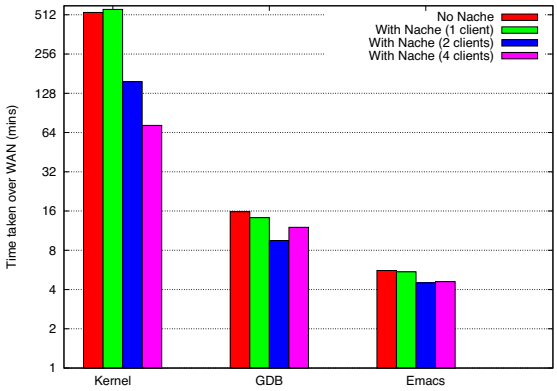


Figure 18: *Effect of Proxy on response time over a WAN.*

delegations is higher as more files may be opened concurrently with two clients than with one client. This is an artifact of the way the server awards delegations (on the second concurrent OPEN from a client) rather than an inherent benefit of the proxy. Similarly, the response time on a WAN reduces by 70% for the same build. Observe that the kernel compile had the best performance improvement when using a proxy for the same reasons as discussed in Section 6.1.

Response time over WAN We repeated the software build experiments over a WAN to measure the latency improvements with Nache. Figure 18 shows the time taken to build the Linux kernel, GDB and Emacs with and without Nache. As in the LAN case, the source code is NFS mounted, but the object files and executables are stored locally.

For a single client accessing data directly from the remote server without Nache, the time taken is 533min, 15.8min and 5.6min respectively. The response time decreases as we increase the number of clients going through Nache. In case of the kernel build, the response time is slightly higher with a single client due to the overhead incurred by an intermediary proxy and the absence of sharing. With two clients, on the other hand, the response time is 3.5 times lower than without Nache. The marked improvement in response time is due in part to the fewer operations sent to the server as we discussed earlier.

6.4 Measuring Proxy Overhead

In certain scenarios such as a single client accessing files or when there is no file sharing, Nache simply adds to the data path without any added benefit of a shared consistent cache. In this section, we measure the overhead of the proxy in such scenarios using micro-benchmark tests. The workloads used for the micro-benchmark are as follows:

- *Create Files:* Creates 25,000 files, writes up to 16KB of data and then closes all of them.
- *Random Reads:* Performs random reads on a large 1GB file using a single thread.
- *Random Writes:* Performs random writes on a large 1GB file using a single thread.

Micro-benchmark	ops/sec (Nache)	ops/sec (NFSv4)	latency (Nache) ms	latency (NFSv4) ms
Create Files	37	40	1283	1172
Random Reads	99	127	1.3	7.8
Random Writes	27	32	37.6	31.3
Random Appends	9	11.9	220	146
Create Files (WAN)	9	16	5217	3009
Random Reads (WAN)	77	48	12.9	20.7
Random Writes (WAN)	8.6	10	116.3	98
Random Appends (WAN)	2.2	2.4	883.4	821

Table 3: *Micro-benchmark results and comparison among configurations with and without Nache*

- *Random Appends:* Creates a dataset with multiple files and does append-fsync on the files with random append sizes ranging from 1KB to 8KB. The file is flushed (fsync) every 10MB up to the maximum file size of 1GB.

We ran all the benchmarks over a LAN and a WAN to measure latencies for the different workloads. Table 3 shows the operation rate (ops/sec), the number of NFS operations sent to the server and the average latency for each of the micro-benchmarks. The results show that overhead is higher (7-40% worse) in create-intensive cases compared to the other scenarios. This is expected because Nache just acts as another router and seems to provide no benefit in terms of data or metadata caching. Some gains are observed in case of random reads on WAN which can be attributed to caching at the proxy. The Nache code is not fine-tuned and some of the overhead should decrease with some optimizations.

7 Related Work

Caching has always been used in distributed filesystems to improve performance. Most popular distributed filesystems rely on a client-server architecture where caching is done primarily at the client. While the various NFS versions have supported client-side caching, they enforce only weak cache consistency. NFS extensions such as Spritely-NFS and NQNFS tried to improve the NFS consistency semantics. Spritely-NFS [35] used the Sprite [27] cache consistency protocols and applied them to NFS. This allowed for better cache consistency by using server callbacks. NQNFS [23] also aimed at improving NFS consistency semantics but differed from Sprite in the way it detected write sharing.

