Cudele: A File System with Programmable Consistency and Durability

Blind Author
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
blindauthor@corporation.com

ABSTRACT

HPC file systems are abandoning POSIX because the synchronization and serialization overheads are too costly – and often uneccessary – for their applications. We provide an API for the client to (1) merge updates into the global namespace and (2) assign policies to subtrees so that other clients cannot interfere with the decoupled namespace. This allows administrators to optimize subtrees within the same namespace for different workloads. We draw conclusions about the performance impact of previously unexplored consistency/durability metadata designs and show that consistency can cause a 104× slow down while merging updates (7× slow down) and maintaining durability (10× slow down) have a more reasonable cost.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Blind Author. 2017. Cudele: A File System with Programmable Consistency and Durability. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Today's client-server based file system metadata services have scalability problems. It takes a lot of resources to service POSIX metadata requests so applications perform better with dedicated metadata servers [10, 13]. Unfortunately, provisioning a metadata server for every client is expensive and complicated.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

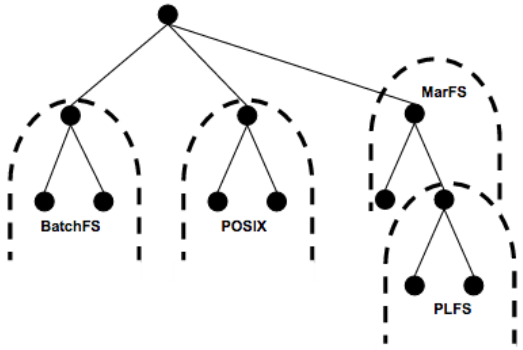


Figure 1: Administrators can assign consistency and durability policies to subtrees to get the benefits of some of the state-of-the-art HPC architectures.

Current hardware evolution and the rise of software-defined storage storage, which uses techniqes like erasure coding, replication, and partitioning, have ushered in a new era of HPC computing; architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to a two layer stack with just a burst buffer and object store [4]. This trend exacerbates the metadata scalability problem

To address this trend, HPC research has advocated client side processing and an emphasis on serverless metadata services. The techniques propose new metadata management techniques that reduce synchronization and serialization overheads by transferring these responsibilities to the client. We call this approach "decoupling the namespace" and the semantics of consistency differ amongst systems. While the performance benefits are obvious for these users, applications that rely on stronger consistency or durability guarantees must be re-written or deployed on a different system.

We propose subtree policies, an interface that lets future programmers control how the system manages different parts of the namespace. For performance one subtree can adopt weaker consistency semantics while another subtree can retain the rigidity of POSIX's strong consistency. Figure 1 shows an example setup where a single global namespace has directories for applications designed for different, state-of-the-art HPC architectures. We present Cudele, a prototype programmable file system that supports different degrees of consistency and

durability by exposing mechanisms used within the file system as a client library. Cudele supports 3 forms of consistency (invisible, eventual, and strong) and 3 degrees of durability (none, local, and global) giving the administrator a wide range of policies and optimizations that can be custom fit to an application. Our contributions:

(1) a prototype that lets administrators program a range of consistency and durability semantics (9 permutations), allowing them to custom fit the storage system to the application.
(2) an API for assigning and programming policies to subtrees in the file system namespace.
(3) an analysis of recently proposed research systems compared against previously unexplored metadata designs.

Our results confirm the assertions of "clean-state" research systems that decouple namespaces; specifically that the technique drastically improve performance (104× speed up) but we go a step further by quantifying the costs of merging updates (7× slow down) and maintaining durability (10× slow down). We also show the effect of having a metadata specific file format in systems that are based on in-memory data structures. Section 2 places Cudele in the context of other related work. Section 3 quantifies the cost of POSIX consistency and system-defined durability and Section 4 presents the Cudele prototype and API. Section 5 describes Cudele's mechanisms and shows how re-using internal subsystems results in an implementation of less than 500 lines of code. The evaluation in Section 6 quantifies the overheads and performance gains of explored and previously unexplored metadata designs.

2 RELATED WORK

The bottlenecks associated with accessing POSIX file system metadata are not limited to HPC workloads and the same challenges that plagued these systems for years are finding their way into the cloud. Workloads that deal with many small files (*e.g.*, log processing and database queries [14]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [7]), are subject to the scalability of the metadata service. The biggest challenge is that whenever a file is touched the client must access the file's metadata and maintaining a file system namespace imposes small, frequent accesses on the underlying storage system [11]. Unfortunately, scaling file system metadata is a well-known problem and solutions for scaling data IO do not work for metadata IO [1–3, 11, 15]. There are two approaches for improving the performance of metadata access.

2.1 Metadata Load Balancing

One approach for improving metadata performance and scalability is to alleviate overloaded servers by load balancing metadata IO across a cluster. Common techniques include partitioning metadata when there are many writes and replicating metadata when there are many reads. For example, IndexFS partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal; it does well for strong scaling because servers can keep more inodes in the cache which results in less RPCs. Alternatively, ShardFS replicates directory state so servers do not need to contact peers for path traversal; it does well for read workloads because all file operations only require 1 RPC and for weak scaling because requests will never incurr extra RPCs due to a full cache. CephFS employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies do not change depending on the balancing technique. Despite the performance benefits these techniques add complexity and jeopardize the robustness and performance characteristics of the metadata service because the systems now need (1) policies to guide the migration decisions and (2) mechanisms to address inconsistent states across servers.

