Scalable, Near-Zero Loss Disaster Recovery for Distributed Data Stores

Ahmed Alquraan
University of Waterloo
ahmed.alquraan@uwaterloo.ca

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Samer Al-Kiswany
University of Waterloo
alkiswany@uwaterloo.ca

ABSTRACT

This paper presents a new Disaster Recovery (DR) system, called Slogger, that differs from prior works in two principle ways: (i) Slogger enables DR for a *linearizable* distributed data store, and (ii) Slogger adopts the *continuous backup* approach that strives to maintain a tiny lag on the backup site relative to the primary site, thereby restricting the data loss window, due to disasters, to milliseconds. These goals pose a significant set of challenges related to consistency of the backup site's state, failures, and scalability. Slogger employs a combination of asynchronous log replication, intra-data center synchronized clocks, pipelining, batching, and a novel *watermark service* to address these challenges. Furthermore, Slogger is designed to be deployable as an "add-on" module in an existing distributed data store with few modifications to the original code base. Our evaluation, conducted on Slogger extensions to a 32-sharded version of LogCabin, an open source key-value store, shows that Slogger maintains a very small data loss window of 14.2 milliseconds which is near the optimal value in our evaluation setup. Moreover, Slogger reduces the length of the data loss window by 50% compared to incremental snapshotting technique without having any performance penalty on the primary data store. Furthermore, our experiments demonstrate that Slogger achieves our other goals of scalability, fault tolerance, and efficient failover to the backup data store when a disaster is declared at the primary data store.

PVLDB Reference Format:
Ahmed Alquraan, Alex Kogan, Virendra J. Marathe, and Samer Al-Kiswany. Scalable, Non-Zero Loss Disaster Recovery for Distributed Data Stores. *PVLDB*, 13(9): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/3397230.3397239

1. INTRODUCTION

The importance of distributed systems has dramatically grown in the cloud era. They provide the highly desirable scale-out and fault tolerance capabilities that are central to distributed infrastructures and services hosted by cloud vendors. This paper focuses on disaster recovery (DR), a critical feature required in production distributed systems, long before the cloud era, and certainly since it started. DR enables tolerance of data center wide outages, where the original data center is rendered inoperable for extended periods. DR of a *distributed data store* (databases, key-value stores, file storage, etc.) is enabled by creating an additional copy of the data store at a remote backup site (data center) while the primary site's data store is online [30]. The backup copy, which typically lags behind the primary data store, serves as the new basis of the data store to create and/or start a new primary data store. The latest data updates at the old primary data store may be lost during disasters. Nonetheless, concerns about data loss due to disasters have forced key DR design decisions in production data center infrastructures and distributed data stores [2, 8, 45, 46].

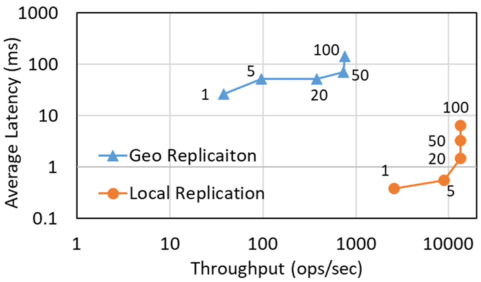


Figure 1: Local vs. Geo replication performance (throughput vs. latency) of LogCabin under 100% writes load, varying number of concurrent clients (numbers on the curves). Full details of the setup are provided in §7.

Traditional means of DR is through snapshots [18, 20, 33, 40, 48] – a data store snapshot is asynchronously created and replicated to the backup site. While a sufficient solution for many use cases, the key limitation of this approach is a potentially large window of data loss (seconds, minutes, hours/days) between the time the last snapshot was replicated, and the time the disaster occurred.

An alternate approach is to build synchronous geo-replicated data stores [5, 8, 46], which can trivially tolerate data center wide failures. The key benefit of geo-replicated data stores is that *zero* data loss is guaranteed even in the presence of data center wide outages. However, synchronous replication across data centers has a significant performance cost in the data store's critical path [17]. Our experiments on LogCabin [37], a highly available Key-Value (K-V) store that uses the Raft consensus protocol [38] for replication, compared performance between synchronous 3-way intra and inter data center replication. Figure 1 shows the experiment's results. We found that the inter data center replicated LogCabin cluster performs over an order of magnitude worse than the intra

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397239

data center cluster. The performance degradation simply reflects the effects of geographic distance between machines interacting in the replication protocol.

Clearly, intra data center replication is highly attractive from a performance perspective. Disasters are naturally relevant to distributed data stores constrained within a single geographic location (e.g. a single data center). As a result, our work's scope is restricted to such systems. The first question we want to answer is whether one can engineer a DR scheme that asynchronously replicates updates to a backup site with a *near zero lag* – in the event of a disaster at the primary site, the DR scheme may lose updates accepted at the primary site from just the last few milliseconds.

Another class of solutions to DR that form a reasonable starting point for our work fall under a category we call *continuous backups*. In these solutions, the primary data store is continuously, asynchronously, and incrementally replicated to the backup site [11, 21, 22, 30]. Unfortunately, none of these solutions work correctly for *linearizable* [16] distributed data stores[1]. In particular, we show that the order in which updates are backed up, using prior techniques, may lead to an update order at the backup site that is inconsistent with the update order observed at the primary site (§3).

In recent times, a growing number of commercially successful data stores that support linearizability, exclusively or optionally, have emerged [1,3,5,8,23,41]. The DR problem we discuss here is relevant to intra data center deployments of these systems. Thus the real question we want to address is, can a near-zero lag DR scheme be designed for linearizable distributed data stores? To that end, we propose a solution based on timestamps generated by a synchronized distributed clocks (§4). Furthermore, from a pragmatic view, can the solution be easily pluggable into existing distributed data stores, where the changes needed in the original data store are few and non-invasive?

We introduce Slogger, a new DR framework, to address the above questions (§5). Slogger plugs into any linearizable (even non-linearizable) distributed data store that uses write-ahead logs [32] to apply changes to its state. Slogger asynchronously replicates and applies the logs to a designated backup site. It preserves linearizability by tracking *temporal order* between log records using synchronized distributed clocks [8, 14, 28, 29, 31]. Specifically, Slogger assumes that the data store can tag each log record with a timestamp derived from the distributed clock that is synchronized across nodes in a data center. This timestamp is used to identify a partial order between updates. Log records are applied to the backup site in tagged timestamp order. To make our solution work in practice, we needed to employ several techniques such as concurrency, pipelining, and batching; we also employed a coarse-grain synchronization technique based on a novel *watermark service* on the backup site. All backup work happens continuously in the background and off the primary data store's critical path.

Slogger is the first backup system that guarantees *prefix linearizability* [49]: The backup system stores updates in the same linearizable order as observed by the primary data center as long as the latter is operational. If the primary data center fails, the system guarantees backup of a prefix of the linearization order of updates that happened at the primary data center. Prefix linearizability, which may lead to some data loss during a disaster, is far more preferable to inconsistencies in the backup site after a disaster at the primary site, particularly for mission critical applications such as databases [4, 11, 35, 44] and enterprise storage systems [36].

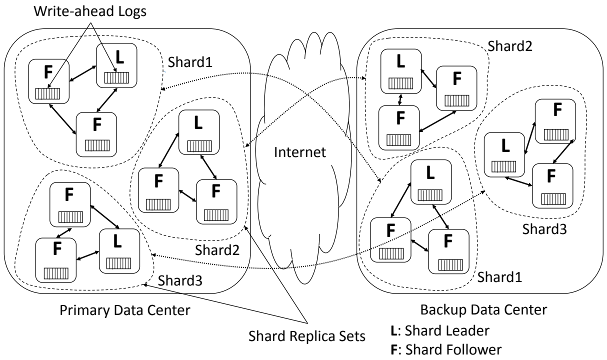


Figure 2: Example data store with 3 shards, each 3-way replicated. The inner workings of our backup system are described in §5.

We present extensive empirical evaluation (§7) of Slogger extensions to a 32-way sharded version of LogCabin, a highly available open source key-value store. Our experiments show that the data loss window stays surprisingly low (as low as 14.2 milliseconds in our experiments) even though the backup happens asynshronously in the background. This low data loss window comes at virtually no performance penalty on LogCabin. Furthermore, our experiments demonstrate that Slogger achieves our other goals of scalability, fault tolerance, and efficient failover to the backup data store when a disaster is declared at the primary data store.

2. DATA STORE ARCHITECTURE

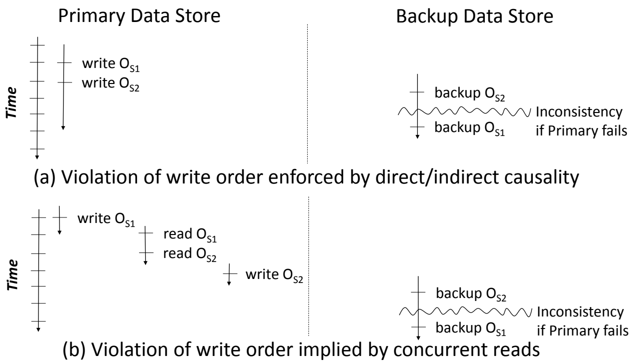
The Primary Data Store. Slogger makes critical assumptions about some properties of the distributed data store's design. We believe these properties are commonly supported by most data stores that provide high-availability guarantees using synchronous replication [2, 8, 23, 45, 46].

