The Latency and Lifetime of a CephFS Request

Michael Sevilla[1], Noah Watkins[1], Carlos Maltzahn[1], Ike Nassi[1], Scott Brandt[1], Sage Weil[2], Greg Farnum[2], Sam Fineberg[3]

[1]UC Santa Cruz, {msevilla, jayhawk, carlosm, inassi, scott}@soe.ucsc.edu

[2]Red Hat, {sage, gfarnum}@redhat.com

[3]HP Storage, fineberg@hp.com

ABSTRACT

1. INTRODUCTION

In a POSIX file system, whenever a file is touched the client must access the file's metadata. Serving metadata and maintaining a file system namespace for today's workloads is sufficiently challenging because metadata requests impose small, frequent accesses on the underlying storage system [15]. This skewed workload is very different from data I/O workloads. As a result, file system metadata services do not scale for sufficiently large systems in the same way that read and write throughput do [1, 2, 3, 18]. Furthermore, clients expect fast metadata access, making it difficult to apply data compressions and transformations [9]. When developers scale file systems, the metadata service becomes the performance critical component.

One approach for servicing metadata requests more quickly is to distribute the metadata service. CephFS, the POSIX-compliant file system for Ceph [18], takes this approach; it distributes metadata amongst metadata servers (MDSs) using dynamic subtree partitioning [19].

Distributing file system metadata across a cluster is difficult because workload hotspots skew requests towards the same servers. This unbalanced behavior overloads those servers and reduces the scalability of the system. The reason that more requests go to certain servers is because POSIX requires path traversals to verify access controls (*e.g.,* permissions).

Getting the metadata for a pathname requires recursing up each parent directory to ensure that the client has access to the object. With this scheme, many files can share the same pathname prefix resulting in more requests for parent metadata. As a result, metadata near the top of the hierarchical namespace (i.e. root inodes) are accessed more frequently.

Previous work [**?**] attempted to balance load across the metadata cluster but the scalability and performance of the system was limited by CephFS's metadata protocols. the metrics used to make these migration decisions have high variance and do not accurately reflect the state of the system. Eventually, we want to examine *request type* locality as a metric for balancing metadata load. Requests have different latencies and behaviors, which adhere to the system's metadata protocols. It would be a good metric because they usually adhere to the same code path, so if latencies start increasing, then it is an indication of system overload. Treating all requests as the same is a fatal mistake because not all requests do the same thing.

Load balancing is a worthwhile goal, especially with the rise of load balancing in other fields (computing [22] and databases [7]) and with movable resources, cores [22], memory [6], IO [13], and network [8], on the horizon. But in this paper, we focus on the communication protocol between MDS nodes to see if a shared log is as effective as custom RPCs.

2. BACKGROUND: MANTLE

We designed a metadata balancer called Mantle that separates metadata migration policies from its mechanisms by accepting policies as injectable code. For example, to control when metadata load is migrated, the administrator can inject one of the policies shown in Figure 1. The greedy spill balancer aggressively migrates load when the current MDS (`whoami`) has metadata load and when its neighbor (`whoami`) has none. The fill & spill balancer conservatively migrates load by sending requests to the current MDS until its CPU utilization eclipses a threshold (45%). For these balancers, the metrics for determining when the MDS is overloaded are the the number of metadata requests and CPU utilization, respectively.

While these metrics work for simple jobs, they lack the important contextual information about the job to adequately making the mapping from traditional resource utilization (*e.g.,* CPU utilization, memory usage, etc). Using the number of metadata requests as a metric treats all requests the same, even though some requests take longer than others. CPU utilization provides an even more macro-level view of the MDS and does not account for how the MDS reacts to different requests.

2.1 Tieing the metric to performance (goodness)

```
-- Balancer option 1: greedy spill

if MDSs[whoami]["load"]>.01 and
   MDSs[whoami+1]["load"]<.01 then

-- Balancer option 2: fill & spill

if MDSs[whoami]["cpu"]>48 then
```

Figure 1: The sample balancing policies injected into Mantle [1] use poor metrics, like metadata load and CPU utilization, that fail to account for the latencies of different requests.

