Logical Synchronous Replication in the Tintri VMstore File System

Gideon Glass, Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo, *Tintri, Inc*

https://www.usenix.org/conference/fast18/presentation/glass

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

Logical Synchronous Replication in the Tintri VMstore File System

Gideon Glass [*], Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo [†]

Tintri, Inc.

{gxglass, arjun91, dattatraya, abhinandh, sumedhsakdeo} @gmail.com

Abstract

A standard feature of enterprise data storage systems is synchronous replication: updates received from clients by one storage system are replicated to a remote storage system and are only acknowledged to clients after having been stored persistently on both storage systems. Traditionally these replication schemes require configuration on a coarse granularity, e.g. on a LUN, filesystem volume, or whole-system basis. In contrast to this, we present a new architecture which operates on a fine granularity—individual files and directories. To implement this, we use a combination of novel per-file capabilities and existing techniques to solve the following problems: tracking parallel writes in flight on independent storage systems; replicating arbitrary filesystem operations; efficiently resynchronizing after a disconnect; and verifying the integrity of replicated data between two storage systems.

1 Introduction

Synchronous replication in enterprise data storage systems allows customers to situate redundant storage systems at campus or metropolitan distances. This provides continuous availability in the event of hardware failures, power or cooling failures, or other disasters. One of the storage systems acts as *primary*, responsible for accepting IO from clients. The peer storage system acts as *secondary* and is responsible for accepting replicated IO from the primary system. In the common case when both storage systems are in a synced state, a cluster failover involving a role reversal between the primary and secondary can occur without loss of data.

To handle temporary outages (e.g. minor network glitches, non-disruptive software upgrades of either storage system), the primary and secondary coordinate to bring both the storage systems into a consistent state, without timing out client operations. To handle outages of arbitrary duration (e.g. power failure, lengthy network disruptions, or maintenance related activities) where the secondary is offline or unreachable by the primary, some mechanism for efficient resynchronization is required—once the secondary comes back on line, the primary system should be able to generate and replicate the delta changes that occurred while the secondary was offline.

A related feature that is often supported is *transparent failover*. In the event of a failure of the primary storage system, it enables the secondary system to take over in a way that does not disrupt client applications—with no loss of data and no loss of availability. This failover can be manually driven or automated. Figure 1 illustrates a typical deployment.

We have addressed the problems of synchronous replication and transparent failover by designing, implementing, and deploying a system that is flexible, efficient, and intuitive to use. Our system supports *logical* synchronous replication—the ability to only replicate a portion of the file system namespace, specified as a top-level directory. Designing this system involved solving several sub-problems:

- maximizing the overlap of write execution on the Primary and Secondary storage systems to minimize latency overhead of mirroring; a novel filesystem metadata mechanism is used for this purpose (Section 4).
- replicating complex, arbitrary filesystem operations; a two-phase commit protocol is used for this (Section 5).
- efficiently resynchronizing the two storage systems after extended disconnects (Section 6).

Finally, we have implemented a novel distributed integrity check that allows us to verify periodically or on demand that Primary and Secondary contain identical data (Section 7).

[*]Currently at Google.
[†]Currently at Lyft.

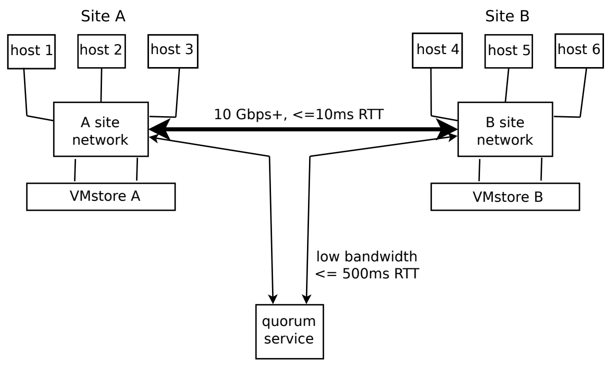

Figure 1: Generic synchronous replication deployment topology with multiple client hosts, a primary storage system, a secondary storage system, and a quorum service to facilitate automatic failover.

2 VMstore Background

In this section we discuss specific characteristics of our workloads and why they motivate logical replication, and our technology base.

2.1 Workload

As its name suggests, VMstore is a specialized storage system designed for virtualization workloads. By default the system exposes a single NFS (or SMB) mount point. On NAS storage, virtual machines (VMs) are typically stored in self-contained directories named after the VM. VM directories typically contain on the order of ten to twenty files, which are either small (e.g. text configuration files) or very large—virtual disk image files corresponding to the virtual disks associated with the VM. VMstore does not support general purpose NFS workloads. As a result, the system can be substantially simplified in one dimension: the number of files it is required to support. VMstore models vary, but support on the order of 100,000 files per system. The median usage is far below these limits, with most systems having under 10,000 files. This simplification affects our replication-related design choices and will be discussed in further sections. A final important aspect of our workload is that file writes comprise in excess of 99.9% of mutation events; file and directory creation, deletion, renames, etc are not uncommon but are generally associated with VM provisioning activities, not with ongoing application workloads running within VM's themselves.

Another aspect of the system is the desire to maximize simplicity for the user. A high priority is placed on making the system usable by IT generalists and virtualization administrators. As a result, the system does not expose traditional storage abstractions such as RAID groups, filesystem volumes, or LUNs to users. Consequently, apart from conceivably replicating the entire storage array, there is no obvious storage abstraction on which to expose synchronous replication for configuration purposes. [1]

From our prior experience with asynchronous replication, we know that many customers choose not to replicate significant portions of their workloads. As an example, a development/test customer may choose to replicate virtual machines housing important data (source code control system, bug database, etc) but not virtual machines running automated continuous integration test workloads. Customers generally are very aware of the relative differences in value of the data inside the different types of virtual machines in their environments. Commonly, a minority of VM's are actually configured for replication. This varies on a continuum, of course, but in rough numbers, it is common for, say, 25% of virtual machines to be configured for replication.

Based on these requirements—that enterprise customers be able to simply express replication policy on a subset of the files and their systems and efficiently replicate that data across geographically disparate locations, we introduce logical synchronous replication as a mechanism to continuously replicate fine-grained subsets of file system state.

2.2 Filesystem Architecture

The VMstore file system is a purpose-built storage system implementing all layers of the storage stack from RAID to file access protocols (NFS and SMB) in a user level filesystem process. Features expected of an enterprise storage system, such as durable writes, automatic crash recovery (from NVRAM), compression, deduplication, snapshots, writable clones, and asynchronous replication are present.

The VMstore file system is a log-structured file system [9]. As such, data is never overwritten in place. Additionally, it implements a transaction system allowing arbitrarily complex metadata updates with ACID semantics. (This system is based on Stasis [10] but with significant local enhancements to integrate with the VMstore log-structured storage system.) We use this facility to construct novel mechanisms to track file writes in progress and to implement per-file content checksums; these are discussed in further sections.

For high availability (HA) within a given system, a VMstore system contains two controllers (x86 servers), each with its own NVRAM and set of network interfaces. The controllers access shared storage devices (SSD and/or HDD, depending on model). High availability is implemented in an active/passive model. For pur-

[1] Per-VM synchronous replication was an option we considered. However, this does not work well with transparent failover, discussed in the next section, because it would require a customer-assigned Cluster IP address on a per-VM basis.

poses of synchronous replication, we do not assume local high availability; replication simply runs on whichever local controller is Active.