The data store is logically partitioned into a multitude of non-overlapping *shards*. Multiple shards can co-habit the same physical machine. Each shard is synchronously replicated for high availability. We assume a shard's *replica set* contains a collection of copies of the shard, termed *replicas*, hosted on different machines. A replica set typically contains a single *leader* replica that processes all the *updates* directed to the shard and propagates them to the rest of the replicas, called *followers*, in its replica set. Leaders can be chosen statically or dynamically using a leader election algorithm. The replication scheme itself may have a simple primary-secondary form [12, 19], or may rely on a more sophisticated consensus protocol such as Paxos [15, 26] or Raft [38]. Figure 2 depicts an example setup.

We assume that *all* updates to a shard are first appended to a local write-ahead log containing logical update records. The leader performs replication by sending log records to its followers. Replication is synchronous. After replication completes, the updates may propagate to the shard's *state machine* layer [43]. We further assume that the data store contains the mechanics to instantiate a new follower copy of a shard from its existing replica set. The data store can tolerate transient network problems such as unreliable delivery, local outages, asynchrony, etc. Finally, we assume a fail-stop failure model for machines used by the data store, in which machines may stop working, but will never send erroneous messages.

The Backup Site. We assume the backup and primary sites have similar infrastructures. The backup site will host a logical mirror copy of the primary site. This means the backup site will have logically identical shards, and use identical algorithms and schemes to control data it receives from the primary site, e.g., same replication, leader election, failure detection, etc. algorithms. However, the

[1] Linearizability is also referred to as *external consistency* in some works [8].



(a) Violation of write order enforced by direct/indirect causality

(b) Violation of write order implied by concurrent reads

Figure 3: Two scenarios that can lead to observed inconsistencies between primary and backup data stores: (a) Order implied by causally related non-overlapping writes to distinct shards. (b) Order implied by visibility of concurrent non-overlapping writes.

backup site will likely have a different physical structure, e.g., relative locations (host machines) of replicas, replication factor, and network topology will likely be different.

We assume that the network between the primary and backup sites is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or even delivered at all, and there is no limit on the time a machine takes to process a packet. Like the primary machines, the backup machines also have a fail-stop failure model.

Disasters. We assume a disaster is declared manually by the data center administrator, during an outage. The backup site will be informed about the disaster by an out-of-band mechanism that will trigger fail-over steps.

3. THE WRITE ORDERING PROBLEM

Our starting point is the aforementioned asynchronous log record shipping technique for backing up a distributed data store [11, 21, 22, 30]. That technique enforces ordering between writes (log appends) on different shards by tracking explicit data dependencies between them. For instance, King et al. [21,22] propose to track the order between log records of distributed transactions by tracking their write-write and write-read ordering dependencies, determined via overlapping read and write sets of transactions using Lamport clock style distributed counters [25]. The dependencies are used to apply transactions in the correct order at the backup site.

While the above dependency tracking may be sufficient for serializable updates, it is insufficient for linearizable updates. The nub of the problem is that ordering dependencies may not necessarily be implied by writes and reads; they may be enforced by logic of the application that uses the data store. Consider an application A that uses a data store D. A issues a write to object O_{S1} that resides in shard $S1$. After the write completes, A issues a write to object O_{S2} that resides in another shard $S2$. Thus, A has enforced an externally visible *happens-before* relation between the two writes. This relation is enforced by a linearizable data store since linearizability guarantees that an operation takes effect in real-time between its invocation and response [16].

Now consider the problem of backing up the two writes. The above happens-before relation must be preserved by the backup system. It however cannot be captured by simply tracking overlapping reads and writes, since there are no overlaps in the example. The order is enforced by A's program logic, which is opaque to D. Thus the backup site cannot correctly determine the order in which writes to O_{S1} and O_{S2} must be applied. In general the two writes may be causally ordered in arbitrarily complex ways that involve multiple threads of execution within A, and even multiple applications that communicate with each other via an out-of-band communication channel.

Interestingly, ordered reads can also enforce an order between concurrent writes. Consider two threads of execution in application A. Now consider a scenario where one thread writes to O_{S1} and the other thread concurrently writes to O_{S2}. Since the writes are concurrent, no specific order is imposed on them. However, if a third thread of execution happens to read the two objects in a specific order, say O_{S1} followed by O_{S2}, the reads may imply an order in which the two writes need to be backed up. For instance, if the read of O_{S1} returns its new value, whereas the read of O_{S2} returns its old value, backing up the two writes in the opposite order – O_{S2} followed by O_{S1} – may lead to an inconsistent view in the backup site if the primary site fails between the two backups. This *visibility-based* order must be correctly preserved in the backup.

One can consider another solution that forces the application to explicitly track ordering dependencies, and communicate them to the data store. This however entails a prohibitively intrusive change in the application, since the dependency tracking may be hard, even infeasible, in sufficiently complex applications.

Figure 3 depicts both of our problem scenarios. Both the scenarios above created a specific order between writes to O_{S1} and O_{S2}. The first is a causal dependency explicitly enforced through program logic, whereas the second is implicit through visibility of the updates. In both these scenarios, the backup system needs to preserve a happens-before relation between the writes that was established at the primary site. But how do we do it?

4. A TIMESTAMP BASED SOLUTION

Recall our assumption from §2 that all writes, data as well as metadata, in the data store are applied through write-ahead logs. We expect that in most cases the data store will contain a unique log per shard. Thus, in this model, the writes discussed in §3 are first appended to the leader log and then to the follower logs. These appended writes can be shipped to the backup asynchronously.

We observe that, at the backup, out-of-order reception of log records for different shards is acceptable. It is the order in which the log records are *applied* to the backup shard that is crucial to preserve correct happens-before relation between data store writes. For instance, in the examples shown in Figure 3, even if the backup receives the write to O_{S2} before it receives the write to O_{S1}, it must guarantee that if O_{S2}'s write is applied to the backup data store, O_{S1}'s write must also be *available* for application at the backup data store. This requirement permits a relaxation in that O_{S2}'s write can be applied to the backup data store *before* O_{S1}'s write is applied to the backup data store. We assume that no application directly reads from the backup data store, so ordering application of O_{S2} before O_{S1} is permitted.

A log record's write can be applied to the backup data store only after all log records' writes it causally depends on have at least been received at the backup in a reliable (replicated, if necessary) way. This causal dependency cannot always be determined by just observing the log records from the different logs, particularly in the problematic cases described earlier (§3). However, the happens-before links between log records can be indirectly embodied by *monotonically increasing timestamps*.

Timestamps tagged on log records can help establish a happens-before relation between log records generated in different logs. If two writes are causally dependent, they will be tagged with different timestamps correctly embodying the order of the writes. This is a conservative approach since it creates unnecessary happens-before relations between writes that are not causally related – absent additional dependency information, any writes that happened

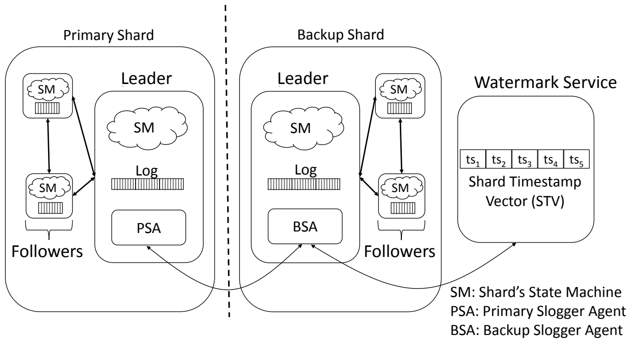

Primary Shard
Backup Shard
Leader
SM
Log
PSA
Followers
Leader
SM
Log
BSA
Followers
Watermark Service
ts$_1$ ts$_2$ ts$_3$ ts$_4$ ts$_5$
Shard Timestamp Vector (STV)
SM: Shard's State Machine
PSA: Primary Slogger Agent
BSA: Backup Slogger Agent

Figure 4: High level architecture of Slogger.

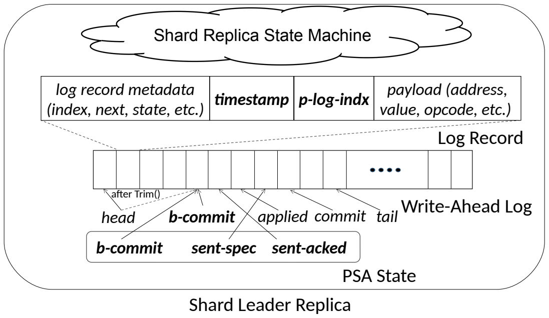


Figure 5: Write-ahead log and PSA state of a shard's replica in the data store. Slogger related changes in the log implementation in bold font (all PSA state variables are in bold font).

at time T are assumed to be causally dependent on *all* writes that happened at *all* shards at time $T-K$, for any integer $K>0$. However, so far it seems to be the only viable approach given that ordering relations established through application logic, or through visibility of updates, are opaque to the data store (e.g., inspecting just the log records on different logs does not reveal such ordering relations). Furthermore, there are interesting optimizations, such as pipelining, batching, and some key design decisions made in Slogger that significantly mitigate performance overheads due to this superfluous order enforced between unrelated log records. However, the fundamental question remains: how do we enable synchronized timestamps in an asynchronous distributed infrastructure?

