C2Cfs: A Collective Caching Architecture for Distributed File Access

Andrey Ermolinskiy
Renu Tewari

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-40
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-40.html

March 15, 2009

Copyright 2009, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

C2Cfs: A Collective Caching Architecture for Distributed File Access

Abstract

In this report we present C2Cfs - a novel collective caching architecture for distributed filesystems. C2Cfs diverges from the traditional client-server model and advocates decoupling the consistency management role of the central server from the data serving role. Our design enables multiple client-side caches to share data and efficiently propagate updates via direct client-to-client transfers, while maintaining the standard consistency semantics. We present an NFSv4-based implementation of our architecture, which works with unmodified NFS servers and requires no changes to the protocol. We also evaluate the implementation and demonstrate the performance benefits of decentralized data access enabled by our approach.

1 Introduction

Medium and large commercial enterprises, engineering design systems, and high performance scientific applications, all require sharing of massive amounts of data across a wide-area network in a reliable, efficient, consistent, highly available, and secure manner. While a data sharing infrastructure that fulfills all of these objectives remains an elusive, if not altogether impossible goal, a number of different approaches and designs have been explored over the years.

At one end of the spectrum, traditional networked filesystems, such as NFS [18] and AFS[11], follow the *client-server* paradigm and enable a collection of client machines to access a shared filesystem exported by a remote central server. In this model, the server plays the role of the authoritative "owner" of the data.

At the other end, we have seen a number of proposals for fully decentralized distributed filesystems such as xFS [5] that employ the *anything-anywhere* principle and eliminate the notion of centralized ownership. In xFS, the owner of a file is not statically defined, but can move among a set of cooperating servers. In general terms, a serverless distributed storage architecture requires maintaining an additional set of mappings between logical data identifiers (e.g., filenames) and the locations of their respective owners. These mappings can either be stored centrally or distributed across the participants. In the extreme case, fully decentralized P2P storage systems such as Ivy [15] and OceanStore [13] use the distributed hash table (DHT) primitive to locate the owner(s) of a given datum and retain availability in the face of membership churn.

While the client-server model is attractive due to its simplicity, centralized file access has several major drawbacks. Fundamentally, it constrains the movement of data to a star-like topology and does not take advantage of better inter-client connectivity that may likely exist. As a result, i) data access throughput is limited by server's capacity and by the network bandwidth to the server, ii) clients may observe poor latency in WAN environments since data is routed triangularly through the server, and iii) data availability is limited by the connectivity and availability of the server.

In contrast, the serverless approach exemplified by systems such as xFS and Ivy helps alleviate the server bottleneck by distributing data ownership and control, but requires complex techniques for locating data, managing consistency, and handling failures. Absence of a central server accountable for the data makes such systems difficult to administer and deploy in a commercial setting. Furthermore, most of the existing serverless filesystem protocols are optimized for high-bandwidth LAN connectivity and are not suited for use over wide-area networks. We believe these to be the prime factors that hinder the adoption of serverless solutions in enterprise settings and today, NFS, CIFS, and AFS remain the three most widely deployed protocols for remote file access within an enterprise.

Caching of data closer to the client is a widely used technique to mask the latency and poor connectivity of WAN links. In the context of client-server file access, both NFS and AFS rely extensively on client-side caching to improve performance. With caching, however, comes the fundamental problem of cache consistency. Strong consistency, where every read operation reflects the most recent write, would incur a high performance penalty and is rarely implemented in practice

in large multi-client distributed filesystems. Both NFS and AFS provide a weaker form of cache consistency, called *close-to-open consistency*, where a client is required to flush all changes when closing a file and a subsequent client is guaranteed, at the time of an *open* request, to see the updates from the last *close* request observed by the server [11, 16].

Regardless of how consistency is implemented, the server, in a traditional client-server setting such as NFS, is the authoritative owner of all data and metadata and can be viewed as playing two additional roles, namely: i) servicing all data cache misses and ii) managing the cache consistency state and driving the cache revalidation and/or invalidation protocols[1].

In this report, we argue for a simple *decoupling of roles*, i.e., the consistency management role of the server from the miss handling role, within the traditional client-server model. Such separation enables a new point in the design space of distributed filesystems, where clients cooperate on servicing each others' cache misses and retrieve data via direct *client-to-client* transfers that are topologically optimal, thereby reducing server load and improving client response times. At the same time, our approach does not forfeit the simplicity and practicality of the standard client-server model, allowing the server to remain the authoritative owner of the data. In enterprise settings this is an important consideration given the need for interfacing with backup and disaster recovery systems.

We propose C2Cfs- a stackable filesystem layered over NFS that enables client caching proxies to cache file data in a local persistent store and share it with other client proxies, while maintaining the traditional close-to-open consistency semantics. The design of C2Cfs is guided by the following practical requirements: i) support well understood consistency semantics ii) use a standard, open, widely-deployed file access protocol. In order to make it a practical solution, our design and implementation rely only on the standard NFS protocol and do *not* require any server-side changes.

The benefits of cooperative client-side caching have been extensively explored in research literature [5, 6, 20] and are well understood. Unlike much of earlier work in this area, C2Cfs is not primarily focused on increasing the effective cache size and reducing the number of misses by deciding on what files to cache as a group and how to evict the files from the set of caches [5]. C2Cfs is more geared toward reducing client-observed latency and the server load in a widely-distributed setting. Furthermore, earlier proposals for cooperative caching in networked filesystems either introduce a new dedicated protocol, rely on integrating with existing DHT systems [6], or require extensions to the NFS protocol [20]. Experience has shown that open and widely-adopted standards such as NFS are remarkably resistant to change and as a result, these systems face a substantial barrier to adoption in an enterprise setting. Moreover, while most of these designs focus on read-only data, none of them support a well-defined consistency model such as close-to-open consistency for handling updates.

In contrast, C2Cfs requires no changes to the foundational file access protocol and can be readily deployed over an existing storage infrastructure. To the best of our knowledge, C2Cfs is the first system to provide the benefits of collective caching while operating within the confines of a standard and widely-used distributed filesystem protocol. While our current implementation is based on NFSv4, it is inherently designed to work with other file access protocols such as NFSv3, AFS, and CIFS.