2.2 The pitfalls of hard-coding thresholds

3. WHAT'S ALREADY THERE

CephFS's MDS nodes already track the latencies for all metadata requests in the logs and performance counters. Latencies collected with the debug logs are heavy weight and may change the behavior of the MDS. Latencies collected with the performance counters do not capture enough context, like what the request was, what the request wanted, and which MDS the request goes to. Tracing with LTTng gets the best of both worlds - the flexibility of logging with without the overhead.

LTTng tracepoints are used into the other Ceph components, like the OSD, RADOS block device, and RADOS library. The RADOS block device tracepoints are the most similar to the MDS tracepoints we add. They are triggered on each operation, like `clone`, `copy`, and `stat` and track metadata like the request name, snapshot name, and the read/write permissions.

One disadvantage of using LTTng is the overhead of adding tracepoints and context to existing tracepoints. The tracepoints are compiled into the source code, so adding new tracepoints or changing existing ones requires recompiling the Ceph source code and restarting the cluster. Some common context, like the pthread ID, can be added while the system is running, but these are specific to the kernel.

4. IMPLEMENTATION

Tracepoints are added to the MDS functions that handle requests and reply to clients. Unlike the RADOS block device library, the MDS is a server that receives requests and responds to them, so requests are funneled through these two functions. The tracepoints track the time, memory address, thread ID, and request type - these are used to for post-processing.

We trace the latencies from the MDS's perspective in an attempt to neutralize the effects of the client processing power and file system implementation and the system's network speed. Clients may have different hardware and different software. For example, CephFS has both a FUSE and kernel client and for clarity, our eexperiments are on the FUSE cient.

The post-processing script shown in Listing 2 extracts the latencies for all requests of a certain type. It loops over each event in the LTTng (line 3), looking for any creates. If

```
inFlight = {} latencies = []
for event in traces.events:
  if event["type"] == CEPH\_MDS\_OP\_CREATE:
    req = (event["pthread\_id"], event["addr"])
  if event.name == "mds:req\_enter":
    inFlight{req} = time
  elif event.name == "mds:req\_exit":
    try:
      latencies.append(time - inFlight{req}) del inF
    except KeyError:
      continue
  weights = numpy.repeat(1.0, window)/window
  avg =numpy.convolve(latencies, weights, 'valid')
```

Figure 2: The post processing script that extracts latencies and a moving average of the latencies for the file create request.

the event is the server receiving a create (line 6), the time stamp is saved using the thread ID and memory address of the request as a key. If the event is the server replying to a request (line 8), the latency is recorded by subtracting the current time from the time when the request arrived. Lines 14 and 15 calculate the moving average of the latencies.

5. RESULTS

Recall the goal of this work:

1. determine capacity of an MDS, including what constitutes "good progress" and what indicators constitute an MDS under strain
2. understand the CephFS metadata protocols, including the cost of migration and caching

Our experiments do not evaluate the performance of single directory, multiple client create workloads. The hashing scheme used in IndexFS [14] has shown to be an effective approach for strong scaling with create-heavy workloads. Instead, we focus on workloads which do not lend themselves to directory hashing with the intent of finding workloads where different namespace distribution techniques are necessary.

5.1 Scaling the number of files

First we look at the effect on `readdir` and `create` when scaling the number of files. This is like a strong scaling experiment in that we vary the amount of work per node but the difference is that we achieve this not by increasing the number of nodes for a fixed workload but instead increase the workload. A traditional strong scaling experiment does not help us determine the capacity of a single MDS.

The experiment varies the following parameters: inode cache (on/off), number of files, numbe of clients. We write n files into the CephFS mount using `mdtest`, then we do an `ls -alh -R /cephfs` to list all the files in the mountpoint. We then we repeat, adding n more files and recurse over all files in the mount. These files are not getting added to the same directory since we want to simulate a workload that serves users in different directories.