High Accuracy Distributed Clocks. Synchronizing clocks between network-connected nodes is an old problem. The Network Time Protocol (NTP) [31] is a standardized distributed clock protocol used most widely in the industry for several decades. NTP achieves *accuracy* (maximum deviation from the time, also called *clock drift*) within 10s of microseconds to 10s of milliseconds [14] based on the network topology and data center size. Events occurring at a coarser granularity can be tagged with NTP timestamps to establish a happens-before relation in terms of global time.

Existing scalable distributed systems such as Google's Spanner database [8] use a combination of atomic clock appliances and GPS devices to determine the clock drift, perceived by all the clock appliances and GPSes. This clock drift is embodied by a time interval abstraction called *TrueTime*. Spanner's transaction commit protocol is managed to ensure that a transaction's beginning and end do not fall in even partially overlapping TrueTime (time ranges). This guarantee is used to establish a linearizable order of Spanner transactions. Spanner's TrueTime range was reported to be in the order of single digit milliseconds [8]. This clock accuracy however spans the multi-data center, geo-replicated system. We require clock synchronization *within* a data center where the data store is hosted; the backup site's clock does not need to be synchronized with the primary site's clock.

Orthogonal to the Spanner work, the Precision Time Protocol (PTP) [28], which was standardized about a decade earlier, uses dedicated hardware resources on network routers, switches, and end points to achieve much higher accuracy, in the order of sub-microsecond ranges [29, 34, 47], within a data center. More recent work [14, 29] has proposed schemes that achieve even greater accuracy, close to single or double digit nanoseconds.

Using Synchronized Clocks. Slogger requires clock synchronization between shard leaders at the primary data center only. No clock synchronization is required across data centers. We assume that the data center used to host our Slogger-augmented distributed data store has PTP or other more recent clock synchronization infrastructure. This delivers in the order of sub-microsecond scale clock drifts.

We also assume that replication of log appends is comparatively a much coarser operation, taking tens to hundreds of microseconds, or even milliseconds. Thus causally dependent updates of O_{S1} and O_{S2}, as depicted in Figure 3 (a), will be separated by timestamps that are consistent with the order of the updates.

Note that if a log record is timestamped at the beginning of its append operation, the ordering indirectly created by concurrent reads, as depicted in Figure 3 (b), is correctly embodied by the timestamps. This is because the append itself is expected to take an interval that is much greater than the clock's drift due to the much coarser replication latency. As a result, the read of O_{S1} will happen at a time strictly greater than the write's timestamp by more that the clock drift window. Furthermore, the read of O_{S2} happens *after* the read of O_{S1}. Even if the two reads happen quickly, within the clock accuracy window, if the read of O_{S2} returns its old value, it is guaranteed that the write of O_{S2} will have a timestamp that is greater than the timestamp of O_{S1}'s write: We assume that the data store's implementation guarantees that if the write of O_{S2} had a timestamp less than the time at which its read happened, the read would be ordered *after* the write in shard $S2$, and thus return its new value. This can be easily enforced by introducing synchronization between the competing read and write of O_{S2}.

If the PTP demon returns a synchronization error indicating that the local clock could be out of sync, that shard will stop accepting new updates. If the PTP implementation on a shard leader fails in a byzantine way (i.e., returns random time), Slogger may not be able to guarantee linearizability among operations that touch a set of shards that includes the faulty shard. This failure semantics is similar to the semantics of modern systems that rely on synchronized clocks [8].

5. THE SLOGGER ARCHITECTURE

Synchronized timestamps are foundational to Slogger's functioning. However, there are several other key components of Slogger that need to be architected carefully to deliver efficient and scalable DR. In particular, we would like to answer the following questions: How does Slogger ensure that the backing up process does not affect performance of the primary data store? How does Slogger absorb and apply a large number of logs from the primary to the backup site correctly and in a scalable way? How does Slogger tolerate local faults in the primary site as well as the backup site? How does Slogger bootstrap a new backup data store from an existing data store? How does failover work in Slogger when a disaster occurs at the primary site? Does integration of Slogger in an existing distributed store entail significant changes to the data store's architecture and implementation? This section addresses all these questions in detail.

Figure 4 depicts the high level architecture of Slogger. It contains three principal components: (i) *Primary Slogger Agent (PSA)*, (ii) *Backup Slogger Agent (BSA)*, and (iii) *Watermark Service*. A PSA is embedded in each replica of every shard in the data store on the primary site. PSA of a shard's leader node asynchronously sends newly *committed* log records to the corresponding backup shard leader's BSA (each backup replica has a BSA). For scalability on the backup site, each shard's log grows independently of other shards' logs. As a result, an out-of-band mechanism is needed to construct the most recent consistent state of the backup data store. The mechanism must ensure that the backup site constructs a version of the data store that is consistent with the current (or a prior) version of the primary data store. Slogger uses the *watermark service* to that end. Slogger also requires a few minor modifications to the data store's write-ahead log.

5.1 The Write-Ahead Log

Slogger's functionality relies on existance of a write-ahead log in the data store it augments. We assume commonly found attributes in this log: The log is finite sized and circular. Appends happen at the tail end of the log, marked by a *tail* index that is advanced by an append. The head end is marked by a *head* index. It is advanced (also called *log trimming* in the literature [32,42]) after a collection of log records at the head end are applied to the shard's state machine. The log also contains a *commit* index marking the index up to which log records have been replicated to a threshold number of replicas on the primary site (e.g. a majority of the replicas in a consensus based replication scheme). In addition, the log contains an *applied* index marking the log index up to which log records have been applied to the shard's state machine.

Figure 5 depicts a data store's write-ahead log and the relevant changes needed to support Slogger. For Slogger, we need to add a new index to the log called *b-commit*, which represents the index up to which the backup shard has received and committed (replicated) the log. b-commit is never greater than commit, and is never less than head. The difference between b-commit and commit essentially represents the *lag* in the backup shard in comparison with the primary shard. There is no correlation between b-commit and applied other than both have to be between head and commit. However, they do have an effect on the log trimming task in that head can be advanced to the smallest of b-commit and applied. In Figure 5, since b-commit lags behind applied, a log trim operation can advance head only up to b-commit (shown by the dashed arrow). Since its value can be reconstructed by consulting the backup shard, b-commit does not need to be a part of the persistent metadata of the log at the primary site.

Slogger augments each log record with a *timestamp* field that contains the time returned by the synchronized clock at the beginning of the corresponding log append. Slogger needs each log record's index embedded in the log record itself. This *p-log-indx* index is used by the backup shard to pass on to the primary shard the log index of the latest log record that the backup shard has appended to its log. Since the backup data store is not necessarily a physical mirror of the primary data store (e.g. replication factors may be different on both sites), the configuration updates at the backup shard may be different from the configuration updates in the primary shard. As a result, the p-log-indx of a log record may be different from its index in the corresponding backup shard's log. For instance, a log record appended at the primary shard's log may have an index of 10, which would become that log record's p-log-indx. However the corresponding backup shard's log may append that same log record at index 8. The backup shard can use that log record's p-log-indx (10) to inform the primary shard that it has received log records up to index 10. The primary shard then knows that it needs to back up log records from index 11.

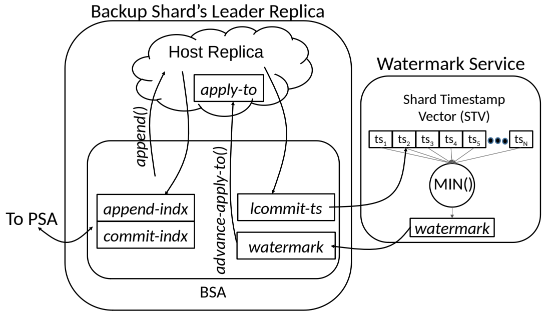


Figure 6: The BSA and Watermark service.

5.2 Primary Slogger Agent (PSA)

A PSA is hosted in each replica of each primary shard. It is an independent thread of execution that continuously monitors the shard's log and forwards any newly *committed* log records to the backup site. At any given time, only the primary shard's leader's PSA (henceforth, just PSA for brevity) is actively engaged in replication to the backup site.

PSA sends log records to the backup site in the order they were appended at the primary shard's log. It needs to track the latest log record that was successfully received (but not necessarily committed) at the backup site. It does so with the *sent-acked* index – after replicating a log record, the backup shard sends back an acknowledgment (ack) back to the primary leader's PSA, which the latter uses to update sent-acked.

We add another field to PSA's metadata, the *sent-spec* index, which indicates the log index up to which log records have been sent to the backup site. This index enables PSA to send additional (committed) log records while acks for previously sent log records are pending. It is advanced by PSA immediately after new log records are sent to the backup site. The number of pending acks can be capped by a configurable threshold after which PSA blocks, waiting for those acks. Log records are resent if their acks are not received in a preconfigured timeout interval. sent-spec always trails the log's commit, and is in turn trailed by sent-acked. sent-spec and sent-acked become equal when PSA receives acks for all sent log records. Figure 5 depicts PSA's sent-acked and sent-spec pointing to different log records.

sent-acked is never behind the log's b-commit. PSA also hosts a local (cached) copy of the log's b-commit for expedited lookup of its value. Note that PSA's state variables do not need to persist; they can be reconstructed from the primary and backup shards' states.