In this report, we highlight our three main contributions. First, we propose a design for a collective caching scheme that can be layered on any distributed filesystem supporting close-to-open consistency. Second, we present an NFSv4-based instantiation of this architecture. Finally, we demonstrate the practicality and quantitative benefits of our scheme by presenting a Linux-based C2Cfs prototype implementation and its evaluation under a range of workloads.

The rest of the paper is organized as follows. In the next section we provide a brief overview of the traditional consistency semantics in distributed filesystems. Section 3 provides an overview of the C2Cfs architecture. The implementation is described in Section 4, followed by Section 5 that evaluates C2Cfs under a range of workloads. Finally, Section 6 provides an overview of the related work and Section 7 concludes.

2 Cache Consistency Semantics

A number of design and implementation choices we made in C2Cfs were driven by our desire to support the *close-to-open* consistency semantics provided by the traditional client-server file access protocols.

Under close-to-open consistency, after opening a file a client is guaranteed to observe all previous updates committed by a *close* request. More formally, we define that in a given global execution history of file requests, a $read(f)$ request issued by client c at time t_r returns **valid** data if it reflects the effects of every prior

[1]More concretely, the consistency state is managed by the server either directly using callbacks as in AFS or NFSv4 with delegations or by storing "change attribute" values and responding to client GETATTR requests as in NFS [18].

	c_1	c_2	c_3
t=1	OPEN (f)		
t=2		OPEN (f)	
t=3	READ (f, 0, 4096)		
t=4		READ (f, 0, 4096)	
t=5	WRITE (f, 0, "X")		
t=6	CLOSE (f)		
t=7		WRITE (f, 1, "Y")	
t=8		CLOSE (f)	
t=9			OPEN (f)
t=10			READ (f, 0, 2)

Figure 1: Concurrent access to a shared file.

$write(f)$ request (from any client) committed at time $t_c < t_o$, where t_o is the time of c's most recent $open(f)$ request. A $write(f)$ request from client c' is considered **committed** when c' issues a $close(f)$ request. We say that a distributed filesystem protocol provides **close-to-open cache consistency** if a $read(f)$ request always returns valid data for any shared file f and any request execution history.

These semantics are fairly easy to achieve in the conventional client-server model. In NFS, clients are required to perform a writeback and send all modified data to the server before closing a file. When opening a file, the client issues a $GETATTR$ request to retrieve the latest file attributes from the server, using which it validates the cached version. In AFS, instead of the client checking with the server on a file *open*, the server establishes a callback to notify the client if the file gets updated on another client. Analogously, all changes must be sent to the AFS server when the client closes the file.

The key question that arises in C2Cfs is how to provide these semantics in a decentralized *client-to-client* model that does not force clients to send all *read* requests to the main server. To illustrate that lack of centralized serialization may lead to complications, consider the example in Figure 1, in which three clients (c_1, c_2, and c_3) are accessing a shared file f of length 4KB that initially resides on the central server. Observe that the *open* request issued by c_3 is preceded by *close* requests from c_1 and c_2, which means that c_3 should observe the effects of both preceding writes and the *read* operation at $t = 10$ should return "XY". However, at the time of c_3's *open*, c_1's local copy of f may not necessarily reflect the update performed by c_2 and vice versa. Hence, a direct client-to-client transfer from c_1 or c_2 would in this scenario fail to return correct data and some additional coordination must take place in order to ensure that both updates are reflected in the version that we make available to c_3.

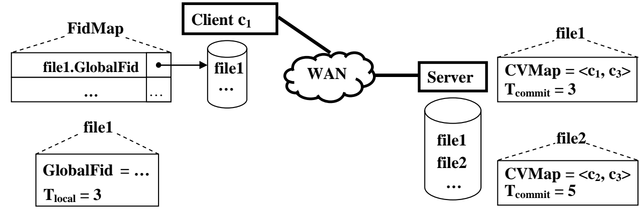


Figure 2: Overview of the C2Cfs architecture.

3 *C2Cfs* Architecture

C2Cfs extends the basic model of client-server filesystem in a way that enables a collection of clients to access the shared filesystem in a consistent and efficient manner without requiring them to always fetch the data from the server. In C2Cfs, each client node plays the role of a caching proxy that stores a subset of the shared files persistently on its local disk. Furthermore, clients revalidate their cache content using direct *client-to-client* transfers, while retaining some minimal coordination with the server to guarantee close-to-open consistency. Figure 2 illustrates the high-level organization of C2Cfs and lists the key pieces of state.

In C2Cfs, each file f in the shared filesystem is assigned a globally-unique persistent identifier, denoted $f.GlobalFid$[2]. Locally at each client node, C2Cfs maintains a *file identifier map* (denoted $FidMap$) that maps the global persistent identifier onto a local identifier ($f.LocalFid$) referring to the clients's own copy of the file in its local cache. For each locally-cached file, we also maintain the reverse mapping $\langle f.LocalFid \rightarrow f.GlobalFid \rangle$ as part of per-file metadata.

C2Cfs uses a straightforward timestamp-based scheme to provide cache consistency guarantees, while supporting direct cache-to-cache transfers. As the example in Section 2 demonstrates, a fully-decentralized model of update propagation may require clients, in a degenerate case, to track validity and fetch updates from remote clients at the granularity of individual bytes - clearly an infeasible requirement. This resulted in our first simplifying design choice: *we maintain the cache validity status at whole-file granularity.*

To guarantee close-to-open consistency in C2Cfs, the central server maintains a per-file *commit timestamp* (T_{commit}) - a monotonic counter that is incremented by 1 every time a client *commits* its updates to a file by sending them to the server and closing the file. The value of T_{commit} at the server establishes the most recent version of the file observed by the server and a client's copy of f is considered *valid* if its local timestamp value ($f.T_{local}$) matches the server's $f.T_{commit}$.

[2] In our current NFS-based implementation, the NFS filehandle from the central server plays the role of the global identifier.

Finally, for each file f in the filesystem, the server maintains the *cache validity map* ($f.CVMap$) which maps $f.GlobalFid$ onto a list of client locations storing valid copies. Observe that instead of relying on an external DHT-based service for locating up-to-date replicas, we maintain the list of locations at the central site and manage it using standard NFS.