5.1.1 Throughput Analysis

The graph in FigureX (a) shows the MDS can sustain a throughput of 500 `creates`/second from 1 client. The spikes are probably from early replies but when buffers are exhausted, the throughput is restricted. FigureX (b) shows the `readdir` operations scale linearly when adding files. This indicates that hashing across an MDS cluster would improve performance by parallelizing metadata retrieval. This would be limited by the speed of the RPCs, since hashing would increase the amount of communication when a client does a `readdir`. These conclusions align with the ShardFS [21] work.

Conclusion: an MDS is has a sustained maximum create throughput and `readdir` scales with the number of files, which implies that hashing would improve performance.

5.1.2 Latency Analysis

The graph in FigureX (a) shows that the latency of `creates` is constant and much shorted than the latency of `readdir`. Figure (b) shows that `creates` affect CPU utilization more than `readdir`, yet we speculated in the previous section the `readdir` would benefit from a more spread and even distribution like hashing. This weakens one of the conclusions in the Mantle paper, that Fill and Spill does better for `creates` in an underloaded environment. Even if the MDS is underloaded, it would still benefit on a `readdir` from parallelization of directory listings.

Another conclusion can be drawn from the accompanying CPU utilization graphs – that good indicators for quantifying the effective load on an MDS must be reliant on both the request type distribrution and the CPU utilization. Again, this weakens the fill and spill balancer in the Mantle paper, which used 30% CPU utilization as a threashold for shedding load to its neighbor. A strong load metric would say something like x% CPU utilization if the workload is mostly `creates` and a y% CPU utilization the workload is made up of `readdir`, where y is lower than x because `creates` slow down with more CPU while `readdir` does not.

Conclusion: an MDS has vastly different latencies across file system operations and with different sized workloads; any balancers should take into account multiple factors when determing how loaded an MDS is.

5.2 Inode Cache Analysis

5.3 Compile Workload

Workload Characterization

What does it look like where won't hashing work?

5.4 Affect of kernel client

5.5 Open questions

Open questions

- why are the lookups and creates the same latencies??? (I only see one request #)
- what is the MDS doing during those lags???
- is there a queue size of 500 that throttles the number of in-flight requests (is this configurable? # of cached inodes??)

6. THE CEPHFS METADATA PROTOCOLS

This section is a deep dive on the CephFS source code. First, we describe the normal POSIX requests as viewed through the FUSE API. The implementation of these operations vary depending on the storage system. In CephFS, the `dispatch_client_request()` function has the switch statement for dispatching requests. A full description of these requests is in `fuse.h`. Briefly:

- `create()`: create and open a file
- `open()`: open a file
- `readdir()`: read directory (used by `ls -al`)
- `lookup()`: get the file attributes
  - translated to `getattr`
  - difference: invalid if the file depth is 0
  - difference: can return null dentry lease
- `getattr()`: get the file attributes; used in `stat()`
- NFS operations: `LOOKUPINO`, `LOOKUPPARENT`, `LOOKUPNAME`

Metadata fields: all metadata fields are protected by locks. By default, the fields are readable but require an exclusive write lock for updates. More complicated attributes (e.g., size, mtime, etc.) have state machines that keep track of a client's mode, which can be: single client, shared read, mixed read/write, or shared write. When dirfrags are distributed, a "scatter-gather" lock controls the `mtime`; this lock allows concurreent updates unless someone starts reading. If so, the capability is dropped.

Client operation: Since the prototype, CephFS has added 4 client capabilities. Now, the full list has: reads, read and update, cache reads, and read. These are managed by sessions.

