Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace

Paper 382

ABSTRACT

HPC developers are abandoning POSIX because the synchronization and serialization overheads of providing strong consistency and durability are too costly – and often unnecessary – for their applications. Unfortunately, designing near-POSIX file systems excludes applications that rely on strong consistency or durability, forcing developers to re-write their applications or deploy them on a different system. We present a file system and API that lets clients specify their consistency/durability requirements and assign them to subtrees in the namespace, allowing users to optimize subtrees within the same namespace for different workloads. We draw conclusions about the performance impact of unexplored consistency/durability metadata designs and show that maintaining strong consistency can cause about a 100× slow down compared to relaxed consistency and no durability. Comparatively, merging updates after a period of relaxed consistency (less than a 10× slow down) and maintaining durability (about a 10× slow down) have more reasonable costs.

ACM Reference format:
Paper 382. 2017. Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 EVALUATION

We did not compare to other HPC systems because we are not claiming that our file system metadata protocols are superior. Instead, we use Ceph as a platform for exploring related work over the same storage system, which facilitates an apples-to-apples comparison of the strategies themselves. So we run experiments on the same cluster and normalize results to control for hardware differences and to make our results more generally applicable. Our goal is not to build a file system optimized for HPC workloads and our prototype is not a competitor to Lustre. Rather it is a solution for letting the techniques of a system like Lustre live in the same namespace as techniques optimized for file creates, like BatchFS.

We evaluate our prototype on the two clusters in Table 1. Microbenchmarks were run on the in-house clusters while client scalability tests were run on CloudLab. We only scale up to 29 clients because that is the maximum capacity of a single metadata server. For the in-house cluster, we need more object storage servers and we have to co-locate clients on the same node because of resource contention [1]. All daemons run as a single process which is the default setting for Ceph. We develop on Ceph's Jewel release, version 10.2.1, which was released in May 2016.

This paper adheres to The Popper Convention[2] [2], so experiments presented here are available in the repository for this article[3]. 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.

Cluster	Hardware	Software
In-House	15 nodes, 4 2GHz CPUs	Ubuntu 12.04, 1 MON
	8GB RAM, SSD	1 MDS, 8 OSDs, 8 Clients
CloudLab	34 nodes, 16 2.4GHz CPUs	Ubuntu 14.04, 1 MON
	128GB RAM, SSD	1 MDS, 3 OSDs, 29 Clients

Table 1: In-House used for microbenchmarks, CloudLab for Use Cases. MON is monitor daemon, MDS is metadata server daemon, and OSDs are object storage daemons.

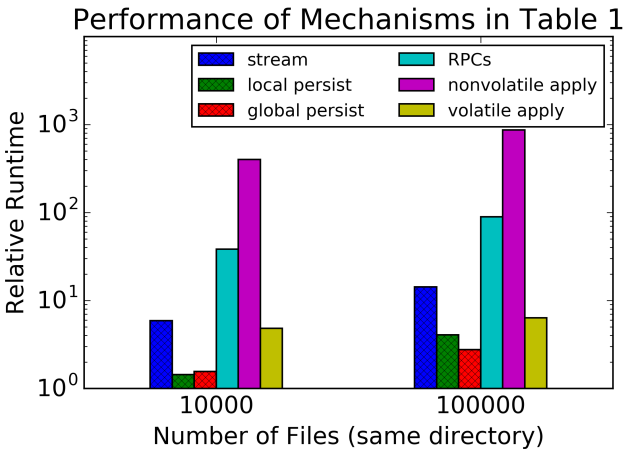

Figure 1: [source] Microbenchmark: performance of each mechanism. Results are normalized to the runtime of the "Append Client Journal" mechanism (the runtime of writing n file creates to the client's in-memory journal).

