C2Cfs: A Collective Caching Architecture for Distributed File Access

Andrey Ermolinskiy
Department of Electrical Engineering and Computer Science
U.C. Berkeley
andreye@cs.berkeley.edu

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

***Abstract*—In this paper we present C2Cfs - a decentralized 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. Finally, we evaluate the implementation and demonstrate the performance benefits of decentralized data access enabled by our approach.**

I. 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 [1] and AFS[2], follow the *client-server* paradigm and enable a collection of clients to access a shared filesystem exported by a remote 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 [3] that employ the *anything-anywhere* principle and eliminate the notion of centralized ownership. Broadly, a serverless distributed storage architecture requires maintaining a 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 [4] and OceanStore [5] 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 pure client-server model is attractive due to its simplicity, it has several major shortcomings. 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 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 centralized data ownership 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.

In a widely-distributed setting, both NFS and AFS rely extensively on client-side caching to (partially) mask the latency and poor connectivity of WAN links. 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. 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 a file $open$ request, to see the updates from the last $close$ request.

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

In this paper, 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 paradigm. Such separation enables a new point in the design space of distributed filesystems, where clients cooperate on servicing each others' cache misses and access data via direct *client-to-client* transfers that are topologically optimal, thereby reducing server load and improving response times. At the same time, our approach does not forfeit the simplicity and practicality of

	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)

Fig. 1. **Concurrent access to a shared file.**

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 clients to cache file data in a local persistent store and share it with other client cache proxies. 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 and are well understood. However, earlier proposals for cooperative caching in networked filesystems either introduce a new dedicated protocol, rely on integration with existing DHTs [6], or require extensions to the NFS protocol [7]. 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 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-deployed 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 paper, we highlight our three main contributions. First, we propose a collective caching architecture that can be layered on any distributed filesystem supporting close-to-open consistency. Second, we discuss an NFSv4-based instantiation of this architecture. Finally, we demonstrate the practicality and quantitative benefits of our scheme by presenting the prototype implementation of C2Cfs on Linux.

II. Cache Consistency Semantics

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

Under close-to-open consistency, after opening a file a client is guaranteed to observe all previous updates committed by a $close$ request. These semantics are fairly easy to achieve in the conventional client-server architecture. In NFS, clients are required to perform a writeback and send all modified data to the server before closing the file. When opening a file, a client issues a $getattr$ request to retrieve the latest file attributes from the server, using which it validates the cached version.

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.

III. *C2Cfs* Architecture

C2Cfs extends the basic model of a 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 $f.GlobalFid$[1]. 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 client'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 II 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

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

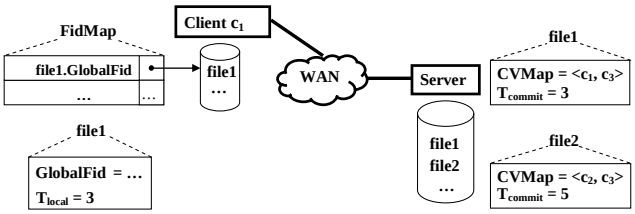

Fig. 2. **Overview of the C2Cfs architecture.**

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 $f.T_{commit}$.

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 a 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 $GlobalFid$ [2]. (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 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 $f.CVMap := \emptyset$. In both cases, the client completes the operation by incrementing $f.T_{commit}$ and writing the new timestamp and $f.CVMap$ to the server.

This mechanism enables us to provide the close-to-open consistency guarantees as defined in Section II. While a formal proof is beyond the scope of this paper, observe that since all updates are written through to the 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 revalidates 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 current design, all metadata operations (e.g., file creation and deletion) are written through to server and the corresponding changes are also applied to the local cache.

IV. 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. 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. A more detailed exposition of the implementation can be found in [8].

A. 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 *clientNFS* and *localFS*, respectively). The stackable architecture of C2Cfs ensures that any POSIX filesystem could be used as the cache backend.

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.

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

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).

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, in localFS. which includes the local commit timestamp T_{local} and its $GlobalFid$. Our current implementation maintains this state in an extended file attribute.

B. 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 with 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 $GlobalFid$.

C. Client-to-client Data Movement

To revalidate 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.

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 lookup, 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.

V. Experimental Evaluation

Due to space constraints, we present only key evaluation results that quantitatively demonstrate the performance benefits of decentralized update propagation and the feasibility of our design. Several additional important measurements are reported in [8].

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 application I/O request size is 256KB.

A. 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 bytes transferred to and from the server.

In the first experiment, we measure 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.