3 Replication System Overview

This section introduces our synchronous replication system, describes its configuration model for users, discusses error situations and how we handle them, and outlines how we support transparent failover.

3.1 Introduction

A requirement from our customer base is to support not just synchronous replication but also client-transparent failover across storage systems; customers do not want to perform any reconfiguration in the event of a total failure of one storage system, or a data center outage. To distinguish between local-system HA-related failover and failover across VMstores in a synchronous replication relationship, we refer to the former as *local failover* or *HA failover* and to the latter as *cluster failover*. By *transparent* we mean specifically that clients are not aware of failovers occurring, except for brief periods of disconnection. In particular, no reconfiguration of client hosts is needed—failover, either local failover or cluster failover—requires no manual intervention.

In our environment, virtualized guest operating systems (Linux and Windows) use internal I/O timeouts of 60 seconds or higher. Within VMstore, we require internal failovers to complete within 30 seconds. This gives the hypervisor clients enough time to reconnect and reissue I/Os so that guest operating systems do not time out.

We introduce the notion of a *mirrored datastore*—essentially an IP address and a mount point—as the basis for configuring synchronous replication. We require each mirrored datastore to have a dedicated *Cluster IP address* associated with it. This IP address is mounted by clients for IO operations and fails over between the two storage systems, similar to how local system Data IP addresses are failed over during local HA failover. To make this concrete, Table 1 provides an example showing a mix of synchronously replicated and unreplicated datastores.

Cluster IP	Mount Point	Replicated?
10.200.200.5	/tintri	No
10.300.100.60	/tintri/alpha	Yes
10.300.100.61	/tintri/beta	Yes

Table 1: Example client view of datastores on a VMstore.

The network requirements for Cluster IP addresses are simple: client hosts must be able to reach this address regardless of which VMstore happens to be the Primary at any point in time.[2]

In our system, synchronous replication is applied recursively to all content under the mirrored datastore. In practice this means that a virtual machine which the customer desires to replicate should simply be placed within a mirrored datastore top-level directory (e.g. /tintri/alpha/ImportantMachine), and one which is not desired to be synchronously replicated should be placed in the top-level directory (e.g. /tintri/LessImportantMachine). Note that the contents of *alpha* and *beta* from Table 1 may be replicated to different peer VMstores, or to the same peer VMstore; they are configured independently and are managed independently within VMstore. The cluster IP addresses associated with these datastores are also independently managed and are exported by whichever VMstore system is the replication primary. To simplify error checking and to avoid problems with conflicting policies, we allow only top-level subdirectories to be replicated. The root directory (exported as /tintri) cannot be configured for replication. Customers wanting to replicate their entire workload can simply place all VM's within one or more mirrored datastore subdirectories.

3.2 Failure Model and Operational Requirements

The failure model we assume is as follows. Machines may fail at any time, e.g. because of software crashes or power failures. Datacenters or the network may fail any time and for extended durations. The network may corrupt packets (and go undetected by the TCP checksum); we detect this via strong checksums in our messaging protocol and handle it as we would handle a transient TCP connection loss (i.e. by simply reconnecting). We assume peers are not malicious/Byzantine and VMstores always authenticate each other as the first step in each connection. Finally, individual devices (SSDs or HDDs) may fail, but lower levels of the filesystem insulate replication from having to handle device errors.

We now discuss the operational requirements of our system. Availability is important but consistency (data integrity) takes precedence over availability when there is a tradeoff. Availability, in turn, takes precedence over absolute redundancy. As discussed in detail in Section 3.3, this is done by taking the Secondary out of sync when necessary, and then resychronizing it when it comes back on line. An optional mode, which we have not implemented, would be for the Primary to *fail* I/Os which it could not replicate due to the Secondary (or network) being down. This might conceivably be useful for customers who prioritize absolute redundancy over availability.

[2]For cross-site replication, this requires the use of a stretched layer 2 or layer 3 network. Customers who deploy synchronous replication typically already have this in place.

3.3 Replication States

Figure 2 depicts a slightly simplified view of the state of a given mirrored datastore within a given VMstore system. Each VMstore maintains its state for a given datastore separately, so for a given mirrored datastore there are really two instances of this state machine operating in a loosely coupled manner.

The following invariants relate to the state machine and affect allowable state changes.

1. Only one VMstore may be Primary at any given time. Whichever system is Primary owns the Cluster IP address and advertises it on the network.
2. Upon initial configuration of synchronous replication, the Primary system may have a significant amount of data in the configured subdirectory. An initialization process is required to bring the Secondary into sync. The initialization process is essentially a special case of resynchronization (discussed in Section 6) in which the Secondary happens to be empty at the start of the process. This process may take a significant amount of time. The top row of the state machine contains states related to initialization/resync. The datastore is not in sync in these states.
3. During initialization and resync, the Secondary does not have a complete copy of data. Cluster failover is not possible until initialization/resync completes.
4. The normal state of operation is that both systems are connected and in sync (the Primary is in state 3; the Secondary is in state 7). In this state, client operations are fully replicated and are persisted to NVRAM on both systems prior to being acknowledged to clients (i.e., normal synchronous replication semantics are in effect).
5. Automatic cluster failover (of Secondary to Primary) requires a quorum—two of three systems. An external quorum service having storage independent of the two VMstores is provided for this purpose. Automatic cluster failover is initiated by an in-sync Secondary after a period of time (30 seconds) for which it has not heard from the nominal Primary, provided the Secondary can communicate with the quorum service.
6. Conversely, to handle a Secondary which is inaccessible, a Primary must undergo a transition to mark the datastore as being "out of sync" (not shown in Figure 2). There is a subtlety here: the Secondary, meanwhile, may have initiated a cluster failover in conjunction with the quorum service. As a result, a Primary must coordinate with the quorum service in order to mark a Secondary as being out of sync, and must be prepared for this to fail (i.e. if the Secondary already took over). If that fails, the former Primary must relinquish ownership of the datastore and must drop in-flight I/O requests and not return errors to clients.[3]

A design choice we enforce is that operations which can succeed on the Primary but which fail on the Secondary result in the Primary taking the Secondary out of sync immediately (a transition from state 3 $\rightarrow$ state 4 $\rightarrow$ state 2). The most likely example of this type would be the Secondary being out of space; less commonly, the Secondary might encounter some other limit (e.g., number of files, number of snapshots) that might prevent an operation that otherwise can succeed on the Primary. Again this policy reflects the choice to prioritize availability over absolute redundancy in some conditions. (To recover from this condition, the Primary will periodically reconnect to the Secondary and attempt to resync it; this will succeed if the user has freed up capacity or otherwise addressed the constraint that earlier had caused failure.)

The state machine also reflects a practical engineering consideration: taking the Secondary out of sync then later resyncing it has a non-trivial minimum cost, and we seek to avoid taking a Secondary out of sync if possible. This corresponds to two practical scenarios: brief network outages, and local HA failovers due to software crashes and restarts.