1.1 Microbenchmarks

Figure 1 shows the runtime of the Cudele mechanisms for a single client creating files in the same directory, normalized to the time it

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

[1] The bandwidth to RADOS is only sufficient with 8 nodes on this older hardware, which uses SSDs and nodes that are almost 10 years old. The CloudLab cluster performs just as well with and without a highly provisioned object storage server cluster.
[2] http://falsifiable.us
[3] Links have been removed for double-blind submission

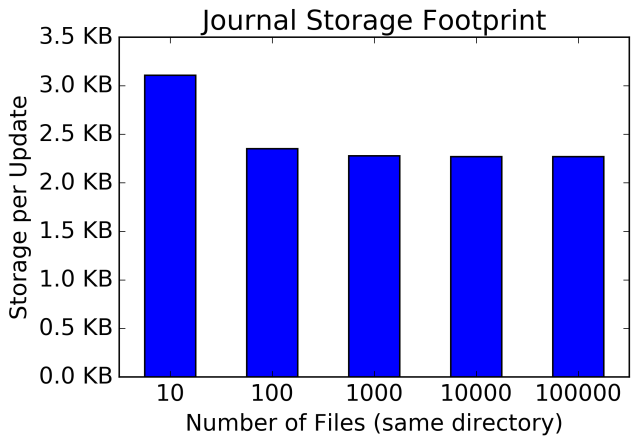

Figure 2: [source] Microbenchmark: storage footprint on client. The size of the client's journal scales with the number of updates.

takes to write n file create updates to the client's in-memory journal (*i.e.* the "Append Client Journal" mechanism). Bars above 10^0 are slower than the "Append Client Journal" mechanism. "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. The largest workload we tested is 100K file creates in the same directory. This is the maximum recommended size of a directory in CephFS; preliminary experiments with larger directory sizes show memory problems.

Poorly Scaling Data Structures: Despite doing the same amount of work, mechanisms that rely on poorly scaling data structures have large slowdowns for the larger number of creates. For example, "RPCs" which relies on the internal CephFS directory structures, goes from a $40\times$ slowdown for 10K files to a $90\times$ slowdown for 100K files. It is a well-known problem that directory data structures do not scale when creating files in the same directory [4] and any mechanism that uses these data structures will experience similar slowdowns. Other mechanisms that write events to a journal (*e.g.* the persists, "Volatile Apply") experience a much less drastic slowdown because the journal data structure does not need to be scanned for every operation. Events are written to the end of the journal without even checking the validity (*e.g.*, if the file already exists for a create), which is another form of relaxed consistency because the file system assumes the application has resolved conflicting updates in a different way.

Overhead of RPCs: "RPCs" is $66\times$ 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 $10\times$ slower.

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 communicating 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.

Parallelism of the Object Store: Comparing "Local" and "Global persist" demonstrates the bandwidth advantages of storing the journal in a distributed object store. For 100K file creates, the "Global Persist" performance is $1.5\times$ 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.

Journal Size: Figure 2 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 payloads of this size on an HPC network are reasonable.

Takeaway: the Cudele mechanisms have overheads and costs that can differ *by orders of magnitude*. Cudele gives users the ability to compose these mechanisms based on their application's correctness requirements and performance goals.

1.2 Use Case 1: Creates in the Same Directory

Clients creating files in private directories is heavily studied in HPC [3–5, 7, 8], mostly due to checkpoint-restart [1]. But the workload also appears in cloud workloads, where systems like Hadoop use the file abstraction to exchange work units to workers or to indicate when jobs complete [6]. A more familiar example is uncompressing an archive (*e.g.*, `tar xzf`), where the file system services a flash crowd of creates across all directories as shown in Figure **??**. We use this as a microbenchmark because it allows us to control the size of the log, since each create creates a single journal event.

Figure 3 shows the runtime of systems employing weak and strong consistency, normalized to the runtime of the "Append Client Journal" mechanism (again, just creating files in the client's in-memory journal). We use the following compositions from the mechanisms in Table **??**: Strong Consistency = "RPCs" + "Stream"; Strong Consistency, No Fault Tolerance = "RPCs"; Weak Consistency = "Append Client Journal" + "Local Persist"; and Weak Consistency, No Fault Tolerance = "Append Client Journal" + "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.