5.3 Backup Slogger Agent (BSA)

A BSA is hosted in each replica of each backup shard. Like the PSA, the BSA is an independent thread of execution that performs a sequence of operations that interact with the corresponding PSA, the backup shard's log, and the watermark service. Only the backup shard's leader's BSA (henceforth, just BSA for brevity) performs all these operations.

Figure 6 depicts BSA's architectural details. BSA's responsibilities include receiving log records sent by PSA. Log records are received in the order they were appended at the primary shard's log; log records received out-of-order are nacked. BSA sends the received log records to its host replica – the append() call in Figure 6. It also stores p-log-indx for the last received log record in the local variable *append-indx*. While the host replica (leader of the backup shard) replicates these log appends, BSA sends an ack for those log records back to PSA. Both the ack/nack messages contain the value of append-indx, which confirms BSA's progress with PSA. At PSA, the received append-indx value overwrites its sent-acked.

Recall that only committed log records are sent by PSA to BSA. We consider the backup shard's log records committed only after they are successfully replicated to its replica set (based on the replication criterion, e.g. majority, as dictated by the data store's implementation). The backup shard's log maintains its own commit that is independent of the primary shard's log's commit. The backup shard's commit is advanced as log records are successfully replicated to the shard's replicas. BSA monitors this commit and copies the last committed log record's p-log-indx into a local variable called *commit-indx*. BSA adds the updated commit-indx in the aforementioned ack/nack it sends back to PSA, which uses the value to advance its b-commit.

Committed log records of the backup shard cannot be immediately applied to the shard's state machine since simply applying those log records can lead to inconsistencies in the backup site if the primary happens to fail at an arbitrary time (see section 3). A committed log record *r* becomes eligible for application only after all shards on the backup site have committed all the log records with timestamps less than *r*'s timestamp. To accomplish that efficiently BSA periodically performs two actions: (i) it queries the backup shard's log to determine timestamp of the latest committed log record, and stores it in a local variable called *lcommit-ts*; and (ii) it thereafter sends the lcommit-ts to the *watermark service*, which maintains a repository of largest timestamps of committed log records observed at each shard.

5.4 The Watermark Service

The watermark service determines what log records can be applied safely to the backup data store. Each BSA periodically sends its lcommit-ts to the watermark service. The watermark service maintains a vector of timestamps, called *shard timestamp vector (STV)*, each element of which represents the largest timestamp received from each backup shard's BSA. The minimum of these timestamps, which we call the backup's *watermark*, indicates the time up to which *all* backup shards have committed log records. After the watermark service receives lcommit-ts from a BSA, it writes the received lcommit-ts to that backup shard's slot in its STV. It then computes a new minimum watermark available in STV and sends it back to BSA. Figure 6 depicts the STV for *N* shards, and shows receipt of a lcommit-ts in the STV slot labeled ts_2. The figure also shows computation of the new watermark using the $MIN()$ function. The watermark service responds to BSA with the newly computed watermark.

The new watermark received at BSA is used to update its watermark, and is then forwarded, via the advance-apply-to() call shown in Figure 6, to the shard to advance its log's *apply-to* index. The backup shard updates apply-to based on the received watermark. In the end, apply-to points to the shard's latest committed log record that can be applied to the shard's state machine. Note that apply-to does not need to be a part of the persistent metadata of the backup shard, and remains a simple counter in the non-persistent DRAM-resident metadata of the backup shard's log. Each shard periodically exchanges timestamp messages with the watermark service to stay up-to-date with the backup data store's progress.

Throughout its execution, the watermark service maintains the invariant that its watermark is monotonically increasing. Note that the watermark service itself is an optimization to the more straightforward solution where each BSA periodically broadcasts its lcommit-ts values to all other BSAs on the backup data store. In the end, the watermark's ground truth, the lcommit-ts values in log records of each shard's log, remains persistently stored in shard logs. Thus even if the watermark service fails, it can be easily reinitialized from all the backup shard logs' lcommit-ts values.

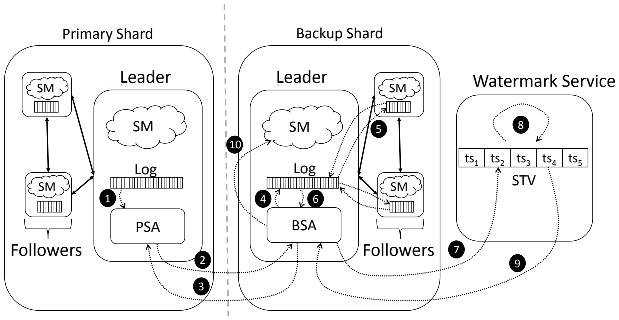


Figure 7: Detailed control flow of Slogger.

5.5 Putting it all together

5.5.1 Replica Initialization

When a replica in a backup shard is created, it first performs all the initialization work relevant to the data store's implementation. It then launches its BSA. The launched BSA remains idle until its host replica becomes the shard's leader. When the BSA detects that its host replica has become the leader, it performs the following steps: First, it determines the shard's lcommit-ts by reading the last committed log record's timestamp. Second, it establishes a connection with the watermark service to determine the backup site's global watermark. Third, if it detects a change in the global watermark, it advances the shard's apply-to to inform the shard that more log records may be applied to the its state machine. Finally, the BSA listens for a request from the PSA in the corresponding primary shard.

Similar to BSAs on the backup site, at the primary site, each PSA is launched in its host replica and remains idle until it detects that the host replica has become the leader of the primary shard. A new leader's PSA builds its state by querying the corresponding backup shard's BSA. BSA responds with its append-indx and commit-indx values to the newly identified PSA. If BSA is not the backup leader, it responds with a NOT_LEADER message. PSA then queries all backup shard replicas to determine the correct leader. The correct leader's BSA responds with its append-indx and commit-indx values. commit-indx serves as the initial value of the primary shard's b-commit, whereas append-indx is used to initialize PSA's sent-spec and sent-acked. Thereafter, PSA can asynchronously start shipping its shard's newly committed log records to BSA.

5.5.2 The Backup Process

Figure 7 shows the full flow for Slogger's asynchronous backup process for each update received at the primary shard's leader. The whole process takes 10 steps (labeled in the figure), some of which happen concurrently.

1. The primary shard's leader achieves consensus on its recently received update request by replicating to a majority of its followers, and advances its log's commit. The leader timestamps the log record with the synchronized clock's value as a part of the append. Its PSA periodically checks commit for changes and detects that a new log record is available for asynchronous backup.

2. PSA forwards the recently committed log record, in the form of a BACKUP request, to its "peer" BSA (on the corresponding backup shard's leader). After sending the log record, PSA advances its sent-spec.
3. On receiving PSA's log record, BSA can send back one of three responses: (i) A NOT_LEADER message if its host replica is not the backup shard's leader. (ii) A nack indicating that the log record was received out-of-order. The nack contains BSA's append-indx, which indicates the last log record it received in the correct order. PSA can use the received append-indx to resend log records that were not received by BSA in the correct order. (iii) An ack response indicating to PSA that the log record will be appended at the backup shard. Once PSA receives an ack it advances sent-acked to the log record for which it just received the ack. In both the nack and ack responses, BSA also embeds its append-indx and commit-indx values to help PSA adjust its sent-spec, sent-acked and b-commit indexes.
4. The newly received log record at BSA is then forwarded to the host replica to append to its log. Note that at this stage BSA assumes that its host replica is still the shard's leader. If that is not the case, BSA detects the change in the leader's state and goes back to a passive state where it no longer processes log records received from its peer PSA.
5. The backup shard's leader attempts to achieve consensus on the newly appended log record, advancing its commit if it succeeds. The details of this step are data store specific.
6. BSA periodically determines if new log records were committed at the backup shard. If so, it updates commit-indx, which is forwarded to PSA in response to a subsequent log append message it sends (see step 3). BSA also updates its lcommit-ts with the last committed log record's timestamp.
7. If lcommit-ts changed (increased), BSA sends its new value to the watermark service.
8. On receiving a new lcommit-ts of a shard from its BSA, the watermark service updates the shard's slot in its STV with the lcommit-ts, and then computes the new global watermark.
9. The watermark service responds to BSA with a new global watermark.
10. On receiving the new watermark value from the watermark service, BSA updates its local watermark and uses it to update the replica's apply-to to the latest log record with timestamp less than or equal to the watermark. The replica is then enabled to apply all its log records up to the updated apply-to.