In practice this means that we attempt to recover from brief disconnects (up to approximately 30 seconds) by pausing client acknowledgments at the Primary, and attempting to reconnect to the Secondary in the background. If the reconnect attempt succeeds within the timeout, the Primary will resend buffered, unacknowledged updates (and only then ack the client), and the system will stay in Sync. The states in the second row of Figure 2 reflect these activities. "Catching Up" in the state descriptions refers to replicating (possibly re-replicating) buffered updates from the Primary; Sections 4 and 5 discuss this in detail.

The protocol between the VMstores and the quorum service solves a standard distributed consensus problem and will not be discussed in detail. The quorum service is provided by a standalone software application which can be installed by the customer in a virtual machine either on premises or in the public cloud; the main operational requirements are that the storage for the quorum service must be independent from the VMstores for which it is arbitrating, and network connectivity to the quorum service should be good.

[3]In practice, a soon-to-be-former Primary that becomes isolated on the network must relinquish ownership of the mirrored datastore, and must give up the Cluster IP, *before* the Secondary takes over, to prevent both systems from attempting to advertise the Cluster IP on the network. The Primary transitions out of the Primary/Synced/Connected state using a smaller timeout (e.g. 25 seconds) than the timeout driving the Secondary's attempt to initiate cluster failover. These transitions are not shown on the diagram for brevity.

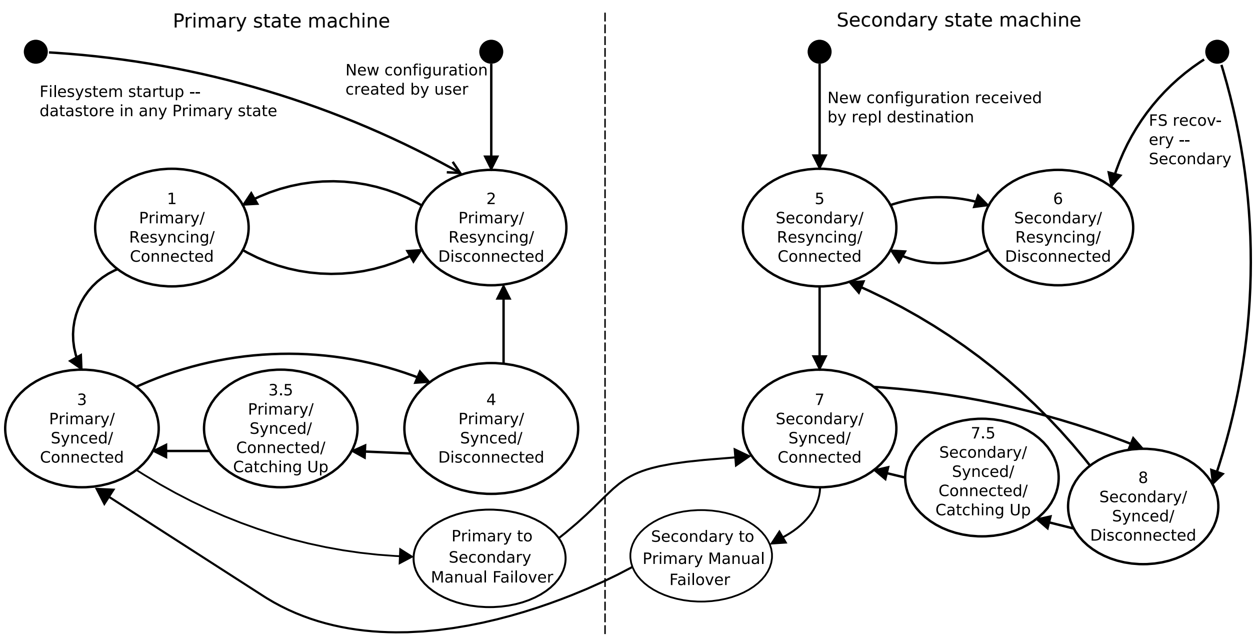
Primary state machine
Secondary state machine
Filesystem startup -- datastore in any Primary state
New configuration created by user
New configuration received by repl destination
FS recovery -- Secondary
1 Primary/ Resyncing/ Connected
2 Primary/ Resyncing/ Disconnected
3 Primary/ Synced/ Connected
3.5 Primary/ Synced/ Connected/ Catching Up
4 Primary/ Synced/ Disconnected
5 Secondary/ Resyncing/ Connected
6 Secondary/ Resyncing/ Disconnected
7 Secondary/ Synced/ Connected
7.5 Secondary/ Synced/ Connected/ Catching Up
8 Secondary/ Synced/ Disconnected
Primary to Secondary Manual Failover
Secondary to Primary Manual Failover

Figure 2: Replication System State Machine. This is shown from the point of view of a mirrored datastore in a single VMstore. The other VMstore will be in one of the other states. For brevity, states relating to automatic cluster failover are not shown.

3.4 Client Transparency

Implementing transparent cluster failover requires that clients of a failed storage system be able to reconnect to the Cluster IP address and see a view of the world exactly consistent with what they previously saw. This includes file content, user-visible metadata (path names, file attributes, etc) and client system-visible metadata (e.g., NFS file handles). To implement this, we assign internal file identifiers and NFS file handles as follows. The Primary makes all such assignments without having to coordinate with the Secondary—the fact that it is Primary gives it the right to assign these values.[4]

- **File Global Identification**. We identify each file in every mirrored datastore with a globally unique FileId. This consists of a 128-bit datastore-specific UUID and a datastore-relative 64-bit monotonically increasing counter. The global FileID value is used extensively within the replication system, as the values are the same on both systems.
- **NFS File Handles** must be stable across cluster failovers. File handles are based on the global FileId. Because only the Primary system assigns FileId's, and because FileId's are unique within a datastore (due to containing the datastore UUID as a prefix), we avoid any need for granular synchronization to negotiate assignment of these values.
- Operation sequencing: **Operation Sequence Numbers** (OSN's) are used to globally order all operations within a mirrored datastore. An OSN consists of a 64-bit cluster generation number, incremented whenever a cluster failover occurs, and a 64-bit local sequence number, assigned and incremented on the cluster Primary for every new incoming operation. Replicated operations are tagged with OSN's for bookkeeping purposes.

During a cluster failover in which a Primary loses ownership of a datastore, the system must be able to identify stale operations and drop them, rather than execute them. A full description of the solution to this problem is beyond the scope of this paper, but to summarize, we tag all requests entering the system with *tokens* which contain among other things the cluster generation number. Requests tagged with generation numbers older than the current generation number must be dropped.

4 Data Path: Writes

Minimizing write latency is an important consideration in primary storage systems. Users expect write latencies on unreplicated systems to be at most small numbers of milliseconds. Replication distances are typically small. Industry-wide guidance typically calls for not more than

[4]A side effect of this scheme is that at initialization, if the source datastore directory contains existing content, that content must be *re-assigned* new file identifiers—and consequently, file handles—to avoid possible conflicts with unrelated existing content on the Secondary. We require that this content be off-line while this occurs, as clients will encounter stale file handles if accessing content during this process. In practice, customers almost always configure synchronous replication on empty source directories and migrate data into the mirrored datastore by using live storage migration features (e.g., Storage vMotion) in the virtualization system.