Speedups of Decoupled Namespaces: Weak consistency uses the decoupled namespace strategy and shows up to a $20\times$ speedup over

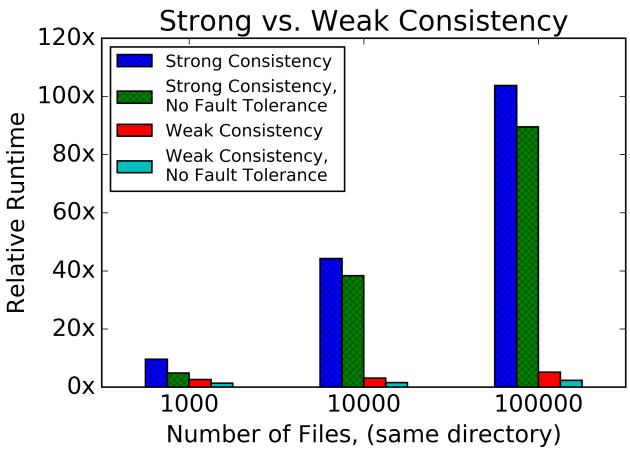

Figure 3: [source] Use Case 1: the RPC per metadata update (strong consistency) has a large overhead compared to decoupled namespaces (weak consistency.)

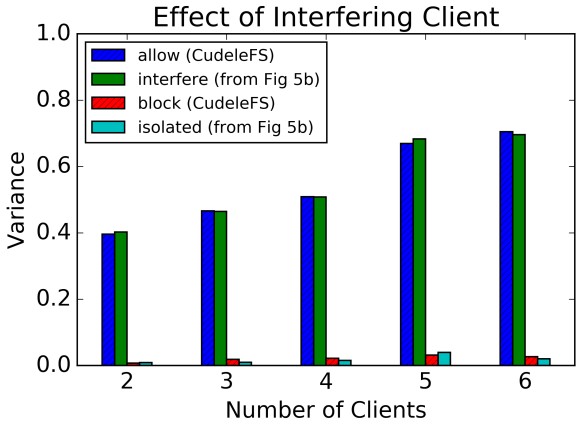

Figure 4: [source] Use Case 2: using the "allow" and "block" API, users can isolate directories from interfering clients. Variance with blocking turned on is the same as "isolated" from Figure ??.

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 Weak Consistency, which emulates DeltaFS.

Durability << Consistency: The $1.15\times$ overhead of Strong Consistency compared to Strong Consistency, No Fault Tolerance for 100K files is negligible. It suggests that the overhead of consistency is much larger than the overhead 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 Weak Consistency because the runtimes are so short that the normalized slowdowns are misleading.

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.

Takeaway: Cudele shows the true benefit of eventual consistency, where we see over a $100\times$ slowdown for achieving strong consistency, in the worst case.

1.3 Use Case 2: Creates with Rogue Client

Clients create files in private directories and a separate client, which we call a "rogue" client, creates files in each directory. This introduces false sharing and the metadata server revokes capabilities on directories touched by the rogue client. While HPC tries to avoid these situations with worklflows [8, 9], it stills happens in distributed file systems when users unintentionally access directories in a shared file system.

Next we show how Cudele can be programmed to block interfering clients. We use the same problematic workload from Figure **??**, where clients write to their own private directories and another client interferes with a stream of creates at 30 seconds. We also have another client write to a decoupled namespace and merge its updates at 90 seconds. We equip each client directory with the configuration:

```
{
    "allocated_inodes": "100000"
    "interfere_policy": "block"
    "consistency": "RPCs"
    "durability": "stream"
}
```

Each directory functions with "RPCs" and "Stream" enabled – which is the default implementation of CephFS. The only difference is that we enable blocking on each subtree so while the tree behaves like CephFS, all interfering operations will be returned with `-EBUSY`. Note that IndexFS does a similar operation with leases except clients block.

To show the benefits of this isolations, our results in Figure 4 are plotted alongside the variance bars from Figure **??**. Because "allow"/"interfere and "block"/"isolated" have the same variability we draw the following three conclusions: (1) clients that use the API to block interfering clients get the same performance as isolated clients, (2) there is a negligible effect on performance for the extra work the metadata server does to return `-EBUSY`, and (3) merging updates from the decoupled client has a negligible effect on performance.

Takeaway: the API lets users isolate directories when applications need better and more reliable performance. This is a way of controlling consistency.

