Abstract

Alternative History: Multi-Log Systems in Theory and Practice

Joshua Z. Lockerman

2019

Large-scale datacenter systems rely on stateful control plane services. Distributing state for such services can be difficult; idiosyncratic interfaces and performance requirements precludes the use of general, fixed API storage systems, while custom solutions are often complicated and difficult to debug and maintain. By funneling all updates through a single, durable, totally ordered log, Shared Log State Machine Replication can make arbitrary state available, durable, and strongly consistent among a number of machines using a single service. The log abstraction hides the complexity of asynchrony and failures from applications, allowing them to be built as simple state machines that process updates in strict sequence. However, this simplicity comes at a cost to speed and flexibility by imposing a system-wide total order that is expensive, often impossible, and typically unnecessary. Furthermore, the State Machine Replication paradigm itself functions as a broadcast domain, forcing servers to see more state than is actually useful. This work investigates the implications of weakening the Shared Log State Machine Replication semantics. By guaranteeing a partial ordering of updates instead of a total order, the FuzzyLog achieves linear scaling with transaction support, weak consistency models, and progress during network partitions. By allowing servers to filter which updates they see within the context of an object, FuzzyViews enable greater privacy and read performance.

Alternative History:

Multi-Log Systems in Theory and Practice

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Joshua Z. Lockerman

Dissertation Director: Mahesh Balakrishnan

December 2018

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisors Zhong Shao and Mahesh Balakrishnan—none of this would have been possible without all they have taught me, and their continued mentorship, support and guidance. I appreciate all you have done for me. I would also like to thank Dan Abadi and Sid Sen; their advice and friendship were critical for seeing me through the PhD process. In particular, Dan introduced me and Mahesh, without which this project would never have happened. The FuzzyLog involved collaborating with many fine researchers. Besides the ones mentioned earlier, I'd like thank James Aspnes, Jose Falerio, Juno Kim, and Soham Sankaran for their part in the project. I would also like to thank Sue Hurlburt for assistance in navigating Yale. Outside of research, I would like to thank my brothers Yitzchak and Eli, my sister-in-law Rachel, my parents, and especially my fiancee Riva Tropp; each of you has helped me in your own way.

This work was sponsored by the NSF under grants IIS-1637385 and IIS-1718581.

Terms and Abbreviations

CAPMap A FuzzyLog-based best-effort key value store that provides linearizability during normal execution and causal consistency during network partitions.

Chain A totally ordered subset of a partial order. In FuzzyLog, a totally ordered sublog of the partially ordered FuzzyLog.

DAG Abbreviation of "directed acyclic graph".

Dapple The implementation of the FuzzyLog used for experiments in this thesis.

Event An element within an objects history. Synonym for update.

FuzzyLog The partially ordered shared log abstraction that serves as a focus for this work.

FuzzyView A State Machine Replication object created based on a subsequence of updates from an underlying object.

Object The result of applying a history into some form better suited for performing queries.

Shard A subset of data from a distributed system which is co-located.

SMR Abbreviation of State Machine Replication.

State Machine Replication A technique in which a object is replicated across multiple machines via broadcasting the events which altered the object [1].

Update An element within an object's history. Synonym for event.

Chapter 1

Introduction

1.1 The FuzzyLog

Large-scale datacenter systems rely on control plane services such as filesystem namenodes, SDN controllers, coordination services, and schedulers. Such services are often initially built as single-server systems that store state in local in-memory data structures. Properties needed to achieve performance at scale, such as durability, high availability, and scalability are then retrofitted by distributing service state across machines. Distributing state for such services can be difficult; their requirement for low latency and high responsiveness precludes the use of external storage services with fixed APIs such as key-value stores. Custom solutions can require melding application code with a medley of distributed protocols such as Paxos [2] and Two-Phase Commit (2PC) [3], which are individually complex, slow and inefficient when layered, and difficult to merge [4, 5].

A recently proposed abstraction, the *distributed shared log*, simplifies this. Shared logs are an extension of State Machine Replication (SMR) in which all updates to the distributed state are funneled through a single, durable, globally-shared log. This paradigm has been used to construct fault-tolerant databases [6–10], metadata and coordination services [11, 12], key-value and object stores [13–15], and filesystem namespaces [16, 17]. Services built over a shared log are simple, compact layers that map a high-level API to append/read operations on the shared log. The log acts as the source of strong consistency, durability, failure atomicity, and transactional isolation. This layering vastly simplifies the construction

of distributed systems. For example, a shared log version of ZooKeeper uses 1K lines of code, an order of magnitude lower than the original system [11].

Despite the usefulness of object history as the underlying abstraction, the simplicity of a shared log requires imposing a system-wide total order that is *expensive*, often *impossible*, and typically *unnecessary*. Previous work has shown that a centralized, off-path sequencer can make such a total order feasible at intermediate scale (e.g., a small cluster of tens of machines) [11, 18]. However, at larger scale—in the dimensions of system size, throughput, and network bandwidth/latency—imposing a total order becomes expensive: ordering all updates via a sequencer limits throughput and slows down operations if machines are scattered across the network. In addition, for deployments that span geographic regions, a total order may be impossible: a network partition can cut off clients from the sequencer or a required quorum of the servers implementing the log. On the flip side, a total order is often unnecessary: updates that manipulate disjoint data, such as different keys in a map, do not need to be ordered, while updates that touch the same data may be able to commute because the application allows weak consistency guarantees (e.g., causal consistency [19]), or due to the inherent commutativity of the update types. We set out to explore the following question: *can we provide the simplicity of a shared log without imposing a total order?*

To answer this question, we propose the FuzzyLog abstraction: a durable, iterable, and extendable order over updates in a distributed system. Crucially, a FuzzyLog provides a *partial order* as opposed to the total order of a conventional shared log. The FuzzyLog is a directed acyclic graph (DAG) of nodes representing updates to a sharded, geo-replicated system (see Figure 1.1). The FuzzyLog materializes a happens-after relation between updates: an edge from A to B means that A must execute after B.

The FuzzyLog captures two sources of partial ordering in distributed systems: data sharding and geo-replication. Nodes in the FuzzyLog are organized into *colors*, where each color contains updates to a single application-level data shard. A color is a set of independent, totally ordered *chains*, where each chain contains updates originating in a single geographical region. Chains within a color are connected by cross-links that represent update causality. The entire DAG—consisting of multiple colors (one per shard) and chains within each color (one per region)—is fully replicated at every geographic region and is

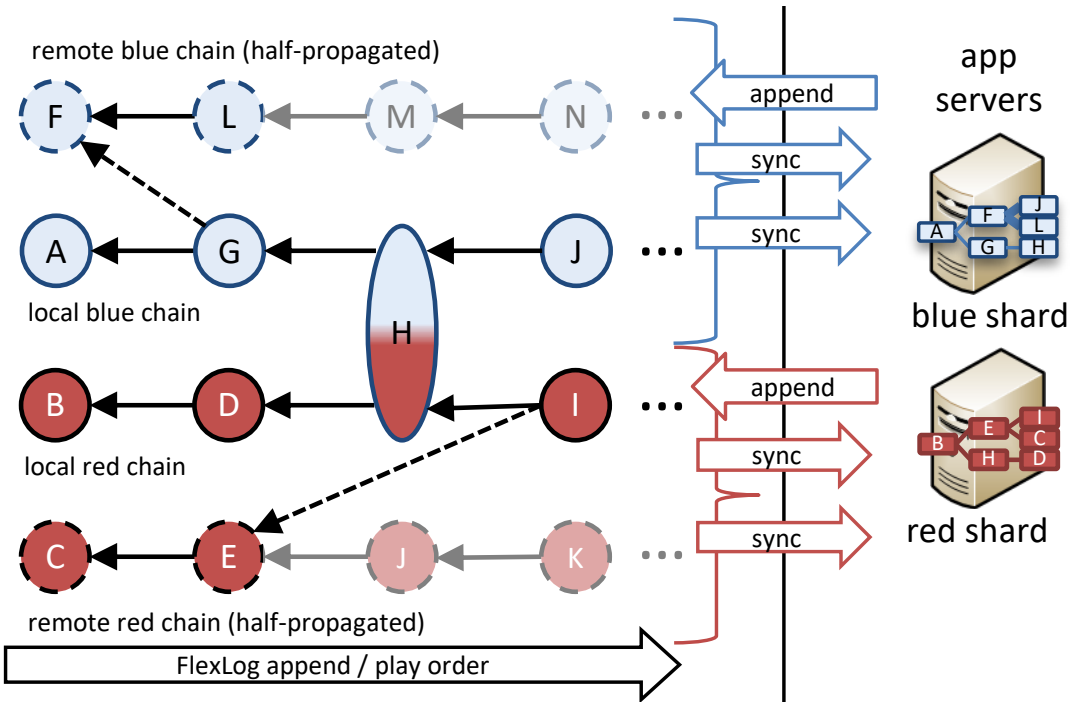


Figure 1.1: In the FuzzyLog, each color is composed of nodes, which in turn contain updates to a data shard. The colors are organized into chains each of which contains updates from a geographical region. App servers materialize this state into a form more appropriate for querying.

lazily synchronized, so that each region has the latest copy of its own chain, and some stale prefix of the chains of other regions. Figure 1.1 shows a FuzzyLog deployment with two data shards (i.e,. two colors) and two regions (i.e., two chains per color).

The FuzzyLog API is simple: a client can *append* a new node by providing a payload describing an update and the color of the shard it modifies. The new node is added to the tail of the local chain for that color, with outgoing links to the last node seen by the client in each remote chain for the color. The client can *synchronize* with a single color, playing it forward by applying new nodes from the local region's copy of that color in a reverse topological sort order of the DAG. A node can be appended atomically to multiple colors, representing a transactional update across data shards.

Applications built over the FuzzyLog API are nearly as simple as conventional shared log systems. As shown in Figure 1.1, FuzzyLog clients are application servers that maintain in-memory copies or views of shared objects. To perform an operation on an object, the application appends an entry to the FuzzyLog describing the mutation; the application

server then plays the FuzzyLog, retrieving new entries from other clients and applying them to its local view until it encounters and executes the appended entry. The local views on the application servers constitute soft state that can be reconstructed by replaying the FuzzyLog. A FuzzyLog application that uses only a single color for its updates and runs within a single region is identical to its shared log counterpart [11, 18]; the FuzzyLog degenerates to a totally ordered shared log, and the simple protocol described above provides linearizability [20], durability, and failure atomicity for application state.

By simply marking each update with colors corresponding to data shards, FuzzyLog applications achieve scalability and availability. They can use a color per shard to scale linearly within a data center; transactionally update multiple shards via multi-color appends; obtain causal consistency [19] within a shard by using a color across regions; and toggle between strong and weak consistency when a network partition occurs by switching between regions.

Implementing the FuzzyLog abstraction in a scalable and efficient manner requires a markedly different design from existing shared log systems. We describe Dapple, a system that realizes the FuzzyLog API over a collection of in-memory storage servers. Dapple scales throughput linearly by storing each color on a different replica set of servers, so that appends to a single color execute in a single phase, while appends that span colors execute in two phases (in the absence of failures) that only involve the respective replica sets. Dapple achieves this via a new fault-tolerant ordering algorithm that provides linear scaling for single-color appends, serializable isolation for multi-color appends, and failure atomicity. Across regions, a lazy synchronization protocol propagates each color's local chain to remote regions.

We implemented a number of applications over the FuzzyLog abstraction and evaluated them over Dapple. AtomicMap (201 lines of C++) is a linearizable, durable map that supports atomic cross-shard multi-puts, scaling to over 5.5M puts/sec and nearly 1M 2-key multi-puts/sec on a 16-server Dapple deployment. CRDTMap (284 LOC) provides causal+ consistency [21] by layering a CRDT over the FuzzyLog. CAPMap (424 LOC) offers strong consistency in the absence of network partitions, but degenerates to causal+ consistency during partitions. We implemented a ZooKeeper clone over the FuzzyLog in 1881 LOC

that supports linear scaling across shards and supports atomic cross-shard renames. We also implemented maps that implemented Red-Blue consistency [22] and a transactional CRDT [23].

Existing implementations of these applications are monolithic and complex; they often re-implement common mechanisms for storing, propagating, and ordering updates (such as protocols for atomic commit, consensus, and causality tracking). The FuzzyLog implements this common machinery efficiently under an explicit abstraction, hiding the details of protocol implementation while giving applications fine-grained control over sharding and geo-replication. As a result, applications can express different ordering requirements via simple invocations on the FuzzyLog API without implementing low-level distributed protocols.

Contributions: We propose the novel abstraction of a FuzzyLog (§2): a durable, iterable DAG of colored nodes representing the partial order of updates in a distributed system. We argue that this abstraction is *useful* (§2.1), describing and evaluating application designs that obtain the simplicity of the shared log approach while scaling linearly with atomicity, obtaining weaker consistency, and tolerating network partitions. We show that the abstraction is *feasible in practice* (§5.1), describing and evaluating a performant, scalable, fault-tolerant implementation called Dapple.

1.2 FuzzyViews

In this thesis we also propose the abstraction of a FuzzyView: a view of state materialized from an arbitrary subsequence of a total order of updates. The FuzzyView enables *selective* SMR, where each server reflects a selection of past updates rather than the entire total order. An implication of selective SMR is diversity: each server can reflect a potentially different view over the same total order. We explore multiple applications of selective SMR:

- *Function accelerators*, where a server efficiently implements a narrow subset of application functionality. For example, in a replicated map, a server might be dedicated to providing fast key existence checks, for which it only needs to see updates that insert and remove keys, not changes to existing keys.

- *Approximate views*, where the state at a server is consistent but approximate. Applications can use approximate views to trade precision for performance while still providing strong consistency. For example, an approximate view for a counter can provide a trade-off between error and response latency.

- *Privacy-aware applications*, where the state at each server satisfies some level of privacy. For example, consider a location-tracking application where each update is a check-in (i.e., the geographical location of a person at a particular time), and the materialized state is a heat-map of such check-ins. In such an application, each server might store check-ins that occur within some geographical area (e.g., near work or near home) or within some time period (e.g., work hours vs. non-work hours).

- *Ensemble learning*, where the state at each server is a different machine learning model over the same set of updates. For example, different models can be constructed from updates tied to users with different characteristics (e.g., whether the user is logged-in or not). The resulting diversity of the ensemble can result in higher accuracy.

The FuzzyView abstraction ties together the concepts of sharding, weak consistency, materialized views [24], and learners in SMR systems.

1.3 Background

Every distributed system generates and manipulates a history in some manner, with varying degrees of transparency. At one extreme are abstractions for explicitly ordering updates. These system have a long history, ranging from Virtual Synchrony [25, 26], State Machine Replication [1], Viewstamp Replication [27], Multi-Paxos [28], and newer approaches such as Raft [29]. Many of these impose a total order on updates; the exceptions track particular partial orders imposed by operation commutativity (pessimistically [30, 31] and optimistically [32]), causal consistency (as in Virtual Synchrony and Lazy Replication [33]), network partitions (as in Extended Virtual Synchrony [34]), or value independence .

At the other extreme are transaction processing systems. In these systems, the transaction history not exposed at all to the user, rather it is implicitly stored and manipulated to

allow for efficient transaction scheduling [35], generally with the intent of allowing as many transactions as possible to commit concurrently. Though the history of such systems is not directly exposed, an understanding of it is important for determining what the outcome of a set of transactions will be. Recent work has investigated the benefits from optimizing based on this implicit history; leading to renewed interest in such techniques as transaction chopping [36–39] wherein a transaction is replaced with multiple ones in a manner which maintains the valid schedules. Such techniques lower the contention between transactions, increasing the throughput of the system, while maintaining its semantics. Meanwhile, techniques such as the scalable commutativity rule [40] and I-Confluence [41] have investigated restricting APIs to a form that is more amenable to concurrent scheduling.

A number of systems provide weaker consistency by removing ordering that a naive user would expect the object's API to provide. COPS [21] and Eiger [42] provide causal consistency in a partitioned store, while Bayou allows for disconnected updates and eventual reconciliation [43, 44]. TARDiS [45] exposes *branch-on-conflict* as an abstraction in a fully replicated, multi-master store. Though such systems are too diverse to allow for many generalizations, they all share one key feature: they allow for different views to have diverging state, and only be reconciled at a later point. Although this divergence can make programmer's lives more difficult [21], it is critical to these system's main use-case: allowing progress to be made with little to no communication. While some systems, such as RedBlue [22], attempt to restrict the set of operations allowed, so that each machine returns answers consistent with the state of the system as a whole, many of these systems simply allow results inconsistent with any serializable order, and simply reconcile differences after the fact [21, 46–48].

The FuzzyLog is an outgrowth of the State Machine Replication [1] paradigm. State Machine Replication (SMR) provides a simple, transparent way to replicate arbitrary state, called an object, in a distributed system. The object replicas are kept in sync by funneling all updates through a total order; to use Lamport's terminology, the total order is stored durably on *acceptor* servers, while the replicas of the object are stored on *learner* servers.

A key aspect of the SMR paradigm is that the views of the objects found on the learners are simply soft-state caches; durability is derived from the acceptor servers. Accordingly,

the state of an object is effectively the total order of updates stored at acceptors, and one or more materialized soft-state copies stored at the learners. In real-world systems, the total order is garbage collected periodically; when this happens, the burden of durability shifts to some other location that stores checkpoints of the materialized state.

An object is mutated by appending an update to the total order. To read the object with linearizable semantics, a learner must play the total order forward, applying new updates to its materialized state. We call the total order the *history* of the object; the materialized copies are *views*.

Accordingly, current SMR systems can be seen as having two halves. The top half is responsible for materializing the views, and translating the arbitrary object APIs to and from a form that can be stored durably, and interpreted by other machines. The bottom half is responsible for realizing the total order. Systems currently implement this bottom half via various abstractions:

- Multi-Paxos [28] systems typically realize the total order as an address space of numbered single-shot Paxos instances.
- Group communication [25] systems implement the total order via a group broadcast/-multicast abstraction that delivers messages to receivers in a total order.
- Shared log [11, 18] systems obtain total ordering via a shared log abstraction with an append/read API.