Authority Pin: or "auth pin", is a reference counter on an object (*e.g.,* inodes, dentries, or directories) the prevents the subtree from being migrated. This is implemented with a "frozen" state machine. When a subtree is "freezing", no auth pins can be stuck to it and no metadata updates are allowed. The subtree transitions to "frozen" when all existing auth pins expire. Parents of freezing or frozen subtrees are auth pinned so that the root inode stays put during a migration. Operations, such as `rdlock_path_pin_ref()`, increment the auth pins with references. This function traverse the path, forwards if necessary, and tries to increment the reference on the auth pin while taking into account snapshots.

6.1 `create()`

This calls `open()` if the CREATE flag is omitted and `openc()` otherwise.

`openc()`: if the file exists, do a regular inode open, if not, create the file and open. First, it gets the file mode, which are the modes from `open/fcntl` and incluces things like `O_WRONLY`, `O_CREATE`, `O_EXCL`, etc. Then it does a path traverse by looking at each inode in the file path.

For each inode, it makes sure its inode, it gets or creates the dirfrag (checking for it remotely if possible), it checks if its frozen, then it does a lookup on the directory entry. If the dirfrag is remote, then the request gets delayed and the remote MDS will call the waiter callback (need to see where it gets put in). It uses this direntry to check if the client should wait on a duped direntry (*i.e.* xlocked). Then it tries to determine if the linked inode is missing (*e.g.* `ENOENT`) by checking for a miss on a null *and* readable dentry.

Possible forwards:

- path traversal

read, write, read write, e, create which can be write-only, read-only, or read-write.

6.2 `readdir()`
6.3 `setfilelock()`
6.4 `create()`
6.5 `mknod()`
6.6 `link()/symlink`
6.7 `mkdir()`
6.8 `close()`
6.9 `getattr()`

First, it prevents the path from being frozen and migrated by ticking the auth pin; if it can lock the object, it returns the inode and if it cannot, it returns. Second, it checks to see if the client has the EXCL capability. If it does, it does not read lock it because the `stat()` value will be valid (since there is only one client or the MDS readlocks and reads the value). It does this by trying to acquire the exclusive locks for the path, whether its a linke, auth, file, or xattr. Third, it stuffs the inode obtained from the first step into the metadata request and responds to the client.

6.10 `setattr()`

First, it gets the inode by trying to read lock the path, just like `getattr()`. Second, it checks to make sure that the path is not a snapshot or inode 0. Third, it tries to acquire the locks for mode, uid, gid, mtime, atime, size, and ctime. Then it to performs the operation on a projected inode, which could be truncating the file, changing its permissions/user/group, setting the timestamp. Finally, it logs the update in the journal, dirties the inode and its parents, and initiates a log flush if readers/writers are waiting.

7. RELATED WORK

7.1 Consensus with RPCs

In state machine replication (SMR), nodes store a copy of the log and use a consensus algorithm to keep the log up to data. Distributed consensus is difficult because processes operating on different copies of the log in parallel need to reach an agreement. In general, the most renowned family of distributed consensus algorithms is PAXOS [**?**]. These algorithms take streams of proposal updates and produce an agreement about the ordering events. Acceptors form a total-ordered cluster and learners replicate the service. Clients send mutations to the TO cluster, which orders the store commands, persists them, and returns the sequence of mutations to the learners. [10] lays out the disadvantages of this approach:

- wastes network/storage resources because it moves data: one solution is to deploy learners as the TO-cluster of acceptors but this approach (1) requires more learners and (2) has clients fight for entries.
- load distribution because of partitioned state: web-scale systems relax consistency but this creates workload hotspots.

Ceph isolates PAXOS as a service in monitor processes (MONs) [17]. Versions for the different services, which include authentication, logging, and MDS/MON/OSD/PG maps, are fed to the PAXOS instance, which gets consensus from the other PAXOS instances in the cluster. The combination of these services helps the MON understand the state of the cluster. This process is not used for consistency in the Ceph File System (CephFS), probably because of performance.

7.1.1 Consensus in Distributed File Systems

One approach for mediating access and maintaining consistency in file systems is to use RPCs. To make file systems scalable, many storage systems focus on reducing the number of RPCs per operation (*i.e.* RPC amplification), consistency (semantics of `readdir`), load balance (how much work each MDS is doing), scalability (depends on the workload), and availability. These RPCs are a way of forming a consensus between MDS nodes but minimizing the number of RPCs has positive effects on network and memory consumption.