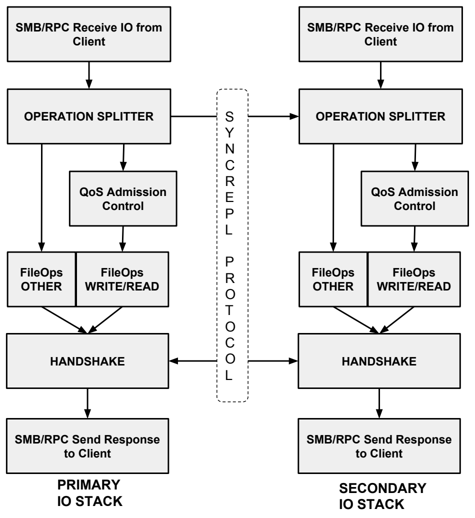

Figure 3: IO stack with operation splitting and handshake for synchronous replication between a pair of storage systems.

10ms RTT between storage systems; customers commonly deploy with 1-2ms RTT. In order to maximize overlap between write execution on the Primary, replication of writes to the Secondary, and write execution on the Secondary, we perform replication processing as early in the pipeline as possible.

Figure 3 depicts logical steps in write processing inside the VMstore file system. The top box indicates protocol processing for Microsoft SMB [1] or RPC/XDR [11, 12] processing for NFS. The module labeled "operation splitter" refers to front-end synchronous replication processing—it splits write operations for local processing and enqueues them for transmission to the Secondary. The operation can then traverse various stages of the pipeline in parallel on both the storage systems. In case one storage system is under contention, the operation would experience queueing delays only on that individual storage system.

The modules labeled "FileOps Write/Read", and "FileOps Other" correspond to existing IO stack processing for reads, writes, and all other operations (file creations, deletions, truncations, directory operations, etc). Once operations have executed locally within the Primary VMstore, flow returns to synchronous replication in the box labeled "Handshake Module". On the Primary, the handshake module will hold onto the write operation until the Secondary sends an acknowledgement for this write or until such time as synchronous replication allows the operation to be acknowledged to the protocol client, whichever is sooner. The latter corresponds to when the response from the Secondary takes longer than the allowed time forcing the Primary to go out of sync (as described in Section 3.3). On the Secondary, the handshake module will hold onto the write operation until the Primary has acknowledged completing the write to the Secondary. This step is required because the Secondary keeps track of writes that are still pending execution on the Primary and the acknowledgement from the Primary is used to free up metadata for the write on the Secondary (discussed below).

4.1 File Locks

The desire to achieve maximum parallelism within the system must be balanced against concerns for correctness. Here are some cases that illustrate these considerations:

1. Suppose a file is created and then is immediately written. What happens on the Secondary if, because of queueing at different points in processing within the system, the write operation is able to be processed on the Secondary before the file create operation completes its processing? The Secondary may have allowed the Primary to acknowledge the file creation to the client before the Secondary has actually finished *processing* the file creation—the operation may have been intent-logged only (this is discussed in Section 5).
2. What happens if a client issues a write to a given file, at a given [Offset, Byte Count], and then issues a separate, overlapping write before the first write has acknowledged? How do we ensure that these writes are processed in the same order on the two systems?

In designing our replication system, an important consideration was to avoid modifying a lot of the existing file system logic as far as possible. In VMstore, metadata operations (e.g. file creates) are executed by threads in a particular thread pool. Write operations are executed separately by a software pipeline involving a different set of threads.

To address these issues, the operation splitter module (Figure 3) imposes two kinds of file locks: *per-file operation locks* and *range locks*. The per-file operation locks are acquired by reads, writes, and metadata operations prior to their execution. Reads and writes acquire operation locks in a shared mode, whereas metadata operations like truncate, file create, rename, etc acquire them in an exclusive mode. This causes metadata operations to be executed in isolation on both the storage systems and solves case 1 above. To address case 2, we introduce range locks maintained on a tuple consisting of $<$ FileID, StartOffset, RequestSize $>$. Write and read operations acquire locks using the respective offset and byte count values specified in the operation. This allows reads and writes to non-overlapping byte ranges to execute in parallel. In practice lock contention is not a problem because clients do not issue update patterns that inherently race. This is because guest operating systems are generally careful to avoid issuing multiple outstanding writes to overlapping ranges of disk blocks.

4.2 Metadata Support and Distributed Recovery of Writes

As noted earlier, one of the challenges of any synchronous replication system is to keep track of all writes in progress. We leverage a combination of NVRAM and our flexible metadata transactional system for this purpose. For each synchronous replication datastore, we maintain a persistent hash table that contains one entry per in-flight write. Entry tuples are <FileId, OSN, byte offset, length> with OSN being the key. We refer to these tuples as *PAW Entries* ("Primary Acknowledgement Waiting") on the Primary. On the Secondary, the same set of tuples are maintained for writes which have been received and executed and we refer to them as *SAW entries* ("Secondary Acknowledgement Waiting"). To summarize, the presence of a PAW entry means that the Primary is waiting for an acknowledgement from the Secondary and the presence of a SAW entry means that the Secondary is waiting for an acknowledgement from the Primary.

PAW/SAW metadata is associated with each write operation as it proceeds through various write processing stages within VMstore. As part of the metadata transaction which updates file metadata to point to a newly-written block, the transaction also commits a PAW entry identifying the block in question. This is done both on the Primary and the Secondary. Identifying all blocks which may be dirty on both sides requires iterating over all PAW/SAW records in the mirrored datastore. The cost of this is proportional to the number of in-flight IOs in the system, which is bounded. If the system undergoes a local HA failover before the transaction commits and persists, the PAW/SAW information can be retrieved from NVRAM along with the data and other information about the write.

The ***PAW/SAW update sequence*** across the two systems is as follows:

1. When processing a write, the Primary will create a local PAW entry;
2. After receiving the write, the Secondary will have stabilized the write to NVRAM and will subsequently create a SAW entry for itself; the Secondary then acks the write to the Primary;
3. At this point (and after its local NVRAM update is finished), the Handshake Module on the Primary acks the write to the client;
4. Simultaneously, the Primary acks the Secondary's ack and releases its PAW entry;
5. Upon receipt of the second ack, the Secondary can free its SAW entry.

Note that the Secondary commits writes to NVRAM and SSD storage independent of the Primary. As a result, scenarios like the following are possible:

1. Primary receives writes W1 and W2 (could be to the same or different files).
2. These writes are mirrored to Secondary and are enqueued for local processing on Primary.
3. Primary writes W2 persistently. However, before writing W1 to NVRAM, the Primary crashes and performs a local HA failover.
4. Secondary writes W1 persistently. However, before writing W2 to NVRAM, the Secondary crashes and also performs a local HA failover.
5. The Primary and Secondary complete local HA failovers independently.
6. After recovering, the Primary is able to connect to the Secondary and must now reconcile W1 and W2.

To handle this situation and really any situation where a Primary and Secondary have become disconnected (and one or both may have restarted), after every reconnect we perform what we call *distributed recovery*. This happens regardless of whether the systems are in-sync or not. To handle in-progress unacknowledged writes, both sides iterate over all the PAW/SAW entries in the datatstore. The Secondary sends its set to the Primary which merges it with its own. Then, the Primary simply reads out its copy of data for all blocks involved and sends the data to the Secondary to rewrite the relevant blocks. This ensures that the datastore contents are identical upon the completion of distributed recovery (for the in-sync case). Once this completes, new writes are allowed into the system on the Primary.