Setting policies for migrations is arguably more difficult than adding the migration mechanisms themselves. For example, IndexFS and CephFS use the GIGA+ [8] technique for partitioning directories at a predefined threshold and using lazy synchronization to redirect queries to the server that "owns" the targeted metadata. Determining when to partition directories and when to migrate the directory fragments are policies that vary between systems: GIGA+ partitions directories when the size reaches a certain number of files and migrates directory fragments immediately; CephFS partitions directories when they reach a threshold size or when the write temperature reaches a certain value and migrates directory fragments when the hosting server has more load than the other servers in the metadata cluster. Another policy is when and how to replicate directory state; ShardFS replicates immediately and pessimistically while CephFS replicates only when the read temperature reaches a threshold. There is a wide range of policies and it is difficult to address with tunables and hard-coded design decisions.

In addition to the policies, distributing metadata across a cluster requiress distributed transactions and cache coherence protocols to ensure strong consistency (e.g., POSIX). For example, ShardFS pessimistically replicates directory state and uses optimistic concurrency control for conflicts; namely it does the operation and if there is a conflict at verification time it falls back to two-phase locking. Another example is IndexFS's inode cache which reduces RPCs by caching ancestor paths – the locality of this cache can be thrashed by random reads but performs well for metadata writes. For consistency, writes to directories in IndexFS block until the lease expires while writes to direcotries in ShardFS are slow for everyone as it either requires serialization or locking with many servers; reads in IndexFS are subject to cache locality while reads in ShardFS always resolve to 1 RPC. Another example of the overheads of addressing inconsistency is how CephFS maintains client sessions and inode caches for capabilities (which in turn make metadata access faster). When metadata is exchanged between metadata servers these sessions/caches must be flushed and new statistics exchanged with a scatter-gather process; this halts updates on the directories and blocks until the authoratitive metadata server

responds. These protocols are discusssed in more detail in the next section but their inclusion here is a testament to the complexity of migrating metadata.

The conclusion we have drawn from this related work is that metadata protocols have a bigger impact on performance and scalability than load balancing. Understanding these protocols helps load balancing and gives us a better understanding of the metrics we should use to make migration decisions (*e.g.*, which operations reflect the state of the system), what types of requests cause the most load, and how an overloaded system reacts (*e.g.*, increasing latencies, lower throughput, etc.).

2.2 Relaxing POSIX

POSIX workloads require strong consistency and many file systems improve performance by reducing the number of remote calls per operation (*i.e.* RPC amplification). As discussed in the previous section, caching with leases and replication are popular approaches to reducing the overheads of path traversals but their performance is subject to cache locality and the amount of available resources, respectively; for random workloads larger than the cache extra RPCs hurt performance [10, 16] and for write heavy workloads with more resources the RPCs for invalidations are harmful. Another approach to reducing RPCs is to use leases or capabilities.

High performance computing has unique requirements for file systems (e.g., fast creates) and well-defined workloads (e.g., workflows) that make relaxing POSIX sensible. One popular approach is to allow clients to "lock" parts of the namespace to improve performance and scalability by avoiding synchronization, false sharing, and serialization. BatchFS assumes the application coordinates accesses to the namespace, so the clients can batch local operations and merge with a global namespace image lazily. Similarily, DeltaFS eliminates RPC traffic using subtree snapshots for non-conflicting workloads and middleware for conflicting workloads. MarFS gives administrators the ability to (1) lock "project directories" and (2) allocate GPFS clusters for demanding directory workloads. TwoTiers eliminates high-latencies by storing metadata in a flash tier; apps lock the namespace so that metadata can be accessed more quickly. Unfortunately, decoupling the namespaces has costs: (1) merging metadata state back into the global namespace is slow; (2) failures are local to the failing node; and (3) the systems are not backwards compatible.

For (1), state-of-the-art systems manage consistency in non-traditional ways: IndexFS maintains the global namespace but blocks operations from other clients until the first client drops the lease, BatchFS does operations on a snapshot of the namespace and merges batches of operations into the global namespace, and DeltaFS never merges back into the global namespace. The merging for BatchFS is done by an auxiliary metadata server running on the client and conflicts are resolved by the application. Although DeltaFS never explicitly merges, applications needing some degree of ground truth can either manage consistency thesmelves on a read or add a bolt-on service to manage the consistency.

	Decoupled Namespace	Global Namespace
Example	BatchFS [17] DeltaFS [18]	CephFS [16] IndexFS [10]
Consistency	eventual	strong
Durability	node local	journal

Table 1: State-of-the-art systems in HPC improve file system metadata performance by relaxing consistency and durability guarantees.

For (2), if the client fails and stays down, all metadata operations on the decoupled namespace are lost. If the client recovers, the on-disk structures (for BatchFS and DeltaFS this is the SSTables used in TableFS) can be recovered. In other words, the clients have state that cannot be recovered if the node stays failed and any progress will be lost. This scenario is a disaster for checkpoint-restart where missed cycles may cause the checkpoint to bleed over into computation time.

For (3), decoupled namespace approaches sacrifice POSIX going as far as requiring the application to link against the systems they want to talk to. In today's world of software defined caching, this can be a problem for large data centers with many types and tiers of storage. Despite well-known performance problems POSIX and REST are the dominant APIs for data transfer.

Decoupling the namespace delays metadata consistency and sacrifices durability. As shown in Table 1, metadata consistency is provided by capabilities and metadata durability is addressed with a journal.

3 POSIX OVERHEADS

In our examiniation of the overheads of POSIX we benchmark and analyze CephFS, the file system that uses the Ceph's object store (*i.e.* RADOS) to store its data and metadata. We choose CephFS because it is an open-source production quality system. This file system is an implementation of one set of design decisions and our goal is to highlight the effect that those decisions have on performance.