High performance computing has unique requirements for file systems (e.g., fast creates) and well-defined workloads (e.g., workflows), which allow many storage systems to "lock" parts of the namespace to improve performance and scalability. IndexFS [] aggressively caches paths and permissions on the client by handing out leases; metadata may only be modified when all leases have expired. BatchFS [] goes a step further and assumes the application coordinates disparate accesses to the namespace, so the clients can do batch local operations and merge with a global namespace image lazily to avoid server synchronization. Similarily, DeltaFS [] eliminates RPC traffic using subtree snapshots for non-conflicting workloads and middleware for conflicting workloads; this reduces false sharing and server synchronization. MarFS [] eliminatse non-essential file system features and instead partition namespace by "project directories"; admins lock subtrees to so that each directory can have its own GPFS cluster. TwoTiers [] eliminates high-latencies by storing metadata in a flash tier; apps lock the namespace so that metadata can be.

IndexFS [14] aggressively caches pathnames and their permissions on the client servers. Modifications to metadata cached by clients is delayed until all client leases have expired. This reduces the RPC amplification to 1 when mutatating directory metadata (*e.g.,* `mkdir`, `chmod`, etc.) because clients are not querying MDSs for path traversals. The disadvantage of this approach is the high latency of mutation operations (reads to filenames). ShardFS [21] replicates metadata (specifically directory lookup state) across the MDS cluster, reducing the RPC amplification to 1 for file operations (e.g., `stat`, `chmod`, `chown`, etc.). Modifications to the directory lookup state are done with optimistic concurrecny control and fall back to retry if verification fails. The disadvantage of this appraoch is the high number of RPCs for maintaining directory metadata mutations (writes to directories). Systems that do both, like CephFS [19, 18] have significant complexity as it is difficult to determine when to replicate vs. cache, when/where to migrate load, and how much load to migrate [16]. It tries to replicate on reads and partition on writes but determining what the workload is doing (reading or writing) is difficult.

TODO - show root inode popularity - show CephFS replication - re-show CephFS partitioning

7.2 Consensus with logs

Consensus algorithms are implemented with RPCs among servers (in our case MDS nodes) but an alternative is to have each MDS node consult a shared log. CORFU [4] demonstrated that a cluster of flash devices can create a shared log that is fast enough to support hundreds of clients appending to the tail. The log enforces strong consistency by ensuring a globally-ordered sequence of updates which achieves the same goal of SMR. It does this without using SMR techniques:

- Time-slicing instead of partitioning. Parallelism is at the IO level instead of splitting data into object streams, which loses atomicity and load balancing across partitioning.
- Decoupling sequencing from I/O: throughput is bounded by the speed of the sequencer instead of a head node that sets up proposal pipelines

Tango [5] demonstrates the power and versatility of the shared log by layering objects on CORFU. These objects provide a consistent view of state across clients and replicas and the interface lets clients efficiently share different objects for services on the same log.

7.3 Achieving Locality

The LARD algorithm [12] sends proxy requests to the same back-end server until it is overloaded. This approach directly improves temporal locality and indirectly redirects requests based on their content, but does directly alter the balancing based on the request behaviors.

Algorithms that get locality

- Jiang: DULO: an effective buffer buffer cache management scheme to exploit both temporal and spatial locality

7.4 Achieving Locality

@inproceedingsbehzad:hpdc14, Address = Vancouver, BC, Canada, Author = Babak Behzad and Surendra Byna and Stefan M. Wild and Prabhat and Marc Snir, Booktitle = HPDC '14, Month = June 23-27, Title = Improving Parallel I/O Autotuning with Performance Modeling, Year = 2014

@inproceedingsbehzad:sc13, Address = Denver, CO, Author = Babak Behzad and Huong Vu and Thanh Luu and Joseph Huchette and Surendra Byna and Prabhat and Ruth Aydt and Quincey Koziol and Marc Snir, Booktitle = SC '13, Month = November 17-21, Title = Taming Parallel I/O Complexity with Auto-Tuning, Year = 2013

7.5 Identifying Locality

[20] finds temporal and spatial locality for blocks by calculating offset differences biased with time and by partitioning accesses into distinct sets. This approach lacks the semantic knowledge of what the application is actually doing and is only applicable for data blocks, not for metadata.