In the scenario described above, W2 is persisted on Secondary for the first time as part of distributed recovery. With respect to W1, the data on Secondary undergoes a rollback to the contents as dictated by the Primary. This is correct because neither W1 nor W2 was acknowledged to the client prior to the sequence of crashes.[5]

At some point it is necessary to remove PAW/SAW entries to bound the work involved in distributed recovery. If the write has persisted on the Secondary, and the Primary has an acknowledgement of that, the corresponding PAW entry is deleted on the Primary. Similarly, if the write has persisted on the Primary, and the Secondary has an acknowledgement for that, the corresponding SAW entry is deleted on the Secondary. PAW and SAW entry deletions correspond to Steps 4 and 5 respectively in the PAW/SAW update sequence described above.

[5]Presumably the client will reconnect per its normal logic and reissue both writes to the Primary. However, if the client also crashes—which is fine—then both the Primary and Secondary end up with only write W2 written persistently. This is also fine: there is no guarantee about whether the storage systems stabilized an unacknowledged write, and there is no ordering guarantee between simultaneously executing unacknowledged writes.

5 General Filesystem Operations

We use the term *metadata operations* (or just *metadata ops*) to refer to all filesystem modifications other than file writes. Many of these operations are familiar POSIX/NFSv3 operations: file creation and deletion, directory creation and deletion, rename, setattr, link creation, and so on. Additionally, we implement several proprietary operations. A full description of these operations is beyond the scope of this paper, but to summarize, the operations are space reservation (in which the system attempts to reserve physical capacity for the full logical size of a given file); file-level snapshot creation and deletion; and file-level clone creation.[6]

Metadata ops differ from writes in several important ways. While they are not infrequent (they may occur tens to hundreds of times per second, in active provisioning workloads), they are much less frequent than writes, giving us more implementation flexibility. Second, whereas writes can be undone simply by reading out data from the Primary and overwriting whatever data may exist on the Secondary (Section 4.2), there is no equivalent mechanism available to undo metadata operations. Thus, a more general mechanism is required to track in-flight metadata ops.

5.1 Operation Logging and States

We implement a scheme similar to two-phase commit to ensure that both systems track metadata ops and agree that they can be executed prior to executing them. Prior to being executed, metadata operations along with their respective OSN's, operation-specific arguments, etc., are *intent logged* on both sides. Physically, the log is simply a space-reserved file in a hidden, unreplicated portion of the file system. One intent log is maintained for each mirrored datastore. Log updates are efficient: we utilize the existing file write path which provides low-latency stable writes via NVRAM. The log size is not large because the log can be logically truncated regularly with no impact on performance. Note that the log is not used when the datastore is out of sync, so there are no limitations arising from log storage capacity.

Figure 4 depicts the general flow for metadata operations. To summarize, both sides log each operation along with an operation state:

- PENDING
- COMMITTED

[6]VMstore implements VM level snapshots as point-in-time snapshots over the set of files comprising the physical embodiment of the virtual machine: various metadata files and the virtual disk files, and possibly files capturing dynamic state, e.g. memory and swap. Within VMstore, a VM level snapshot consists of a set of file-level snapshots, taken atomically, and a certain amount of snapshot-wide metadata. VM level clones are implemented by instantiating a set of file-level clones, writable files which reference an underlying base snapshot in a read-only manner.

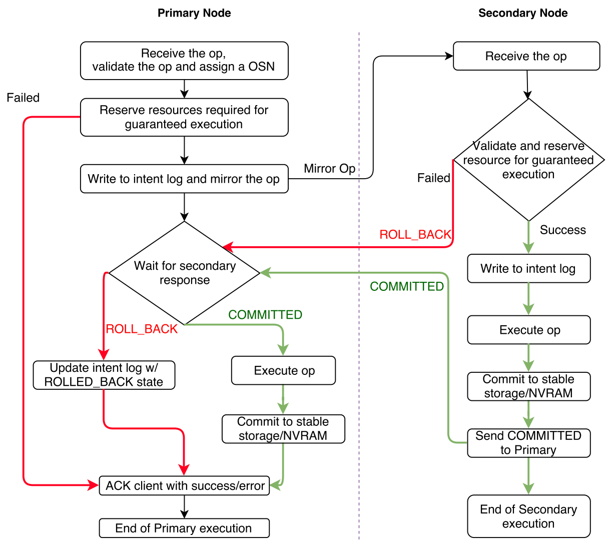
Primary Node
Secondary Node
Receive the op, validate the op and assign a OSN
Failed
Reserve resources required for guaranteed execution
Write to intent log and mirror the op
Mirror Op
Wait for secondary response
ROLL_BACK
COMMITTED
Update intent log w/ ROLLED_BACK state
Execute op
Commit to stable storage/NVRAM
ACK client with success/error
End of Primary execution
Receive the op
Validate and reserve resource for guaranteed execution
Failed
ROLL_BACK
Success
Write to intent log
COMMITTED
Execute op
Commit to stable storage/NVRAM
Send COMMITTED to Primary
End of Secondary execution

Figure 4: Metadata operation execution scheme that implements a two-phase commit protocol.

- ROLLED_BACK

Either side may decide that an operation may not succeed, for a variety of operation-specific reasons. For example, space reservation may fail on the Secondary but not on the Primary. In general, the protocol gives the Secondary the opportunity to determine if an operation should fail.

The first step in processing is for the Primary to validate the request (arguments are valid, resources are available, etc). If this fails, an immediate failure is returned to the client. If this succeeds, the operation is logged in the PENDING state and the operation is replicated to the Secondary. This Secondary then decides whether the operation can succeed. For operations which the Secondary cannot execute, it simply sends a ROLLED_BACK reply to the Primary. For successful operations, the Secondary first logs them in the COMMITTED state, then executes them, then sends a COMMIT reply back to the Primary. At this point the Primary executes the operation.

As part of executing metadata operations, the existing code paths all utilize the VMstore transaction mechanism. We augment these code paths to tag each file with the OSN of each completed operation for reasons discussed in the next section. Finally, when both sides finish executing the operation, file locks are released, and the Primary is allowed to acknowledge the operation to the client.

5.2 Distributed Recovery For Metadata Operations

Crash recovery for metadata operations must handle the same sort of considerations that were discussed above for write operations: messages may be lost; local HA failovers may occur at any time; etc. To recover in-flight metadata operations, we adopt an approach conceptually similar to what we used for in-flight write operations. We consolidate intent log entries on both sides by finding all entries in the COMMITTED state in both systems' intent logs. Note that, as it is implemented currently, the Secondary always commits a given entry first: it does this before sending a reply to the Primary that, upon receipt, allows the Primary to commit the operation. Thus, distributed recovery for metadata ops involves scanning the live portion of the intent log on the Primary for committed operations and simply sending them all to the Secondary. By definition, every log entry in the intent log that is in the COMMITTED state needs to be reapplied if it has not already been applied. Both the Primary and Secondary also take care of this during system start up.

In general, metadata operations are not idempotent. Some are, but we handle the general case and ensure that all metadata operations are executed exactly once. Log replay handles this simply and efficiently by comparing each operation's OSN with the last-executed OSN on the respective files. Operations which have already been done are simply ignored. The same OSN based comparison is also used in the replay of write operations where these writes are just discarded if the corresponding files have subsequently been deleted.