At a high level, cache revalidation in C2Cfs proceeds as follows: When an application issues a file *open* request, the client first resolves the supplied filename into a $GlobalFid$ [3]. (Note that the $filename - to - GlobalFid$ mapping may change as result of metadata operations, e.g., file creation, deletion, and renaming). Next, given a file's global identifier, the client reads its current T_{commit} and $CVMap$ from the server. The client then consults its $FidMap$ and if a copy exists in the local cache, obtains the corresponding $LocalFid$ and the local version timestamp T_{local}. If T_{commit} matches T_{local}, the cached replica of the file is valid. Otherwise, the client immediately discards the cached version and requests a fresh copy from the client(s) that hold a valid copy according to $CVMap$. As a fallback, if no remote client has the most recent data in its cache, the client retrieves the file directly from the server.

After obtaining the latest copy of the file and writing it to the local cache, the client sets $f.T_{local} := f.T_{commit}$, updates the validity map to reflect the fact that it now holds an up-to-date copy, and writes the modified $CVMap$ to the server. Once a client's local copy of the file has been revalidated, all *reads* are fulfilled from the local cache, while *updates* are written through to the cache and the central server.

When an application at client c closes a file f, the client must flush all outstanding updates to the file, increment $f.T_{commit}$ at the server, and determine whether its cached version is still valid, which would be the case in the absence of concurrent updates to the same file from other clients. That is, after closing the file, c's cached version remains valid if no other client performed a *close* operation between c's own *close* and its preceding *open*. To make this determination, c stores the value of server's $f.T_{commit}$ at the time of opening the file in a local variable ($f.T_{open}$). When closing the file, c re-reads the server's timestamp and compares it to the original value. If $f.T_{commit} = f.T_{open}$ then there have been no concurrent updates and c's cached version of f remains valid. In this case, the client adds itself to the list of valid cache locations by setting $f.CVMap := \{c\}$. Otherwise, a remote client must have committed some conflicting updates to f, in which case none of the clients are guaranteed to have observed all committed updates and we set $f.CVMap := \emptyset$. Hence, at the end of this step, the set of valid client caches in $f.CVMap$ is either empty or contains precisely one entry. In both cases, the client completes the operation by incrementing $f.T_{commit}$ and writing the new timestamp and $f.CVMap$ to the server.

[3] In the current implementation, this is accomplished via an NFS LOOKUP request to the server, which returns a persistent opaque NFS filehandle.

This mechanism enables us to provide the close-to-open consistency guarantees as defined in Section 2. While a formal proof is beyond the scope of this paper, observe that since all updates are written through to the central server, revalidating a stale cache entry from the server is always safe. Furthermore, we can demonstrate using an inductive argument that client-to-client revalidation is also safe: c may revalidate a file f from another client c^* only if $c^* \in f.CVMap$, whereas c^* adds itself to $f.CVMap$ only after revalidating its own copy to reflect all preceding committed updates.

In our initial example, client c_1 would set $f.CVMap := \{c_1\}$ when closing the file. When c_2 issues a *close* request, however, it would detect a concurrent update and clears the list of valid caches by setting $f.CVMap := \emptyset$. At a later stage, c_3 finds the list of caches to be empty and proceeds to accessing the file from the primary server.

In our current design, all metadata operations (e.g., file creation and deletion) are written through to central server and the corresponding changes are also applied to the local cache.

4 Prototype Implementation

In this section we detail our implementation of C2Cfs on Linux 2.6.21 using NFSv4 as the file access and cache coordination protocol. Figure 3 illustrates the high-level organization of the C2Cfs implementation and below, we focus our discussion on the three core components, namely: i) the client-side kernel module that implements a stackable filesystem layered over NFSv4 and ext3, ii) the management of server-side state, and iii) client-to-client data transfer.

4.1 Client-side filesystem layer

We have implemented a C2Cfs prototype on Linux as a stand-alone loadable kernel module. Our prototype exposes a new file system type into the Linux VFS layer, providing the collective caching facility on top of the unmodified NFS client and a local persistent cache back-end. (In the discussion that follows, we refer to these underlying filesystem layers as *client-NFS* and *localFS*, respectively). The stackable archi-

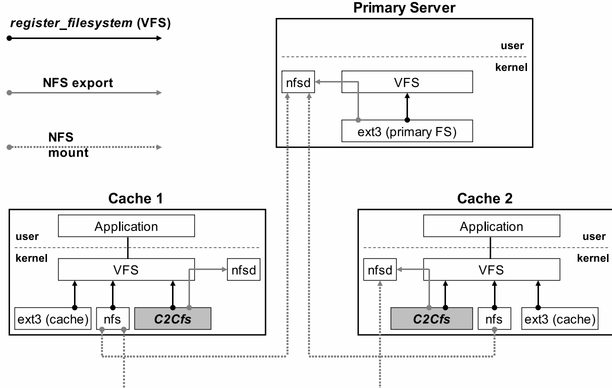


Figure 3: C2Cfs implementation.

tecture of C2Cfs ensures that any POSIX filesystem could be used as the cache backend. Some aspects of the local client cache implementation are similar to xCacheFS [19].

Conceptually, a C2Cfs filesystem on Linux is a collection of in-memory kernel data structures (*super_block*, *inodes*, *dentries*, and *files*) without a persistent embodiment on the local disk. These data structures are created dynamically in response to application requests sent through the VFS layer.

To enable direct client-to-client data transfers, the root of a C2Cfs file system is exported via NFS so that the contents of the local cache are available to remote C2Cfs-capable clients. It is important to note that we export the root of the C2Cfs file system rather than the actual on-disk cache (localFS). As we explain below, this enables us to implement a special *lookup-by-filehandle-as-name* procedure that permits the local client to retrieve data from remote peers efficiently via its $GlobalFid$ (the primary server's NFS filehandle).

During initialization, our module registers the C2Cfs filesystem type with the VFS layer. To mount a C2Cfs filesystem, the *mount*() system call is invoked with a number of arguments, which include: i) the path to the root of the *localFS*, ii) the path to the NFS mount point for the server.

The C2Cfs layer ensures that the cached data and metadata stored within localFS mimics the namespace, directory structure, and inode attributes of the remote server by controlling the file lookup and cache population code path.

The $FidMap$ maintains mappings from a file's primary NFS filehandle to an identifier in localFS. A file in the local cache is typically identified by the device_id, the inode number, and the inode generation number. To ensure compatibility across various filesystem types, we obtain the $LocalFid$ using the kernel's *exportfs* facility, which constructs an opaque identifier from a given localFS dentry by invoking a filesystem-specific callback. Currently, our implementation maintains the $FidMap$ in a simple in-memory hash table, backed by a hidden file in localFS.