Some file server systems try to predict which files will be accessed next:

- Amer: File access prediction with adjustable accuracy
- Doraimani: File grouping for scientific data management: lessons from experimenting with real traces
- Kroeger: Predicting file system actions from prior events
- Kroeger: design and implementation of a predictive file prefetching algorithm
- Sivathanu: semantically-smart disk systems
- Arpacci-Dusseau: semantically-smart disk systems: past, present, and future

Locality in file system workloads

- Amer: Aggregating caches: A mechanisms for implicit file prefetching
- Ari: ACME: adaptive caching using multiple experts
- Doraimani: File grouping for scientific data management: lessons from experimenting with real traces
- Li: C-miner: mining block correlations in storage systems

Figuring out what the application is doing

- Yadwadkar: discovery of application workloads from network file traces

To optimize latency, per-request behaviors have been studied extensively, especially to find which requests are slow because of synchronous I/O for cache coherence and consistency. Speculator [11] identifies synchronous I/O for cache coeherence/consistency as a source of latency, so it masks multiple round trip delays, like listing directories, by allowing computation on stale data while waiting for server's response to a previous request.

8. CONCLUSION

References

[1] C. L. Abad, H. Luu, Y. Lu, and R. Campbell. Metadata Workloads for Testing Big Storage Systems. Technical report, Citeseer, 2012.

[2] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell. 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, pages 125–132, 2012.

[3] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel I/O and the Metadata Wall. In *Proceedings of the 6th Workshop on Parallel Data Storage*, PDSW'11, 2011.

[4] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, NSDI '12. USENIX, April 2012.

[5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures over a Shared Log. In *24th ACM Symposium on Operating Systems Principles*, SOSP '13, November 2013.

[6] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-memory Multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, 2009.

[7] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, 2013.

[8] L. Georgiadis, R. Guérin, V. Peris, and K. N. Sivarajan. Efficient Network QoS Provisioning Based on Per Node Traffic S]haping. volume 4 of *Trans. Networking '96*, Aug. 1996.

[9] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, 2008.

[10] D. Malkhi, M. Balakrishnan, J. Davis, V. Prabhakaran, and T. Wobber. From Paxos to CORFU: A Flash-Speed Shared Log. *ACM SIGOPS Operating Systems Reviews*, 2012.

[11] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, 2005.

[12] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, 1998.

[13] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-virtualized Devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC'07, 2007.

[14] K. Ren, Q. Zheng, S. Patil, and G. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the 20th ACM/IEEE Conference on Supercomputing*, SC '14, 2014.

[15] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 4–4, 2000.

[16] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the 21st ACM/IEEE Conference on Supercomputing*, SC '15, 2015.

[17] C. Team. Ceph's new monitor changes.

[18] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'06, 2006.

[19] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing*, SC'04, 2004.

[20] A. Wildani, E. L. Miller, and L. Ward. Efficiently Identifying Working Sets in Block I/O Streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, 2011.

[21] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. In *Proceedings of the 6th ACM Symposium on Cloud Computing*.