Our system extends on this model. We relax the ordering requirement to allow for partially ordered system histories; *within* an object we do this to enable weakened consistency, progress during network partitions, and low-latency geo-replication; *between* objects, we relax ordering to improve throughput in sharded scenarios. Further, we examine the implications of relaxing materialization requirements, allowing views to be constructed from part of an object's history. Such FuzzyViews enable a tradeoff between completeness, performance, and privacy, and suggest an even richer event-oriented perspective to come.

Other systems improve performance not by altering the consistency of their queries, but by relaxing their answers along some other dimension. Approximate databases, such

as BlinkDB [49], extend a SQL-based database interface with the ability to specify queries over samples of data. Such sample-based queries achieve significant speedup by sacrificing exact answers in favor of ones with only statistical guarantees. In many applications such approximate answers are all that are necessary (and in some domains, all that are possible due to measurement error), resulting in a worthwhile trade-off.

Our FuzzyViews exist in a similar domain, and can be used to provide statistical queries, as in our sampled counter, though the lack of a convenient way to generate a new independent sample prevents them from being a strict generalization. However, unlike statistical mechanisms which rely on a quantitative data model and can only provide probabilistic guarantees, SMR and FuzzyViews can work with arbitrary data models and provide absolute guarantees.

Chapter 2

The FuzzyLog Abstraction

A FuzzyLog is a type of directed acyclic graph (DAG) that can be constructed and traversed concurrently by multiple clients. The nodes in this graph are used to store events, and the graph's edges are used to represent dependency information. For clarity, in the remainder of this thesis we will use the term 'node' exclusively to refer to nodes in the FuzzyLog DAG and not for other uses like nodes in a distributed system.

Each node in the DAG is tagged with one or more colors; these colors divide an application's state into logical data-shards. Nodes tagged with a particular color correspond to updates against the corresponding data shard.

Each color is a set of totally ordered sub-logs called "chains"; there is one chain per region, with cross-edges between chains to indicate causality. Every region has a full but potentially stale copy of each color; the region's copy has the latest updates of its own chain for the color, but stale prefixes of the other per-region chains for that color. Clients only interact with their own region's local copy of the DAG; they can modify this copy by

```
//constructs a new handle for playing a color
FL_ptr new_instance(colorID color, snapID snap=NULL);
//appends a node to a set of colors
int append(FL_ptr handle, char *buf, size_t bufsize, colorset *nodecolors);
//synchronizes with the log
snapId sync(FL_ptr handle, void (*callback)(char *buf, size_t bufsize));
//trims the color
int trim(FL_ptr handle, snapID snap);
```

Figure 2.1: The FuzzyLog API.

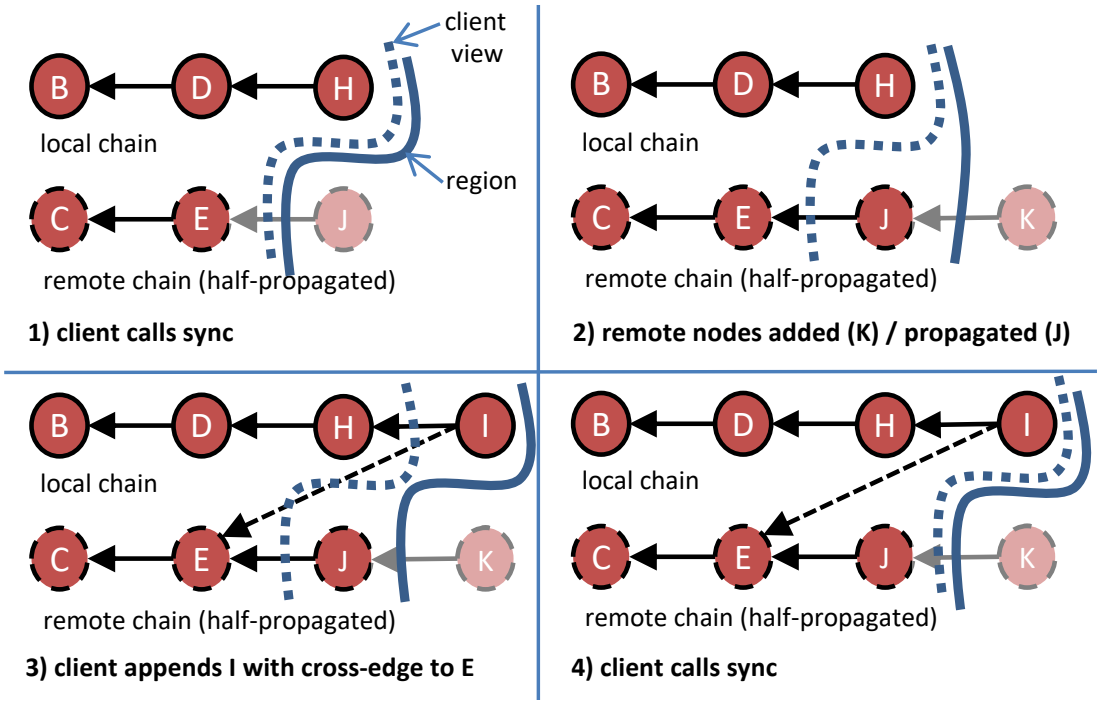


Figure 2.2: The evolution of a single color.

appending to their own region's chain for a color, and can play-back the DAG by reading from their color's chains in a procedure described later.

Figure 2.1 shows the FuzzyLog API. A client creates an instance of the FuzzyLog with the `new_instance` call, supplying a single color to play forward. It can play nodes of this color with the `sync` call, and it can `append` a node to a set of colors. We will first describe the operation of these calls in a FuzzyLog deployment with a single color (i.e., an application with a single data shard), and later describe how this is extended to multiple shards.

The `sync` call is used by the client to synchronize its state with the FuzzyLog. A `sync` takes a snapshot of the set of nodes currently present at the local region's copy of a color, and plays all new nodes that have been added since the last `sync` invocation. Once all of these new nodes have been provided to the application via the passed-in callback, the `sync` returns with an opaque ID describing the snapshot. The nodes returned by `sync` are seen in a reverse topological sort order of the DAG: nodes in each chain are seen in the reverse order of edges within the chain, and nodes in different chains are seen in an order that respects cross-edges. Nodes in different chains that are not ordered by cross-edges can be seen in any order. Each node effectively describes a list of prior nodes – via its position

in a totally ordered chain, and via explicit pointers for cross-edges. Figure 2.2 shows the evolution of a client's view of a color. In panel 1 the client synchronizes with the color; in panels 2 and 3 the client trails behind as new nodes are replicated into the local region from afar; and in panel 4 the client synchronizes once again, catching up. After each of the `sync`s the client receives a Snapshot ID. These IDs can be used by clients to check if the set of nodes seen by one client subsumes the set of nodes seen by another.

When a client appends a node to a color with `append`, an entry is inserted into the local region's chain for that color. The entry becomes the new "tail" of the chain, and it has an edge in the DAG pointing to the previous tail; we define the tail as the only node in a non-empty chain with no incoming edge. In this manner, the local region chain imposes a total order over all updates generated at that region. The node also has outgoing cross-edges pointing to the last node played by the client from every other per-region chain for the color; in effect, the newly appended node is ordered after every node of that color is seen by the client. For example, in Figure 2.2 panel 3, a client appends a new node I to the region's local chain (after node H), with a cross-edge to E, which is the latest node in the remote chain seen by the client.

To garbage collect the FuzzyLog, clients can call `trim` with a snapshot ID, indicating that the nodes in it are no longer required (because e.g., the client stored the corresponding materialized view in a durable external store). A snapshot ID can also be provided to the `new_instance` call, in which case playback skips nodes within the snapshot; this allows a new client to join the system without playing the FuzzyLog from the beginning.

While the `sync` and `trim` calls operate exclusively over a single color, the FuzzyLog supports appending to multiple colors at once: an `append` to a set of colors atomically appends the entry to the local chains for each color. The new node is then reflected by `sync` calls on any one of the colors involved. If a node is in multiple colors, trimming it in one color does not remove it from the other colors it belongs to, the node must be trimmed from every color it inhabits individually.

Semantics: Operations within a single region are serializable. All `append` and `sync` operations issued by clients within a region execute in a manner consistent with some serial execution. This serialization order is linearizable if the operations are to a single color

within the region (i.e., on a single chain); it does not necessarily respect real-time ordering when `append` operations span multiple colors.

Operations to a single color are causally consistent across regions. In other words, two `append` operations to the same color issued by clients in different regions are only ordered if the node introduced by one of them has already been seen by the client issuing the other one. In this case, an edge exists in the DAG from the second node to the first one. The internal structure of the DAG ensures that the copies at each region converge even though concurrent updates can be applied in different orders at them: since the clients at each region modify a disjoint part of the DAG (i.e., they append to their own per-region chain), there are never any conflicts when the copies are synchronized.

2.1 FuzzyLog Applications

This section describes how applications can use the FuzzyLog API, with a case study of an in-memory key-value storage service. In this section, the term 'server' refers exclusively to application servers storing in-memory copies of the key-value map, which in turn are FuzzyLog clients. We start with a simple design called LogMap that runs over a single color within a single region (i.e., it effectively runs over a single totally ordered shared log). Each LogMap server has a local in-memory copy of the map and supports `put`/`get`/`delete` operations on keys. The server continuously executes a `sync` on the log in the background, applying updates to keep its local view up-to-date. A `get` operation at the server simply waits for a `sync` that started after the `get` was issued. Once this `sync` completes, the `get` accesses the local view and returns. Waiting for a latter `sync` in this manner ensures that any updates that were appended to the FuzzyLog before the `get` was issued are reflected in the local view at the time the `get` accesses it, providing linearizability. A `put/delete` operation appends a node to the FuzzyLog describing the update; it then waits for a `sync` to apply the update to the local view, at which point it returns.

This basic LogMap design – implemented in just 193 lines of code – enables durability, high availability, strong consistency, concurrency control and failure atomicity. It is identical to previously described designs [18] over a conventional shared log, however, and its reliance

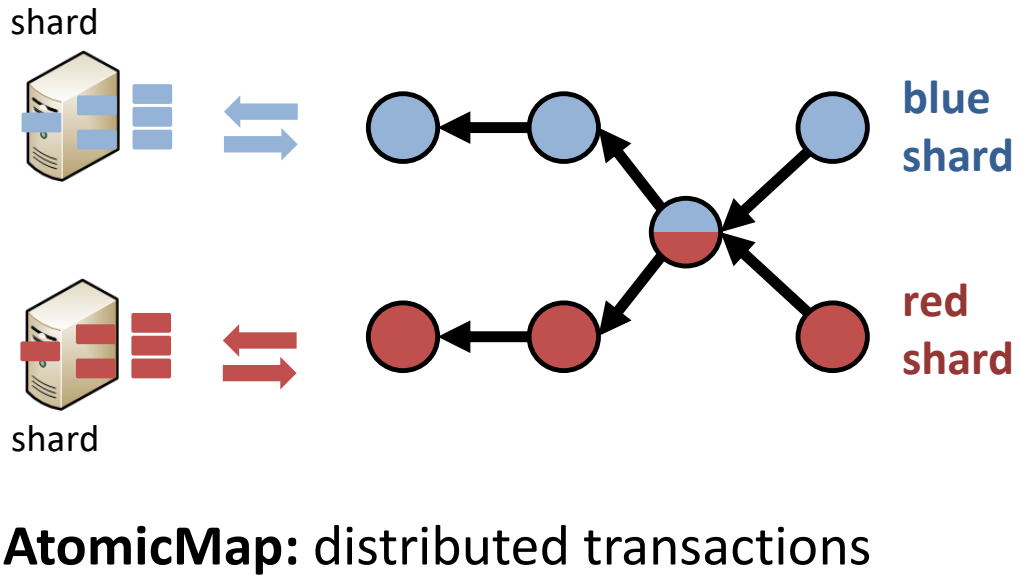


Figure 2.3: FuzzyLog capabilities: AtomicMap scale with the number of servers while still allowing multi-server transactions.

on a single total order comes at the cost of scalability, performance, and availability. The remainder of this section describes how LogMap can be modified to use the FuzzyLog to circumvent each of these limitations.

2.1.1 Scaling with atomicity within a region

We first describe applications that run within a single region and need to scale linearly. In ShardedMap (193 LOC), each server stores a shard of the map; each shard corresponds to a FuzzyLog color. Updates to a particular shard are appended to the FuzzyLog as nodes of the corresponding color; each server syncs its local state with the color of its shard. This simple change to LogMap – requiring only that the color parameter is set appropriately on calls to the FuzzyLog – provides linear scalability for linearizable `put`/`get` operations.

The FuzzyLog supports atomicity across shards. If the atomic operation required is a simple blind `multi-put`, that doesn't return a value, all we require is that the appends update a set of colors instead of a single one. This simple change allows a server to modify multiple shards at one, one shard for each of the colors the server appends to. AtomicMap (201 LOC, Figure 2.3) realizes this design. One subtle point is that since FuzzyLog multi-

color appends are serializable, AtomicMap is also serializable, but not strictly serializable nor linearizable.

To implement read/write transactions with stronger isolation levels, we use a simple two-phase commit [3] variant. To commit, the server appends an intention node into the FuzzyLog to the set of colors corresponding to the shards being read and written. When a server encounters the intention node in the color it's playing, it appends a second node with a yes/no decision and read-set for that color, to the set of colors. To generate this decision, the server examines the sub-part of the transaction touching its own shard and independently (but deterministically) validates it (e.g. checking for constraint violations). A server only applies the transaction to its local state if it encounters both the original intention and a decision marked yes for each color involved, blocking conflicting operations until it can determine if the transaction commits or aborts.

Interestingly, this protocol provides strict serializability even though the FuzzyLog itself is only serializable. Intuitively, within a single color the intention-nodes serialize transactions: if, after appending an intention-node for a transaction T, a client waits until it plays the node before declaring the transaction complete, the client is guaranteed to have seen all transactions—that could appear earlier than T in the serial order—before declaring that T itself is completed. As a result, future transactions must appear later in the serial order, ensuring strict serializability. In a multi-color transaction we need to ensure that this applies to all the involved colors: the client must have seen all transactions, in all the colors involved, that could appear before T. A decision node conveys this information: for a decision node to be written at a color, all earlier transactions must be completed. Thus, as in Tango [11], our protocol requires at least one application server to be available for each shard in order to generate decision records, to ensure progress.

2.1.2 Weaker consistency across regions

Applications can often tolerate weaker consistency guarantees. One example is causal consistency [19], which roughly requires the following: if a server performs an update U_1 after having seen an update U_0, then any other server in the system must see U_0 before U_1. If U_1 and U_2 were performed independently by servers that did not see each other's update,

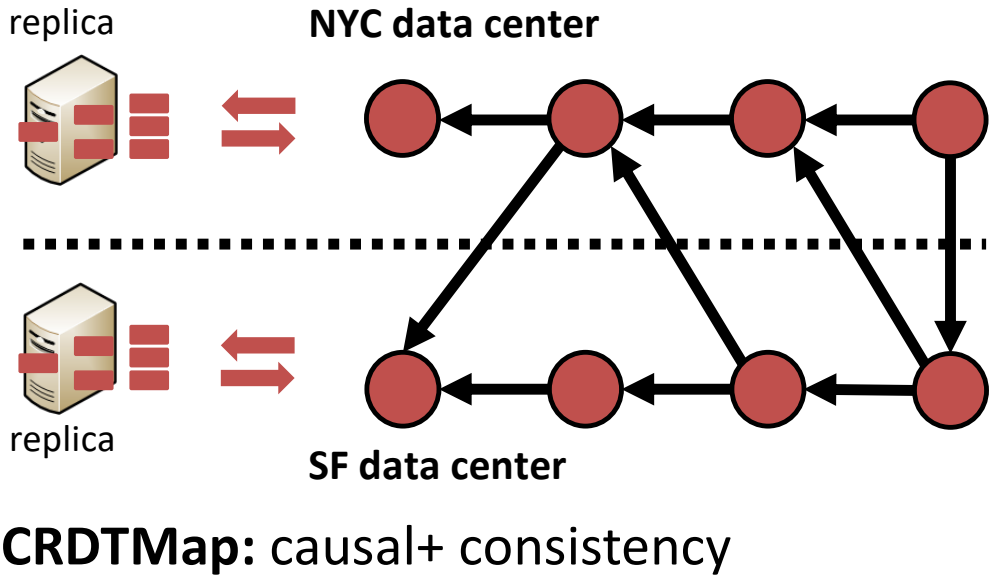


Figure 2.4: FuzzyLog capabilities: CRDTMap implements causal consistency, allowing low-latency across distant regions.

U_1 and U_2 can be seen in any order. Causal consistency is appealing as it is in some senses the strongest consistency level that can be provided during a network partition [21].

CRDTMap implements a causally consistent map. In Figure 2.4, the map is replicated across two regions, one in NYC and another in SF. CRDTMap simply uses a single color for all updates to a map. In each region, put operations are appended to the local chain for the color; these operations are asynchronously propagated to the other regions, ensuring all regions eventually see all of the updates to the map. Since the partial order within a color is exactly the causal order of updates, each server playing the color observes updates in a causally consistent order.

To achieve convergence when servers see causally-independent updates in different orders, we employ a design for CRDTMap based on the Observed-Remove Set CRDT [50], which exploits commutativity to ensure that concurrent updates result in the same final state, without requiring rollback logic even when the updates are seen in conflicting orders. The CRDT design achieves this by predicating the deletions performed by a server on non-deleting puts that the server has already seen; accordingly, each delete node in the DAG lists the put operations that it subsumes.