C2Cfs also maintains a small amount of per-file metadata, which includes the local commit timestamp T_{local} and its $GlobalFid$. Our current implementation maintains this state in an extended file attribute, but alternative options (e.g., storing these structures in a flat file indexed by the cacheFS inode number) can be considered for back-end file systems that do not support extended attributes.

When an application issues a file *open* request to C2Cfs, we obtain the object's dentry in localFS, revalidate the cached copy via the protocol described in Section 3, and call the *dentry_open* VFS function to obtain an open file object in localFS. If write-mode access is requested, we also obtain the corresponding open file structure in clientNFS, which is necessary for write-backs to the server. The localFS and clientNFS open file structures are linked to the main C2Cfs open file object via its *private_data* field.

The processing of the actual *read* and *write* requests on an open file is, in fact, quite straightforward. A read request is handled by reading the corresponding region from the local cache and all write requests are propagated to both clientNFS and the local cache.

4.2 Server-side State

For each file in the shared filesystem, the server maintains the current commit timestamp T_{commit} (a 32-bit integer) and the cache validity map. Currently, the $CVMap$ is represented by a bitmap which contains 1 bit per client indicating whether the respective client caches the last committed version of the file.

Since our design requirements dictate an unmodified server, clients are responsible for creating, updating, and deleting these objects. Hence, the client that first creates a file will also create and initialize the associated T_{commit} and $CVMap$ data structures at the server.

The per-file metadata at the server could be stored as an extended attribute along with the file, but we found that the Linux 2.6.21 implementation of the NFSv4 server only supports ACLs and not general extended attributes. Hence, our current implementation keeps the metadata in a separate file much like a named attribute. The file is stored in a hidden well-known directory with the name of the file being the string representation of its $GlobalFid$.

By storing the server state in a separate file we pay the additional overhead of open, close, read, and write operations every time the server state is modified.

These overheads are an artifact of the NFS server implementation on Linux and not a design issue. Apart from that, clients must coordinate their access to these shared objects and our current implementation relies on NFSv4 advisory locking, which incurs some additional synchronization overhead.

4.3 Client-to-Client Data Movement

When revalidating a cached copy of a file f, a client retrieves $f.T_{commit}$ and $f.CVMap$ from the server. It consults the validity map to determine the list of peer locations from which a valid copy of f can be fetched.

Client-client mount: The $CVMap$ may only represent a numerical client identifier *client_id*, in which case we need an external means of resolving *client_id* into $\langle hostname : export_path \rangle$ and currently, we maintain these mappings in a static configuration file. Given a peer client's hostname and export path, the client mounts its C2Cfs export over NFS. The mounting is done on demand, as we do not want all clients to be mounting from all their peers.

Observe that the above steps of getting a list of client locations and performing an internal mount is very similar to the NFSv4 client's behavior when sub-mounting from a replica location on a referral. In that case the client receives $\langle hostname : export_path \rangle$ as part of the $fs_locations$ attribute. However, we could not leverage this feature, since in current NFSv4 implementations referrals are only possible at filesystem boundaries.

Lookup using filehandle as name: To fetch an up-to-date version of a file f from a remote peer, the client must perform a lookup over NFS and obtain the corresponding filehandle from the peer. Instead of issuing a lookup requests on the filename of f, the client requests a lookup on the string representation of $f.GlobalFid$. This unusual lookup technique serves two important purposes: i) avoiding the overhead of a multi-stage pathname lookups, ii) avoiding ambiguities due to a rename of any component in the path leading to the file or a rename of the file itself (the filehandle is guaranteed to be persistent and unique).

The lookup from a remote peer proceeds as follows: i) The client issues a lookup request for $f.GlobalFid$ with an intent to open, which results in an NFS OPEN request with the name $f.GlobalFid$ to the peer client. ii) When the peer client receives an OPEN request, the NFS server daemon issues a VFS lookup with an intent to open to the C2Cfs layer. iii) The C2Cfs layer detects that this is a lookup by *filehandle as a name* and consults the $FidMap$ to obtain $f.LocalFid$. iv) Using the kernel's *exportfs* interface, it converts $f.LocalFid$ to a dentry in its localFS and the dentry is used to obtain an open file pointer to the local copy of f.

The current implementation splits up the file into n equal-sized segments, where n is the number of available locations and prefetches the entire file into the local cache in a piece-wise manner. Prefetching can be made more efficient by utilizing the knowledge of various network-level metrics such as the end-to-end latency and bandwidth, but we have not yet explored these optimizations. They are orthogonal to the core C2Cfs design and can be retrofitted if needed.

4.4 Discussion

Leveraging delegations: NFSv4 has read and write delegations support that can temporarily make an NFS client the delegated owner of the file. While the delegation is outstanding, all $OPEN$, $READ$, $WRITE$, $CLOSE$, $LOCK/UNLOCK$, and attribute requests for the file can be handled locally by the client. Our implementation could use delegations in multiple ways. Clients can obtain a write delegation on a shared file and the associated metadata file and this would allow a client to update the two files together across multiple open-close cycles without contacting the server. However, we found that although the Linux NFSv4 client supported write delegations, the Linux server implementation never awarded them. Hence, our current implementation does not take advantage of this feature.

Security and access checking: In a distributed setting, security becomes an important concern. The client cache can provide other peer clients access to data without the server knowing or verifying their authority to do so. In our implementation the client always checks with the server to get the cache map and the server can perform access control checks at that time. In other cases the client resorts to sending an *access* request to the server to verify the access permissions for the peer client on every *open*.

5 Experimental Evaluation

In this section, we evaluate our Linux-based C2Cfs prototype under a variety of micro-benchmarks and applications workloads. Below, we present only key results that quantitatively demonstrate the benefits of decentralized data propagation and confirm the feasibility of our design.

First, we quantify the extent to which the C2Cfs architecture improves upon the traditional client-server model with respect to the following key metrics: i) The load on the primary server, as given by the number of NFS operations and its network bandwidth consumption. ii) Application-observed response time that is

affected both by the server utilization and the WAN latency. We conclude this section by examining the worst-case overhead incurred by our design.