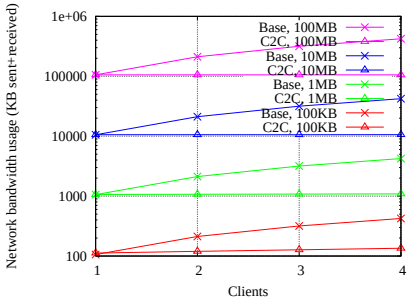

Fig. 3. **Overhead at the primary NFS server in the synthetic read experiment. The X-axis shows the number of clients doing a staggered read. The y-axis (in log scale) shows the number of NFS operations and the total bytes sent and received by the server.**

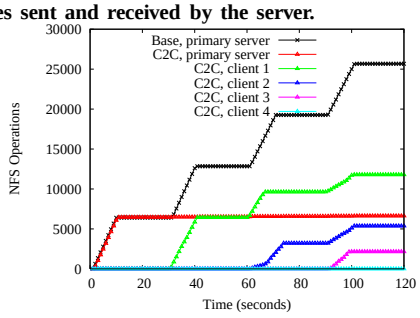

Fig. 4. **Distribution of reads across clients in the synthetic read experiment. The x-axis shows time in seconds (client requests are staggered by 30 seconds). The y-axis shows the cumulative number of NFS operations at the server and each of the clients.**

In the baseline case, every client reads the file from the server and thus, the number of NFS operations at the server grows linearly with the number of clients, as reported in [8]. With our scheme, only the first clients reads the data from the server. The second client fetches it from the first client after getting the cache map from the server. Similarly, the third client reads from both the first and the second clients and so on. Hence, the number of requests observed by the server with C2Cfs remain nearly constant, as only the first client fetches the file directly from the server. 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.

Figure 3 measures the amount of data transferred to/from the server. The amount of traffic 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 server's network bandwidth usage by 75% for a 100MB file. In [8], we analyze the behavior further and consider the breakdown of NFS operations in this experiment.

Figure 4 shows a time-series plot illustrating the distribution of NFS requests across the clients and the central server. In the baseline case, the number of requests at the server increases 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 2, and so on.

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 I
Application response time (seconds) for reading a 100-MB file.

B. 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 I 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 the next experiment 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 response time compared to the baseline case. The complete set of measurements is reported in [8].

C. Evaluation with Realistic Application Workloads

In the above experiments, we used synthetic workloads to evaluate C2Cfs in a controlled setting. We also used Filebench [9] - a soon-to-be-standard filesystem benchmark filesystems - to study the performace under realistic application workloads.

In the first experiment, we configure Filebench with a Web server workload. We first generate a directory tree containing

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 II
Latency overhead for a single-client sequential read.

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 main server, whereas with C2Cfs, client 2 redirects a fraction of its requests to client 1; client 3 redirects to 1 and 2, and so on. For the first client, the baseline scheme produces 169MB of traffic at the server, compared to 189MB produced with C2Cfs. For each subsequent client, however, our design reduces the network bandwidth usage at the server to 19MB. Full results from this and other experiments can be found in [8].

D. Overhead of 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 [8]. iii) The latency overhead incurred by the revalidation protocol due to the additional time taken to access the C2Cfs metadata at the server.

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 II 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.

VI. Related Work

NFS [1] and AFS [2], [10] are among the most widely-used distributed networked filesystems. Coda [11] extends the core architecture of AFS to support server replication and disconnected mode of operation. 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 client-server protocol that supports: i) close-to-open consistency, ii) persistent unique file identifiers, iii) file locking or atomic creates.

The advent of P2P file sharing ([12], [13]) and the immense research interest in DHT-based content location techniques have led some to propose a fully-decentralized, serverless architecture as a viable architectural model for general-purpose filesystems. CFS [14] 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 based on Chord [15] to distribute filesystem blocks over a set of storage servers. The Ivy architecture [4] is a read-write P2P filesystem that provides NFS semantics and strong integrity properties without requiring users to fully trust other users of the filesystem. Ivy relies on cryptographic techniques to protect the data and hence incurs a substantial performance penalty. In contrast, application performance represents the primary focus of our work. The PRACTI replication framework [16] 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 NFS. The Shark cooperative file cache architecture [6] has similar goals as C2Cfs. However, it does not provide close-to-open consistency guarantees and does not support an unmodified NFS server and protocol - an important consideration for commercial deployments.

References

[1] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "NFS version 4 Protocol," RFC 3530. [Online]. Available: http://www.ietf.org/rfc/rfc3530.txt

[2] J. Howard and et al., "An overview of the Andrew filesystem," in *Usenix Winter Techinal Conference*, 1988.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless Network File Systems," in *SOSP 1995*.

[4] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *SOSP 2002*.

[5] 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*, 2000.

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

[7] Y. Xu and B. D. Fleisch, "NFS-cc; tuning NFS for concurrent read sharing," *Int. J. High Perform. Comput. Netw.*, vol. 1, no. 4.

[8] A. Ermolinskiy and R. Tewari, "C2Cfs: A collective caching architecture for distributed file access," *UC Berkeley Technical Report UCB/EECS-2009-40*.

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

[10] 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.*, vol. 6, no. 1.

[11] 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 Trans. Comp.*, vol. 39, no. 4.

[12] "Bittorrent," http://www.bittorrent.com.

[13] "Kazaa," http://www.kazaa.com.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP 2001*.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM 2001*.

[16] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI replication," in *NSDI 2006*.