Note that the above described algorithm leads to an interesting backup model: an ever-evolving prefix of the history of the data store's state. It does so by creating a *consistent cut* across *all* shard logs, delineated by the monotonically increasing global watermark's value. In a sense, each advance in the global watermark represents a consistent snapshot of the entire data store. However, the snapshot's lifetime exists only briefly until the global watermark advances to a higher value. This makes Slogger's approach quite distinct from traditional snapshot-based backups that identify (and embody) the consistent state of the data store at a specific point in time. In the traditional snapshot approach, a user can refer to the specific snapshot generated in the past even though the primary data store may have changed substantially since the snapshot was taken. Slogger creates a continuously evolving backup, where retrieving the state of the data store at a distinct point in the past is not possible. Nonetheless, Slogger achieves DR in its unique way. Prior works supporting continuous backups like Slogger [11,21,22,30] do not use timestamps and watermarks to advance the backup data store's state evolution. Furthermore, they do not achieve DR for linearizable distributed data stores.

5.5.3 Keeping up with the Primary

As stated earlier, in Slogger's replication algorithm, all backup shards indefinitely receive (and replicate) log records independently of each other. However, backup shard logs are finite in size and hence must be applied to shard state machines to make space available for future log records.

As long as the watermark service keeps receiving monotonically increasing lcommit-ts values from *all* shards in the backup site, Slogger's replication algorithm ensures that the global watermark remains monotonically increasing, thus guaranteeing liveness. We require a foundational assumption for liveness though – each primary shard keeps sending log records with monotonically increasing timestamps at regular intervals. To that end, we require each primary shard to generate log records at regular intervals, even when the shard does not receive any update requests from the data store's client applications. This can be achieved by making the primary shard's leader append NOP log records at regular intervals in their logs. This capability is available in most, if not all, production quality distributed data stores. Thus, even if a shard does not receive update requests for prolonged intervals, it will keep producing timestamped NOP log records that help continuously increase the global watermark, thus guaranteeing liveness.

For Slogger to work, the whole backup process must be able to keep up with the primary site's update rate. This implies that the log at each primary shard has enough capacity to receive new log records while older log records get replicated and applied to the backup site.

Another crucial aspect of Slogger that helps the backup site keep up with the primary is its ability to do batching at various stages in the entire backup lifecycle: First, PSA can send a collection of contiguous log records batched in a single request to its peer BSA. Second, the log records forwarded by BSA to its host replica can also leverage any batching capabilities supported by the data store itself. Third, determination of the lcommit-ts by BSA is amenable to covering a large batch of recently committed log records in one shot – BSA needs to simply determine the timestamp of the last committed log record. Lastly, traffic to the watermark service can also be modulated by controlling how often BSA sends its lcommit-ts to the watermark service.

5.5.4 Fault Tolerance

We assume that the original data store tolerates non-byzantine fail-stop failures. We leverage this capability of the data store to achieve fault tolerance in Slogger. The obvious points of failure are PSA, BSA, and the watermark service. Under the fail-stop failure model that we assume, failure of the PSA implies failure of its host replica. If it is a follower replica that fails, nothing needs to be done since its PSA (or BSA) is passive. However, if the leader replica fails, the data store initiates its leader election protocol to elect a new leader, after which the leader's PSA (or BSA) becomes active.

When a new PSA begins execution, it needs to first initialize its variables (b-commit, sent-spec, and sent-acked). It does so by sending a BACKUP request to its peer BSA. BSA's response to the message, whether an ack or a nack, contains its current append-indx and commit-indx values that are used to initialize PSA's variables. Even if the primary shard contains multiple active PSAs at the same time, Slogger works correctly since PSAs replicate only committed log records, and BSAs ignore redundantly received log records.

BSA's failure is similarly tied to the failure of its host replica. Initialization of BSA's variables is done by consulting its host replica's log. In addition, BSA needs to communicate with the watermark service to take over the responsibility of forwarding lcommit-ts values for the shard. It also initializes its watermark variable by querying the watermark service.

The backup algorithm for a shard may experience some delays if the leader of the primary or backup shard fails. However, we expect this delay to be largely related to the leader election process that is triggered after a leader fails. In most leader failure scenarios, we expect this delay to be insufficient to stall the backup process to the extent that the backup shard cannot keep up with the primary shard.

As stated earlier, the watermark service is itself an optimization. If it fails, a new instance for the service is spun up and its STV is initialized by communicating with all the backup shards.

In principle, there exists a pathological scenario where one or more primary shards fail or cannot communicate with their backup counterparts. In such circumstances, the failed shard's lcommit-ts will not advance, thus stalling the global watermark. If other shards continue to serve client requests, their logs may fill up on the backup site. This would create a "back pressure" on the corresponding primary shards, which may need to be stalled, thus escalating a single shard "outage" to multiple shards in the data store. Such a pathological scenario can be handled pragmatically by dynamically allotting greater space for logs and, at the same time, informing the data store administrator of such circumstances.

5.5.5 Bootstrapping Backup Shards

We assume the distributed data store has capabilities to bootstrap a new replica from an existing replica set of a shard. Slogger leverages these same capabilities to spin up the backup shard (and its replicas) on the backup site.

5.6 Handling Disasters

While highly available data stores can tolerate failures of a significant number of resources, it is not possible to do so beyond a particular threshold (e.g. partial or full destruction of a data center due to an earthquake that makes some or all shards unavailable). We call such a scenario a disaster, and leave it up to the data store administrator to make that determination. Thus a disaster is designated, and recovery from it is triggered manually by the administrator. Designating a disaster at the primary site eventually leads to its shutdown. This can be done by leveraging *metadata servers* that are typically present in production quality distributed data stores. The data store administrator can flag a disaster by switching any live metadata servers to the *disaster* mode. Each primary replica periodically pings the metadata servers to determine its health. If it is still available, a metadata server responds with a DISASTER flag that signals the primary replica that it needs to shut down. If the metadata servers are unavailable the ping times out. In both cases, the primary replica must assume a disaster and shut down. Thus eventually all replicas in the primary site will shut down.

Triggering recovery at the backup site leads to a number of steps that helps the backup bring up a new primary site. First, the backup site's metadata servers are flagged by the administrator for a RECOVERY mode. Each backup replica periodically pings the backup metadata servers to determine its health. If a metadata server responds with a RECOVERY flag, the replica goes in a *recovery* mode. If the replica is the leader of a shard, it computes its final lcommit-ts value, and sends it to the watermark service. The leader also marks its message as the final message. The watermark service thus receives final messages from all backup shards, and broadcasts the final global watermark to all backup shards. Then, the leader of each shard applies all log records based on the received watermark and sends a *recovery completion* message to the watermark service. Once the watermark service receives recovery completion messages from all shards, the backup is designated to have recovered. It can thereafter act as the new primary site for the data store or can be used to bootstrap a new backup site.

6. IMPLEMENTATION

To verify the safety property of Slogger, we wrote a complete system specification and used TLA+ model checker [24, 27] to check it. Our specification verified the safety property under a range of failure scenarios including primary leader failure, backup leader failure, and failure of the watermark service.

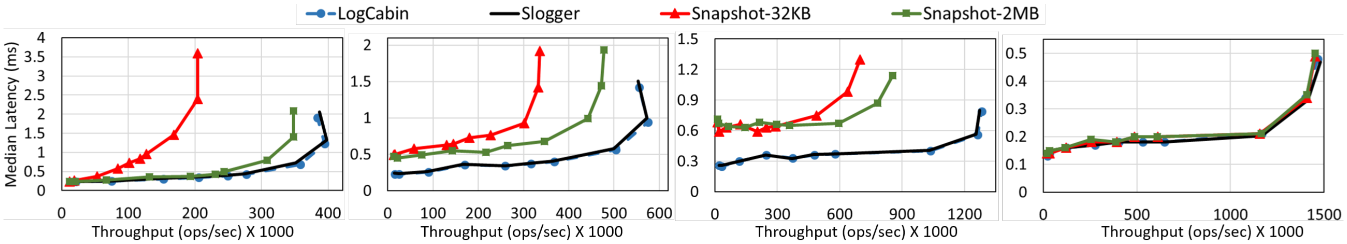
To evaluate Slogger's effectiveness, we integrated it with LogCabin [37], a linearizable key-value (K-V) store that uses the Raft consensus protocol [38] for synchronous replication. LogCabin is a single shard K-V store. To build a prototype distributed K-V store, we added a key-space sharding layer that hosts each shard in a 3-way replicated LogCabin instance. The key-space sharding map, which is static in our implementation, is directly available to client's through a thin client-side stub library. The replica set for each shard is also statically defined in a configuration file accessible to all replicas. This configuration file was augmented with details of the backup shard replica sets as well as the watermark server. The static primary-backup configuration setting helped us simplify our prototyping effort for expedited experimentation. We assume that the machine local clocks are synchronized with a synchronized clock.

We implemented Slogger using C++, which was also used to implement LogCabin. We implemented Slogger as a module in LogCabin. Slogger's code implements all its components – PSA, BSA, and watermark service. LogCabin uses an operation log to replicate its leader's updates to the follower nodes. We modified this log implementation to incorporate all the fields and processing relevant to Slogger. These modifications amounted to 130 lines of code, which attests to our assertion of little disruption in a data store to integrate Slogger with it. To support failover after a disaster (§5.6), we added the metadata server functionality to the watermark service.