1.4 Use Case 3: Read while Writing

Scientists use the file system to check the progress of jobs using `ls` [**?**]. The number of files or size of the files is indicative of the progress. This practice is not too different from systems that use the file system to manage the progress of jobs; Hadoop writes to temporary files, renames them when complete, and creates a "DONE" file to indicate to the runtime that the task did not fail and should not be re-scheduled on another node. In this scenario, Cudele users will not see the progress of decoupled namespace since their

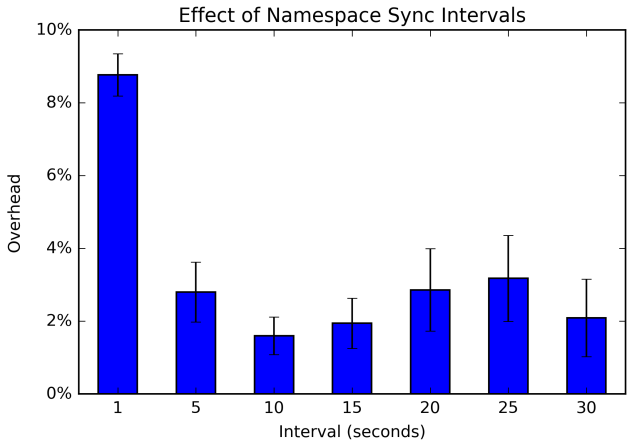

Figure 5: [source] Use Case 3: slowdown of a single client periodically syncing updates to the gobal namespace. The crossover point represents the trade-off of frequent updates (with small journal files) and infrequent updates (with larger journal files).5.

updates are not globally visible. To help scientists judge the progress of their jobs, Cudele clients have a "namespace sync" that sends batches of updates back to the global namespace at regular intervals.

Figure 5 shows the performance degradation of a single client writing updates to a decoupled namespace and pausing to send updates to the metadata server. We scale the namespace sync interval to show the trade-off of frequently pausing or writing large logs of updates. We use an idle core to log the updates and to do the network transfer. The client only pauses to fork off a background process, which is expensive as the address space needs to copied. The alternative is to pause the client completely and write the update to disk but since this implementation is limited by the speed of the disk, we choose the memory-to-memory copy of the fork approach.

To show the contention at the metadata server of this approach, we scale the number of journal merge events, each hitting different directories in the namespace, in FigureX. The size of each journal is 100K files, the maximum recommended directory size for CephFS. We also plot the cost of RPCs with and without journaling.

Merge requests are serialized at the metadata server; merging concurrently is an optimization for future work. The current implementation has race conditions because the metadata server recovery code was not designed to run in parallel (*i.e.* the metadata server never recovers with multiple threads). As a result, the metadata server versions inodes and directory entries to ensure that the recovered metadata is consistent.

Performance is better than RPCs because we place no restrictions on the validity of metadata inserted into the journal and avoid touching poorly scaling data structures. In other words, we can scale linearly if we weaken our consistency by never checking for the existence of files before creating files. only appends `open()` requests to the journal. When the updates are merged by the metadata server into the in-memory metadata store they never scan growing data structures. Had we implemented the "Create" mechanism to include `lookup()` commands before `open()` requests, we would have seen the poor scaling that we see with the "RPCs" mechanism.

Takeaway: the weakest form of consistency for creating files (*i.e.* not checking data structures for the validity of an update or existence of a file) shows linear scalability and stable performance.

1.5 Use Case 4: Reading Large Directories

The final use case is reading large directories. At job completion, scientists might use `ls` again to see the results. This causes great strain on the file system as paths need to traversed and the entire directory, with all its entries, must be transferred back to the client. Here we show how decoupling a large namespace and materializing the view in memory on the client is faster than doing RPCs for walks of the file system namespace.

FigureX shows how fast a client can materialize decoupled namespaces in memory. This takes the on-disk file system metadata format and parses them into events that can be manipulated and appended to. Again, we scale the number of up

REFERENCES

[1] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009) *(SC '09)*.

[2] 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.

[3] 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)*.

[4] 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)*.

[5] 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)*.

[6] K onstantin V. Shvachko. 2010. HDFS Scalability: The Limits to Growth. *login; The Magazine of USENIX* (2010).

[7] 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)*.

[8] 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)*.

[9] 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)*.