[22] Q. Zhang, L. Cheng, and R. Boutaba. Cloud Computing: State-of-the-art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

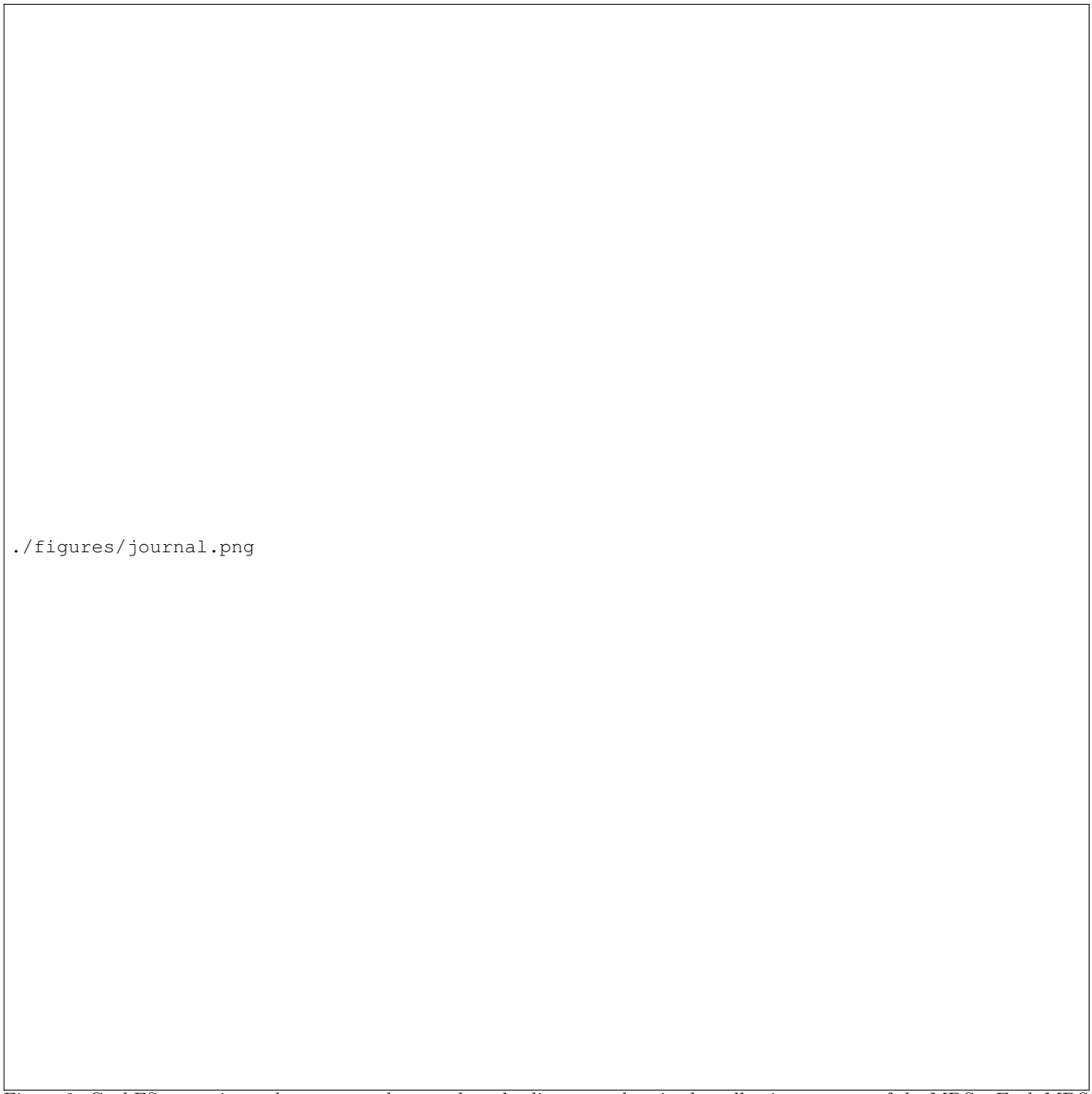

Figure 3: CephFS uses a journal to stage updates and tracks dirty metadata in the collective memory of the MDSs. Each MDS maintains its own journal, which is broken up into 4MB segments. These segments are pushed into RADOS and deleted when that particular segment is trimmed from the end of the log.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