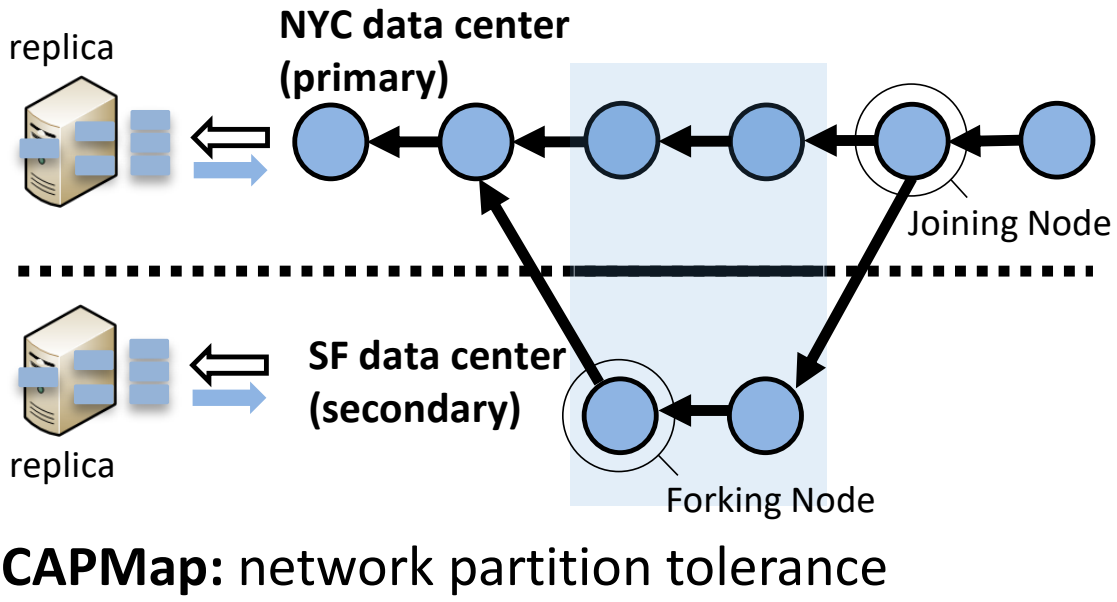


Figure 2.5: FuzzyLog capabilities: CAPMap provides linearizability in normal operation while falling back to causal consistency during a network partition.

2.1.3 Tolerating network partitions

While CRDTMap can provide availability during network partitions, it does so by sacrificing consistency even when there is no such partition in the system. CAPMap (named after the CAP conjecture [51]) provides strong consistency in the absence of network partitions and causal consistency during them (see Figure 2.5).

As with our other map designs, CAPMap appends entries on `put` operations and then syncs until it sees the appended node. Unlike the other designs, CAPMap requires servers to communicate with each other, albeit in a simple way: servers route FuzzyLog appends through proxies in other regions. To perform a `put` in the absence of network partitions, the server routes its append through a proxy in a primary region; it then syncs with its own region's copy of the FuzzyLog until it sees the new node before declaring the `put` completed. As a result, when no network partition is occurring a total order is imposed on all updates (via the primary region's chain for the color), and the map is linearizable.

When a secondary region is partitioned away from the primary region, servers switch over to appending to the FuzzyLog in the local region, effectively 'forking' the total order.

CAPMap sets a flag on these updates to mark them as secondary nodes (i.e., appends occurring at the secondary). When the network partition heals, servers in the secondary region stop appending locally and resume routing appends through the proxy in the primary region. Every routed append includes the snapshot ID of the last `sync` call at the secondary client; the proxy blocks the append until it sees a subsuming snapshot ID on a `sync`, ensuring that all the nodes seen by the secondary client have also been seen by the proxy and are available at the primary region.

Any server playing the DAG after the partition heals enforces a deterministic total order over nodes in the forked section: when it encounters any secondary node, it buffers them until the next primary node (i.e., the joining node). All buffered nodes are then applied immediately before the joining node, ensuring that all servers observe the same total order and converge to the same state. As a result, we obtain causal+ consistency [21] during network partitions and linearizability during time periods when there is no such partition.

2.1.4 Other designs

TXCRDTMap: Two properties discussed so far – transactions within a single region and weak consistency across regions – can be combined to provide geo-distributed transactions. With 80 LOC of code change in CRDTMap, we can obtain a transactional CRDT that provides cross-shard failure atomicity [23] (or equivalently, an isolation guarantee similar to Parallel Snapshot Isolation [52]).

RedBlueMap: The FuzzyLog can support RedBlue consistency [22], in which blue operations commute with each other and with all red operations, while red operations have to be totally ordered with respect to each other, but not with blue operations. RedBlue consistency can be implemented with a single color. One of the regions is designated a primary, and 'Red' operations are routed to the primary via a proxy (and thus totally ordered, similar to CAPMap). 'Blue' operations are performed at the local region. We implemented RedBlueMap in 330 LOC.

COPSMap: While CRDTMap can be scaled by sharding system state across different per-color instances, an end-client interacting with such a store will not get causal consistency across shards [21,42]. Concretely, in a system with two regions and two colors, an end-client

in one region may issue a put on a red server, and subsequently issue a put on a blue server. Once the blue put propagates to the remote region, a different end-client may issue a get on a blue server, and subsequently a get on a red server. If the end-client sees the blue put, it must also see the red put, since they are causally related. To provide such a guarantee, the map server can return a snapshot ID with each operation; the end-client can maintain a set of the latest returned snapshot IDs for each color and provide it to the map server on each operation, which in turn can include it in the appended node. In such a scheme, when the blue server in the remote region sees the blue put, it contacts a red server to make sure the causally preceding red node has been seen by it and exists in the region. Such a design requires servers playing different colors to gossip the last snapshot IDs they have seen for their respective colors. We leave the COPSMap implementation for future work.

2.1.5 Garbage collection

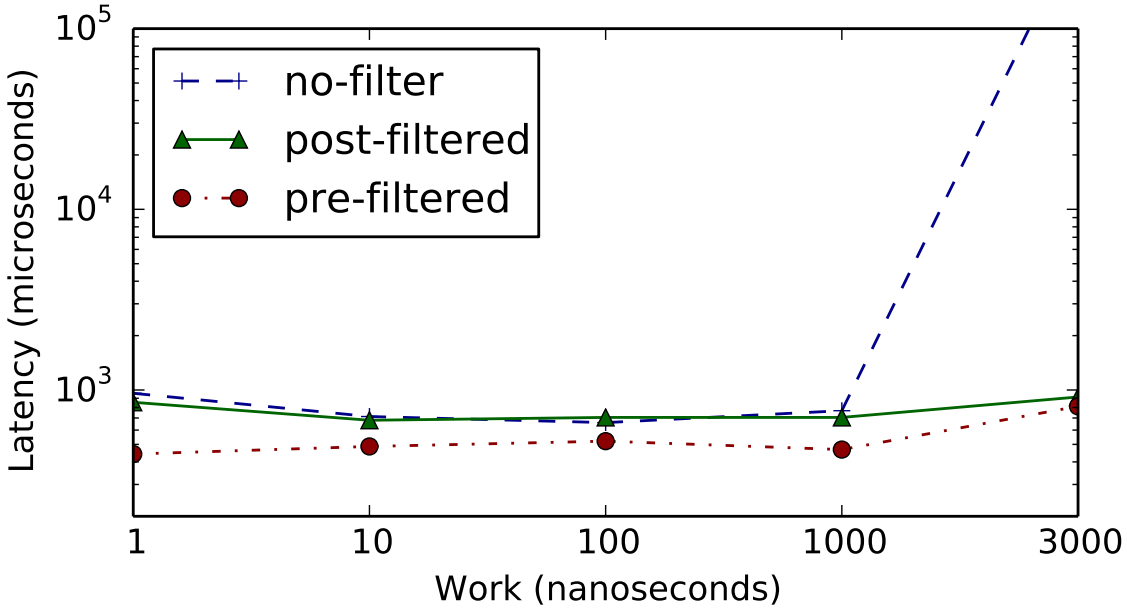
As with shared log systems, garbage collection is enormously simplified by the nature of the workload: the log is used to store a history of commands rather than first-class data, and can be trimmed in increasing prefixes. At any time, the application can store its current in-memory state (and the associated snapshot ID) durably on some external storage system, or alternatively ensure that enough application servers have a copy of it. Once it does so, it can issue the `trim` command on the snapshot ID. Clients that are lagging behind may encounter an `already_trimmed` error, in which case they must retrieve the latest durable state from the external store, and then continue playing the log from that point.

Chapter 3

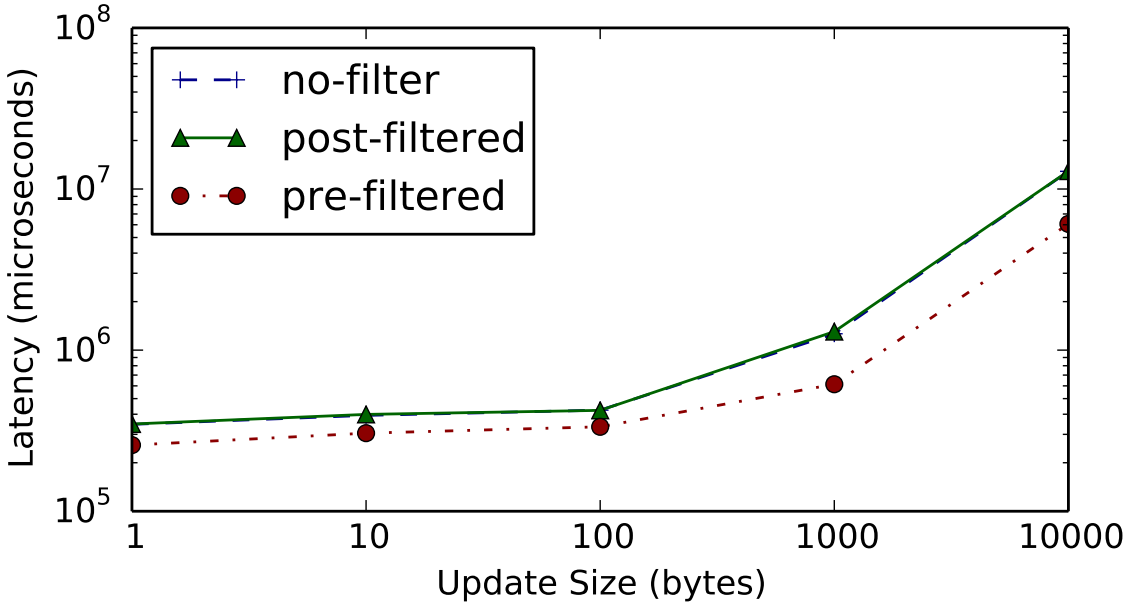
New Applications

Having discussed the applicability of the FuzzyLog to classic distributed systems challenges, data-sharding and geo-replication, we now move on to more surprising systems that can result from altering the guarantees provided by SMR. As opposed to classic SMR views, which materialize logically-independent sub-histories, with *selective SMR* we explore the consequences of filtering our events *within* a single history. These FuzzyViews are materialized from a *subsequence* of the history of an object, and such subsequences can be arbitrarily defined. Despite only materializing part of an object, FuzzyViews offer linearizable semantics for updates and queries; a query will view any updates in the total order that completed before it started, and the history of the view will satisfy the definition of the subsequence. Conceptually, a FuzzyView can be created and kept up-to-date by applying a filtering predicate to the totally ordered stream of updates. The design space for the FuzzyView can be bisected based on where and how the filtering occurs:

- **Post-hoc filtering:** In principle, a FuzzyView can be implemented simply by applying the filtering predicate at the view upon playback. Post-hoc filtering can be implemented in the context of any SMR design and does not require any special support from the ordering engine. However, its benefits are limited. Figure 3.1a shows that post-hoc filtering does not provide a performance boost: if the amount of 'work' required to process and apply an update to the local view is not excessively high, the benefit provided by a FuzzyView that performs post-hoc filtering is minimal. Further,



(a) Post-hoc sync latency vs Work. Post-hoc filtering only helps performance if the work done per update is large (Note log scale).



(b) Post-hoc sync latency vs Update size. Time taken to synchronize against the log increases with update size as the fetch bottleneck gets worse. This renders post-hoc filtering ineffective when increases in workload have a corresponding increase in update size. (Note log scale)

Figure 3.1: Filter Performance.

since updates are played and interpreted at the learner, we do not obtain particularly strong privacy guarantees.

- **Pre-hoc filtering:** Alternatively, we can filter updates within the ordering engine, such that each learner only receives updates of interest to it. Such an approach reduces network overhead and provides privacy benefits, since the learner machine does not see updates that are filtered out. This approach does require the ordering engine to be modified in order to support selective playback.

Building an ordering engine – such as a shared log or a multicast primitive – that supports general-purpose in-network filtering can be daunting; we would effectively be building a database engine. However, we already have an efficient mechanism by which we can build pre-hoc filters over our existing shared log implementation: we can use our color mechanism to have nodes determine *at append time* which filters are applicable to a proposed update, and tag them appropriately. Machines thus only receive events they need to materialize their views, and only require a small amount of additional bookkeeping in the append path to handle writes to the object.

Regardless of where filtering is done, FuzzyViews are created from the same history as the underlying object, with all of the unwanted updates ignored. Since we do not distort the history of the underlying object, we retain the benefits of the shared log: there remains a pristine version of the object's history in the log, which clients can still play to materialize the full object; and all requests made to FuzzyViews are fully linearizable with respect to each other, and the full views. Even though they have less information, reads from *any* of the views are as consistent and up-to-date as reads from the pristine version would be.

3.1 Accelerating applications

The simplest use-case for FuzzyViews is to accelerate queries. In many situations, a query on an object may not require a complete view of the object's history in order to provide an exact answer, or alternatively may be able to tolerate an approximate answer, if the answer's error is bounded. Such queries can be issued against a FuzzyView of the object

based on a subsequence that only contains "important" updates; since these views are able to skip many of the updates in their objects history, they can respond to queries sooner, while imposing less load on the system as a whole.

We provide an illustrative example for each of these use-cases: a key-value map which provides a precise FuzzyView for determining if a given key is present within the map; and a bounded-error counter, whose FuzzyView can be used to obtain an estimate of the counter's current value. We further discuss the techniques used to derive these FuzzyViews from their underlying objects, and how such techniques can be used to construct novel FuzzyViews.

3.1.1 Function accelerator

Consider a map that can be mutated with the following API: `insert`(`key`, `value`), `update`(`key`, `value`), and `remove`(`key`, `value`). A query that checks if a given key is present in the map, *i.e.*,`contains`(`key`), can be answered by a replica that only keeps track of `insert` and `remove` operations. Such a replica maintains the key-set of a map, and implements a FuzzyView that answers the `contains` query exactly. To implement this, we simply associate each operation with a different color in the shared log: the key-set can be materialized by reading only the color corresponding to `insert` and `remove`. Since they only read a subsequence of operations from the shared log, these replicas can update their views faster, and thus answer queries quicker—*e.g.*,to efficiently check if a user has already been registered for a service— without sacrificing consistency.

The map example above is a particular case of a more general construction: whenever an object consists of multiple logical components, each component can be given its own FuzzyView. Many data structures have this property, and naturally expose it through their APIs. For example, a broadcast queue can be viewed as having separate enqueue and dequeue components, which is useful in a multi-consumer scenario. The clients only look at enqueues, and insert a dequeue for each entry they read. A garbage collector reads only the dequeues and removes those entries that have been seen by everyone. Another example is a list that supports `add`, `remove` and `set` operations: keeping track of the `add`s and `remove` tells clients the length of the list, while keeping track of `set` says something about invalidations. In general, any container structure that allows items to be modified

(*e.g.*,lists, trees, maps) could benefit from function accelerators that separate changes to items from changes to the set membership.

These examples are by no means exhaustive. We expect accelerators to be highly specific to individual data structures and applications.

3.1.2 Approximation

Sometimes it is unnecessary to provide an exact answer to a query and one that approximates the true value suffices. Such queries are common in statistical analysis as the results are only correct to within a certain degree of error anyway [49]. Views that provide approximation can be constructed by ignoring parts of the history which only contribute a small amount to the result. We illustrate this by constructing a simple counter object that supports FuzzyViews with bounded absolute error: the views are accurate to within $\pm\frac{t}{2}$ for some threshold t. We start with a version that only supports a single incrementor and generalize it to a construction similar to the consistency metrics in [53, Section 3.4].

Single incrementor. A single incrementing client maintains a local accumulator containing the total amount the client has changed the counter. The accumulator can be decomposed into the form $q \cdot t + r$, *i.e.*, some multiple of the threshold plus a remainder $r < t$. If an increment increases q, the update is marked "significant" by associating it with a different color in the shared log. By reading only the "significant" updates, we can construct a FuzzyView that can return estimates of the counter, by multiplying the number of updates by t. Specifically, after n increments the real and estimated values are:

$$real = \sum_{1}^{n} 1 = n \qquad est = \sum_{1}^{\lfloor n/t \rfloor} t = \lfloor n/t \rfloor \cdot t$$

Let $n = q \cdot t + r$. Since $est = \lfloor n/t \rfloor \cdot t = \lfloor (q \cdot t + r)/t \rfloor \cdot t = \lfloor q \rfloor + \lfloor r/t \rfloor \cdot t = q \cdot t$, we know the actual value of the counter lies within $[est, est + t)$. Thus by choosing the midpoint of this range, $(est + \frac{t}{2})$, we obtain an estimate for the counter that is within $\pm\frac{t}{2}$ of the actual value.

It is important to note that marking an update as significant to a view does more than

just let the view observe the update; the existence of the update implies information about prior (non-significant) updates in the history that the view does not have access to. For instance, if the FuzzyView sees a significant update of magnitude less than t, it can infer that there was at least one other non-significant update it did not see. Although this information may be useful from a performance or functionality perspective, it may be problematic from a privacy standpoint. We explore this issue in Section 3.2.

Non-uniform increments. Generalizing to non-uniform increments requires a slightly more advanced strategy. To see why, consider a counter with threshold $t = 10$; if an increment of 11 is received there are two possibilities:

1. The accumulator's remainder is less than 9. The estimated value needs to be t greater than its current value. For example, if the real value was 15 and the estimate 10 before the increment, the real value should be 26 and the estimate 20 after the increment.
2. The accumulator's remainder is at least 9. The estimated value needs to be $2t$ greater than its current value. For example, if the real value was 19 and the estimate 10 before the increment, the real value and estimate should both be 30 after the increment.

The required increase will not always be t or $2t$, as it depends on the magnitude of the increment, but in general there will only be two cases. Unfortunately, the basic counter we have constructed has no way of distinguishing between these two cases.