To show how the file system behaves under high metadata load we use a create-heavy workload. Create-heavy workloads are studied the most in HPC research because of the checkpoint/restart use case but they also happen to stress the underyling storage system the most. Figure 2 shows the resource utilization of compiling the Linux kernel. The untar phase, which is characterized by many creates, has the highest resource usage which suggests that it is stressing the consistency and journalling subsystems of the metadata server the most. Traditional file system techniques for improving performance, such as caching inodes, do not help for create-heavy workloads.

In this section, we quantify the costs of strong consistency and global durability in CephFS. We use the kernel client

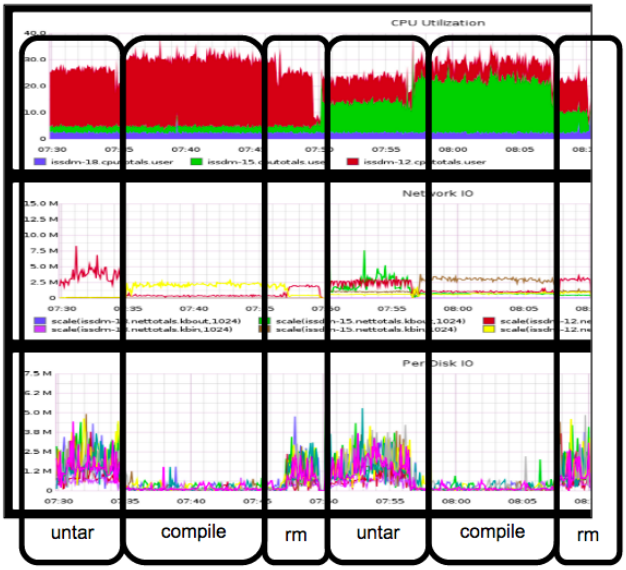


Figure 2: Create-heavy workloads (untar) incurr the highest disk, network, and CPU utilization because of the consistency and durability demands of CephFS .

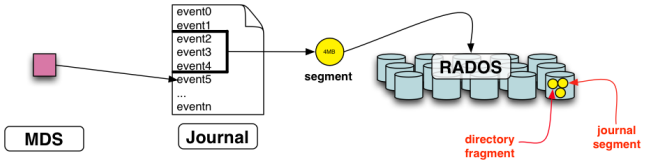


Figure 3: CephFS uses a journal to stage updates and tracks dirty metadata in the collective memory of the MDSs. Each MDS maintains its own journal, which is broken up into 4MB segments. These segments are pushed into RADOS and deleted when that particular segment is trimmed from the end of the log. In addition to journal segments, RADOS also stores per-directory objects.

so that we can find the true create speed of the server; our experiments show a low CPU utilziation for the clients which indicates that we are stressing the servers more.

3.1 Durability

While durability is not specified by POSIX, users expect that their metadata is safe. We define three types of durability: global, local, and none. Global durability means that the client or server can fail at any time and metadata will not be lost. Local durability means that metadata can be lost if the client or server fails and stays failed. None means that metadata is volatile and that the system provides no guarantees when clients or servers fail.

CephFS Design: a journal of metadata updates that streams into the resilient object store. Similar to LFS [12] and WAFL [5] the metadata journal can grow to large sizes ensuring (1) sequential writes into RADOS and (2) the ability for daemons to trim redundant or irrelevent journal entries. The journal is striped over objects where multiple journal updates can reside on the same object. There are two tunables for controlling the journal: the segment size and the number of parallel segments that can be written in parallel. The former is memory bound as larger segments take up more memory but can reduce the time spent journaling and the latter is CPU bound.

As shown in Figure 3, in addition to the metadata journal, CephFS also represents metadata in RADOS as a metadata store, where directories and their file inodes are stored as objects. The metadata server applies the updates in the journal to the metadata store when the journal reaches a certain size. The metadata store is optimized for recovery (*i.e.* reading) while the metadata journal is write-optimized.

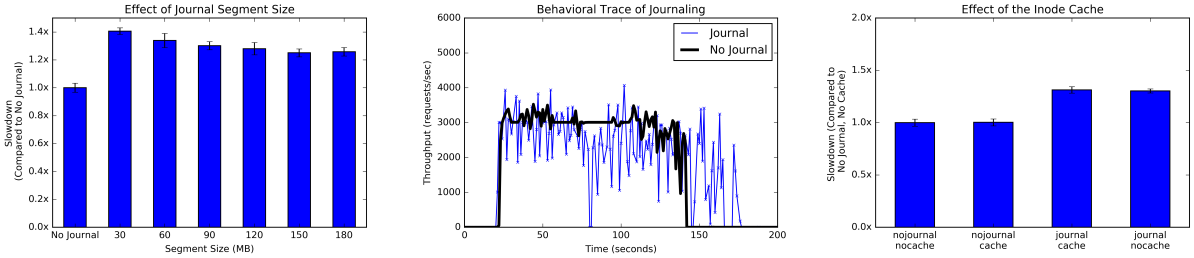
Figure 4 shows that journaling metadata updates into the object store has an overhead. Figure 4a shows the effect of journaling of different journal segment sizes; the larger the segment size the bigger that the writes into the object store are. The trade-off for better performance is memory consumption because larger segment sizes take up more space with their buffers. Figure 4b shows how the metadata server periodically stops serving requests to flush (*i.e.* apply journal updates to) the metadata store. The journal overhead is sufficient enough to slow down metadata throughput but not so much as to overwhelm the bandidth of the object store. We measured our peak bandwidth to be 100MB/s, which is the speed of our network link.