7. EVALUATION

We now evaluate several aspects of Slogger's design. First, we compare the performance overhead of Slogger and incremental snapshotting (§7.1). Second, we measure the lag between primary and backup sites (§7.2). Third, we evaluate scalability of our watermark service (§7.3). Fourth, we measure how the backup lag is affected by different failure scenarios; i.e. SPA, SBA, and Watermark Service (§7.4). Fifth, we evaluate how our design details pertaining to pipelining, batching and selection of the right log size, contribute to the viability of Slogger (§7.5). Finally, we show how quickly can Slogger support failover to the backup site once a disaster has occurred at the primary site (§7.6).

Testbed. We used machines from two CloudLab [13] data centers – Wisconsin and Clemson – to run our experiments. We ran the primary site on a 16-machine cluster in the Clemson data center, and the backup site on a 16-machine cluster in the Wisconsin data center. We used five Clemson data center machines to run clients and generate the workload. Hence, clients communicate with the primary site machines over the local network. Each machine in Clemson data center has a dual socket Intel E5-2660 CPU with 10-cores per socket, 256GB of DRAM, a 10Gbps NIC for local network traffic, and a 1Gbps NIC for non-local network traffic. Each



(a) Workload w: 0% R, 100% W (b) Workload a: 50% R, 50% W (c) Workload b: 95% R, 5% W (d) Workload w: 100% R, 0% W

Figure 8: Throughput vs. latency for different systems for various read (R) to write (W) ratios when varying the number of clients between 1 and 750.

machine in Wisconsin data center has a dual-socket Intel Xeon Silver CPU with 10-cores per socket, 192GB of RAM, a 10Gbps NIC for local network traffic, and a 1Gbps NIC for non-local network traffic. The network round trip time (RTT) between Clemson data center and Wisconsin data center is 25.5 milliseconds. The network RTT time within Clemson and Wisconsin data centers are 0.068 milliseconds and 0.033 milliseconds, respectively. In all experiments, unless explicitly stated otherwise, we used a cluster of 32 shards and each shard is 3-way replicated.

Alternatives. We compare the following alternatives for designing a disaster recovery system:

- Baseline. We used LogCabin original implementation as a performance baseline. This option only serves client requests at the primary data center, i.e., the data is not georeplicated. This alternative presents the upper bound for system performance.
- Incremental Snapshotting. LogCabin supports snapshotting, but it generates a complete system snapshot on every snapshot interval. We modified LogCabin to perform incremental snapshotting [4]; a new snapshot will only include the set of objects that have been updated since the previous snapshot. The write-ahead log is used to identify the set of updated objects since previous snapshot. Then, these objects are sent to the backup site and used to update the state machine of the backup shards. In our implementation, a new snapshot is taken when the snapshot size exceeds a configurable threshold, or when the duration since the last snapshot exceeds a configurable threshold. To simplify our implementation, each shard snapshots its state machine independently without coordinating with other shards. We note that, while the individual snapshots may not preserve causality across shards, the performance of our prototype presents an upper bound of incremental snapshotting using LogCabin as adding a synchronization phase between shards will add additional delay. In all our experiments we set the snapshotting interval to 500ms.
- Slogger. A modified version of LogCabin that integrates Slogger to provide DR.

Our evaluation of the synchronous geo-replication (Figure 1) approach indicates that it reduces the system throughput by more than an order of magnitude compared to intra date center replication. We omit this alternative for clarity of presentation.

Workload. We used the standard YCSB benchmark [7] to evaluate Slogger. We used a uniform workload, as Slogger reads committed log records from the replicated log which is agnostic to the workload distribution; we also experimented with a Zipf distribution and got similar results. We experimented with a pre-populated data store containing 50 million key-value pairs, each with key-size of 24 bytes and value-size of 512 bytes. In all our experiments, the clients issue *blocking* requests in a closed loop, i.e. they do not issue a new request until the old request is successfully acknowledged by the system. Consequently, this workload has a long chain of happens-before relation as each new operation causally depends on all previous operations issued by the same client. Before sending a request, the client hashes the key to determine the destination shard.

7.1 Performance Evaluation

In this section, we compare throughput and latency of Slogger with its baseline and incremental snapshotting. As the throughput of incremental snapshotting is directly affected by the snapshot size (we evaluate the effect of snapshot size in the next section), we use two snapshot sizes: small snapshots of 32KB, and large snapshots of 2MB.

Figure 8 shows the throughput and latency for various systems for different read-write ratios when varying the number of clients. LogCabin presents the baseline performance without geo-replication. Slogger does not lead to noticeable overheads compared to LogCabin for any of the workloads as it has been architected to do continuous backup of a distributed data store off the client-facing critical path. On the other hand, incremental snapshotting has a significant overhead on the performance of the system; for write-only workload, using small snapshots reduces the throughput of LogCabin by up to 50% and increases the latency by up to 75%. Large 2MB snapshots reduces the overhead on the system throughput, yet achieving considerably lower throughput than both LogCabin and Slogger. At the same time, larger snapshots significantly increase the lag of the backup site (§7.2.3). For read-only workload (Figure8 (d)), all systems have the same performance as no objects are being updated.

7.2 Data Loss Window

Slogger is a continuous backup system that has the promise of delivering a distributed data store backup that is closely behind the primary data store. The immediate consequence is a backup system with a tiny (millisecond scale) data loss window. To assess this data loss window we conducted an experiment that measures a metric called the backup site's *lag*. Intuitively, the lag measures how far behind, in terms of time, is the backup site's state from the primary site's state. For example, a lag of 100 milliseconds implies that the backup site did not receive at least some log records that were committed at the primary site during the last 100 milliseconds. If the primary site shuts down at that point, the backup will lose updates that happened over the last 100 milliseconds – the data loss window.

Understandably, the lag varies over time. Capturing the precise lag in a non-intrusive way over the duration of the distributed data store's lifetime is perhaps an infeasible undertaking. However, we can get an approximation of a typical lag by finding its bounds using some lightweight instrumentation. We now define the upper and lower bounds of the lag, detail the experiments we used to

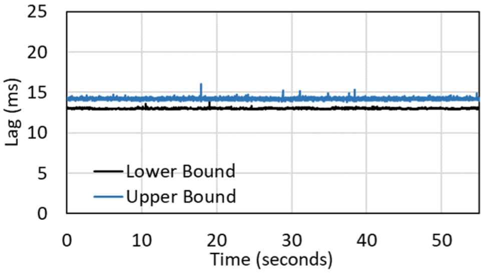


Figure 9: Lower and upper bounds of the typically observed lag at the backup site. This is a write-only workload

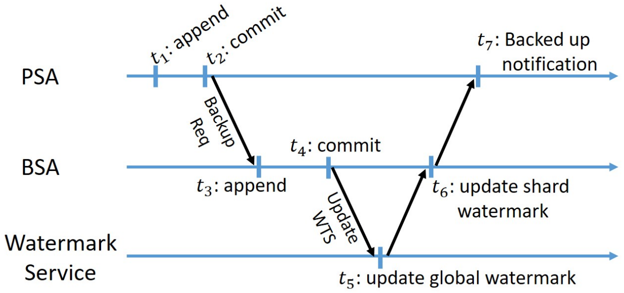


Figure 10: Sequence of events until PSA learns that a log record is backed up.

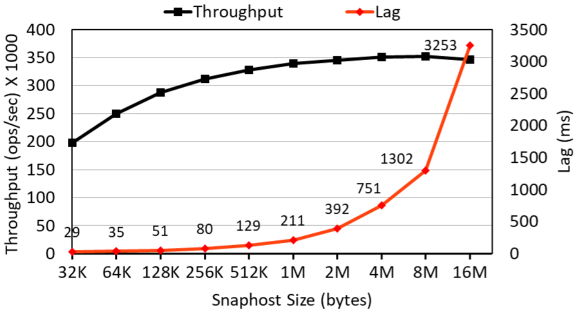


Figure 11: The throughput and the lag for different snapshot sizes with write-only workload. In this experiment, all snapshots are taken due to exceeding the size threshold. Labels next to points represent the lag.

measure them, and show the results of these experiments.

7.2.1 *Lower Bound of The Lag*

The lower bound (l_b) of the lag is the minimum lag that can be achieved between the primary and the backup sites. It is the minimum time needed to send a log record from the primary site (i.e. PSA) to the backup site (i.e. BSA). Clearly l_b cannot be lower than half the round-trip time (RTT) between the primary and backup sites. In Slogger, the lower bound is the time needed to perform step 2 of the backup process (Figure 7).

We measured the lower bound of the lag on a cluster of 32 shards with a ping-pong experiment. In this experiment, each shard's PSA sends a BACKUP request that has only one log record to the corresponding BSA. When BSA receives the request, it replies immediately with an ack. When PSA receives the ack, it immediately sends another BACKUP request. We measured the lag's lower bound from the BSA side by taking the time difference between consecutively received BACKUP requests and dividing it by 2 to get an estimate of the one-way latency (we assume symmetric latencies between the PSA and BSA). Figure 9 shows the output of running this experiment for 60 seconds. In the figure, we present the output of one shard as it is representative of other shards' output. The average value of lag's lower bound is 13.01 milliseconds.

7.2.2 *Upper Bound of The Lag*

The upper bound (u_b) of the lag is the time difference between the commit time of a log record at the primary site and the time at which the output of the watermark service, the global watermark, becomes greater than or equal to the timestamp of the log record (Figure 7 - step 8).