There are two solutions to this problem. One is to split the increment into two updates. Assuming the increment value is $s \cdot t + u$, the incrementing client would split the increment into two updates, the first with magnitude u and the second with magnitude $s \cdot t$. While this provides a correct estimate, the estimate $(q + s + 1) \cdot t$ being within t of the actual value $(q + s + 1) \cdot t + (r + u) \operatorname{rem} t$, it has the adverse effect of turning an increment that would have been atomic into a pair of updates. This may not be an issue in practice—the estimate after the first update is still a valid estimate of the counter—but we prefer an implementation that preserves atomicity.

Fortunately, we can support atomic increments by adding one extra bit of metadata to each "significant" update. The protocol proceeds as in the uniform version, except that if

an increment has magnitude greater than t, a bit is set iff the remainder of the increment modulo t increases the counter over t. That is, if the increment is $s \cdot t + u$ and the accumulator is $q \cdot t + r$, the bit is set iff $s > t$ and $r + u \geq t$. The correctness of this construction is a little subtle and benefits from a proof.

Proof. By induction. Assume the current estimate is $est = q \cdot t$, real value is $n = q \cdot t + r$ with $r < t$, and $est \in [n - t, n + t)$. Let the increment be $inc = s \cdot t + u$ with $u < t$ and the new real value n'. There are four cases:

a. $inc < t$ and $r + u < t$.

Then $s = 0$ and the new value $n' = q \cdot t + (r + u)$. Since $r + u < t$, $est' = q \cdot t \in [n' - t, n' + t)$

b. $inc < t$ and $r + u \geq t$.

Since $r < t$ and $u < t$, $r + u < 2t$. So $n' = (q + 1) \cdot t + (r + u) \bmod t$. Since the update is significant, $est' = (q + 1) \cdot t$ which maintains the invariant.

c. $inc = s \cdot t + u$ with $s \geq 1$ and $r + u < t$.

Then $n' = (q + s) \cdot t + (r + u)$ and $est' = (q + s) \cdot t$.

d. $inc = s \cdot t + u$ with $s \geq 1$ and $r + u \geq t$.

Then $n' = (q + s + 1) \cdot t + (r + u) \bmod t$

and $e_{new} = (q + s + 1) \cdot t$.

Since at the start $est = n = 0$, the base case holds, which concludes the proof.

Multiple incrementors. Generalizing to multiple incrementors simply changes the error bound. With w incrementors and threshold t, the estimate falls in the range $[est, est + w \cdot t)$. That is, each incrementor has at most $t - 1$ magnitude of increments that a reading client does not see.

In all of these cases the counter estimates remain linearizable and, if a client desires, it can recover a completely accurate value of the counter.

The above construction demonstrates two symmetrical techniques that aid in the construction of FuzzyViews. One is update splitting, which is a powerful technique for ensuring

that an object's history has enough granularity to make FuzzyViews useful. In this technique we decompose a single, logical, operation at the application level into multiple physical updates at the log level. For example, if the map example in Section 3.1.1 only supported a combined `insert_or_update` operation, our key-set FuzzyView would not be viable; if we wanted to construct the view, we would have to split such operations into separate `insert` and `update` entries in the log. While update splitting can create the opportunity for FuzzyViews where otherwise there would be none, it is not a panacea: update splitting changes the number of updates a view sees, which can be a pessimization in some workloads. Moreover, forcing a mutation to use multiple updates instead of one may alter the semantics of the object, in particular reducing its atomicity, meaning it cannot always be used.

The second technique adds metadata to updates to summarize information about the object's state. This enables a FuzzyView to gain information it could not otherwise obtain, which is particularly useful in objects that are very history based. The drawbacks are that it forces a full view to read redundant data, and requires a mutator to maintain state.

Approximation can be combined with function accelerators to create yet more performant objects. For instance, if in the key-set example above (3.1.1) if the replica keeps track of only `insert` operations, instead of both `insert`s and `remove`s, it would implement a different FuzzyView that answers the `contains` query approximately, by reporting if a key is probably within the map or definitely not in the map—akin to a Bloom filter [54].

The objects discussed in this section are by no means exhaustive; there are likely many more such FuzzyViews that have yet to be discovered.

3.2 Privacy-Aware Applications

As we have seen, FuzzyViews can answer meaningful queries even if they only read a subsequence of an object's history, though in some instances, only by discarding information. However, the fact that a FuzzyView excludes some information can itself be advantageous from a privacy standpoint.

It is often desirable to expose part of an object's state to less trusted consumers, while

maintaining the full version for those with sufficient authority [55]. Providing such censored access to a copy of the full object can be risky, since bugs in the authentication layer can easily turn into security breaches [56]. These issues, privacy and partial declassification, motivate a large body of work around information flow control [55, 56]. By filtering based on authorization, FuzzyViews provide a mechanism for a system to expose only those parts of an object that are actually required. Since these views contain only information the client is allowed to access, any query on them is valid.

As an example of the kinds of views one might wish to provide, consider an employee database. Managers need fairly complete information about their own employees, but do not require detailed information about other employees. The finance department may need to know about each employee's salary, but not other information about them. An informatics department may need all non-personal data to train models to predict employee welfare. All of these views need to be kept in sync without accidentally exposing information where it is not needed. FuzzyViews provide a mechanism for this.

Alternatively, consider a location tracking service. There are many demands for such a service. Employers wish to track their employees locations during business hours. Parents want to track their children to protect them. During a crisis, people want their friends, family, as well emergency personnel to know whether they are in a safe location. A tracked individual wants as much privacy as possible, without preventing those who need their location from discovering it. It is tempting to use cell phones, which continuosly record their location, to track users, but such devices record excessive information. For instance, an employer should not be able to track an employee outside of work hours, as that would be an invasion of privacy. By constructing a FuzzyView of the user's location, such as one filtered by time-of-day, users can ensure their employers, or others who need circumstantial access to their location, only access the location data they have permission to see.

Figure 3.2 shows examples of maps generated based on FuzzyViews of one of the author's locations over several days. Each location update is tagged with a color based on whether it occurs during business hours or not. Figure 3.2a shows the subject's full path, spanning all hours of the day. Figures 3.2b shows a FuzzyView of the path based on locations visited during work hours; Figure 3.2c shows the complementary map of locations visited during

(a) Full path (b) Work hours (c) Non-work hours

Figure 3.2: A location tracking service example. The full view (a) captures the subject's locations at all hours, whereas the FuzzyViews (b,c) capture the locations during work hours and non-work hours.

non-work hours. As can be seen, the work-hours filter hides most of the travel; there is no way to tell from that map the extent to which the subject traveled during non-work hours. These maps also illustrate a potential flaw with this approach: there is a gap in the non-work-hours path which is suggestive of the locations occupied during work-hours.

This is a risk in any cutoff-based filtering: an adversary may be able to infer additional information based on patterns in the information available. This particular case can be mitigated by adding some gap time which is neither work-hours nor non-work-hours, to allow the user to obfuscate their earlier location. Whether this gap is desirable depends on who exactly has access to these views, and what guarantees they require; depending on the times which need to be covered by the views, there may be no acceptable time to have the gap. In general, hiding this kind of trajectory information from an adversary with multiple samples is much more difficult than hiding information at specific point in time, and may not be possible without more statistical techniques such as [57].

A FuzzyView guarantees that any query issued against it exposes no information beyond what is provided by the updates the view has access to. Unlike differential privacy [57] or other anonymization, our privacy guarantees are *qualitative*: updates are either entirely hidden or entirely exposed, and can be applied to a single data-source, not just (aggregations over) groups. This presents us with a set of security trade-offs that are much more similar

to information flow-based systems [55, 56] than statistical ones, and FuzzyViews can likely be used to simplify the implementation of an information flow system.

FuzzyViews enable simple interfaces because any query on the view is allowed (the disallowed updates have been pre-filtered). Although pre-filtering is not strictly required for privacy, it is desirable: unlike post-hoc filtering, sensitive data never reaches the local machine at all, making it much less likely that a side-channel or read-exploit could expose sensitive information. Furthermore, FuzzyViews have low implementation overhead: in many cases, the update logic need not be aware of the filtering at the lower layers, and the same code can be used to materialize both filtered and unfiltered histories.

3.3 Ensemble Learning Applications

A somewhat unexpected application of FuzzyViews occurs in ensemble learning, where diverse machine learning (ML) models are trained and combined to improve the accuracy of a prediction task [58]. In this setting, the data used to train an ML model is the "object": a full view enables training on all datapoints, whereas a FuzzyView enables training on a subsequence of points.

Several ensemble learning techniques use all datapoints for training, achieving diversity by varying the type of ML algorithm used. These techniques do not benefit from the performance savings of FuzzyViews. However, other ensemble techniques train each model on a subsample of the datapoints: for example, bagging [59] is an algorithm for contextual bandit learning [59, 60] that trains multiple predictors on different subsamples of the data, with the goal of creating diverse predictions. FuzzyViews can be used to realize these predictors: to train N different predictors, assign one of N colors uniformly at random to each datapoint[1], and construct a FuzzyView for each color.

One may question the benefit of FuzzyViews—and specifically, pre-filtering—if the output of the predictors is going to be combined anyway, as in the case of bagging. There are two answers to this. First, if the data append rate to the shared log exceeds the network

1. To ensure model reproducibility, colors should be assigned deterministically using a pseudorandom generator.

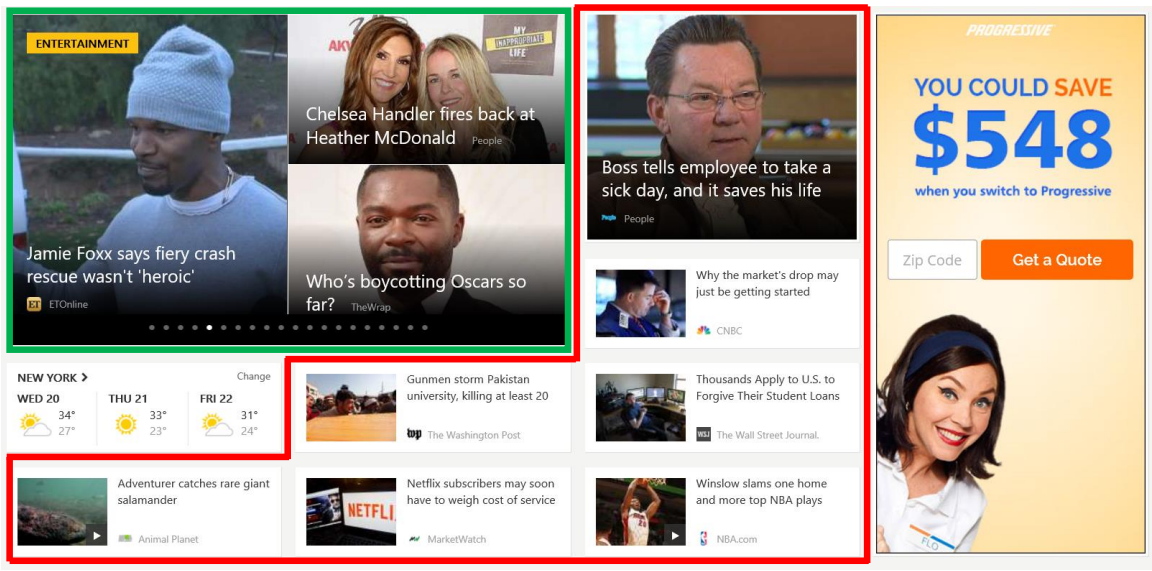


Figure 3.3: MSN homepage (image from [61]). The Slate (boxed red) and Panel (boxed green) use the Decision Service in production as of early 2016.

capacity of a single machine, FuzzyViews reduce the inbound network traffic and allow predictors to still be trained on single machines, which is better than the alternative of doing parallel learning. Second, diverse models need not always be combined: in many ML deployments, models trained from a common data source are kept separate. For example, an online service may find that training separate content-recommendation models for different submodules of the site, or for different classes of users, leads to better prediction accuracy.

As a concrete example, consider the case of the Decision Service [61], an open-source platform for contextual bandit learning[2]. The Decision Service personalizes news stories displayed on MSN's homepage, shown in Figure 3.3, which serves 10s of millions of users issuing thousands of requests per second. When a user requests the homepage, MSN's frontend servers must decide how to order the articles on the page. If the user is logged in, there is context: demographics (e.g., age, location) and the topics of news stories they have clicked on in the past; otherwise only location is available. The goal is to maximize the number of clicks on articles.

When the Decision Service was first deployed, it was applied to the Slate of the MSNhomepage (see Figure 3.3), and only for logged-in users. Later, it was applied to the Panel and

2. We obtained access to the data and experiences of the service by contacting the research team [61].

Full data	Logged-in / Non-logged-in	Male / Female
1.000	1.021	0.982

Table 3.1: Estimated click-through rate of partitioned models (realized using FuzzyViews) based on one day of MSNdata from April 2016, normalized against the full data model.

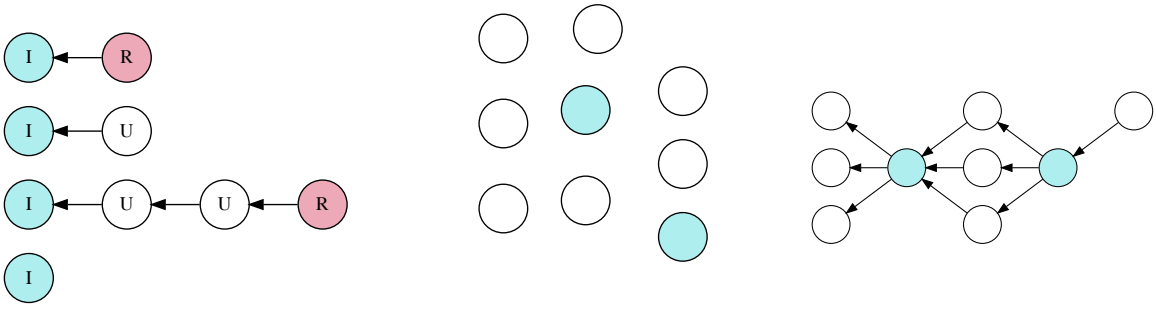
10+ other modules and markets (*e.g.*,Brazil, Europe). All of these deployments are in production and are isolated from each other, poorly utilizing the underlying cloud services required by the Decision Service. FuzzyViews could enable a common data collection pipeline across these deployments, amortizing the cost of those resources, by using a different color per deployment.

For example, when the Slate deployment was extended to non-logged-in users, offline experiments revealed that training a separate model for logged-in vs. non-logged-in users yields better overall accuracy than a combined model. Similar experiments were done for other partitions of the data, *e.g.*,based on the user's location or gender. Table 3.1 shows the estimated click-through rate of some partitioned models compared to the single full data model, based on one full day of real MSNtraffic from April 2016. (We adopt the same offline evaluation methodology used by the Decision Service [61].) All partitioned models are realized using FuzzyViews with appropriate colors to distinguish each partition's datapoints. As the table shows, separating the models for logged-in vs. non-logged-in users improves overal performance, whereas separating by gender does not.

3.4 Formalizing FuzzyViews

In this work we demonstrate how to construct views of an object which are linearizable yet trade off precision to gain other benefits. It is helpful to define what we mean by precision. Intuitively, we can say that one query is more precise than another if performing the query requires more of an object's history, that is, if it is more detailed. We formalize this notion below.

We can define an object as having a current state s, drawn from some set $\mathbf{S}$, a set of mutations $\mathbf{M} \subseteq \mathbf{S} \to \mathbf{S}$, and a set of observations $\mathbf{O} \subseteq \{o | \exists R, o : \mathbf{S} \to R\}$.



(a) Key-value map

(b) Randomly sampled counter

(c) Threshold counter

Figure 3.4: Example mutation graphs from our formalism of objects discussed in this paper. Shaded nodes are possibly visible in the filtered view. In (a) letters are used to distinguish inserts (I), updates (U), and removals (R).

A mutation m is *invisible* to an observation o if $\forall m' \in \mathbf{M}, s \in \mathbf{S} : o(m(m'(s))) = o(m'(s))$. If a mutation is not invisible to an observation it is said to be *visible*.

Example. In the key-value map construction of Section 3.1.1, `update`s are invisible to `contains` observations, but are visible to `get` if the key for `update` and `get` match.

We define the mutation-DAG for an object's state as a directed acyclic graph that has a node for every mutation in the history, and an edge from node b to node a if b depends on a. Conservatively, we can think of this "depends on" relationship as: if two updates do not commute in some view, then whichever update occurs second must depend on the one that occurred first, though this may be too strong for some systems. These edges capture the flow of information in the object; an edge implies that information from an earlier mutation could possibly remain in later versions of the object. Some examples of mutation-DAGs for the objects described earlier are shown in Figure 3.4.

To find the state needed to perform an observation, it suffices to start with the set of observable nodes in the mutation-DAG and take the transitive closure.

Example. The set of nodes needed for `contains` on the key corresponding to the second row of Figure 3.4a is just the insert in that row, while for the key corresponding to the third row would include all nodes in the row. In a randomly sampled counter (Figure 3.4b), the

needed nodes are only those that were sampled, while in the threshold counter described above (Figure 3.4c), observations of the estimated value implicitly depend on earlier, invisible, increments, hence the more complex structure of that graph.

Let needed(o, s) be the number of nodes required to perform observation o in state s. An observation o is said to be more detailed than another o' if $\forall s, \text{needed}(o, s) \geq \text{needed}(o,' s)$ and $\exists s, \text{needed}(o, s) > \text{needed}(o,' s)$. That is, for every state of the object the needed state for o is at least as large as the needed state for o', and there exists some state where it is greater. This fits with our intuition: a randomly sampled counter only gives probabilistic guarantees and is thus less detailed than a threshold based counter, and a `contains` provides less information than a `get`, since the former merely determines if a value exists, while the latter actually provides it. This further argues for a relationship between our function accelerators and approximate views. A function accelerator takes advantage of the invisibility of mutations to some observations to accelerate queries by ignoring some updates. An approximate view does something similar, but, since the mutations it ignores are not invisible, it must lose some information.

Relationship between detail and consistency: it is easy to construct objects that are very precise but are only eventually consistent; for instance, a counter which receives increments 2, 1, 3, 1000 is more accurately represented by a view based on the increments 2, 3, 1000 than a view based on the increments 2, 1, 3, even though the latter is arguably more consistent than the former. This implies that consistency and detail are not the same and are partially independent of each other. Our formalism helps define that difference: detail is a function of the mutation-DAG, while consistency says how the mutation-DAG is allowed to evolve over time, and which version of the mutation-DAG an observation must be run against. Though constraints on one may imply constraints on the other, to some extent they can be considered independently. This is supported by earlier work such as [40] and [22], which show that with proper semantics a relatively large amount of ambiguity in the ordering of updates can be tolerated.