While NFS was more suited for LAN access, the AFS [2, 17, 18] filesystem was designed for wide-area access. For this, AFS relied extensively on client-side file caching and supported cache consistency through callbacks. The successor to AFS was the DFS [20] filesystem which had most of the features of AFS but also integrated with the OSF DCE platform. DFS provided better load balancing and synchronization features along with transparency across domains within an enterprise for easy administration. AFS also led to the Coda [22] filesystem that dealt with replication and client-side persistent caching for better scalability while focusing on disconnected operations.

Along with NFS and AFS, which are more prevalent on Unix platforms, Microsoft Windows clients use the CIFS (Common Internet File System) [34] protocol to share data over a network. CIFS provides various optimizations such as batched messages, opportunistic locks for stronger cache coherency, and local buffering to improve response times and save network round trips. The Microsoft DFS filesystem leverages the CIFS protocol to create a filesystem federation across multiple hosts [25].

In case of AFS, along with client-side caching, Muntz-Honeyman [26] analyzed the performance of a multi-level cache for improving client response times in a distributed filesystem. They concluded that multi-level caching may not be very useful due to insufficient sharing among client workloads. While it is known that the effectiveness of an intermediate cache is limited by the degree the sharing across clients, we believe that remote collaboration has significantly increased in the last decade due to advances in network bandwidth and improvements in collaborative tools. Current web workloads, for example, show a high degree of sharing of "hot" documents across clients [11]. Similarly, distributed collaborative projects have increased with global outsourcing. In Nache, we show that even when sharing is low (say 8-10%), the gain in response time can be high when data is accessed across a WAN. Moreover, Muntz-Honeyman's paper shows that an intermediate proxy can substantially reduce the peak load at the server. Thus, along with client response time, a proxy can also improve the server scalability by reducing server overload. We observed with Nache that even a very low degree of sharing can eliminate all the gains of a pure client-side cache due to the recall of delegations on a conflicting

access. This suggests that a shared proxy cache is beneficial to reduce conflicts if cache consistency is desired.

The client-side cache can also be optimized by making it persistent and policy-based. Nache relies on CacheFS for on-disk caching. Xcachefs is similar to CacheFS in that it allows persistent caching of remote filesystems but further improves performance by de-coupling the cache policy from the underlying filesystem [33]. It allows clients to augment the caching mechanism of the underlying filesystem by specifying workload specific caching policies.

Recently a plethora of commercial WAFS and WAN acceleration products have started offering caches for NFS and CIFS protocol for improving wide-area performance. These often use custom devices both in front of the server and the client with an optimized protocol in between [5, 3].

Although proxy caching is not that prevalent in file serving environments, it has been widely used in the Web due in part to the read-only nature of the data and the high degree of WAN accesses. The Squid proxy cache that grew out of the Harvest project [14, 15] uses a hierarchical cache organization to cache FTP, HTTP and DNS data.

While Nache focuses on improving the wide-area access performance of existing file servers, numerous research efforts have focused on building scalable file servers. Slice [7] implements a scalable network-wide storage by interposing a request switching filter on the network path between clients and servers. Clients see a unified file volume and access it over NFS. Slice is mainly designed to provide a scalable, powerful NAS abstraction over LAN whereas our main goal is to improve file serving performance over a WAN. The Tiger file server [10] provides constant rate delivery by striping the data across distributed machines (connected via high speed ATM) and balancing limited I/O, network and disk resources across different workloads. Farsite [6] implements a server-less distributed filesystem that provides the benefits of shared namespace, location transparency and low cost. Thus it transforms unreliable local storage at clients to a more reliable, logically centralized storage service. xFS is another server-less distributed filesystem that uses cooperative caching to improve performance [8]. Lustre [1] is an object based distributed filesystem that is designed to work with object based storage devices where controllers can manipulate file objects. This leads to better I/O performance, scalability, and storage management. While these and other efforts [31, 30] have focused on improving file serving performance, they are not designed for improving the performance of existing file servers and NAS appliances.

8 Conclusion

In this paper, we have presented the design and implementation of a caching proxy for NFSv4. Nache leverages the features of NFSv4 to improve the performance of file accesses in a wide-area distributed environment. Basically, the Nache proxy sits in between a local NFS client and a remote NFS server caching the remote data closer to the client. Nache acts as an NFS server to the local client and as an NFS client to the remote server. To provide cache consistency Nache exploits the read and write delegations support in NFSv4. We highlighted the three main contributions of the paper. First, we explored the performance implications of read and write open delegations in NFSv4. Second, we detailed the implementation of the Nache proxy cache architecture on the Linux 2.6 platform. Finally, we discussed how to leverage delegations to provide consistent caching in the Nache proxy. Using our testbed infrastructure, we demonstrated the performance benefits of Nache using the *Filebench* benchmark and different workloads. In most cases the Nache proxy can reduce the number of operations seen at the server by 10 to 50%.