Comparison to decoupled namespaces: In BatchFS and DeltaFS, as far as we can tell, when a client or server fails there is no recovery scheme. For BatchFS, if a client fails when it is writing to the local log-structured merged tree (implemented as an SSTable) then those batched metadata operations are lost. For DeltaFS, if the client fails then on restart the computation does the work again – since the snapshots of the namespace are never globally consistent and there is no ground truth. On the server side, BatchFS and DeltaFS use IndexFS. Again, IndexFS writes metadata to SSTables but it is not clear whether they ever vacate memory, get written to disk, or are flushed to the object store.

3.2 Strong Consistency

Access to metadata in a POSIX-compliant file system is strongly consistent, so reads and writes are globally ordered. The synchronization and serialization machinery needed to ensure that all clients see the same state has high overhead.

CephFS Design: capabilities keep metadata strongly consistent. To reduce the number of RPCs needed for consistency,



(a) Journal Segment Size **(b) 180MB Segment vs. No Journal** **(c) Inode Cache**

Figure 4: The overhead of the file system metadata journal. The segment size is the threshold that the metadata server starts trimming.

clients can obtain capabilities for reading, reading and updating, caching reads, writing, buffering writes, changing the file size, and performing lazy IO.

To keep track of the read caching and write buffering capabilities, the clients and metadata servers agree on the state of each inode using an inode cache. If a client has the directory inode cached it can do metadata writes (*e.g.*, create) with a single RPC. If the client is not caching the directory inode then it must do an extra RPC to determine if the file exists. Unless the client immediately reads all the inodes in the cache (*i.e.* `ls -alR`), the inode cache is less useful for create-heavy workloads.

The benefits of caching the directory inode when creating files is shown in Figure 5a. If only one client is creating files in a directory ("isolated" curve on $y1$ axis) then that client can lookup the existence of new files locally before issuing a create request to the metadata server. If another client starts creating files in the same directory ("interfere" curve on $y1$ axis) then the directory inode transitions out of read caching and the first client must send `lookup()`s to the metadata server ("interfere" curve on $y2$ axis). These extra requests increase the throughput of the "interfere" curve because the metadata server can handle the extra load but the overall performance decreases. This degradation is shown in Figure 5b, where we scale the number of clients and show increased runtime and variability. The performance is compared to a single isolated client and he error bar is the standard deviation of the runtime of all clients. For the "interfere" bars, each client creates files in private directories and at 30 seconds we launch another process that creates files in those directories. For less than 7 clients, the runtime and deviation is worse when clients interfere. At 7 clients, the metadata server is overloaded so the directory caching in the "isolated" bars has no benefit. Zooming in on runs with 7 clients in Figure 5c we see how different interferring operations affect performance, again when compared to the performance of a single isolate client. "create" has little effect but "stat" and "create many" cause noticeable slowdowns because of the extra work they impose on the metadata server.

Comparison to decoupled namespaces: Decoupled namespaces merge batches of metadata operations into the global namespaces when the job completes. In BatchFS the merge is delayed by the application using an API to switch between asynchronous to sychronous mode. The merge itself is explicitly managed by the application but future work looks at more automated methdologies. In DeltaFS snapshots of the metadata subtrees stays on the client machines; there is no ground truth and consistent namespaces are constructed and resolved at application read time or when a 3rd party system (*e.g.*, middleware, scheduler, etc.) needs a view of the metadata. As a result, all the overheads of maintaining consistency that we showed above are delayed until the merge phase.

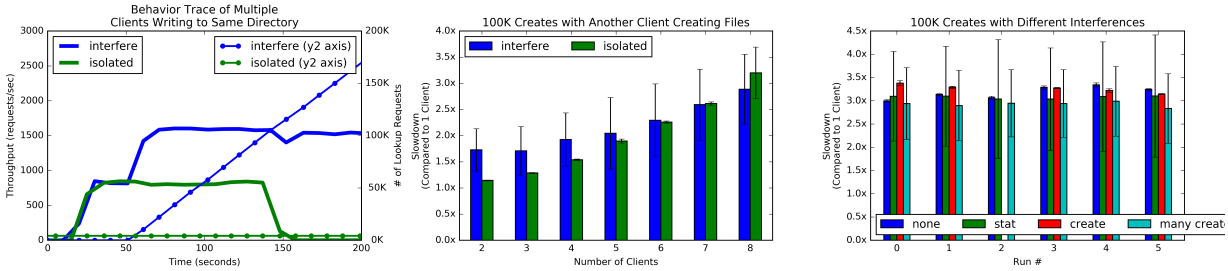
4 METHODOLOGY: DECOUPLED NAMESPACES

In this section we describe Cudele, our prototype system that lets future programmers compose mechanisms (Section §4.1) to provide the necessary guarantees (Section §4.2) for their application.

4.1 Cudele's Mechanisms

Figure 6 shows the mechanisms (labeled arrows) in Cudele and which entity they are performed by (gray boxes). The metadata store and journal are different ways of representing the namespace. The metadata store represents the namespace as a tree of directory fragments and is easier to read/traverse. On the other hand, the journal represent the namespace as a list of events. It a "pile system"; writes are fast but reads are slow because state must be reconstructed. Specifically, reads are slow because there is more data to read, it is unorganized, and many of the updates may be redundant.