3.5 Discussion

Other filters. FuzzyViews apply pre-defined filtering predicates to the stream of updates. There are other kinds of general-purpose filters that may also be useful, but do not quite fit our model. A *prefix filter* only returns an update once sufficient updates have occurred after it. A view materialized from one of these is *guaranteed* to be stale, which may be useful for privacy reasons. Similarly a *suffix filter* discards data from sufficiently old updates, which could be useful for temporally local queries. Given updates that are uniform enough, and do not require too much history, *random sampling* can give a probabilistic approximation of an object, while a *rate-based filter*, which returns exactly 1 out of every n updates, gives stronger guarantees on the updates returned, but weaker statistical guarantees. This last filter has a real-time counterpart, the *Hz filter*, which returns a fixed number of updates per unit time, instead of per updates written.

Color Allocation. Constructing FuzzyViews is currently a largely manual task. Though we developed a library to assist in mapping colors to more human readable predicates, deciding what predicates are used and thus what views should exist remains a manual process. It would be interesting to see if it were possible to automatically derive such views from an API specification; *e.g.*,the key-set example would be a good candidate for that. Additionally, our color allocation scheme allows for redundancy, where predicates cause updates to be tagged with multiple colors. A more advanced scheme could put the set of predicates used for filtering into a "disjunctive normal form" to ensure only the minimal number of colors are used, one per unique condition.

Chapter 4

Latency Bounds for (Strict) Serializability

During the design of the FuzzyLog we experimented with a number of concurrency control protocols in an attempt to create a strictly serializable multiappend which only sends messages to servers involved in the append, yet still completes after a constant latency in non-failing cases. It turns out that, given the constraints the FuzzyLog was built under, this is impossible, and such strictly serializable transactions always have potential executions in which a transaction takes an amount of time linear in the number of servers involved.

We use a message log formalism, inspired by the classic one in [35], to prove some latency lower bounds for scalable serializable and strictly serializable systems, an issue which confounded the FuzzyLog project for some time in our attempt to scale cross server transactions.

4.1 (Strict) Serializable Latency Lower Bounds

4.1.1 Model

A transaction is allowed to perform arbitrary read-modify-write actions on each server.

Protocol correctness is evaluated under the usual asynchronous network model, with client failures (various places including [62]); however, as there is no natural way to define

latency under this model, we evaluated latency under a partial synchrony model [63]: there exists some maximum message delivery time d, and neither the protocol nor the machines are aware of what this time may be.

4.1.2 Notation

For all of these proofs time goes left-to-right. T_1 is used to mean the receipt of the messages for transaction 1 at a server. In the event that the receipt of multiple transaction's messages are interleaved, T_{1a} is used to mean the receipt of the first group of messages from transaction 1 at a server, T_{1b} the second; we never deal with cases where the messages are interleaved in more than two groups. We use $T_i < T_j$ to mean transaction i happens before transaction j in some partial order, which will be specified in the prose. Therefore the following:

$$\begin{aligned} server\ 1: &\quad T_1 \quad T_2 \\ server\ 2: &\quad T_{1a} \quad T_3 \quad T_{1b} \end{aligned}$$

means that at server 1 first all messages from T_1 are received then all messages from T_2, while on server 2 the first group of messages from T_1 are received, then all messages from T_3 and finally the second group of messages from T_1.

4.1.3 Bounds

We concern ourselves with scalable, client-server transactions. Scalable, in that servers only receive messages regarding transactions they take part in; if a transaction does not access a server, that server does not need to know of it. This prevents bottlenecks from forming, and allows the system as a whole to scale with an increasing number of servers. Client-server, in that we only allow clients to communicate with servers; we do not allow clients to communicate with each other, nor do we allow servers to communicate with other servers. Disallowing server-to-server communication simplifies both server complexity and workload, which is useful when building a simple data storage server, such as a shared log, or for high-performance protocols [64].

We make the following choices for the set of transactions we use to construct our worst-

case lower bounds, it is possible that by relaxing these one could construct an even worse bound, but such bounds are not necessarily useful:

- Every transaction both reads a writes at every server it accesses.
- At this point we only care about transactions which are partitionable: at each server, the transaction only requires data on that server, as these are the minimal useful form of transactions, and can be generalized to arbitrary transactions with additional round trips [9].
- Transactions never abort; aborts caused by the system force clients to retry transactions, implying unbounded histories if a client never finds a non-aborting order; aborts induced by the client are simply transactions which performs no writes. In either case, an aborted transaction need not be ordered with respect to other transactions, so removing them does not improve latency.
- All transactions which access a server conflict.
- Servers never fail; in a real system this would be achieved by replicating the servers and postponing ACKs for operations until they are sufficiently replicated. Allowing histories to contain server failures and reinitialization creates the potential for unbounded histories, as servers repeatedly go down and are brought back up.

A transaction T is said to be *complete* at a server at the point when all future transactions which access that server are guaranteed to see the results of T; once a transaction is *complete* at a server it is guaranteed to commit. A transaction is *fully complete* once it is *complete* at every server it accesses, *and* the client which started the transaction is aware of this.

Our proofs rely on transactions being partially *completed*: *complete* at one server even though the transaction is still in progress at another. Since the protocols must be correct in an asynchronous network model, the only way for a server to be aware of changes in the outside world is through message receipt; in particular, a server cannot determine the duration that has passed since a transaction has received its latest message. Therefore, once a server has received the final message for a transaction, the transaction must either

be *complete* at that server, or blocked by some other transaction for which the server has received a message; if we allow a transaction to block without there being any expected message to unblock it, the transaction will potentially wait forever.

Theorem 1. *It is impossible to guarantee serializable transactions complete in 1 round trip with only client-to-server communication.*

Proof. Consider the following message receipt history:

$$\begin{aligned} server\ 1: &\quad T_1 \quad T_2 \\ server\ 2: &\quad T_2 \quad T_1 \end{aligned}$$

As there are no messages left for either transaction, both must be *complete*. Further, since each transaction can only send one message, each transaction must be *complete* at a server as soon as its message is received. By the order of receipt at server 1, T_1 must happen before T_2, by server 2, T_2 must happen before T_1. This causes a cycle in transaction order $T_1 < T_2 < T_1$, violating serializability. □

By Theorem 1, serializable protocols require at least two message receipts at the servers, and therefore two round trips. As we are in an asynchronous setting, there is no purpose in a client sending a message after the first one except due to message receipt (all such messages could be combined into the previous message). Therefore, without loss of generality, we assume protocols are of the form: send a message to all servers, wait for an ACK from at least one of the servers, repeat; servers, and clients which have started their transaction do not send messages except due to message receipt.

Definition 1. *A transaction T_2 is said to be "pulled before" another transaction T_1 if T_{1a} is received before T_{2a} on some server, but T_1 happens after T_2 in the serialization order.*

Example If the message-receipt history

$$server\ 1: T_{1a} \quad T_2 \quad T_{1b}$$

corresponds with a serial ordering $T_2 < T_1$, then T_2 would be said to be "pulled before" T_1.

Lemma 1. *In any serializable protocol involving at least two messages, there must exist potential executions in which one transaction is pulled before another.*

Proof. Consider the following message-receipt history involving two transactions: T_1 and T_2:

$$\begin{aligned} server\ 1: &\quad T_{1a} \quad T_2 \quad T_{1b} \\ server\ 2: &\quad T_{2a} \quad T_1 \quad T_{2b} \end{aligned}$$

Since all messages have been received, the three transactions involved must all be *complete*. If no transaction is pulled before another, then in the serialization $T_1 < T_2$ due to the order of message receipt on server 1, and $T_2 < T_1$ due to server 2. This leads to a cycle $T_1 < T_2 < T_1$, violating serializability. □

We use these to prove that the latency lower bound for a strictly serializable transaction is at least linear in the number of servers touched by the uncompleted portion of a transaction's dependency graph. The dependency graph for a transaction T_i, as defined in Concurrency Control [35], consists of the set of transactions which are serialized before T_i and which transitively conflict with T_i.

Intuition To *complete* strictly, a transaction must know that nothing new can be added to its dependency graph; if a transaction is *complete* before this point a new, later transaction can be serialized before it, causing a cycle between real-time and the serialization, and violating strict serializability. In the worst case this forces the transaction to wait until it hears from all of the other servers before it is allowed to *complete*. We can force the servers to only communicate as in a linear network, causing the communication of this information to take a linear number of hops and thus linear time.

A system of scalable transactions can be viewed as a message passing system in which only some of the servers are connected. In particular, two servers share a direct link if and only if there is some pending transaction T_m which touches them both. If there is such a transaction, then the servers can communicate by appending information to their ACKs, and using the client for said transaction as a proxy. Servers which lack such a direct connection can communicate if they are connected by multiple such hops. In the worst case,

each server shares a transaction with exactly one other server, requiring a linear number of hops for the two furthest servers to communicate. Since each hop corresponds to a server sending some client a message, and the client forwarding said message to another server, in the worst case each hop take a RTT.

Lemma 2. *In the worst case, if two servers within some transaction's dependency graph wish to communicate, it takes time linear in the depth of the uncompleted portion of a dependency graph to propagate information from one to the other.*

Proof. By our scalability property, servers can only exchange information with clients, and then only if they share a transaction. Servers can thus only communicate with each other by using the clients as a proxy. Consider the following allocation of transactions to servers:

$$\begin{aligned} server\ 1: &\quad \{T_1, T_2\} \\ server\ 2: &\quad \{T_2, T_3\} \\ server\ 3: &\quad \{T_3, T_4\} \\ \vdots &\quad \vdots \\ server\ n: &\quad \{T_n, T_{n+1}\} \end{aligned}$$

Each server i is only able to communicate with servers $i \pm 1$. Thus, for server n to communicate with server 1, it takes at least $|n-1|$ hops, and, since every hop takes a server-client-server trip, in the worst case this corresponds to linear RTTs. □

Theorem 2. *Strict serializability takes time at least linear in the number of servers in a transaction's dependency graph.*

Proof. A strictly serializable execution implies that the transactions are serializable, and that this order obeys real time [65]: if a transaction T_j starts after some other transaction T_k ends, then the serial order must not contain $T_j < T_k$.

Assume there exists a protocol that *completes* in less than a linear number of RTTs. By Lemma 2 this implies that transactions can *complete* before receiving information from

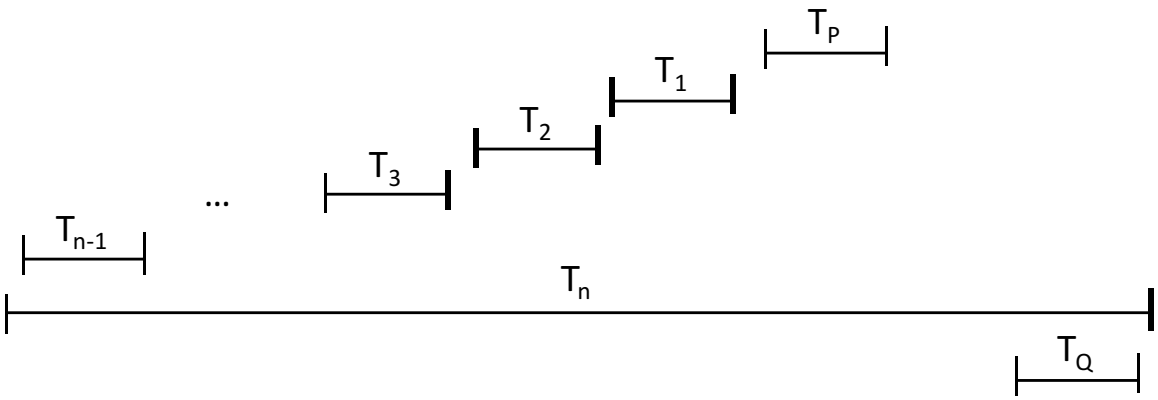


Figure 4.1: Transaction timing which causes a cycle in the case where sub-linear messages are used.

one of the servers. Consider the following history in which T_Q is pulled before T_n:

$$\begin{array}{rccc} server\ 1: & T_1 & T_P & \\ server\ 2: & T_2 & T_1 & \\ server\ 3: & T_3 & T_2 & \\ \vdots & \vdots & \vdots & \\ server\ n: & T_n & T_{n-1} & \\ server\ n+1: & T_{na} & T_Q & T_{nb} \end{array}$$

By the completeness condition, in the serial order $T_1 < T_P$ and $\forall i \in [1, n)$, $T_{i+1} < T_i$, further $T_Q < T_n$ by assumption. Further, since transactions *complete* in sub-linear time, T_P *completes* before it hears from all of the servers, in particular, it does not hear from *servern*. Thus, T_P is able to *complete* before T_n does. This being the case, transaction timings such as the one in Figure 4.1 causes a cycle with $T_P < T_Q$ in real time. (Though such a history would in fact take less than linear RTTs, this is irrelevant; since the machines have no clocks they cannot take advantage of this fact). □

Upper bound There are strictly serializable protocols which can *complete* in time linear in the depth of a transaction's dependency graph. One such protocol is as follows:

1. Perform a serializability protocol, which doesn't allow new transactions to be serialized before conflicting *completed* ones[1]. There are some, such as Skeen's algorithm (described in 5.1.2 as well as in [66]) which *complete* in time no greater than linear in the depth of the dependency graph
2. After the serializability protocol *completes* at a server, instead of completeing the transaction immediately, buffer it until all transactions in its dependency graph have fully *completed*, and only then handle it in the serialization order.

Proof. Assume there is a cycle in the order. Since the serialization protocol is correct by assumption, and there can be no cycles in the real-time ordering of transactions (i.e., a transaction cannot finish if it has not yet started), this must be due to transactions T_1 and T_2 such that $T_1 < T_2$ in the serialization while $T_2 < T_1$ in real time. However, T_2 doesn't *complete* until after every transaction in its dependency graph has fully *completed*, and T_2 itself has been serialized, so T_1 being serialized first would require T_1 to be serialized before an already committed transaction, violating an assumption. □

Since every transaction waits for all the transactions which are serialized before it, and transactions *complete* in parallel unless they conflict, this takes time linear in the depth of the dependency graph.

4.1.4 Other Protocols

Contemporary protocols exhibit both extremes of the tradeoff implied by these bounds, with some tending towards better latency while others attempt scalability. Paxos [2] and centralized sequencers [18] have sub-linear latency in exchange for centralization: with a centralized sequencer a single server sees every transaction, while in Paxos every server sees everything. Ordered two phase locking [35] takes time either linear in the number of transactions, when waiting for locks to be released, or time linear in the number of servers, when waiting for locks to be acquired.

1. Specifically it is stated by Bernstein and Hadzilacos [35, Chapter 1, pg 15-16] that if a user submits a transaction after a conflicting one has been ACKed, it is guaranteed that the new transaction will execute after the ACKed one. For this work it is only necessary that some protocol has this property, and we do not require that all serializable algorithms provide it.

Chapter 5

System Implementation

5.1 Dapple Design and Implementation

Dapple is a distributed implementation of the FuzzyLog abstraction, designed with a particular set of requirements in mind. The first is *scalability*: reads and appends must scale linearly with the number of colors used by the application and the number of servers deployed by Dapple, assuming that load is balanced evenly across colors. The second requirement is *space efficiency*: the FuzzyLog partial order has to be stored compactly, with edges represented with low overhead. A third requirement is *performance*: the `append` and `sync` operations must incur low latency and I/O overhead.

Dapple implements the FuzzyLog abstraction over a collection of storage servers called chainservers, each of which stores multiple in-memory log-structured address spaces. Dapple partitions the state of the FuzzyLog across these chainservers: each color is stored on a single partition. Each partition is replicated via chain replication [67]. Our current implementation assumes for durability that storage servers are outfitted with battery-backed DRAM [68, 69]. We first describe operations against a single color on an unreplicated chainserver.

5.1.1 Single-color operation

Recall that each FuzzyLog color consists of a set of totally ordered chains, one per region; each region has the latest copy of its own local chain, but a potentially stale copy of the

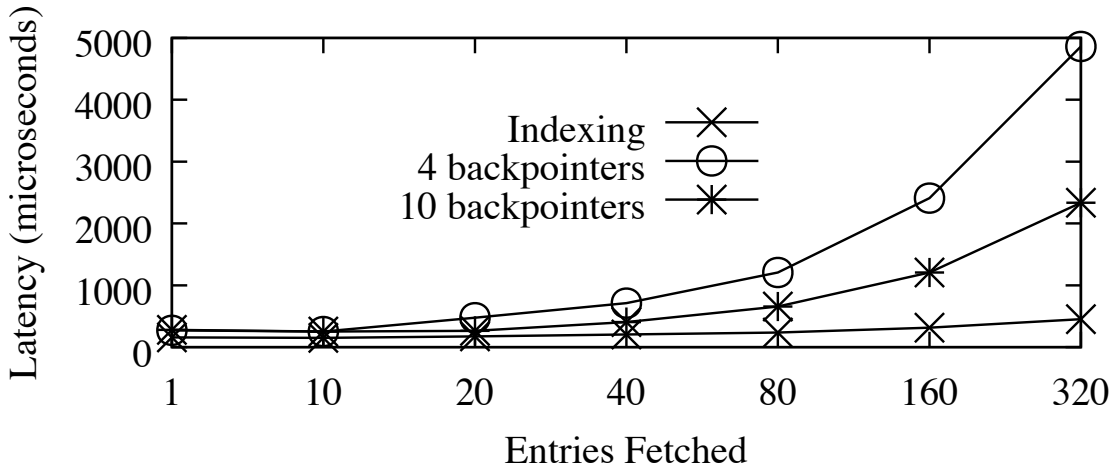


Figure 5.1: Fetching within a chain via backpointers takes time linear in the number events fetched while laying out a chain in an address space allows us to pipeline.