5.1 Methodology

The experiments were conducted in a controlled test environment consisting of 5 server-grade x86 machines running Linux 2.6.21, 3.2GHz, 2GB memory, 72 GB storage, interconnected via a 100Mbps switched Ethernet. Four of these machines were assigned clients running C2Cfs and the fifth machine played the role of the central server holding the master replica of the filesystem and exporting it to clients via NFSv4. Each of the four clients was configured to use a local cache hosted on an ext3 partition large enough to store a complete replica of the shared filesystem.

In the following experiments, we compare C2Cfs, in which we point the benchmark application to the C2Cfs filesystem root, and the baseline scenario, in which client applications are configured to access the shared filesystem directly from the server via the NFS mount point. Unless stated otherwise, all NFS *read/write* requests are sent on the wire in 32KB chunks and the application I/O request size is 256KB.

5.2 Server Load Reduction

One of the benefits of sharing data across caches is that the server will receive fewer requests and can scale to support more clients. We measure the server load reduction using two metrics: i) the number of NFS operations received at the server, and ii) the number of bytes transferred to and from the server.

Synthetic reads : In the first set of experiments, we evaluate sequential read access to a file whose size ranges between 100KB and 100MB. Each client opens and reads the entire file, and then closes it. The reads are staggered among the clients with a 30-second delay.

In the baseline case, all the clients read the file from the server and thus, the number of NFS operations at the server grows linearly with the number of clients, as shown in Figure 4(a). With our scheme, only the first clients reads the data from the server. The second client fetches the file from the first client after getting the cache map from the server. Similarly, the third client reads from both the first and the second client and so on. As expected, the number of requests received by the NFS server with C2Cfs remain nearly constant as the number of clients increases, since only the first client fetches the file from the central server.

NFS Command	Client 1		Client 2	
	Base	**C2C**	**Base**	**C2C**
CLOSE	1	3	1	2
GETATTR	5	15	5	13
GETFH	1	4	1	3
LOCK		2		2
LOCKU		2		2
LOOKUP		1		1
OPEN	1	3	1	2
PUTFH	325	338	325	17
READ	320	322	320	2
WRITE		1		1
Total	660	702	660	54

Table 1: NFS operations at the primary server in the synthetic read experiment with 10MB file size.

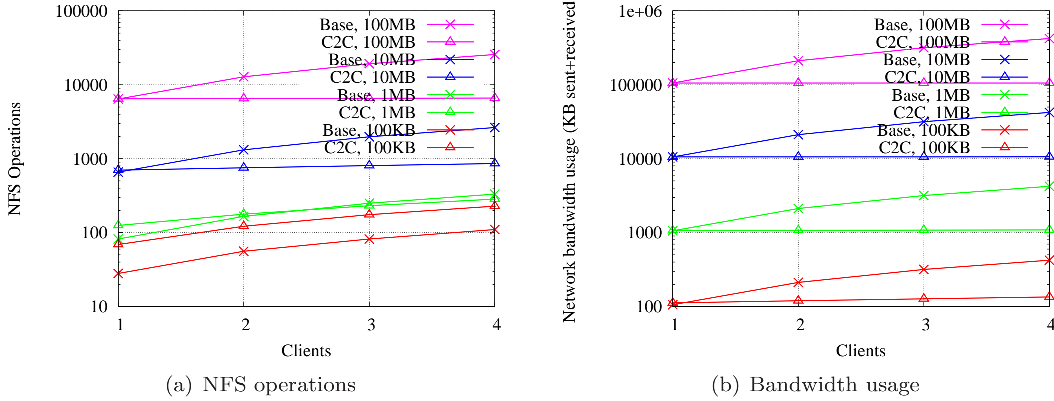
Note, however, that this is true only for files larger than 1MB because we pay an additional overhead to OPEN, LOCK, READ/WRITE, UNLOCK, and CLOSE the file holding the C2Cfs metadata. As we discuss in Section 4, this is an artifact of the current Linux NFS server implementation that does not completely support extended attributes and write delegations. For larger files, the overhead is amortized over the larger number of reads.

Figure 4(b) measures the amount of data transferred to/from the server for the synthetic read experiment. The number of bytes transferred grows linearly with the number of clients in the baseline case, but stays nearly flat when collective caching is enabled. With 4 clients, C2Cfs reduces the network bandwidth usage at the server by 75% for a 100MB file.

In Table 1, we further analyze the behavior by considering the breakdown of NFS operations seen by the central server in the synthetic read experiment with a 10MB file. For the single client case, C2Cfs incurs a 6% overhead due to accessing the metadata file at the server (2 additional OPEN and CLOSE requests, 2 READs, and 2 LOCK/UNLOCK requests). With more than one client, this overhead is alleviated by the reduced number of READs done at the server.

Figure 5 shows a time-series plot illustrating the distribution of reads across the clients and the central server. In the baseline case, the cumulative NFS requests at the server increase linearly with the number of clients. With collective caching, the first client starts at time 0 and all its requests are sent to the server. After a 30-second delay, client 2 starts and reads the data from client 1 and the server sees only a marginal increase in the number of requests. After another 30-second delay, client 3 accesses the file and fetches the data from clients 1 and client 2, and so on.

Synthetic reads and writes : In our next experiment, we consider a sequential read/write access pat-



(a) NFS operations (b) Bandwidth usage

Figure 4: Overhead at the primary NFS server in the synthetic read experiment. The X-axis shows the number of clients doing a staggered read. The y-axis (in log scale) is the number of NFS operations and the total bytes sent and received by the server.

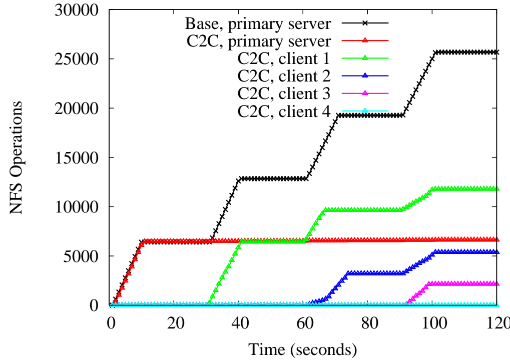


Figure 5: Distribution of reads across clients in the synthetic read experiment. The x-axis shows time in seconds and the clients are staggered by 30 seconds each. The y-axis shows the cumulative number of NFS operations at the server and each of the clients.

tern on files of varying sizes that range between 100KB and 100MB. Each client opens and sequentially reads the entire file, then writes the entire file and closes it. As before, the operation are staggered and each client begins 30 seconds after the previous client has completed its access. In the baseline case, all the clients read the file from the server and thus, the number of NFS operations at the server increases linearly with the number of clients, as shown in Figure 6(a). With collective caching, all the writes are sent to the server, but the reads are served by the other clients. Figure 6(b) shows the network bandwidth consumption at the server and confirms that C2Cfs yields substantial savings. Figure 7 shows the time-series plot illustrating the distribution of reads across the clients.