Because of the need to correctly interleave metadata operations with file writes, the relationship between metadata distributed recovery and file write distributed recovery is simple: metadata distributed recovery is done first, then file writes are recovered. This ensures that files are created prior to writes being recovered.

6 Data Path: Resync

Efficient resynchronization is important in any synchronous replication scheme, because the alternative is basically untenable: rereplicate the entire copy of data from the Primary, possibly tens to hundreds of terabytes. This section describes how we perform resynchronization using file-level snapshots.

As discussed in Section 3, our threshold for extended disconnects is 30 seconds, after which one of two things can happen—the Secondary takes over and becomes Primary (in conjunction with a Quorum Server), or the Primary marks the Secondary as being "out of sync". In the latter case, the Primary stops replicating operations to the Secondary but continues to execute them locally. This will be the state of the datastore on the Secondary until the Secondary becomes reachable again, at which point we begin the process of resync. To allow for efficient resync some method is needed to track incremental changes that occurred after the systems went out of sync.

In VMstore we leverage efficient per-file snapshots that are implemented internally as a linked list of persistent delta B-trees by the filesystem metadata layer. At the time of going out of sync, the Primary will create a special *resync snapshot* on all file(s) within the affected mirrored datastore. (This is possible because our snapshots are relatively cheap, and the number of files, as mentioned earlier, is bounded and small.) The state of the Secondary can be determined by the Primary in the future when it is time to perform resync, from the combination of the data captured in the resync snapshots and the metadata about writes in-flight tracked via the PAW/-SAW entries. When resync begins, the Primary is able to efficiently identify data written after going out of sync by observing the delta between the time at which the resync snapshots were created and the current filesystem state. No work is required to materialize these deltas; they are maintained directly by the underlying filesystem metadata layer and can simply be read out on a per-file basis.

When the systems are out of sync, arbitrary filesystem manipulations may occur on the Primary—files and directories may be deleted, renamed, created from scratch, etc. One of the goals for resync is to avoid replicating updates to files which have subsequently been deleted on the Primary. Of course, the basic requirement is that resync must bring the Secondary into a state of being identical with the Primary. With this in mind, we perform resync processing in three steps:

1. Bring the Secondary into a state of being identical *with the content in the resync snapshots*. This applies to files which existed at the time the Secondary went out of sync, and is skipped for files created after the systems went out of sync. Arbitrary metadata operations that were in-flight when going out of sync are also reapplied on the Secondary.
2. Bring the Secondary directory namespace into sync with the Primary. This handles all deletions, renames, and file creations that occurred while out of sync. This also enables us to replicate new namespace manipulation operations while resyncing.
3. Resync file content on a file by file basis. Within each file, resync on an offset range by offset range basis.

Step (1) is similar to the distributed recovery procedure discussed above that we run immediately after connecting in-sync systems that have been briefly disconnected; the difference is that with resync, the file content must be read from the file-level resync snapshots on the Primary, not from the current live version of the file. As with the distributed recovery scheme, the PAW/SAW persistent metadata identifies blocks which were subject to in-flight writes at the time the systems went out of sync. Note that after going out of sync, the PAW/SAW metadata is essentially frozen to preserve the knowledge of which blocks had ongoing writes until the time we can use this information in resync. For files which exist on both Secondary and Primary (this is the normal case for long lived workloads), after this step, the Secondary is now identical in content to the Primary at the time the resync snapshot was taken on the Primary.

Step (2) allows us to optimize out writes that occurred to files which were subsequently deleted on the Primary, and to generally reclaim these files on the Secondary as early as possible. This reduces pressure for filesystem capacity on the Secondary and avoids scenarios where the Secondary may run out of space simply because it hasn't yet deleted files that we know have been deleted from the Primary.

Finally in Step (3) we iterate over all files on the Primary. The delta between the current file's content and the resync snapshot can be extracted efficiently on the Primary and the data read out and sent. Internally in VMstore, files are identified by a local FileId value, a 64-bit monotonically increasing sequence number. Files are resynced in increasing order of local FileId. This makes it fairly simple to persistently track resync progress; within a given mirrored datastore, we store a single local FileId value persistently during resync. Similarly, the offset within the resync snapshot is checkpointed as well. This allows resync to resume without performing a large amount of re-replication of data in the event of a local crash and restart on the Primary while it is performing resync. Checkpointing resync progress at a granular level is important because large virtual disk files (e.g., in excess of 10TiB) are not uncommon.

New writes to files which have been created after the systems begin resync and writes to offset ranges in files that have already been resynced are handled by mirroring them synchronously. This ensures that resync converges toward completion, i.e. it does not run the risk of falling behind incoming live writes and never completing.

6.1 Handling user-created snapshots

VMstore implements VM-level snapshots (scheduled or manual) using per-file snapshots. These per-file snapshots are atomically created across the set of files comprising a given VM. This complicates resync. At the start of resync, there may exist file level snapshots on the Primary which were created while the systems were out of sync. Conversely, there may exist file-level snapshots on the Secondary which were deleted from the Primary while the systems were out of sync. Similar to how file deletions are replicated during resync prior to sending incremental data, we replicate discrete snapshot deletion operations first, prior to replicating snapshot contents.

With the exception of clone create, snapshot create, and snapshot delete operations, most of the metadata operations that the Primary receives while resyncing are replicated to the Secondary immediately because the namespaces are in-sync. New snapshot creates are only replicated if the files involved are in-sync on the Secondary. Snapshot deletes are only replicated if the snapshot has been resynced to the Secondary. Clone creates are only replicated if the required backing snapshot is present on the Secondary. The resync process must eventually take care of replicating any of these operations if they are delayed from being replicated at the time they were issued to the Primary. Note that these operations themselves are not logged; the resulting file system state (the set of snapshots and clones) is discovered by the local FileId-based iteration described above.

7 Distributed Integrity Verification

The VMstore file system implements an incrementally-updated per-file content checksum for purposes of data integrity verification. The checksum is neither a cryptographic checksum nor a guarantee that corruption has not occurred; rather it is a probabilistic mechanism designed as an extra check on top of many other mechanisms (transactions, crash recovery, NVRAM, careful design, code review, thorough testing, etc) used collectively to ensure data integrity.

The checksum physically comprises approximately 1KB of metadata; file writes update portions of this checksum based on the file offset being written and the block content itself. Each block write updates the checksum using 7 bits derived from data in the write. The checksum metadata updates are performed efficiently using the transactional metadata mechanism noted in Section 2 (essentially the related metadata updates—system-wide statistics, file statistics, B-Tree updates to point to new blocks, etc—are logged together). Additionally, at the time of snapshot creation, a file's current checksum is stored with the associated file-level snapshot metadata.

The checksum values are used in several places. During file deletion, each block's checksum contribution is logically subtracted from the remaining file checksum value, and at the end of deletion, the checksum must be logically zero. Similar logic is used when truncating a file to zero bytes in size.

In synchronous replication, the basic requirement to maintain identical copies of files on both systems (as long as the systems are in sync) enables us to leverage the file content checksums for integrity verification. Integrity verification involves the following steps. First, writes and other operations are temporarily paused using the exclusive file-level lock mechanism described in Section 4.1. Next, in-flight operations are flushed through the system. Following this, the Primary reads out its per-file checksum values and sends them to the Secondary, which reads out its values and compares them. These checksums are expected to match across the two systems.