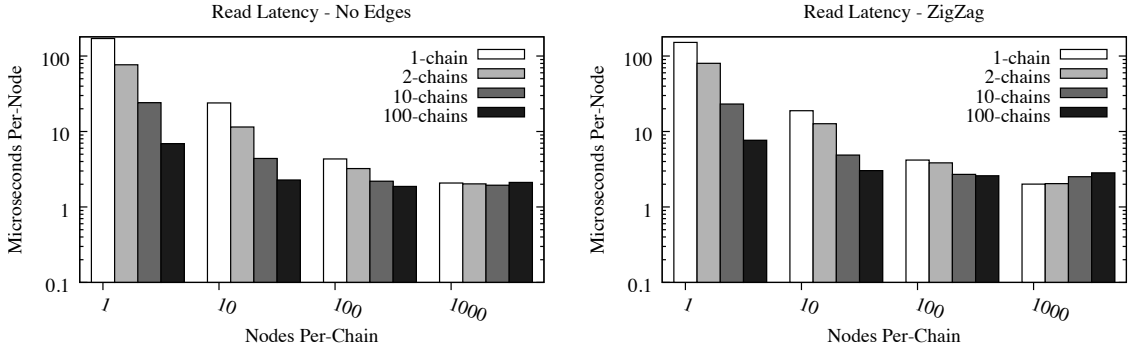


Figure 5.2: Cross-chain dependecies do not cause latency to degrade since fetches are still pipelined.

other regions' chains. Dapple stores each chain on a single log, such that the order of the entries in the log matches the chain order (i.e., if a chain contains an edge from B to A, B appears immediately after A in the corresponding log). In a deployment with R regions, each region stores R logs, one per chain; one of which clients in the region actively write to (the local log), while the remaining are asynchronously replicated from other regions (we call these shadow logs). We choose to store each chain in its own address space, as opposed to fetching chains via backpointers (as in *e.g.*, Tango [11]) due to the latency hit of following pointers. Figure 5.1 shows the time to fetch a number of entries when each node stores 4 backpointers within the chain, 10 such backpointers, or if clients can index directly into the chain. For longer sequences of fetches the backpointer based systems incur an approximately linear slowdown, while the latency of indexing remains roughly constant as the time to fetch later entries is hidden by pipelining. Dependencies between entries within different chains are still stored as backpointers; since we must be able to add a dependency without access to the canonical version of entry which is depended on, this is the only implementation that can work. Fortunately, this does not cause any noticeable slowdown. An example of this can be seen in Figure 5.2 where we compare reading a number of entries and chains without dependencies versus reading the same amount with dependencies arranged so that the entries are forced into a linear order. Despite this being the worst case for playback there is no appreciable latency hit, since, after a single round trip to discover which entries exist, the entries can still all be fetched in parallel; it is only their return to the application which must be serialized. Each server exposes a low-level API consisting of three primitives: `log-append`, which appends an entry to a log; `log-snapshot`, which accepts a set of logs and returns their current tail positions; and `log-read`, which returns the log entry at a given position.

Clients implement the `sync` on a color via a `log-snapshot` on the logs for that color, followed by a sequence of `log-read`s. The return value of `log-snapshot` acts as a vector timestamp for the color, summarizing the set of nodes present for that color in the local region; this is exactly the snapshot ID returned by the `sync` call. The client library fetches new nodes that have appeared since its last `sync` via `log-read` calls. When the application calls `append` on a color, the client library calls `log-append` on the local log for that color.

It includes the vector timestamp of nodes seen thus far in the new entry; as a result, each appended entry includes pointers to the set of nodes it causally depends on (these are the cross-edges in the FuzzyLog DAG). On a `sync`, the client library checks each entry it reads for dependencies and recursively fetches them before passing them up to the application. In this manner, the client ensures that playback of a single color happens in DAG order.

Each chainserver periodically synchronizes with its counterparts in remote regions, updating the shadow logs with new entries that originated in those regions. To fetch updates, the chainserver itself acts as a client to the remote chainserver and uses a `sync` call; this ensures that cross-chain dependencies are respected when it receives remote nodes. Copied-over entries are reflected in subsequent `sync` calls by clients and played; new entries appended by the clients then have cross-edges to them.

Dapple replicates each partition via chain replication. Each `log-append` operation is passed down the chain and acknowledged by the tail replica, while `log-snapshot` is sent directly to the tail. Once the client obtains a snapshot, subsequent `log-read` operations can be satisfied by any replica in the middle of the chain. The choice of replication protocol is orthogonal to the system design: we could equally use Multi-Paxos to accomplish this.

5.1.2 Multi-color operation

The FuzzyLog API supports appending a node to multiple colors. In Dapple, this requires atomically appending a node to multiple logs: one log per color corresponding to its local region chain. To do so, Dapple uses a classical total ordering protocol called Skeen's algorithm (which is unpublished but described verbatim in other papers, e.g. Section 4 in Guerraoui et al. [66]) to consistently order appends.

The original Skeen's algorithm produces a serializable order for operations by multiple clients across different subsets of servers. Unfortunately, it is not tolerant to the failure of its participants. In our setting, each 'server' is a replicated partition of chainservers and can be assumed to not fail. However, the clients in our system are unreplicated application servers that can crash. We assume that such client failures are infrequent; this pushes us towards a protocol that is fast in the absence of client failures and slower but safe when such failures do occur. Accordingly, we add fault-tolerance mechanisms such as leases, fencing,

and write-ahead logging to produce a variant of Skeen's that completes in two phases in a failure-free 'fast' path, but can safely recover if the origin client crashes.

Each chainserver maintains a local logical Lamport clock [70]. All client operations are predicated on relatively coarse-grain leases [71] (e.g., 100 ms), which they obtain from each server (or the head of the replica chain for each partition); if the lease expires, or the head of the replica chain changes, the operation is rejected.

We now describe failure-free operation. The fast path consists of two phases, and has to execute from start to completion within the context of a single set of leases, one per involved partition. For ease of exposition, we assume each partition has one replica chainserver.

In the first phase, an origin client (i.e., a client originating a multi-append) contacts the involved chainservers, each of which responds with a timestamp consisting of the value of its clock. Further, the chainserver inserts the multi-append operation into a pending queue along with the returned timestamp. In addition, the origin client provides a WAL (write-ahead log) entry that each chainserver stores; this includes the payload, the colors involved, and the set of leases used by the multi-append.

Once the client hears back from all the involved chainservers, it computes the max across all received timestamps, augments it with a unique nonce to break ties, and transmits that back to the chainservers in a second phase: this max is the timestamp assigned to the multi-append and is sufficient to serialize the multi-appends in a region. When a chainserver receives this message, it moves the multi-append from the pending queue to a delivery queue; it then waits until there is no other multi-append in the pending queue with a lower returned timestamp, or in the delivery queue with a lower max timestamp (i.e., no other multi-append that could conceivably be assigned a lower max timestamp). Once this condition is true, the multi-append is removed from the delivery queue and processed.

The protocol described above completes in two phases. A third step off the critical path involves the client sending a clean-up message to delete the per-append state (the WAL, plus a status bit indicating the last executed phase) at the chainservers; this is lazily executed after a multiple of the lease time-out, and can be piggybacked on other messages. If a lease expires before the two phases are executed at the corresponding server, or the origin client crashes, it leaves one or more servers in a wedged state, with the multi-append stuck

in the pending queue and blocking new appends to the colors involved. After a time-out, the chainserver begins responding to new append requests with a *stuck-err* error message, along with the WAL entry of the stuck multi-append. A client that receives such an error message can initiate the recovery protocol for the multi-append.

A client recovering a stuck multi-append (i.e., a recovery client) proceeds in three phases: it fences activity by the origin client or other recovery clients; determines the wedged state of the system; and completes the multi-append. The fencing phase involves accessing the lease set of the original client (which is stored in the WAL), invalidating it at the servers, and writing a new recovery lease set at a designated test-and-set location on one of the chainservers. If some other recovery client already stored a lease set at this location, we wait for that client to recover the append, fencing it after a time-out. Fencing ensures that at any given point, only one client is active; the WAL allows clients to deterministically roll forward the transaction.

Correctness: Skeen's protocol has been proved by others to generate a total order [66, 72]. To prove our recovery protocol correct, we wrote a machine-checked proof in Coq, which we now summarize.

5.1.3 Validating the Recovery Protocol

The correctness of the recovery protocol can be decomposed into three parts:

1. The fencing protocol provides mutual exclusion.
2. A client acting in isolation can recover the state of the protocol for a multiappend from the servers.
3. Skeen's underlying protocol which provides ordered multicast as discussed in [66].

The system consists of a set of client threads, whose messages are written $message_{send}$, and a set of server threads, whose messages are written $message_{ack}$. A history is an arbitrary serialization of the message sends in the system. For history h, and any two events a and b, $a \prec_h b$ if a occurs before b in the history; we write $a \prec b$ if the history we are referring to is apparent from context. We only concern ourselves with histories consistent with both

```
client:
 state: id

 take_over(chains, failed_client_id):
  forall chain ∈ chains,
   fence(failed_client_id)
  forall chain ∈ chains,
   wait_for_fence_ack()
  test−and−set(failed_client_id, id)

server:
 state:
  live_clients: set Id

  on receive(fence(id)):
   live_clients = live_clients − {id}
```

Table 5.1: Fencing Pseudocode

program order (if the program requires receipt of message a at a thread before said thread can send message b then $a \prec b$ in the history), and network order (the sending of a message a, a_{send}, must precede the acknowledgment of said message, a_{ack}) as these are the only histories which can be exhibited by a real system. For proof engineering purposes, our histories only contain message sends, and we do not model message receipt directly, instead leaving it implicit in the ordering constraints; if program order requires that the receipt of a must precede the send of b then the receipt of a must happen at some point between $a_{send/ack}$ and $b_{send/ack}$. Since our proofs cannot rely on the specific timing of message-receipt, beyond what is specified by program-order, they should be robust to arbitrary receipt ordering. Recovery is inherently per-multiappend. In the real system each of these functions, except `fence`, would be indexed based on message-id.

Fencing Protocol

The fencing protocol in Table 5.1 provides mutual exclusion: at any point in the history there is at most one client which is able to mutate the servers. For this protocol, each client is required to have a unique ID, and each of the servers stores a set of IDs `live_clients`. Additionally there is a test-and-set, stored on one of the servers, which contains a client ID,

starting off with the ID of the client which originated the multiappend. At start all client ids are found in `live_clients` on all the servers, and servers only respond to messages if the sending clients ID is found in their `live_clients`. If a client wishes to take over the protocol, it first fences off the failed client, removing it from `live_clients` and making it impossible for the failed client's messages to be received. Then it attempts to take over a test-and-set from the failed client's ID; if it succeeds the client can now perform mutations.

This protocol provides mutual-exclusion via uniqueness of test-and-set. At every point in the protocol an invariant is maintained that at most one client is capable of mutating the servers' state. At start, the original client owns the test-and-set, and is the only client able to perform the append. Before a client can attempt to take over the test-and-set it must first fence-off the client currently owning the test-and-set, ensure that said client can no longer append, and thus no client can append. After this, at most one client can succeed in taking over the test-and-set, and thus continue the mutation.

Recovery Protocol

The fencing protocol allows us to treat clients as if they are acting in isolation; for as long as a client is capable of mutating the servers' state it is the only client that is capable of doing so. This implies that on switchover between clients we need not concern ourselves with what the failed client was in the process of doing, but only with the state which is actually found on the servers.

Every server goes through the state-machine seen in Figure 5.3: they start off storing no state, then stores which message they last received, until they are garbage collected. Due to the definition of Skeen's protocol, a server can only see stage 2 after all servers have seen stage 1, and garbage collection is only valid after all servers have seen stage 2.

Lemma 3. *In a valid history, there are only 4 states the servers can be in:*

1. *No server stores anything*
2. *Some servers store Skeens 1, some servers store nothing, and no server has gotten anything after Skeens 1.*
3. *Some servers store Skeens 1, the rest store Skeens 2.*

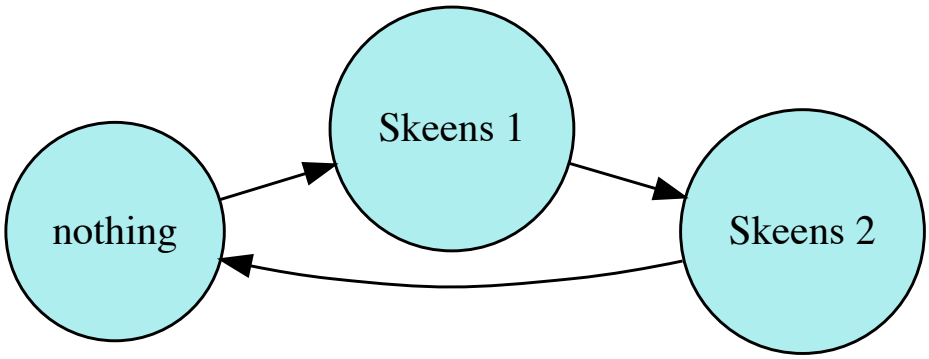


Figure 5.3: The state machine of Skeen's protocol.

4. Some servers store Skeens 2, the rest store nothing.

To see why this is, consider a history which results in one server storing *Skeens 1*, one storing *Skeens 2* and one storing nothing. Since a server has received *Skeens 2* in this history, every server must have received *Skeens 1*. This implies that the server which stores nothing must have been garbage collected. However garbage collection can only occur after every server has received *Skeens 2* leading to a contradiction with the server storing *Skeens 1*. Similar arguments show that the other invalid states cannot occur.

The above lemma gives us the following algorithm for determining where in Skeen's protocol a recovering client should continue:

1. If no server stores anything, Skeen's protocol must have either finished, or never started. Either way, there is nothing for the client to do.

2. If some servers store *Skeens 1* and the others servers store nothing, the previous client must have failed during *Skeens 1*. We must send *Skeens 1* to the remaining servers, and continue from there.

3. If some servers store *Skeens 1*, the rest store *Skeens 2*, the previous client must have failed while sending *Skeens 2*. We need to send *Skeens 2* to the servers which have not gotten it, and continue from there.

4. If some servers store *Skeens 2*, and the rest store nothing the previous client must have failed during garbage collection. We can simply garbage collect the remaining descriptors.

For liveness we additionally need to assume that, after some time has passed, an upper bound to message delay exists.

Performance and availability: The append protocol takes two phases in the fast path and three in the recovery path. The protocol can block if the logs being appended to reside on different sides of a network partition; however, the semantics of colors in FuzzyLog ensure that we only append to logs within a single region. Single-color appends follow the same protocol as multi-appends, but complete in a single phase that compresses the two phases of the fast path.

A subtle but critical point is that a missed fast path deadline will block other multi-appends from completing, but will not cause them to miss their own deadlines; they are free to complete the fast path and receive a timestamp, and only block in the delivery queue. As a result, a crashed client will cause a latency spike but not a cascading series of recoveries. Like all protcols, this protocol is subject to FLP [62]; since recovery clients can fence each other perpetually it can be susceptible to livelock. Our implementation mitigates this by having clients back-off for a small, randomized time-out if they encounter an ongoing recovery, before fencing it and taking over recovery.

5.1.4 Implementation

Dapple's design can be seen in Figure 5.4. On the chainserver, incoming requests are handled by a set of worker cores. When an append, or a snapshot of multiple chains, arrives, the worker core contacts a dedicated ordering core with the colors of the request to determine when the request should occur. All other work, including reads, IO and datastructure lookup is performed at a worker core. This is enabled by using a lock-free trie as the primary index for each locally stored chain.

When reading, the client fetches nodes from the server in a parallel manner, and relies on a node scheduling module to return nodes to the application in an order consistent with

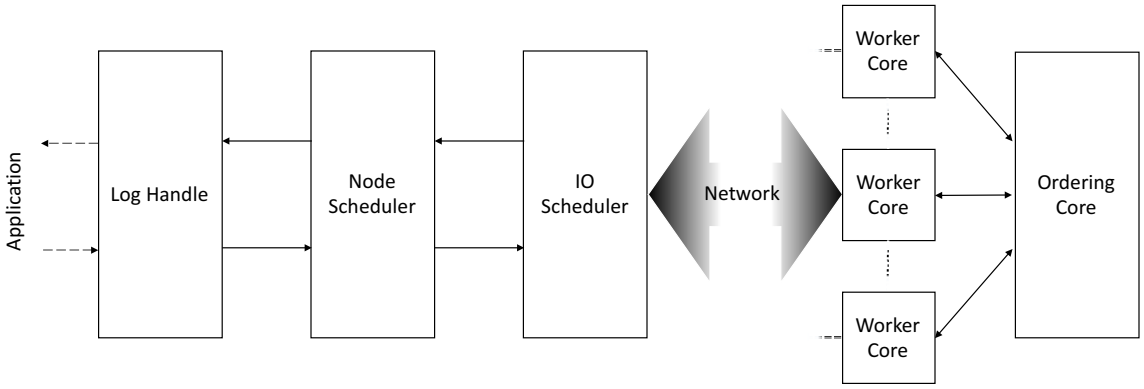


Figure 5.4: The module layout for Dapple.

the DAG. This module is also responsible for adding causal edges. The application interacts with the client library through a log handle, which provides the API described in section 2.

Chapter 6

Evaluation

We run all our experiments on Amazon EC2 using *c4.2xlarge* instances (8 virtual cores, 15 GiB RAM, Intel Xeon E5-2666 v3 processors). Most of the experiments run within a single EC2 region; for geo-distributed experiments, we ran across the us-east-2 (Ohio) and the ap-northeast-1 (Tokyo) regions, which are separated by an average ping latency of 168ms. In all experiments, we run Dapple with two replicas per partition, unless otherwise specified. All throughput numbers are without any application-level batching.

We first report latency micro-benchmarks for Dapple on a lightly loaded deployment. Figure 6.1 shows the distribution of latencies for 16-byte appends involving one color (top) and two colors on different chainservers (middle), as well as the latency to recover stuck multi-appends due to crashed clients (bottom). In all cases, latency increases with increasing replication factor due to chain replication. At every replication factor, single-color appends are executed with lower latency than two-color appends, which in turn require lower latency than two-color recovery. This difference in latency arises because single-color appends execute in a single phase, while two-color appends execute in two phase and two-color recoveries execute in three phases

The remainder of our evaluation is structured as follows: First, we evaluate the differences between Dapple and prior shared log designs (§6.0.1). Second, we use the Map variants from §2.1 to show that Dapple provides linear scaling with atomicity (§6.0.2), weaker consistency guarantees (§6.0.3), and network partition tolerance (§6.0.4). Finally, we describe a ZooKeeper clone over Dapple (§6.0.5).