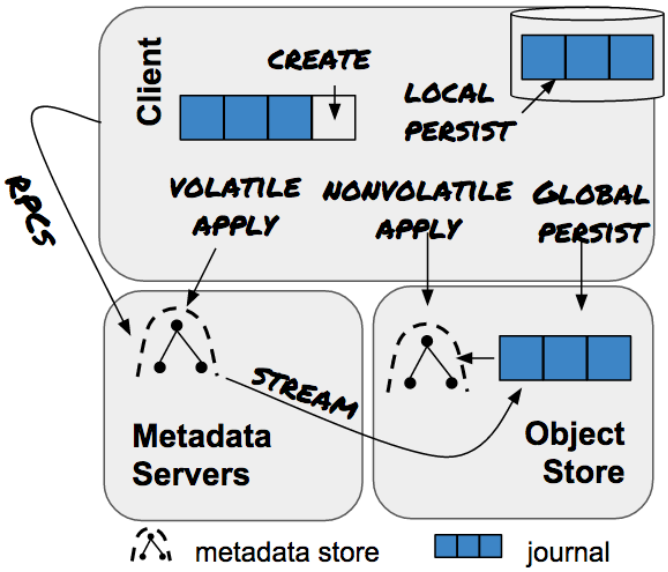
Cudele presents 6 mechanisms: RPCs, Stream, Create, Volatile Apply, Save, and Persist. "RPCs" does round trip remote procedure calls to establish consistency; it is the default implementation for complying with POSIX in CephFS. "Stream" has the metadata servers stream a journal of metadata updates into the object store. "Create" allows clients to append metadata events to an in-memory journal. "Volatile apply" takes the in-memory journal on the client and applies



(a) Inteference forces `lookup()`s **(b) Interference hurts performance** **(c) Different interferring requests**

Figure 5: When a client create stream is "isolated" then lookups resolve locally but when a second client "interferes" by creating in the same directory, the directory inode capability is revoked forcing all clients to centralize lookups at the metadata server.

Figure 6: Applications can decouple the namespace, write updates to a local journal, and delay metadata updates. Table 2 shows how these mechanisms (represented by the arrows) can be combined to provide weaker consistency or durability semantics.



it directly to the in-memory metadata store of the metadata server cluster. "Save" takes the in-memory journal and writes it to the client's disk. "Persist" saves the journal as a an object in the object store from the client.

Next, we discuss how these mechanisms can be composed to get different consistency and durability semantics.

4.2 Setting Policies with Cudele

The spectrum of consistency and durability guarantees that adminstrators can construct is shown in Table 2. The columns are the different consistency semantics and the rows cover the spectrum of durability guarantees. For consistency: "none"

Table 2: Future programmers can explore the consistency (C) and durability (D) spectrums by composing Cudele mechanisms. The consistency and durability properties are not guaranteed until all mechanisms in the cell are complete (i.e. the compositions should be considered atomic) and there are no guarantees while tranisitioning between policies.

C → D ↓	invisible	eventual	strong
none	create	create +volatile apply	RPCs
local	create +local persist	create +local persist +volatile apply	RPCs +local persist
global	create +global persist	create +global persist +volatile apply	RPCs +stream

means the system does not handle merging updates into a global namespace and it is assumed that middleware or the application manages consistency lazily; "eventual" merges updates either when the system has time (e.g., by a background daemon) or when the client is done writing; and updates in "global" consistency are seen immediately by all clients. For durability, "none" means that updates are volatile and will be lost on a crash of any component. Stronger guarantees are made with "local", which means updates will be retained if the client node recovers, and "global", where all updates are always recoverable.

The cells in Table 2 encompass many exisiting storage systems. POSIX systems like CephFS and IndexFS have global consistency and durability; DeltaFS has consistency and durability set to "none"; and BatchFS uses "eventual" consistency and "local" durability. These are just a few of the HPC examples.

To compose the mechanisms administrators inject which steps (described in Section §4.1) to run and which to use in parallel using a domain specific language. For example, to

get the semantics of BatchFS, the administrator would inject the following pipeline:

```
create+save+volatile apply
```

Although we can achieve all permutations of the different guarantees in Table 2, not all of them make much sense. For example, it makes little sense to do `creates+RPCs` since both steps do the same thing or `stream+save` since global durability is stronger and has more overhead than local durability. Formulaically, valid steps include:

```
<create|RPCs> [+v_apply][+save][+persist][+stream]
```

4.3 Cudele Namespace API: Per-Subtree Policies

To assign consistency and durability to the subtrees we store policies in the directory inode. This approach uses the File Type interface from the Malacology programmable store system [] and it tells clients how to access the underlying data or metadata. The underlying implementation stores executable code in the inode that calls the different Cudele mechanisms. Of course, there are many security and access control aspects of this approach but that is beyond the scope of this paper.

The interface for setting the subtree policies is with {path, ¡block—overwrite¿, pre-allocated inodes} tuples. For example:

`(msevilla/mydir, policies.yml)`

would decouple the path `msevilla/mydir` and would apply the policies in `policies.txt`. The policies file supports the following values:

- `allocated_inodes`: the number of inodes to allocate to the decoupled namespace (default 100)
- `interfere_callback`: how to handle a request from another client targeted at the now decoupled subtree (default `block`)
- `consistency_callback`: which consistency model to use (default `RPCs`)
- `durability_callback`: which durability model to use (default `stream`)

Given these default values decoupling the namespace with an empty policies file would give the application 100 inodes but the subtree would behave like the existing CephFS implementation. Table **??** shows how one would use the Cudele API to implement policies from related work.

For `block`, any requests to this part of the namespace return with "Device is busy", which will spare the MDS from wasting resources for updates that may get overwritten. If the application does not mind losing updates, for example it wants approximations for results that take too long to compute, it can select `overwrite`. In this case, metadata will be written and the computation from the decoupled namespace will take priority because the resutls are more accurate.

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "create"
    "durability": "local_persist"
}
```

Listing 1: Implementing DeltaFS with Cudele.

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "create+volatile_apply"
    "durability": "local_persist"
}
```

Listing 2: Implementing BatchFS with Cudele.

```
{
    "allocated_inodes": "-"
    "interfere_policy": "-"
    "consistency": "RPCs"
    "durability": "stream"
}
```

Listing 3: Existing CephFS on Cudele.