In order to avoid blocking file operations for an extended period of time, which could be the case if the datastore contains several thousands of files, distributed integrity checking is done in a batched manner. This allows us to acquire file-level locks one batch at a time as opposed to for the entire datastore. The batch size is chosen such that integrity checking is transparent to clients—it lasts at most a few seconds for any given file—and such that it minimizes network communication.

Content checksum mismatches are expected never to occur in practice. However, if they are encountered, the system takes the Secondary out of sync and lets the Primary continue servicing client IOs, to avoid interruption of service. Additionally the system logs the affected files and their checksum values on both sides. The differences in the checksums allow us to identify a set of candidate file blocks that may be different, and if the number of blocks in this set is below a threshold, the systems additionally read out and save off the affected blocks for later inspection. This mechanism has been occasionally useful in debugging the system during development.

In production we run the verification procedure such that each file is checked once every 24 hours, provided that the datastore is in-sync. Additionally, we also proactively perform checksum verifications at certain points, e.g. just prior to user-initiated cluster failover and at the end of resync.

8 Evaluation

We have implemented a heavily multithreaded write pipeline, each stage of which does asynchronous processing. This improves performance and also isolates processing of mirrored and non-mirrored datastore requests. We evaluated our implementation to answer the following questions:

- What is the overhead of synchronous replication on read and write throughput?
- What is the impact of the VMstore network RTT on client latency for various write workloads?

CPU	Xeon E5-2630 v2 (2x6 cores, 2.60GHz)
RAM	64GB DDR3 at 1600 Mhz
Flash	11x480GB SATA SSDs
Disks	13x4TB SAS SED HDDs
NIC	Intel X540-T2 at 10Gbps

Table 2: VMstore hardware configuration used in experiments.

Experiment setup: We used two VMstores running Tintri OS 4.3 (see Table 2 for hardware configuration[7]); one as the Primary and the other as the Secondary connected to each other through a 10Gbps ethernet link. We also had a Linux-based physical client machine which was connected to the Primary through a 10Gbps ethernet link. A tool that drives synthetic IO traffic over NFS was used to generate random read and write IO traffic with 8KiB, 64KiB and 256KiB block sizes. These IO sizes were chosen because they represent the majority of workload sizes observed in VM workloads. The network RTT between both the VMstores and between the client and the primary VMstore as measured by *ping* using a packet size of 64 bytes was observed to be 100μs on average. In some experiments, we used the *tc* (traffic control) Linux utility to vary the RTT between the two VMstores.

[7]As it happens, we used model T850, introduced in 2014, for these experiments. Two generations of newer hardware families have succeeded this model, so performance on current systems will be higher.

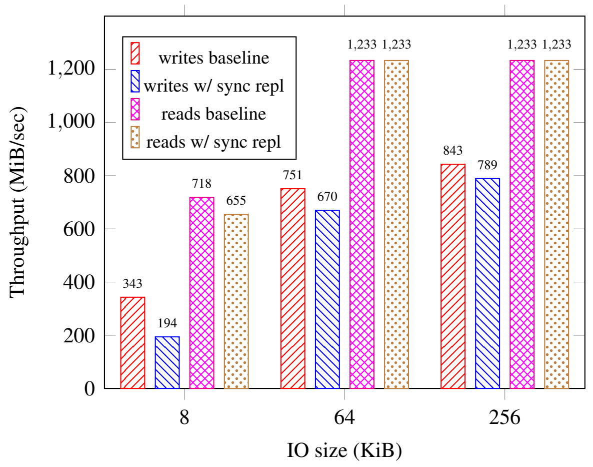

Figure 5: Throughput comparison between baseline performance and performance with synchronous replication for different IO sizes. The RTT was 100 μs; no additional delay was induced in the network.

8.1 Throughput

Figure 5 graphs read and write throughput for various IO sizes—8KiB, 64KiB and 256KiB under two scenarios: i) baseline performance when synchronous replication is not enabled, and ii) performance when synchronous replication is enabled.

Synchronous replication imposes a 43% overhead in throughput for 8KiB writes, 11% for 64KiB writes and 6-7% for 256KiB writes. The difference is significant for 8KiB writes because of the per-request processing overhead of replication in our system. This is due to file range locks, sending the request through various queues, memory allocation, and other assorted software overhead.

Synchronous replication imposes a very minor overhead on read performance; about 8% for 8KiB reads. This is because reads to files in synchronously replicated datastores also have to acquire shared file locks. Larger 64KiB and 256KiB IO size reads end up saturating the 10 Gbps network link even when synchronous replica-

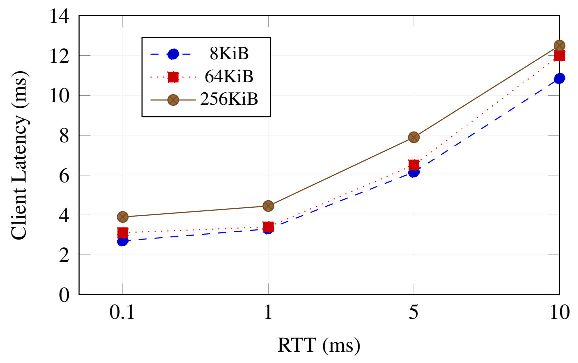

Figure 6: Impact of VMstore network RTT on client latency for various write workloads. For each workload, the client-side load was kept constant for all RTT values.

Workload	Throughput(MiB/sec)			
	0.1 ms	1 ms	5 ms	10 ms
8KiB	205	167	80	50
64KiB	725	660	340	250
256KiB	754	671	380	240

Table 3: Write throughput for different RTT values. The client-side load was fixed for each workload as the RTT was varied.

tion is enabled because bulk data movement dominates the fixed processing costs in the file system.

Performance in our system is subject to continuous improvement; with techniques like batching of small writes and write acknowledgements over the network, tuning of TCP connection performance and optimizing read-only workloads, we are confident that we can improve the performance of small reads and writes in subsequent releases of our software.

8.2 Latency

Figure 6 graphs the average client visible latency for various write workloads and for different values of RTT between the two VMstores. For all IO sizes, the trend observed is expected. The client visible latency increases as the RTT increases because every write has to be synchronously replicated to the Secondary. Additionally, as discussed in Section 4, the execution of writes on the Primary and Secondary is allowed to overlap. So, for lower values of RTT, the individual VMstore IO processing times will dominate the client latency and there is a value of RTT beyond which the RTT will start to dominate the client latency. This RTT crossover point depends on the cost of IO processing in the file system as well as the cost of mirroring and is hence workload dependent.

We also observe that the client latencies for 8KiB writes and 64KiB writes are close to each other at lower RTT values even though the latter has a higher mirroring cost. This is because various parts of the Tintri VMstore file system are optimized for 64KiB IO requests.

The difference in the client latency and the RTT gives the average overhead from synchronous replication and IO processing. From Figure 6, we observe that this overhead remains constant at around 3-4 ms for all workloads. This is expected because for a fixed client-side queue depth, any increase in the RTT should only affect the end-to-end client latency and not the latency overhead from synchronous replication. Of course, another consequence of this is reduced throughput. Table 3 captures the actual throughput observed at different RTT values.