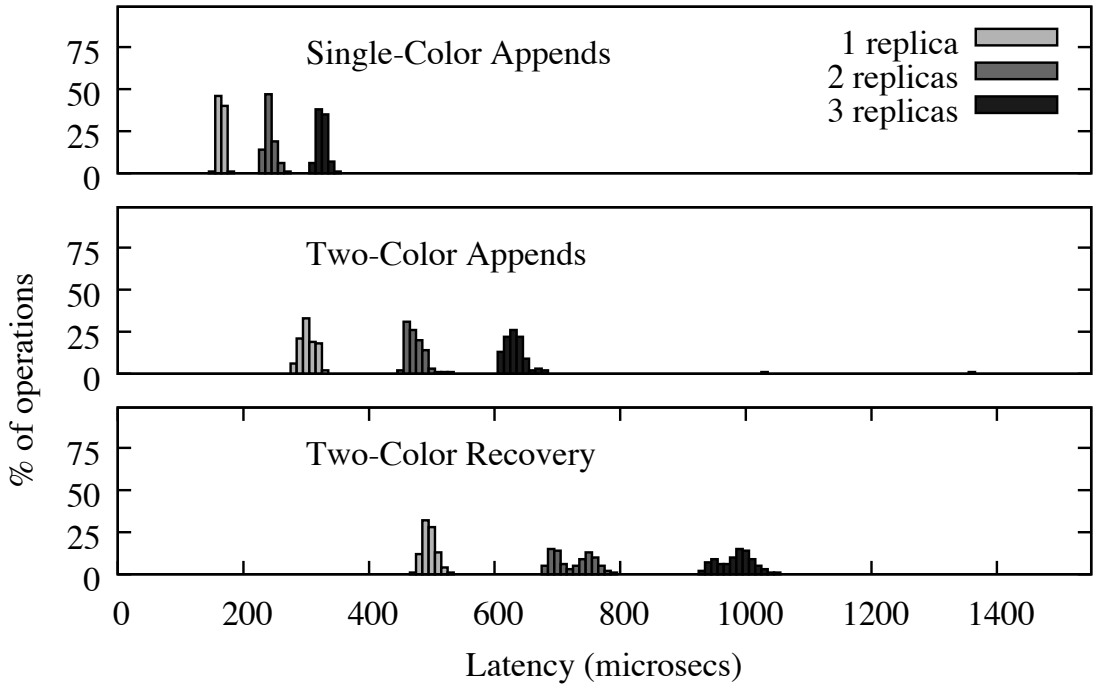


Figure 6.1: Dapple executes single-color appends in one phase; multi-color appends in two phases; and recovers from crashed clients in three phases.

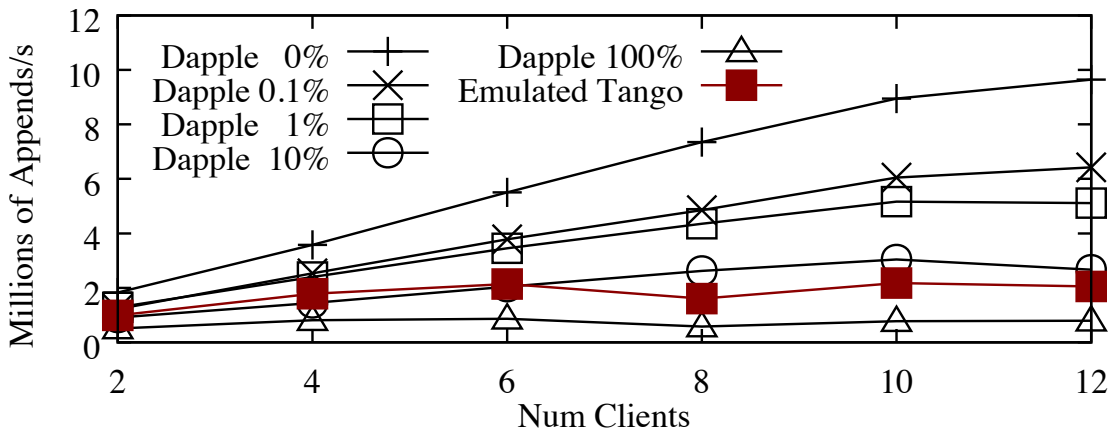


Figure 6.2: Dapple scales with workload parallelism, but a centralized sequencer bottlenecks emulated Tango.

6.0.1 Comparison with shared log systems

In this experiment, we show that centralized sequencers in existing shared log systems fundamentally limit scalability. Shared log systems, such as Tango [11] and vCorfu [14], use a centralized sequencer to determine a unique monotonic sequence number for each append. Based on its sequence number, each append is deterministically replicated on a different set of servers. The sequencer therefore becomes a centralized point of coordination, even when requests execute against different application-level data-structures or shards. In contrast, Dapple allows applications to naturally express their sharding requirements via colors, and can execute appends to disjoint sets of colors independently.

We emulate Tango's append protocol in Dapple by using five chainserver partitions to store data, and a single unreplicated server to disperse sequence numbers; given a sequence number, appends are deterministically written (via a Dapple-`append`) to one of the five chainserver partitions in a round-robin fashion. We compare this to a FuzzyLog deployment that uses five chainserver partitions. The number of partitions and replication factor in emulated Tango and Dapple are identical, while emulated Tango uses an extra server for sequencing. We run a workload where each client appends to a particular color, mixing single-color appends with a fixed percentage of appends that include a second, randomly picked color. Figure 6.2 shows average throughput over a 10-second run for workloads with different percentages of two-color appends. Emulated Tango cannot scale beyond four clients

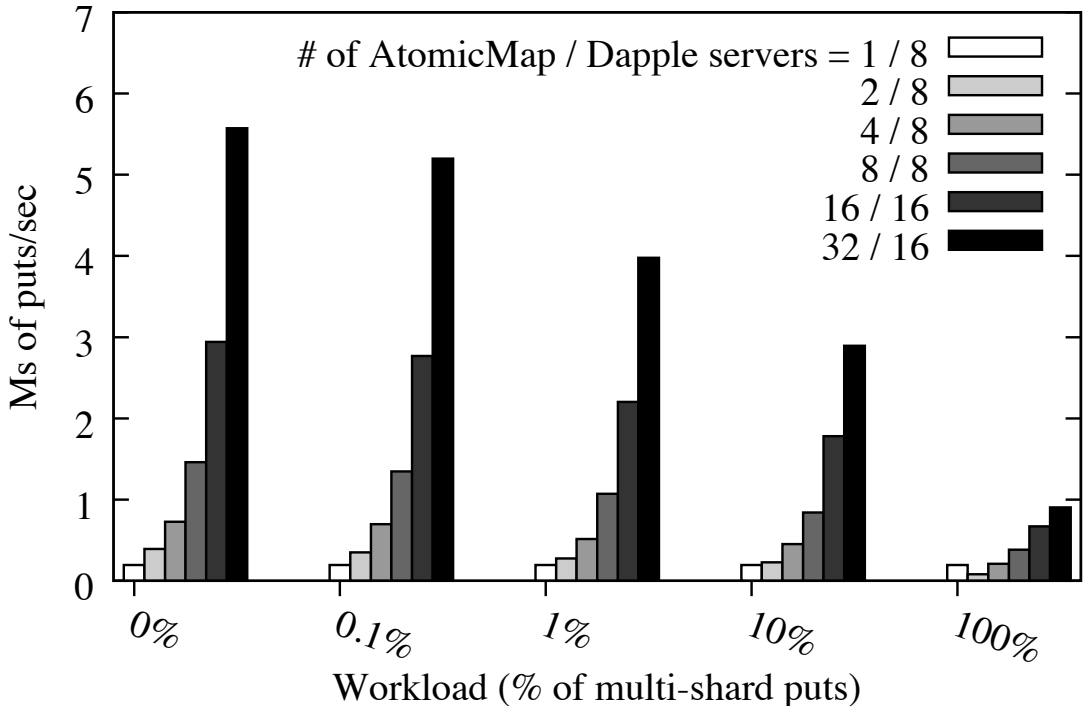


Figure 6.3: AtomicMap scales throughput while supporting multi-shard transactions.

due to its use of a centralized sequencer. Dapple scales near-linearly when the workload is fully partitionable (0% multi-color appends), is 2X faster at 1% multi-color appends, and matches Tango at 10% multi-color appends. At 100% multi-color appends, Dapple performs worse because the required partial order is nearly a total order, which Tango provides more efficiently.

6.0.2 Scalable multi-shard atomicity

The FuzzyLog allows applications to scale within a region by sharding across colors, and supports multi-shard transactions via multi-color appends. We now demonstrate the scalability of an AtomicMap (Section 2.1.1), which partitions its state across multiple colors. Each AtomicMap server is a Dapple client, and is affinitized with a unique color (corresponding to a logical partition of the AtomicMap's state). Each client performs a combination of single puts against its local partition and multi-puts against its partition and a randomly selected remote partition.

Figure 6.3 shows the results of the AtomicMap experiment. For different percentages

of multi-puts in the workload (on the x-axis), we vary system size and plot throughput on the y-axis. We use between 8 and 16 chainservers in Dapple (deployed without replication since we ran into EC2 instance limits). We use 8-byte keys and 8-byte values to emulate a workload where the AtomicMap acts as an index storing pointers to an external blob store. Keys for put operations are selected uniformly at random from a key space of 1M keys.

Figure 6.3 shows that under 0% multi-shard puts, throughput scales linearly from 1 to 16 AtomicMap servers. The throughput jump from 16 to 32 servers is slightly less than 2x because we pack two Dapple clients per AtomicMap server at the 32 client data point (due to the EC2 instance limit). As the percentage of multi-shard puts increases from 0.1% to 100%, scalability and absolute throughput degrade gracefully. This is expected due to the extra cost of executing multi-shard puts (each requires a two-phase multi-color append).

6.0.3 Weaker consistency guarantees

Dapple allows geo-distributed applications to perform updates to the same color with low latency. By composing a single color out of multiple totally ordered chains, one per geographical region, a client in a particular region can append updates to a color without performing any coordination across regions in the critical path. This section demonstrates this capability via a CRDTMap.

In Figure 6.4, we host a single, unpartitioned CRDTMap on five application servers (i.e., Dapple clients); we locate each in a virtual region with its own Dapple copy, all running in the same EC2 region. Four of these servers are writers issuing puts at a controlled aggregate rate (left y-axis), while the fifth is a reader issuing gets on the CRDTMap. Each writing server uses four writer processes. The gets observe some frontier of the underlying DAG, and can therefore lag behind by a certain number of puts (right y-axis), but are fast, local operations. Midway through the experiment, we spike the put load on the system; this does not slow down gets at the reader (not shown in the graph), but instead manifests as staleness.

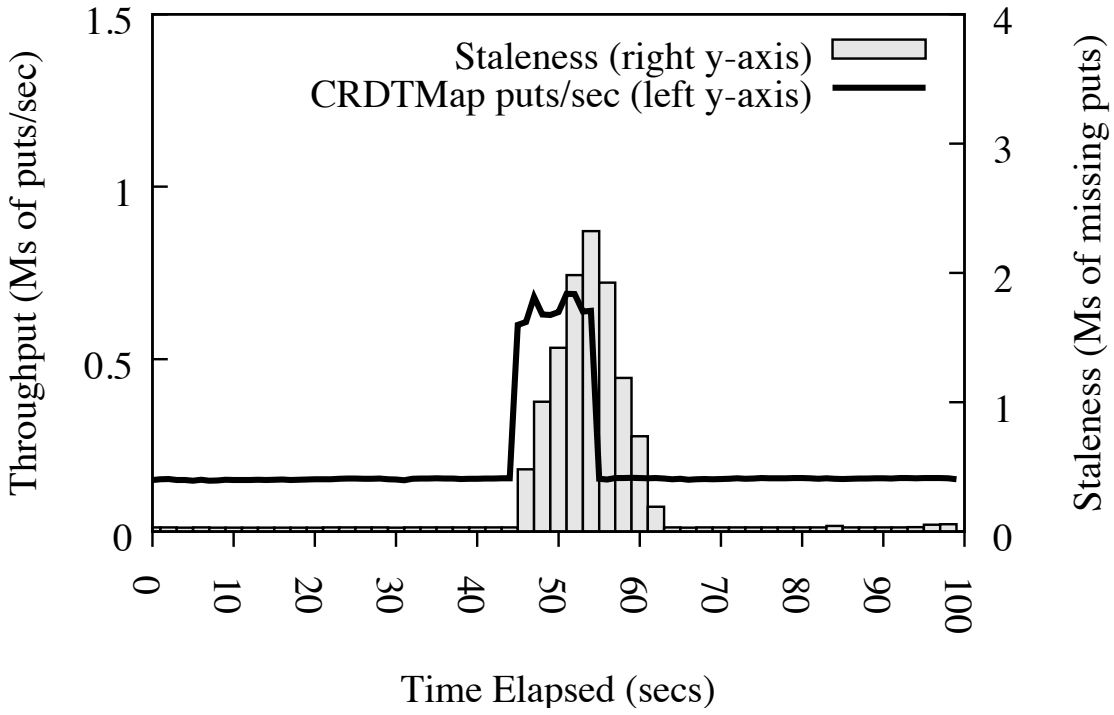


Figure 6.4: CRDTMap provides a trade-off between throughput and staleness.

6.0.4 Network partition tolerance

Dapple allows applications to provide strong consistency during normal operation and weak consistency under network partitions. In this experiment, we demonstrate this capability by running CAPMap across a primary and a secondary region (us-east-2 and ap-northeast-1, respectively). The experiment lasts for 14 seconds. From 0-6 seconds, the primary and secondary regions are connected. Between 6-8 seconds, we simulate a network partition between the primary and secondary. Finally, from 8-14 seconds, connectivity between the primary and secondary is restored. Each region runs two servers, one issuing puts and one issuing gets. We measure the latency of gets and puts (y-axis), against the wall-clock time they are issued at (x-axis).

Figure 6.5 shows the results of the experiment. In normal operation (0 to 6 seconds), all updates are stored in a single primary chain, and both regions get strong consistency; the secondary has high latencies for puts and gets due to the 168 ms inter-region roundtrip it incurs to access the primary chain. At 6 seconds, the network between the regions partitions; the primary continues to obtain strong consistency and low latency, but the

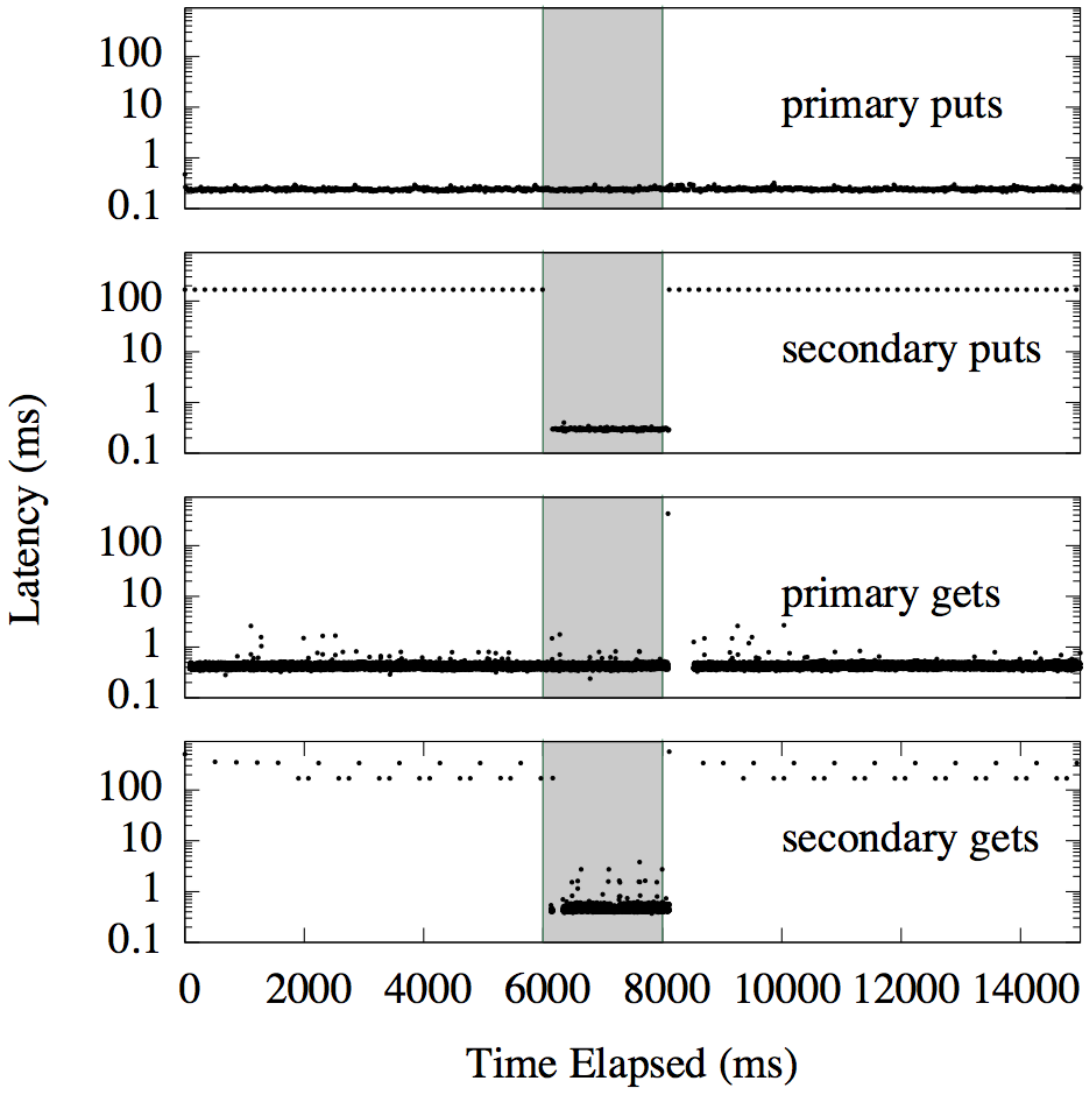


Figure 6.5: CAPMap switches between linearizability and causal+ consistency during network partitions.