As part of on going work we are exploring different policies for awarding read and write delegations to lower the probability of a conflict. Also the Nache architecture is being integrated with the federated filesystem architecture that provides a common file-based view of all data in an enterprise.

References

[1] Cluster File Systems Inc., Lustre: A Scalable, High-Performance File System. `http://www.lustre.org/docs/whitepaper.pdf`.

[2] OpenAFS: `http://www.openafs.org`.

[3] Packeteer Inc., Wide Area File Services: Delivering on the Promse of Storage and Server Consolidation at the Branch Office. `http://www.packeteer.com/resources/prod-sol/WAFS_WP.pdf`.

[4] Acopia Networks, Intelligent File Virtualization with Acopia. `http://www.acopianetworks.com/pdfs/adaptive_resource_networking/Intelli%gent_File_Virtualization_wp.pdf`, Nov. 2006.

[5] Expand Networks, WAN Application Acceleration for LAN-like Performance. `http://www.expand.com/products/WhitePapers/wanForLan.pdf`, Nov. 2006.

[6] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment, Dec 2002.

[7] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed Request Routing for Scalable Network Storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, Oct 2000.

[8] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *"Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles"*, Dec 1995.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] W. Bolosky, J. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, and R. Rashid. The Tiger Video Fileserver. In *Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'96)*, Apr 1996.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. IEEE INFOCOM*, pages 126–134, 1999.

[12] B. Callaghan. *NFS Illustrated*. Addison-Wesley Longman Ltd., Essex, UK, 2000.

[13] CITI. Projects: NFS Version 4 Open Source Reference Implementation. `http://www.citi.umich.edu/projects/nfsv4/linux`, Jun 2006.

[14] J. Dilley, M. Arlitt, and S. Perret. Enhancement and validation of the Squid cache replacement policy. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[15] D. Hardy and M. Schwartz. Harvest user's manual, 1995.

[16] J. Haswell, M. Naik, S. Parkes, and R. Tewari. Glamour: A Wide-Area Filesystem Middleware Using NFSv4. Technical Report RJ10368, IBM, 2005.

[17] J. Howard and et al. An Overview of the Andrew Filesystem. In *Usenix Winter Techinal Conference*, Feb 1988.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[19] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proceedings of the Linux Symposium*, volume 1, Jul 2006.

[20] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Buttos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and E. R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer USENIX Technical Conference*, 1990.

[21] S. Khan. NFSv4.1: Directory Delegations and Notifications, Internet draft. `http://tools.ietf.org/html/draft-ietf-nfsv4-directory-delegation-01`, Mar 2005.

[22] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[23] R. Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 261–278, San Fransisco, CA, USA, 1994.

[24] R. McDougall, J. Crase, and S. Debnath. FileBench: File System Microbenchmarks. `http://www.opensolaris.org/os/community/performance/filebench`, 2006.

[25] Microsoft. Distributed File System (DFS). `http://www.microsoft.com/windowsserver2003/technologies/storage/dfs/default.mspx`.

[26] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 305–313, San Fransisco, CA, USA, 1992.

[27] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[28] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *USENIX Summer*, pages 137–152, 1994.

[29] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. In *Proceedings of Second International System Administration and Networking (SANE) Conference*, May 2000.

[30] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, 1999. IEEE Computer Society Press.

[31] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[32] S. Shepler and et al. Network File System (NFS) version 4 Protocol. RFC 3530 `http://www.ietf.org/rfc/rfc3530.txt`.

[33] G. Sivathanu and E. Zadok. A Versatile Persistent Caching Framework for File Systems. Technical Report FSL-05-05, Computer Science Department, Stony Brook University, Dec 2005.

[34] SNIA. Common Internet File System (CIFS) Technical Reference. `http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf`.

[35] V. Srinivasan and J. Mogul. Spritely nfs: experiments with cache-consistency protocols. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 45–57, Dec. 1989.

[36] H. Stern, M. Eisler, and R. Labiaga. *Managing NFS and NIS*. O'Reilly, Jul 2001.