9 Implementation Experience and Lessons Learned

Currently the system is in production use at dozens of sites globally. This section discusses our experiences in designing, building, testing, and deploying the system.

Usability: Space limitations prevent us from presenting our user interfaces for configuring the system, operational monitoring, and latency visualization. However, it is fair to say that the feedback from customers about the usability of the system has been extremely positive. The one area where there is a usability challenge relates to functionality which we deferred implementing, discussed next.

Functionality: From the beginning we designed for automated cluster failover. However, there was acute pressure to deliver *some* functionality to customers as quickly as possible. As a result, we elected to deliver functionality in a phased manner, and did not make automatic cluster failover available initially. In retrospect, demand for automatic cluster failover was higher than anticipated, and lack of this support has delayed adoption of the system to some extent.

Performance: Apart from the up-front design work to integrate replication carefully with the existing write pipeline (Section 4) to allow maximal parallelization, we did a modest amount of performance tuning specifically on the replication data paths. There is more that could be done to reduce the throughput gap between unreplicated writes and replicated writes, especially at 8KiB. However, as anticipated, the ability to let customers easily *not replicate* large portions of their workloads, combined with the performance-related work that we did do, has had the net result that there have been minimal performance problems in practice.

Complexity: Prior to undertaking synchronous replication, we had implemented asynchronous snapshot-based replication in VMstore. Synchronous replication is significantly more complex for a number of reasons. First, it necessarily involves fairly significant surgical modifications to various write paths and high level filesystem operation implementations. By comparison, asynchronous replication on the source system only has to consume snapshots after their creation, and the snapshot abstraction generally insulates asynchronous replication from the dynamic churn of ongoing filesystem operations. Second, synchronous replication must attempt to ensure both low latency and high throughput; asynchronous replication only needs to deliver sufficient throughput. Third, asynchronous replication has no equivalent of client transparent failover or automated cluster failover; failovers involve external reconfiguration of the customer's virtualization environment, implemented by higher level disaster recovery orchestration software, not by the filesystem replication system.

As a result, the synchronous replication implementation required roughly three times as many lines of code compared to asynchronous replication (approximately 100,000 and 35,000, respectively). Additionally, the asynchronous replication code is more self-contained and hence simpler to reason about. To a first approximation synchronous replication took perhaps five times the number of person-months of engineering effort, spread over roughly twice as much calendar time.

Correctness: The distributed data integrity verification mechanism (Section 7) proved invaluable during development and testing. The per-file content checksums on which this mechanism depends had previously been implemented a number of years before we began the synchronous replication project, and had been used extensively in internal testing. We had not enabled file content checksums in production due to lingering performance issues in certain scenarios. As part of the synchronous replication project, we decided early on to do the work necessary to allow us to enable the file content checksums in production. This allowed us to build the distributed integrity checking mechanism on top of the file content checksum mechanism. This took several months of effort on the part of several engineers, but was surely worth it. This mechanism caught a handful of subtle bugs during development and internal testing. However, in a year of shipping the system to dozens of customers, we have not experienced a single data path related customer found defect; the distributed data integrity check has not failed in production.

10 Related Work

In general, synchronous replication schemes in commercial enterprise storage systems are not well described in the literature. Seneca [7] describes a detailed taxonomy of design choices for remote mirroring, and a design for a remote mirroring protocol with correctness validation using I/O automata-based simulation. It also presents some details of existing systems as of 2003, many of which are still in use. Snapmirror [8] discusses an asynchronous replication scheme of using self-consistent snapshots of the data mirrored from a source to a destination volume. The focus is on system performance at the cost of data loss. The tolerance to data loss is proportional to the frequency of taking and mirroring snapshots.

For resynchronization, some enterprise data storage systems (e.g., Symmetrix [4] and Linbit [2]) use bitmaps to keep track of writes that have been processed on the primary but not yet replicated to the secondary. Other systems (e.g., MetroCluster [6]) use filesystem volume level snapshots or system-wide snapshots to achieve this. In contrast, our system uses granular per-file metadata and file-level snapshots.

Some enterprise storage systems also implement synchronous replication to guarantee zero divergence in data between a pair of storage systems. EMC RecoverPoint [4] supports synchronous replication over IP or over FibreChannel network. Their host-based I/O splitting technology is used to mirror application writes with minimal perceivable impact on host performance. HP 3PAR Peer Persistence [5] maintains a synchronized copy of data between a pair of storage nodes, with the host maintaining an active path to one array and a standby path to the other array. A transparent failover and failback between this pair of storage nodes is made possible using a Quorum Witness. These systems operate on the basis of LUNs and thus require significantly more operational expertise compared to our system.

Veritas Volume Replicator [3] is a host-based software system. It makes use of Storage Replicator Log which is essentially a circular buffer to persistently remember writes to be queued for replication to the secondary. Writes have to be first written to this storage replicator log, then replicated to the secondary for persistence. This serialization of IOs is suboptimal compared to our scheme where writes occur in parallel on the primary and secondary.

11 Conclusion

We have implemented logical synchronous replication, a new approach to solving an old problem. We have introduced novel mechanisms to track writes in-flight and reconcile them across systems after reconnects. Additionally, we leverage two-phase commit to replicate complex filesystem operations, and granular per-file snapshots to implement efficient resynchronization. A datastore-wide distributed data integrity verification procedure built on a novel per-file checksum scheme ensures that the system is operating correctly. The flexibility of replicating only a selected portion of a filesystem has proven intuitive and easy to use by users.

12 Acknowledgments

We would like to thank Fred Douglis, Mark Gritter, Tyler Harter, Ed Lee, and Ashok Sudarsanam for their helpful comments on early drafts of this paper. The anonymous reviewers and our shepherd, Andy Warfield, provided insightful feedback that greatly enhanced the presentation of our material. We would like to thank our management at Tintri (Ashok Sudarsanam, Tom Theaker, Tony Chang, and Kieran Harty) for their support in publishing this paper. Finally, we would like to thank the many engineers at Tintri, past and present, who contributed to building this system.

References

[1] [MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3. https://msdn.microsoft.com/en-us/library/cc246482.aspx.

[2] The quick-sync bitmap. https://docs.linbit.com/doc/users-guide-83/s-quick-sync-bitmap/.

[3] Veritas volume replicator option by symantec. http://eval.symantec.com/mktginfo/products/White_Papers/Storage_Server_Management/sf_vvr_wp.pdf, 2006. White paper guide to understanding volume replicator.

[4] EMC VNX Replication Technologies an overview. https://www.emc.com/collateral/white-papers/h12079-vnx-replication-technologies-overview-wp.pdf, 2015. White paper highlighting EMC VNX replication technology.

[5] Implementing vsphere metro storage cluster using hpe 3par peer persistence. https://www.hpe.com/h20195/V2/GetPDF.aspx/4AA4-7734ENW.pdf, 2016. White paper highlighting HPE 3PAR Peer Persistence.

[6] MetroCluster management and disaster recovery guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2495113, 2017. White paper highlighting MetroCluster.

[7] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 253–268. USENIX, 2003.

[8] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror®: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 9–9, Berkeley, CA, USA, 2002. USENIX Association.

[9] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[10] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[11] Raj Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, August 1995.

[12] Raj Srinivasan. XDR: External Data Representation Standard. RFC 1832, August 1995.