4.4 Programmability Approach

A programmable storage system exposes internal subsystem as building blocks for higher level services. This 'dirty-slate' approach limits redundant code and leverages the robustness of the underlying storage system. Cudele uses this approach to great success and requires only:

- 354 lines of library code
- 219 lines of non-destructive metadata server code, which is not used unless it is turned on
- 4 lines of destructive client/server code to check whether a namespace is decoupled

4.4.1 Metadata Store.

4.4.2 Journal.

4.4.3 Journal Tool. The journal tool is used for disaster recovery and lets administrators view and modify the journal of metadata updates; it can read the journal, erase events from the journal, and apply the updates in the journal to the metadata store. To apply journal updates to the metadata store, the journal tool reads the journal segments from object store objects and applies the update to the metadata store (which are also stored as object store objects).

The journal tool imports journals from binary files stored on disk. First the header of the dump is sanity checked and written into RADOS to the "header" object. The "header" object has metadata about the journal as well as the locations of all the journal pointers (e.g., where the tail of the journal is, where we are currently trimming, etc.). Then the journal events are cleaned (erasing trailing events that are not part of the header) and written as objects into RADOS. Note that while the journal is in RADOS, the metadata servers do do not have the namespace reconstructed in memory so the metadata cluster will not service requests relating to the journal of imported events. To construct the namespace in the collective memory of the metadata servers we need to first construct the namespace in RADOS. The journal tool can explicitly do this by applying the journal to the metadata store in RAODS. This will pull the objects containing journal segments and replay them on the metadata store. Finally, we delete the journal in RADOS and restart the metadata servers so they rebuild their caches.

The journal tool exports journals to binary files stored on disk. First the journal is scanned for the header and then journal is recovered. To recover the journal the "header" object is read off disk and then objects are probed in order and starting from the write position saved in the header. Probing will update the write position if it finds objects with data in them.

When exporting a journal of events, the journal tool first scans the journal to check for corruption. Then it recovers the journal by reading the "header" object out of RADOS. After reading the header, the journal tool can pull journal segments from RADOS because it knows how many objects to pull and how far to seek within those objects.

4.4.4 *Inode Cache.*

4.4.5 *Large Inodes.*

5 IMPLEMENTATION

Each section below corresponds ot the labeled arrows in Figure 6. This implementation decouples policies from mechanisms allowing applciations to choose the consistency and durability semantics they need.

Of the 6 mechanisms in Figure 6 4 had to implemented and only 1 required changes to the underlying storage system itself. RPCs and stream can be achieved with tunables in configuration files for the storage system (e.g., metadata cache size, logging on/off, etc.). Persist", save, and create are implemented as library and does not require modifications to the storage system. Volatile apply requires changes to the metadata server to inject updates back into the global namespace.

5.1 No Changes: RPCs, Stream

RPCs is the default behavior of the storage system. Depending on the cache state and configuration each operation incurrs at least RPC. For example, if a client or metadata server is not caching the directory inode, all creates within that directory will result in a lookup and a create request. If the directory inode is cached then only the create needs to be sent.

Re-using the debugging and testing features in the storage system, we can turn stream on and off. If it is off, then the metadata servers will not save journals in the object store and the daemons that apply the journal to the metadata store will never run. Performance numbers for enabling and disabling the stream mechanism are presented back in Section §3.1.

The journal segments are saved as objects in RADOS. The journal has 4 pointers, described in 'osdc/Journaler.h':

- write position: tail of the journal; points to the current session where we are appending events
- unused field: where someone is reading
- expire position: old journal segments
- trimmed position: where daemon is expiring old items

Journal segments in RADDS have a header followed by serialized log events. The log events are read by hopping over objects using the read offset and object size pulled from the journal header. After decoding them, we can examine the metadata (1) about the event (e.g., type, timestamps, etc.) and (2) for inodes that the event touches.

The metadata for inodes that the event touches are called metadata blobs and the ones associated with events are **unordered**; this layout makes writing journal updates fast but the cost is realized when reading the metadata blobs. It makes sense to optimize for writing since reading only occurs on failures. To reconstruct the namespace for the metadata blob, the journal tool iterates over each metadata blob in the events and builds mappings of inodes to directory entries (for directories) and parent inodes to child inodes/directory entries.

5.2 External Library: Create, Save, Persist

For "create", "save", and "persist" Cudele provides a library for clients to link into. Recall that CephFS uses the object store (1) as a metadata store for all information about files including the hierarchical namespace and (2) as a staging area for the journal of updates before they are applied to the metadata store. As discussed previously, the metadata store is optimized for reading while the journal is optimized for writing. Cudele re-uses the journal tool to read/write log events to memory and persistent storage.

For create, clients write to an in-memory journal and updates are merged into the global namespace by replaying them onto the metadata store in the object store.

For save, clients write serialized log events to a file on local disk and for persist, clients store the file in the objects store. The overheads for save and persist is the write bandwidth of the local disk and object store, respectively.

This required no changes to Ceph because the metadata server knows how to read the events we were writing into the object store. The client writes metadata updates locally and merges the updates with the journal tool. The client that decouples the namespace operates without any consistency

and any conflicts at the merge are resolved in favor of this client. Updates by other clients (*i.e.* metadata writes to the global namespace) are overwritten. We leverage the journal tool's ability to reconstruct metadata events in memory. The client library is shown in Figure 6. Cudele adopts the following process when an application decouples the namespace:

(1) "**decoupled**": metadata server exports the journal events to a file
(2) "**transfer**": the file is pulled by the client from the metadata server
(3) "**snapshot**": client reads the file and materializes a snapshot of the namespace in memory

By using re-using the journal subsystem to implement the namespace decoupling, Cudele leverages the write/read optimized data structures, the formats for persisting events (similar to TableFS's SSTables), and the functions for replaying events onto the internal namespace data structures.