Figure 10 shows all events that occur until a log record is backed up. Note that a log record that has committed on the backup site cannot be applied until the global watermark advances to the timestamp of that log record. So ideally, what we would like to measure is $t_5 - t_2$. However, computing this value is not trivial as timestamps t_2 and t_5 are captured by different entities – PSA and watermark service respectively, which are running on different machines in different data centers. To tackle this issue, we measure the upper bound using only PSA's clock by computing $t_7 - t_2$ at the PSA, where t_7 is the time when PSA receives the notification from the corresponding BSA. However, this period includes the time needed to send the notification from BSA to PSA, l_b, which should not be included in the upper bound. As a result, we compute the final value of the upper bound as $t_7 - t_2 - l_b$. Note that what we are effectively computing is $t_6 - t_2$. This is a conservative approximation of the precise u_b, which would be the time when the watermark is advanced to a time t_w, where $t_w \geq t_1$.

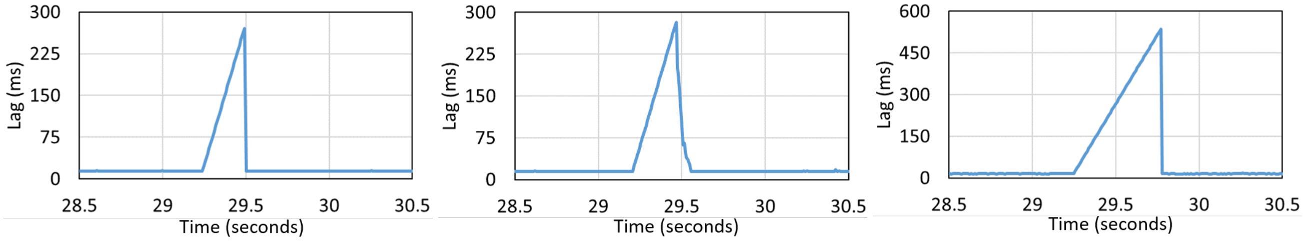
Figure 9 shows the upper bound of the lag of Slogger when measured using a cluster of 32 shards over a period of 60 seconds. Reported results were taken from a representative shard. The average value of the lag of Slogger remains at a somewhat steady 14.2 milliseconds with the maximum value at 16.05 milliseconds. The measured lag varies over time due to several factors: First, PSA of each shard ships newly committed log records at the end of every 1 millisecond window. This introduces a noise of up to 1 millisecond for any log record. Second, at the backup shard, up to lower single digit millisecond noise is introduced by the Raft consensus protocol and batching used by LogCabin. Finally, synchronization with the watermark service leads to sub-millisecond noise.

The above results are largely dominated by network latencies between the data centers hosting the primary and backup stores (half of the RTT is 12.5 milliseconds). We expect the lag to ordinarily fall between the upper and lower bounds in common cases. We note that those numbers do not reflect the worst case scenarios for the lag, which can be triggered by certain rare failures (§7.4).

7.2.3 *Incremental Snapshotting Lag*

In this section we evaluate the lag of the incremental snapshotting approach. This lag is directly affected by the frequency with which a snapshot is created, which, in turn, is affected by the snapshot size. Figure 11 shows the throughput and the lag observed for different snapshot sizes. We define the lag for incremental snapshotting as the difference in receive time of consecutive snapshots at the backup site. As shown in Figure 11, increasing the snapshot size improves the throughput, however, it increases the lag of the backup site significantly. For instance, the lag observed with a snapshot size of 8*MB* is at least 2 orders of magnitude higher than that of 32*KB* snapshot size. The minimum lag that can be achieved using snapshots is 29 milliseconds with 32*KB* snapshot size.

We note that Slogger's lag of 14.2 *milliseconds* is lower than the lag of best incremental snapshotting (i.e., 32*KB*) by 50%, and it does not lead to any performance degradation. Slogger outperforms incremental snapshotting for multiple reasons; first, Slogger is designed to perform the backup process off the client-facing critical path of the date store. As a result, it does not lead to any



(a) PSA failure

(b) BSA failure

(c) Watermark service failure

Figure 12: Slogger backup lag measured during failures of PSA, BSA, and watermark service.

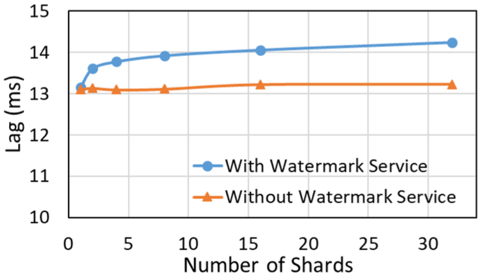


Figure 13: Watermark service scalability under write-only workload.

performance degradation. In incremental snapshotting, LogCabin is paused for a short duration to take a snapshot of the in-memory keys. Second, Slogger reduces the backup site lag by continuously shipping log records once they are committed. In contrast, in incremental snapshotting, updated objects are shipped all at once when the snapshot is taken which increases the lag of the backup site.

7.3 Watermark Service Scalability

The watermark service is, in a way, a central point of synchronization between all backup shards. Its scalability is paramount to the viability of Slogger. Figure 13 shows the lag's upper bound for different number of shards. We observe slight performance degradation as the watermark service gets increasing load; The upper lag of 32 shards is 4.6% higher than that of two shards. This indicates that watermark service can scale for large number of shards and it only imposes a moderate overhead.

However, in Figure 13 we observe a significant increase in the lag's upper bound for two shards compared to that of one shard. The reason for this degradation is that each shard's PSA sends committed log records to its BSA counterpart every 1 millisecond. Since PSAs of different shards are unsynchronized, this may result in increasing the lag by up to 1 millisecond. To better evaluate this synchronization cost, we present another performance curve in Figure 13 (Without Watermark Service). This curve depicts the lag in a variant of Slogger where BSAs do not coordinate with the watermark service. Instead, log records committed on the backup shard are immediately applied to it. The comparison is somewhat unfair since this variant does not guarantee correctness in the face of primary site disasters at arbitrary times. Nonetheless, it helps us get a better insight into understanding the cost of synchronizing multiple shards. Clearly, performance degradation of the lag's upper bound is largely dominated by the 1 millisecond delay that every PSA introduces before sending newly committed log records.

7.4 Fault Tolerance

As discussed in §5.5.4, when a shard leader for a primary (or backup) shard or the watermark service fails, the log backup process for the affected shard stalls immediately. This invariably stalls the global watermark's progress, which in turn stalls backup progress on other shards. Once failure is resolved, backup for the temporarily stalled shard continues, and Slogger quickly returns to a steady state.

Figure 12 depicts the behavior of Slogger when failures happen in a shard leader (PSA or BSA) or the watermark service. LogCabin is configured to detect replica failure at 200 millisecond granularity, after which it triggers a leader election cycle. Leader election takes about 1 millisecond, after which the leader spins up a PSA instance that establishes a connection with the corresponding BSA in 2 network round trips. After that, PSA resumes shipping log records to BSA. The overall recovery takes less than 300 milliseconds. In case of watermark service failure, we fail the watermark service for 500 milliseconds, after which a new watermark service instance is created, which quickly communicates with all backup shards and refreshes the global watermark.

As expected, in all cases, after a failure the observed lag at the backup shards grows steadily as time elapses. While the lag continues to grow, all the healthy primary shards continue to back up their log records to their healthy backup counterparts. In case of a primary shard's leader failure, the affected shard eventually gains a new leader that starts communicating with its backup counterpart. As a consequence, the lagging shard's timestamp is quickly updated to the latest time that is bigger than timestamps of almost all log records received in the interim by all other backup shards. This results in a quick and significant "bump" in the global watermark because of which the lag rapidly drops to the steady state observed before the failure. In case of a backup shard's leader failure, the log records committed at the corresponding primary shard's leader are all shipped by its PSA to the new backup leader's BSA *en mass*, which also results in a quick bump in the global watermark. Similar reasoning holds for watermark service failures.

7.5 Pipelining, Batching, and Log Size

A combination of pipelining and batching appears to be absolutely necessary for a backup shard to keep up with its primary counterpart – if the primary shard produces log records at a rate greater than the rate at which the backup shard receives and absorbs them, the primary shards will have to stall and stop processing client requests. Clearly, owing to the much longer communication latency between primary and backup shards, shipping one log record at a time and waiting for its ack before shipping the next one is insufficient for the backup shard to keep up with the primary shard. What we need is the right combination of pipelining and batching that yields a shard backup rate that matches with the maximum rate at which the primary shard can create log records.

Assuming a maximum sustained rate at a primary shard of N ops/millisecond, with $1KB$ of space per log record, we need to find the right mix of pipelining and batching that can help the backup shard keep up with its primary counterpart. (We assume similar hardware resources available at the backup and primary site – CPUs, memory, network bandwidth.) Clearly, sending a batch of N requests every millisecond will match the production rate at the primary shard.