Finally, Table 2 further analyzes the behavior by considering the breakdown of NFS operations seen by the primary server in the synthetic read-write experiment with a 10MB file. As before, our scheme incurs

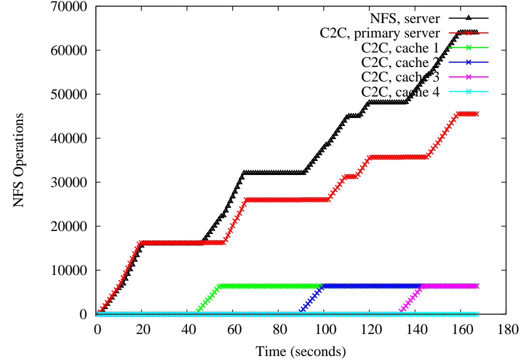
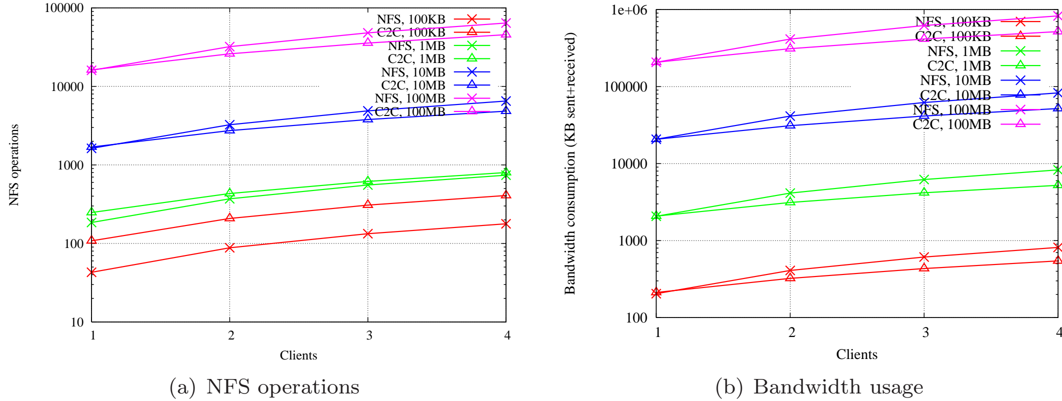


Figure 7: Distribution of NFS operations across clients in the synthetic read/write experiment. The x-axis shows time in seconds and the clients are staggered by 30 seconds each. The y-axis shows the number of NFS operations at the server and each of the clients.

NFS command	Client 1		Client 2	
	Base	**C2C**	**Base**	**C2C**
CLOSE	1	4	1	4
GETATTR	327	342	327	342
GETFH	1	4	1	4
LOCK		3		3
LOCKU		3		3
OPEN	1	4	1	4
PUTFH	647	668	647	348
READ	320	323	320	3
WRITE	320	322	320	322
Total	1625	1687	1625	725

Table 2: NFS operations at the primary server in the synthetic read-write experiment for a 10MB file.



(a) NFS operations (b) Bandwidth usage

Figure 6: Overhead at the primary NFS server in the synthetic read/write experiment. The X-axis shows the number of clients doing a staggered read+write. The y-axis (in log scale) is the total number of NFS operations and the bytes transferred by the server.

some additional overhead (3%) for the first client, but substantially reduces the server load during subsequent access by other clients by distributing the reads.

Workload	Seq. read		Seq. read+write	
	Base	**C2C**	**Base**	**C2C**
Client 1	23.7	24.3	197.4	198.8
Client 2	23.5	10.3	198.0	194.7

Table 3: Application response time (seconds) for reading a 100-MB file from a server across a WAN.

Workload	Seq. read	
	Base	**C2C**
Client 1	185	182
Client 2	204	205
Client 3	186	98
Client 4	204	91

Table 4: Overloaded server experiment: Response time (seconds) for a sequential read of a 1GB file.

5.3 Application Response Time

Masking WAN latency: In a typical enterprise with a central-office branch-office setting, the server can be across a WAN while multiple peer clients in a branch office are in close proximity to each other and have a better connectivity among themselves than with the remote server. In this experiment, we measure application response time in a scenario where the server is separated from clients by a wide-area link. We demonstrate that our architecture can help mask WAN latencies and improve application performance. To simulate the link latency, we use the standard Linux *tc* packet filter and configure it to impose a mean latency of 100 ms (which is what we had observed as the round-trip-time between servers in California and New York). We measure the application response time for two synthetic workloads (sequential read and sequential read/write) on a single file of size 100MB from 4 clients and Table 3 reports the response time for both schemes. For sequential read access, C2Cfs incurs a 2.5% overhead for the first client, but reduces the response time by 50% for the second client and for all clients that follow. In the sequential read/write experiment, the response time is dominated by the writes to the server and thus the two schemes exhibit similar performance.

Overloaded server: The application response time is also affected by the load on the server and Table 4 demonstrates the performance benefits of decentralized cache revalidation of C2Cfs in case of an overloaded primary server. In this two-stage experiment, two distinct files, each of size 1GB, are read sequentially by four clients. In the first stage, clients 1 and 2 read *fileA* and *fileB*, respectively. Since both reads are handled by the central server, each client receives approximately 1/2 of the available server-side bandwidth. In both schemes, all the requests go to the server and thus, we observe similar performance. In the second stage, clients 3 and 4 perform concurrent reads on *fileA* and *fileB*, respectively. In the baseline scenario, these reads are handled by the server as well and hence observe similar performance as the first two clients. By contrast, with C2Cfs clients 3 and 4 can fetch the file directly from clients 1 and 2 and avoid overloading the server. With our architecture, these clients observe a 47% reduction in the response time compared to the baseline case.