5.3 Storage System Changes: Volatile Apply

The volatile apply mechanism (*i.e.* the "v_apply" arrow in Figure 6) takes an in-memory journal on the client and applies the updates directly to the in-memory namespace maintained by the metadata servers. We say volatile because Cudele makes no consistency or durability guarantees. If a concurrent update from a client occurs there is no rule for resolving conflicts and if the client or metadata server crashes there may be no way to recover. Relaxing these constraints gives volatile apply the best performance.

In contrast, apply uses the the object store to merge the journal of updates from the client to the metadata server. Apply is safer than volatile apply but has a performance overhead because objects in the metadata store need to be read from and written back to the object store. Apply was already implemented by the journal tool: it reads the journal from the object store, adds events to the journal, and writes the metadata updates out to the metadata store in the object store. Cudele's volatile apply executes "save" (described in §5.2), transfers the file of metadata updates to the metadata server, and then merges the updates directly into the in-memory metadata store of the metadata cluster.

Creating many files in the same directory would touch the same object but the existing implementation results in this object being repeatedly pushed/pulled.

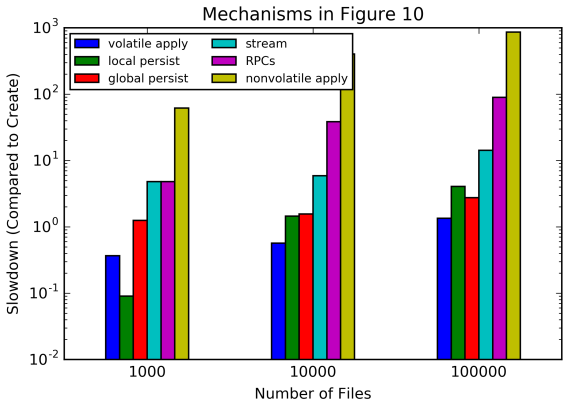


Figure 7: [source] The performance of the Cudele mechanisms normalized to the runtime of the create mechanism. The runtime of the create mechanism is the time it takes to write 100 file creates to the client's in-memory journal of metadata updates.

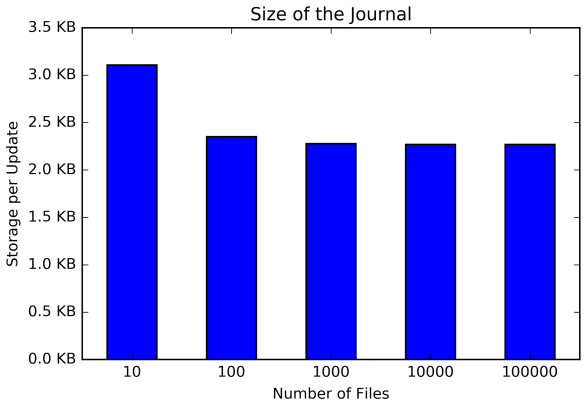


Figure 8: [source] In-memory client journal scales with the number of updates.

6 EVALUATION

We evaluate Cudele on a 15 node cluster, partitioned into 8 object storage servers, 3 metadata servers, and 2 monitor servers. The object storage servers double as clients which is fine because clients are CPU and memory bound while object storage servers are disk IO bound. All daemons run as a single process which is the default setting for Ceph and the nodes have 2 dual core 2GHz processors with 8GB of RAM. There are three daemons per object storage server (one for each disk formatted with XFS) and they share an SSD for the journal. The nodes are running Ubuntu 12.04.4, kernel version 3.2.0-63 but all experiments run in Docker containers; this makes it easier to tear down and re-initialize (*e.g.*, dropping the kernel cache) for the cluster between experiments.

This paper adheres to The Popper Convention[1] [6], so experiments presented here are available in the repository for this article[2]. Experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure. That link points to a Jupyter notebook that shows the analysis and source code for that graph, which points to an experiment and its artifacts.

6.1 Cudele Mechanism Performance

Figure 7 shows the runtime of the Cudele mechanisms, normalized to the time it takes to write 100K file create updates to the client's in-memory journal (*i.e.* the create mechanism). Bars above 10^0 are slower than the create mechanism and bars below are faster. "Stream" is an approximation of the overhead and is calculated by subtracting the runtime of the job with the journal turned off from the runtime with the journal turned on. All the slowdowns and speedups reported are for the 100K file create job, the largest workoad we tested.

6.1.1 Overhead of RPCs. "RPCs" is 66× slower than "volatile apply" because sending individual metadata updates over the network is costly. While "RPCs" sends a request for every file create, "nonvolatile apply" writes all the updates to the in-memory journal and applies them to the in-memory data structures in the metadata server. While communicating the decoupled namespace directly to the metadata server is faster, communicating through the object store ("nonvolatile apply") is 10× slower.

6.1.2 Overhead of "nonvolatile apply". The cost of "nonvolatile apply" is much larger than all the other mechanisms. That mechanism was not implemented as part of Cudele – it was a debugging and recovery tool packaged with CephFS. It works by iterating over the updates in the journal and pulling all objects that *may* be affected by the update. This means that two objects are repeatedly pulled, updated, and pushed: the object that houses the experiment directory and the object that contains the root directory (*i.e.* /). The cost of communcating through the object store is shown by comparing the runtime of "volatile apply" + "global persist" to "nonvolatile apply". These two operations end up with the same final metadata state but using "nonvolatile apply" is clearly inferior.