Note however, that the primary shard must hold on to a shipped

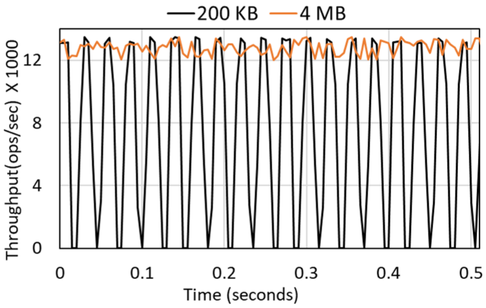


Figure 14: Pipelining, batching, and log size interplay.

log record until it receives an ack from the backup shard confirming that the log record has been replicated and committed. In our experimental setup, the upper bound for the typical lag ($t6 - t2$ from Figure 10) averages to under 15 milliseconds (§7.2). We need to approximately double that time to get a conservative estimate of the round-trip time – 30 milliseconds. Thus, to keep a sustained rate of N ops/millisecond, we need a pipeline depth that matches this round-trip delay. A pipeline with a depth of 30 slots, where each slot contains a batch of N log records or more is sufficient to help a backup shard keep up with its primary counterpart. Some overprovisioning is obviously needed to counter any jitters and spikes in the log record production rate at the primary shard.

In our experiments, each LogCabin shard processed client update requests at an approximate maximum rate of $13K$ ops/sec (13 ops/millisecond). A pipeline of 30 slots, with dynamic batching – grouping together all the newly committed log records in the shard's log – turned out to be sufficient. Dynamic batching typically yielded a batch size of $12-14$ log records in our experiments.

The next question to address is: what is the minimum log capacity required to sustain the above log backup rate? In our experiments, the above constraints lead to a stream of around 400 in-flight log records in the pipeline between the primary and backup shards. This amounts to $400KB$ of sustained bandwidth consumption in the 30 millisecond window. Thus, in the absence of jitters, a log of size of $400KB$ should be sufficient to help the backup keep up with the primary. In practice however, occasional network and other jitters cause unexpected delays. We can compensate those delays by providing $10X$ the minimum required log capacity – $4MB$.

Figure 14 demonstrates the interplay between pipelining, batching, and the log size. In particular, it validates our above analysis by testing with different log sizes – a sufficiently big log (4 MB) will be able to tolerate intermittent failures and delays, but a smaller log (200 KB) will lead to significant interference. A log size of 200 KB can tolerate a RTT of at most 15 milliseconds, however, the RTT in our setup is around 27 milliseconds. Hence, the primary shard's log is filled up after 15 milliseconds. Consequently, it stalls and stops processing client requests until an ack is received from the backup shard after 27 milliseconds resulting in freeing the primary shard's log. Thereafter, the primary shard starts processing client requests again. This behavior repeats every 27 milliseconds (RTT).

7.6 Disaster Recovery

Slogger has been designed to enable rapid failover to the backup site after a disaster is declared at a distributed data store's primary site. We measured the latency of this failover mechanism. More specifically, we measured the interval between the event when the backup site (watermark service, in our implementation) receives a RECOVERY message from the data store administrator, and the event when the watermark service receives recovery completion messages from all the backup shards. The backup site does not accept new client connections during this interval, which is observed as down-time by all clients. This "failover readiness" process happened in less than 7 milliseconds. This happens primarily because the backup site is continuously applying log records it receives from the primary site during normal execution. Thus the amount of log records needed to be applied in the backup shards remain relatively small (at most 45 KB in our experiments).

8. RELATED WORK

Snapshots. Several projects [18, 20, 33, 40, 48] use snapshots to maintain a mirror replica in a remote data center. In these solutions, a snapshot of the primary data store is asynchronously generated and then transferred to the remote replica. SnapMirror [40] exploits the active block maps in WAFL file system to identify modified blocks and avoid sending deleted ones, which reduces the amount of data transferred to the remote replica. The main disadvantage of snapshoting, as our evaluation demonstrates, is the large data loss window, which might range from seconds to hours or days.

Remote Mirroring. Numerous commercial products provide DR to disk volumes such as IBM Extended Remote Copy [10], HP XP arrays [6], and EMC SRDF [9]. In remote mirroring, a backup copy of the primary volume is maintained in a remote location. Typically, these solutions supports multiple replication protocols with various reliability guarantees. For instance, HP XP arrays can operate in both synchronous and asynchronous mode. These solutions do not preserve causality between writes to different volumes as the mirroring of different volumes is not coordinated.

Continuous Backups. Another class of solutions to DR is *continuous backups*, where the primary data store is continuously, asynchronously, and incrementally replicated to the backup site. Inadvertantly, these prior works asynchronously ship log records generated by the primary data store to the backup site [4, 11, 21, 22, 30, 35, 36, 44]. While this approach can be used to drastically reduce the data loss window for a data store, we note that all existing solutions apply to data stores that either support *serializability* [39] or more relaxed consistency models. None of these solutions work correctly for *linearizable* [16] distributed data stores [8]. In particular, we show that the order in which updates are backed up, using prior techniques, may lead to an update order at the backup site that is inconsistent with the update order observed at the primary.

9. CONCLUSION

We presented Slogger, the first asynchronous DR system that provides correct continuous backup of a linearizable distributed data store with a tiny, millisecond scale, data loss window. Slogger backs up the data store continuously by asynchronously shipping log records of individual shards to the backup site. Slogger leverages synchronized distributed clocks to enable the backup site to apply distributed log records in the correct order. It strategically employs techniques such as concurrency, pipelining, batching and a novel watermark service based coarse synchronization mechanism to create a viable backup system for distributed data stores. Empirical evaluation done using LogCabin demonstrates Slogger's capability to deliver a millisecond range backup site lag (14.20 in our experiments), along with several other desirable characteristics such as high scalability, fault tolerance, and efficient "failover readiness" (less than 7 milliseconds in our experiments).

Acknowledgments

We thank the anonymous reviewers for their insightful feedback that improved the paper. We thank Matteo Frigo, Dan Nussbaum, and Michael Frasca for early discussions and feedback on ideas in this paper.

10. REFERENCES

[1] Aerospike: Strong Consistency mode. *https://www.aerospike.com/docs/architecture/consistency.html*.

[2] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[3] Apache Cassandra. *http://cassandra.apache.org/*.

[4] Ann L. Chervenak, Vivekanand Vellanki, and Zachary Kurmas. protecting file systems: A survey of backup techniques.

[5] CockroachDB. *https://github.com/cockroachdb/cockroach*.

[6] Hewlett-Packard Company. *HP StorageWorks Business Copy XP user's guide*.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 251–264, 2012.

[9] EMC Corporation. *EMC Symmetrix Remote Data Facility (SRDF) Product Guide*.

[10] IBM Corporation. *DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference*.

[11] Oracle Corporation. Disaster Recovery for Oracle Database, 2015.

[12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414, 2014.

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pages 1–14, 2019.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, pages 81–94, 2018.

[15] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *CoRR*, cs.DC/0408036, 2004.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[17] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, pages 629–647, 2017.

[18] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference,*, pages 253–268, 2003.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 185–201, 2016.

[20] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 5–5, 2004.

[21] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Overview of disaster recovery for transaction processing systems. In *10th International Conference on Distributed Computing Systems*, pages 286–293, 1990.

[22] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[23] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *2019 USENIX Annual Technical Conference*, 2019.

[24] Leslie Lamport. The TLA+ Home Page. *https://lamport.azurewebsites.net/tla/tla.html*.

[25] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[26] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[27] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] Kang Lee and John Eidson. IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *In 34th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 98–105, 2002.

[29] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[30] David B. Lomet. High speed on-line backup when using logical log operations. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 34–45, 2000.

[31] David L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[32] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[33] C. Mohan and Inderpal Narang. An efficient and flexible method for archiving a data base. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 139–146, 1993.

[34] Pedro Moreira, Javier Serrano, Tomasz Wlostowski, Patrick Loschmidt, and Georg Gaderer. White rabbit: Sub-nanosecond timing distribution over ethernet. pages 1 – 5, 11 2009.

[35] MySQL Documentation: Making a Differential or Incremental Backup. *https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/*.

[36] NetApp: SnapCenter 3.0 Archived Documentation. *http://docs.netapp.com/ocsc-30/index.jsp?topic=%2Fcom.netapp.doc.df-cloud-bu-wg-cli%2FGUID-BD75E784-EDC0-4875-B6F4-ED40E000ECE2.html*.

[37] Diego Ongaro. LogCabin. *https://github.com/logcabin/logcabin*.

[38] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320, 2014.

[39] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[40] R. 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*, 2002.

[41] Riak KV. *https://riak.com*.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 1–15, 1991.

[43] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[44] Tail-Log Backups (SQL Server). *https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/tail-log-backups-sql-server?view=sql-server-ver15*.

[45] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 373–389, 2018.

[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[47] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, and H. Ahmed. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)*, pages 1–7, 2015.

[48] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and mirrors: Reflecting files at a geographically remote location without loss of performance. In *Proccedings of the 7th Conference on File and Storage Technologies*, pages 211–224, 2009.

[49] Lidong Zhou, Vijayan Prabhakaran, Rama Ramasubramanian, Roy Levin, and Chandu Thekkath. Graceful degradation via versions: specifications and implementations. In *Symposium on Principles of Distributed Computing (PODC 2007)*, August 2007.