5.4 Evaluation with Realistic Application Workloads

In the above experiments, we used synthetic workloads to evaluate C2Cfs in a controlled setting. In

NFS command	Base Srv	C2C Srv	C2C C1	C2C C2
CLOSE	1570	3925	401	384
GETATTR	8740	19257	1606	1155
GETFH	3140	5498	402	385
LOCK		3140		
LOCKU		3140		
LOOKUP	1570	1573	1	1
OPEN	1570	3925	401	384
PUTFH	95395	68744	23209	22732
READ	88202	47096	22004	21959
READDIR	32	33		
WRITE		1570		
Total	203391	164209	48831	47773

Table 5: Breakdown of NFS operations in the RPM tar experiment.

NFS command	Base Srv	C2C Srv	C2C C1	C2C C2
CLOSE	40944	102365	6668	13799
GETATTR	218723	468271	24105	41401
GETFH	85515	146919	6669	13800
LOCK		81888		
LOCKU		81888		
LOOKUP	44571	44554	1	1
OPEN	40944	102365	6668	13799
PUTFH	233021	641166	26087	42519
READ	49236	105463	8649	14916
READDIR	3075	3074		
WRITE		40944		
Total	800860	2026586	92188	167838

Table 6: Breakdown of NFS operations in the Linux kernel tar experiment.

the next set of experiments, we study the performance with real application and Filebench [14] - a soon-to-be-standard benchmark for filesystems that can simulate a wide range of realistic workloads.

Tar archive creation (large files): In this experiment, clients create a tarball of a directory tree exported by the server and mounted at each client over /mnt. The directory contains a collection of relatively large binary RPM files (part of the Fedora Core Linux distribution). The top-level directory /RPMS consists of two sub-directories /dir1 and /dir2. Client 1 issues the command "tar cf ./dir1.tar /mnt/RPMS/dir1" to create a new archive with the contents of /dir1 and client 2 issues "tar cf ./dir2.tar /mnt/RPMS/dir2". In both schemes, (baseline and C2Cfs) all the requests for the files in /mnt/RPMS go to the central server. Next, client 3 creates a tarball of the top-level directory ("tar cf ./RPMS.tar /mnt/RPMS"). In the baseline case, all requests are sent to the server, while collective caching enables client 3 to fetch the contents of /dir1 and /dir2 from clients 1 and 2, respectively. Table 5 reports the total number of NFS requests observed by the server and each client after all clients have completed the tar operation. With collective caching, client 3 fetches the contents of /mnt/RPMS from clients 1 and 2, thus reducing the number of READ requests observed by the server by 46%.

Tar archive creation (small files): In the next experiment, we measure the performance of tar archive creation for a directory tree consisting mainly of small files. For this, we use the Linux kernel source tree whose top-level directory /linux consists of 17 subdirectories, which we rename to /subdir1-17. Client 1 issues "tar cf ./dir1-9.tar /mnt/linux/dir1-9" for the first 9 sub-directories and client 2 issues 'tar cf ./dir10-17.tar /mnt/linux/dir10-17" for the remaining 8 subdirectories. In both schemes, all requests from clients 1 and 2 go to the central server. Next, client 3 creates an archive of the top-level directory by issuing "tar cf ./linux.tar /mnt/linux". In the baseline case, all requests are sent to the server, while with collective caching, client 3 fetches the subdirectories directly from its peers. Table 6 reports total number of NFS requests processed by the server and each of the clients. Observe that with C2Cfs, the total number of READs seen by the central server is significantly higher than in the baseline case. Although this appears non-intuitive, recall that C2Cfs requires maintaining a metadata for each file at the server. In this scenario, the overhead of accessing and updating the metadata overshadows the cost of simply reading the file from the server, which suggests that our approach may not be advantageous for very small files.

Filebench (web server workload): In this experiment, we use the Filebench benchmark [14] with a Web server workload. We first generate a directory tree containing 5000 files with a mean file size of 100KB. The workload consists of 5000 file accesses (open, sequential read, close) with a Zipfian popularity distribution. Additionally, a 16-KB block is appended to a simulated log for every 10 reads. We run the workload on each of the four clients and measure the total network bandwidth consumption at the server (bytes sent and received). Each client starts out with an empty cache and begins when the previous client has completed. Each client reads 5000 randomly-chosen files (different clients may choose different files). In the baseline case, all requests go to the central server, whereas with our design, client 2 redirects a fraction of its requests to client 1; client 3 redirects to 1 and 2, and so forth. Table 7 shows the bandwidth consumption at the server for each of the clients. For the first client, the baseline scheme produces 169MB of traffic at the server, compared to 189MB produced with C2Cfs. For all subsequent clients, however, our scheme reduces the network bandwidth usage at the server by approximately 90%.

Client	Network Traffic at the server (bytes)	
	C2C	Base
Client 1	189,707,017	169,217,953
Client 2	19,905,369	169,334,493
Client 3	19,921,029	169,189,073
Client 4	19,863,437	169,337,505

Table 7: Network bandwidth usage at the central server in Filebench web server experiment.

File size	Baseline	C2C	overhead (%)
1 MB	0.094 sec.	0.117 sec.	**24.5%**
10 MB	0.897 sec.	0.923 sec.	**2.9%**
100 MB	8.937 sec.	8.978 sec.	**0.5%**

Table 8: Latency overhead for a single-client sequential read for varying file sizes.

5.5 Overhead of C2Cfs

In this section, we evaluate the worst-case overhead incurred by C2Cfs. Clearly, for a single-client access with no sharing across clients, C2Cfs provides no benefit and imposes some additional overhead, which includes: i) The space overhead of storing the per-file metadata $\langle T_{commit}, CVMap \rangle$ at the server. Our current implementation adds 8 bytes of per-file state - a negligible overhead for all but very small files. ii) The server load overhead due to the additional operations done at the server for cache revalidation. Here, we also pay a non-negligible penalty for small files, as the results in Section 5.4 suggest. iii) The latency overhead incurred by the revalidation protocol due to the additional time taken to access the C2Cfs metadata at the server. We quantify this overhead in the next experiment.