6.1.3 Parallelism of the Object Store. Comparing "local" and "global persist" demonstrates the bandwidth advantages of storing the journal in a distributed object store. As the journal size increases, the "global persist" performance is 1.5× faster because the object store is leveraging the collective bandwidth of the disks in the cluster. This benefit comes from the object store itself but should be acknowledged when making decisions for the application; the size of the object store can help mitigate the overheads of globally persisting metadata updates.

6.1.4 Journal Size. Figure 8 shows the amount of storage per journal update (y axis) for the range of file creates we tested (x axis). The increase in file size is linear with the number of metadata creates and suggests that updates for a million files would be $2.5\text{KB} * 1 \text{ million files} = 2.38\text{GB}$. Transfer times for files this large on an HPC network are reasonable.

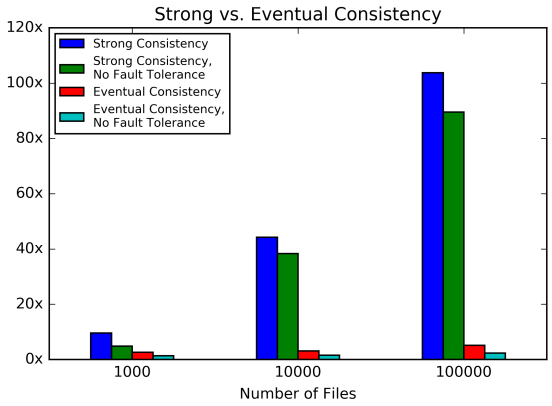


Figure 9: The RPC per metadata update of "Strong Consistency" has a large overhead compared to the decoupled namespace strategy of "Eventual Consistency".

6.2 Eventual vs. Strong Consistency

Figure 9 shows the runtimes of eventual and strong consistency based systems implemented on Cudele, normalized to the runtime of the create mechanism (again, just creating files in the client's in-memory journal). We scaled the number of files up to 100K which is the maximum size of a directory by default in CephFS. We use the following compositions from the mechanisms in Table 2:

- Strong Consistency
  `RPCs + stream`
- Strong Consistency, No Fault Tolerance
  `RPCs`
- Eventual Consistency
  `creates + local persist`
- Eventual Consistency, No Fault Tolerance
  `creates + local persist + volatile apply`

We compare these semantics because the final metadata states are equivalent. Cudele makes no guarantees during execution of the mechanisms or when transitioning semantics – the semantics are guaranteed *once the mechanism completes*. So if servers fail during a mechanism, metadata or data may be lost.

6.2.1 Speedups of Decoupled Namespaces. Eventual consistency uses the decoupled namespace strategy and shows up to a 20× speedup over the traditional namespaces that use RPCs. Compared to the baseline the slowdown is $5 - 7\times$ for Strong Consistency, which emulates BatchFS and $90 - 104\times$ for Eventual Consistency, which emulates DeltaFS.

6.2.2 Durability << Consistency. The 1.15× overhead of "Strong Consistency" compared to "Strong Consistency, No Fault Tolerance" for 100K files is neglible. It suggests that the overhead of consistency is much larger than the overhead

[1] http://falsifiable.us
[2] https://github.com/michaelsevilla/cudele-popper/

of durability. This conclusion should be stronger as we scale the number of files because the cost of streaming the journal into the object store is constant. We omit the same analysis for "Eventual Consistency" because the runtimes are so short that the normalized slowdowns are misleading.

6.2.3 Metadata Formats. Because the metadata formats are the same for all schemes we argue that the performance gain for decoupled namespaces comes from relaxing the consistency guarantees and not from the metadata formats, as was argued in previous work outlining the benefit of SSTables [9, 10].

6.3 Isolation from EBUSY

In this section, we show the isolation benefits of subtree policies. We repeat the experiment from Figure 5b, where clients write to their own private directories and another client interferes at 30 seconds. We also have 2 clients write to decoupled namespaces and merge their updates 90 seconds. The

6.3.1 Benefits of Isolated Subtree Policies.

6.3.2 Cost of Merging.

6.3.3 Scaling Concurrent Merges.

6.4 Isolation from Global Writes, Cost of Overwrites

6.5 Partial Directory Listings

REFERENCES

[1] Christina L. Abad, Huong Luu, Yi Lu, and R Campbell. 2012. *Metadata Workloads for Testing Big Storage Systems.* Technical Report. Citeseer.

[2] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. 2012. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*. 125–132. http://dx.doi.org/10.1109/UCC.2012.27

[3] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. 2011. Parallel I/O and the Metadata Wall. In *Proceedings of the 6th Workshop on Parallel Data Storage (PDSW'11)*.

[4] John Bent, Brad Settlemyer, and Gary Grider. Serving Data to the Lunatic Fringe. (????).

[5] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference (WTEC'94)*.

[6] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. 2016. *Popper: Making Reproducible Systems Performance Evaluation Practical, UCSC-SOE-16-10.* Technical Report UCSC-SOE-16-10. UC Santa Cruz.

[7] Kirk McKusick and Sean Quinlan. 2010. GFS: Evolution on Fast-forward. *Commununications ACM* 53, 3 (March 2010), 42–49.

[8] Swapnil V. Patil and Garth A. Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*.

[9] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*.

[10] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the 20th ACM/IEEE Conference on Supercomputing (SC '14)*.

[11] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. 4–4.

[12] M. Rosenblum and J.K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. In *ACM Transactions on Computer Systems*.

[13] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[14] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*.

[15] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation (OSDI'06)*.

[16] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing (SC'04)*.

[17] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Workshop on Parallel Data Storage (PDSW' 14)*.

[18] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Workshop on Parallel Data Storage (PDSW' 15)*.