Latency overhead: To quantify the latency overhead, we measure the response time seen by a single client that sequentially reads a file residing at the central server. The server is across a 100Mbps LAN connection with no simulated delay. Table 8 reports the client-observed response time. As expected the overhead falls from 24% for the 1MB file to 0.5% for a 100MB file. While clearly non-trivial, in scenarios with large files and multiple clients that were examined above, this overhead is masked by the benefits of a collective consistent cache.

6 Related Work

The C2Cfs architecture presented in this report draws on a large body of prior work in filesystems and the broad topic of replica consistency maintenance in distributed storage systems. Below, we review the most relevant pieces of related work.

Client-server distributed filesystems: NFS [8, 18] and AFS [11, 12] are among the most widely-used distributed networked filesystems. Coda [17] extends the core architecture of AFS to support server replication and disconnected mode of operation, allowing for improved data availability in the event of a network partition. As with most "pure" client-server architectures, the flow of data and cache invalidation requests in these systems is constrained to a star topology with very little direct inter-client coordination.

While in this paper, our implementation was presented in the context of NFSv4, the C2Cfs architecture can leverage any distributed filesystem protocol that supports: i) close-to-open consistency, ii) persistent unique file identifiers, iii) file locking or atomic creates. Hence, our system could also be easily stacked on top of NFSv3, NFSv4, AFS, etc. to provide the benefits of collective caching.

P2P file sharing systems: In recent years, the phenomenal popularity of peer-to-peer file sharing services such as Gnutella, Kazaa, and BitTorrent [3, 1, 2] has sparked substantial research interest in the applications of P2P techniques to storage systems and, more broadly, the benefits of decentralization over the traditional client-server model. However, systems such as these might be best viewed as playing the role of a search engine rather than that of a a storage architecture; the high-level problem they address is that of *locating immutable content* in a dynamic and unreliable client population. Many P2P file sharing systems operate by organizing their clients into an unstructured or semi-structured overlay and clients cooperate on propagating each others' search queries.

In contrast, C2Cfs is a fileserver architecture that exposes the standard POSIX filesystem interface, supports read/write access, and provides familiar consistency semantics. BitTorrent [2] is one of today 's most widely used (and eagerly studied) file sharing protocol. Its chunk-based data retrieval method that enables clients to fetch data in parallel from multiple remote sources is similar to the idea of parallel access in C2Cfs.

P2P serverless storage systems: The advent of P2P file sharing and the immense research interest in DHT-driven content storage techniques have led some to propose a fully-decentralized, serverless architecture as a viable architectural model for general-purpose distributed filesystems. The Cooperative File System (CFS) [9] is a peer-to-peer read-only filesystem that provides provable efficiency and load-balancing guarantees. Internally, CFS uses a DHT-based block storage layer (DHash) to distribute filesystem blocks over a set of CFS storage servers. The Ivy architecture [15] is a read-write P2P filesystem that provides NFS semantics and strong integrity properties without requiring

users to fully trust the other users of the filesystem. Ivy relies on cryptographic techniques to protect the data and hence incurs a substantial performance cost.

The Oceanstore project [13] proposes a large-scale globally-distributed cooperative storage infrastructure and an economic model in which consumers pay their providers for access to reliable storage. Oceanstore assumes untrusted servers and uses Byzantine agreement techniques to coordinate access between the replicas. Farsite [4] is a serverless distributed storage system that enables a group of cooperating (but possibly unreliable and misbehaving clients) to combine their resources into a highly-available and reliable file storage facility. Farsite achieves data availability and persistence through aggressive randomized replication, while relying on cryptographic tools (encryption and digital signatures) to safeguard against unauthorized access to user data. The design of Farsite is based on an optimistic update propagation scheme that does not ensure close-to-open consistency, which represents a significant departure from C2Cfs.

In summary, serverless storage systems such as CFS, Ivy, Oceanstore, and Farsite were designed with the fundamental goal of providing safe and reliable storage in inherently unreliable peer-to-peer environments that are typically characterized by high rates of membership churn, imperfect network connectivity and presence of unreliable and misbehaving clients. The reliance on cryptography and Byzantine agreement techniques in these systems imposes a considerable performance penalty, which makes us suspect that these systems would be a poor fit for supporting applications that operate on large volumes of data and demand high I/O throughput.

In contrast, application performance represents the primary focus of our work. C2Cfs retains the client-server separation but enables filesystem data to travel directly between the participant trusted cache sites. Unlike Farsite and Oceanstore, the design of C2Cfs does not assume an adversarial environment, which in turn allows us to sidestep many of the issues related to data authenticity and trust.

The Bayou [10] project is a platform of replicated, highly-available, variable-consistency, mobile databases for collaborative applications. Bayou introduced new definitions of consistency for mobile applications which differ from NFS.

The Shark cooperative file cache architecture [6] has similar goals as C2Cfs. However, it does not provide the traditional close-to-open consistency guarantees and does not support an unmodified NFS server and protocol - an important consideration for commercial deployments.

Finally. the PRACTI replication framework [7] illustrates the benefits of separating the flow of cache invalidation traffic from that of data itself and C2Cfs demonstrates how such separation can be realized within the confines of a standard client-server file access protocol.

7 Conclusion

In this report, we argued for decoupling the data and cache consistency traffic in the client-server filesystem model. We presented *C2Cfs* - a collective caching architecture that supports decentralized *client-to-client* data flow, while retaining standard consistency semantics.

References

[1] Kazaa. `http://www.kazaa.com`.

[2] Bittorrent. `http://www.bittorrent.com`.

[3] gnutella. `http://www.gnutella.com`.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[5] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, 1996.

[6] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc. of NSDI*, 2005.

[7] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[8] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 Protocol Specification. RFC 1813.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[10] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[11] J. Howard and et al. An overview of the Andrew filesystem. In *Usenix Winter Techinal Conference*, Februrary 1988.

[12] 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.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press.

[14] R. McDougall, J. Crase, and S. Debnath. FileBench: File System Microbenchmarks. 2006.

[15] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[16] 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.

[17] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[18] S. Shepler and et al. NFS version 4 Protocol. RFC 3530.

[19] G. Sivathanu and E. Zadok. A versatile persistent caching framework for file systems. Technical report.

[20] Y. Xu and B. D. Fleisch. NFS-cc; tuning NFS for concurrent read sharing. *Int. J. High Perform. Comput. Netw.*, 1(4):203–213, 2004.