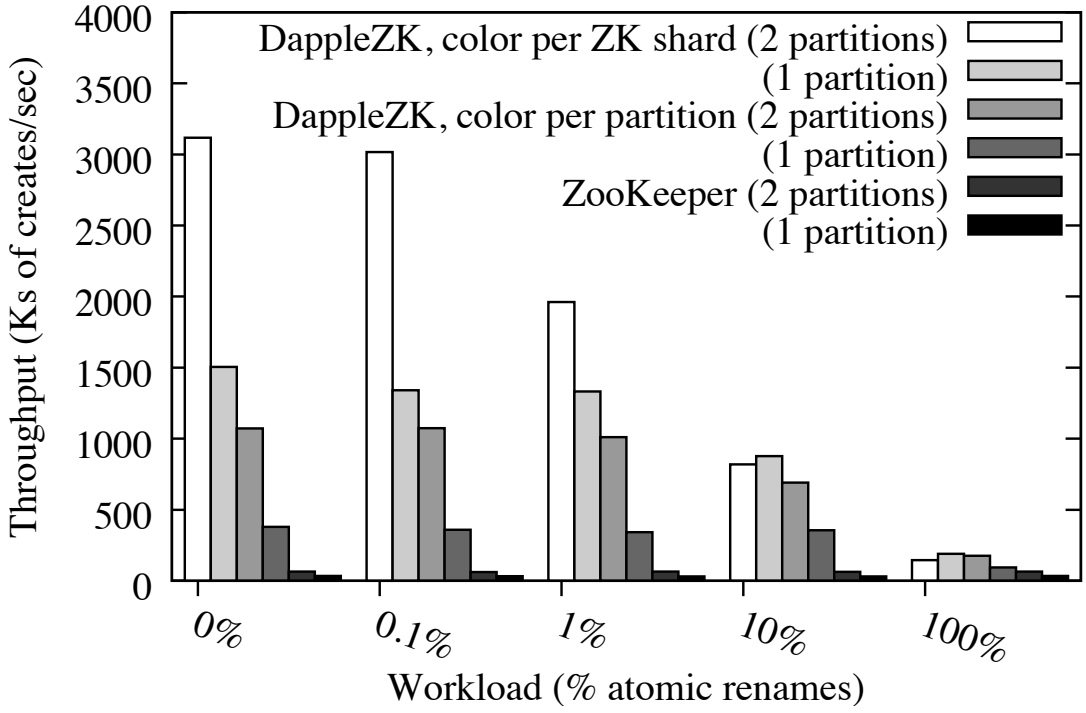


Figure 6.6: DappleZK exploits Dapple's partial ordering to implement a scalable version of the ZooKeeper API.

secondary switches to weaker consistency, storing its updates on a local secondary chain (and obtaining much lower latency for puts/gets in exchange for the weaker consistency). At 8 seconds, the network heals; the secondary appends a joining node to the primary chain via a proxy in the primary zone. As part of this joining request, the secondary provides a snapshot ID reflecting its last appended node to its local chain. The proxy at the primary waits until the nodes in the snapshot are replicated to the primary zone and seen by it before completing the joining append. The joining append causes a high latency put by the secondary just after the partition heals, and a spike in get latency on the primary as it plays nodes appended to the secondary chain during the partition.

6.0.5 End-to-end applications

We implemented a ZooKeeper clone, DappleZK in 1881 LOC of Rust. DappleZK partitions a namespace across a set of servers, each of which acts as a Dapple client, storing a partition of the namespace in in-memory data-structures backed by a FuzzyLog color.

This section compares DappleZK's performance with ZooKeeper. Each DappleZK server is responsible for an independent shard of the ZooKeeper namespace, and atomically creates and renames files using optimistic read-write transactions. Create operations are restricted to a single DappleZK shard. Each rename moves a file from one DappleZK shard to another via the distributed transaction protocol described in Section 2.1.1.

We partition the ZooKeeper namespace across 12 DappleZK shards, and run one DappleZK server per shard. We deploy Dapple with either one or two partitions. Each partition is configured with three replicas. DappleZK uses two coloring schemes; a color per partition and a color per DappleZK shard. In the color per partition deployment, each color holds updates corresponding to multiple DappleZK server shards.

We run conventional ZooKeeper with three replicas, and also include a partitioned ZooKeeper deployment with two partitions. Our ZooKeeper deployments keep their state in a memory-backed ramdisk. Note that ZooKeeper does not support atomic renames; we emulated renames on it by executing a delete and create operation in succession. We include the ZooKeeper comparison for completeness; we expect the FuzzyLog single-partition case to outperform ZooKeeper largely due to the different languages used (Rust vs. Java) and the difference between prototype and production-quality code.

Figure 6.11 shows the results of the experiment. We vary the percentage of renames in the workload on the x-axis, and plot throughput on the y-axis. Each x-axis point shows a cluster of bars corresponding to the four DappleZK configurations, and two ZooKeeper configurations. With a single color and a single partition, every DappleZK server stores its state on the same color. DappleZK servers perform their appends and reads against the same color, which limits their throughput. With two partitions, the number of DappleZK servers per color is halved, which increases throughput. When we switch to a color per DappleZK server, throughput increases dramatically because requests from different DappleZK servers do not need to be serialized against the same color. The addition of another partition further increases throughput because the colors can be spread across two partitions. When deployed with a single partition, Dapple servers were overloaded, which led to extra scheduling overhead and caused the two partition case to outperform a single partition by over 2X (in both color per ZK shard and color per partition cases). With an increasing

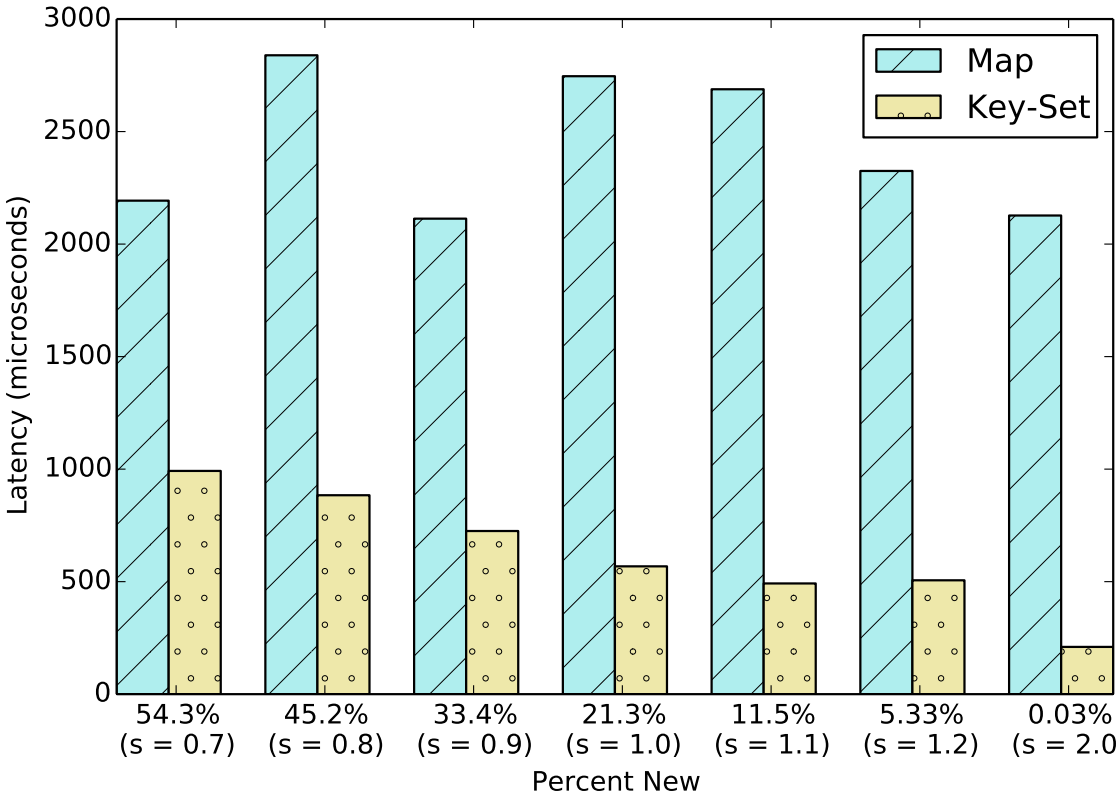


Figure 6.7: Key-set sync latency. The time taken to synchronize a key-set view is a fraction of the time taken to synchronize the full view, based on the proportion of new entries.

fraction of atomic renames, throughput decreases because DappleZK must perform a distributed transaction across the involved DappleZK servers. In comparison to DappleZK, ZooKeeper provided 36K and 66K ops/s with one and two partitions respectively. Further, the DappleZK views still had a sufficient amount of resources remaining to serve a similar number of reads.

6.1 FuzzyViews

For FuzzyViews our evaluation focuses on a key metric for SMR systems that we call *sync latency*: this is the latency to synchronize the local view at a learner with the underlying total order. Equivalently, sync time is the latency to fetch and locally apply any relevant updates which have completed before the query was issued. Sync latency is a fundamental metric for any kind of SMR system, regardless of whether the underlying total order is

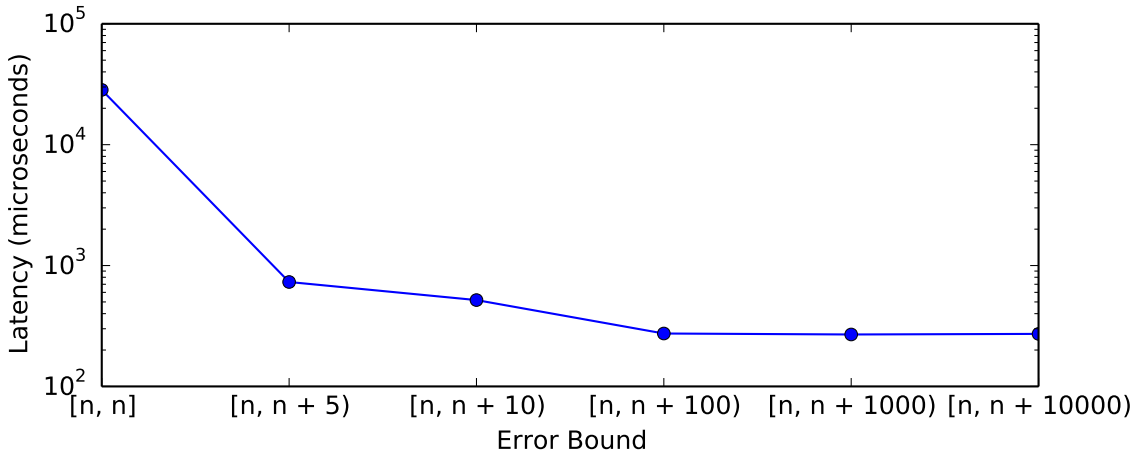


Figure 6.8: Counter sync latency. Latency to synchronize a bounded error counter (y-axis) decreases as the acceptable error (x-axis) increases

implemented via Multi-Paxos, group communication, or a shared log.

For micro-benchmarks we used versions of the bloom-filter like key-set and absolute-error counter. For a case study on using FuzzyViews in a real application, we extended a Zookeeper implementation. Unless otherwise specified, all our experiments are based on servers synchronizing with the shared log while a concurrent writer appends at roughly 350K updates per second.

Key-set (function acceleration). In Figure 6.7, we compare the key-set (described in Section 3.1) against a server that materializes the full map. Note that our key-set can only answer queries for whether a key exists in the map. We use a zipfian distribution for the keys, and the ratio of new inserts to updates is determined by the distribution: the less skewed the distribution, the higher this ratio. As expected, as the proportion of new inserts decreases, and thus the number of updates that are irrelevant to the key-set increases, the time to synchronize the key-set decreases, until it eventually reaches a single round-trip time to the sequencer node of the shared log. Shared log systems are poll based, so learners always need to check to see if any new updates have appeared before they are confident that their view is valid; in the optimal case, no relevant updates have been added and the old view can be used to satisfy the new query. The round trip time to the sequencer thus represents the absolute minimum time a query can take.

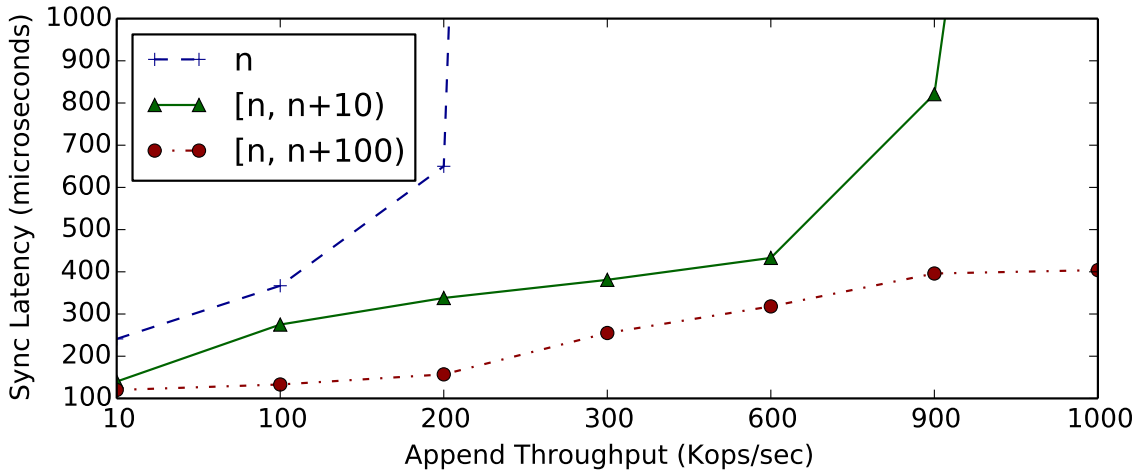


Figure 6.9: Sync latency vs Append rate. Counters with greater acceptable error need to see fewer updates, and therefore can handle a greater append rate before becoming oversaturated. (Note log scale)

Counter (approximation). The connection between approximation error and synchronization latency can be seen more directly in our counter implementation. In Figure 6.8, we evaluate our bounded-error counter, with thresholds of 5, 10, 100, 1000, and 10000, against fixed increments at 500 kHz. As the acceptable error increases, the number of updates the client needs to fetch, and the time taken to update the view decreases. With a high enough error bound, the view is nearly always valid and it need not fetch any updates a majority of the time; the sync latency reduces to a single round trip, validating that the current view is not yet stale.

In a log based system, sync latency is roughly proportional to the number of updates fetched, until updates are added to the log more rapidly then they are read. After that point, sync latency grows without bound, as the readers fall more and more behind the writers; once this occurs, readers can never catch up until writers spend a significant duration appending at a rate below the average synchronization rate, and the pending updates get drained. The effect of this on FuzzyViews can be seen in Figure 6.9, where we show the append rate versus synchronization latency for a number of error-bounds in our bounded-error counter. As can be seen, not only does latency decrease as acceptable error increases, but the fuzzier views can support a greater rate of increments before slowing down precipitously. This implies a system based mainly on FuzzyViews could support a greater

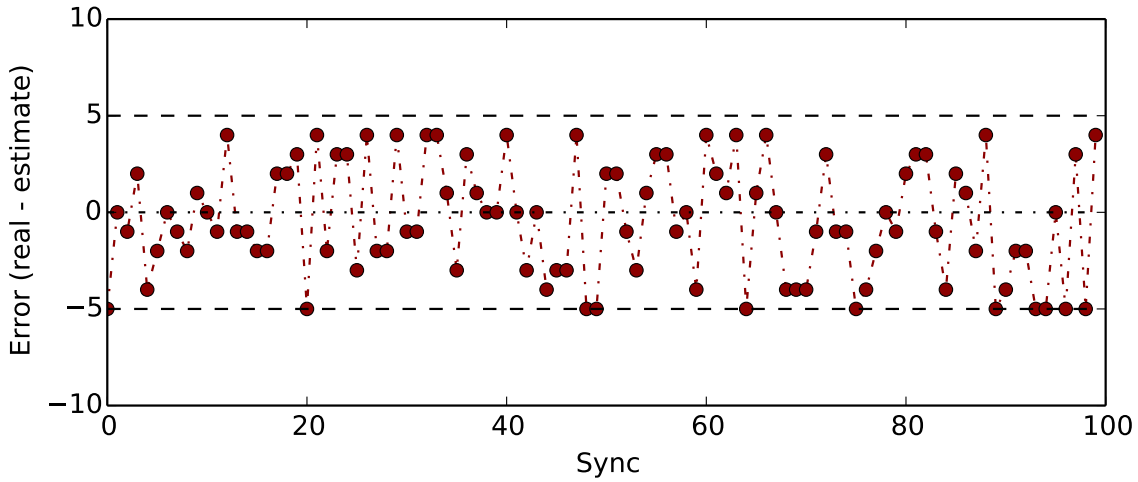


Figure 6.10: Counter error over time. Despite the underlying structure of the bounded error counter (threshold = 10), reads remain randomly distributed within its acceptable range during a series of increments.

write throughput, if the workload in amenable. If the client only needs various FuzzyViews and never materializes the baseline object then the increased throughput would be sustainable. However, if the application needs to materialize a full view on occasion, then the average throughput will still be bottlenecked by that view; while FuzzyViews allow write spikes greater than what the full view would be able to handle, eventually this will need to matched by a quiescent period in which the full view is allowed to catch up.

Conveniently, the log based nature of the system allows us to empirically evaluate the accuracy of our counter; by recording the size of *both* the ground-truth color, and our approximation color when we sync, we can materialize the state of the counter our view would have seen, had it been reading the entire state. Materializing the full view needs to be done off-line due to the faster rate at which we can materialize the FuzzyView; if we were to wait to read in the full state of the counter, our FuzzyViews would be updated much less frequently than they would be in a real system. However, since the full state remains in the log, this does not pose an issue.

We instrumented a version of the bounded error counter with a threshold of 10, and ran it against an incrementor running at roughly 600 kHz; the delta between the real and approximate values for a series of 100 syncs from this run can be seen in Figure 6.10. Despite the implementation tending to have something of a sawtooth pattern on this workload, with

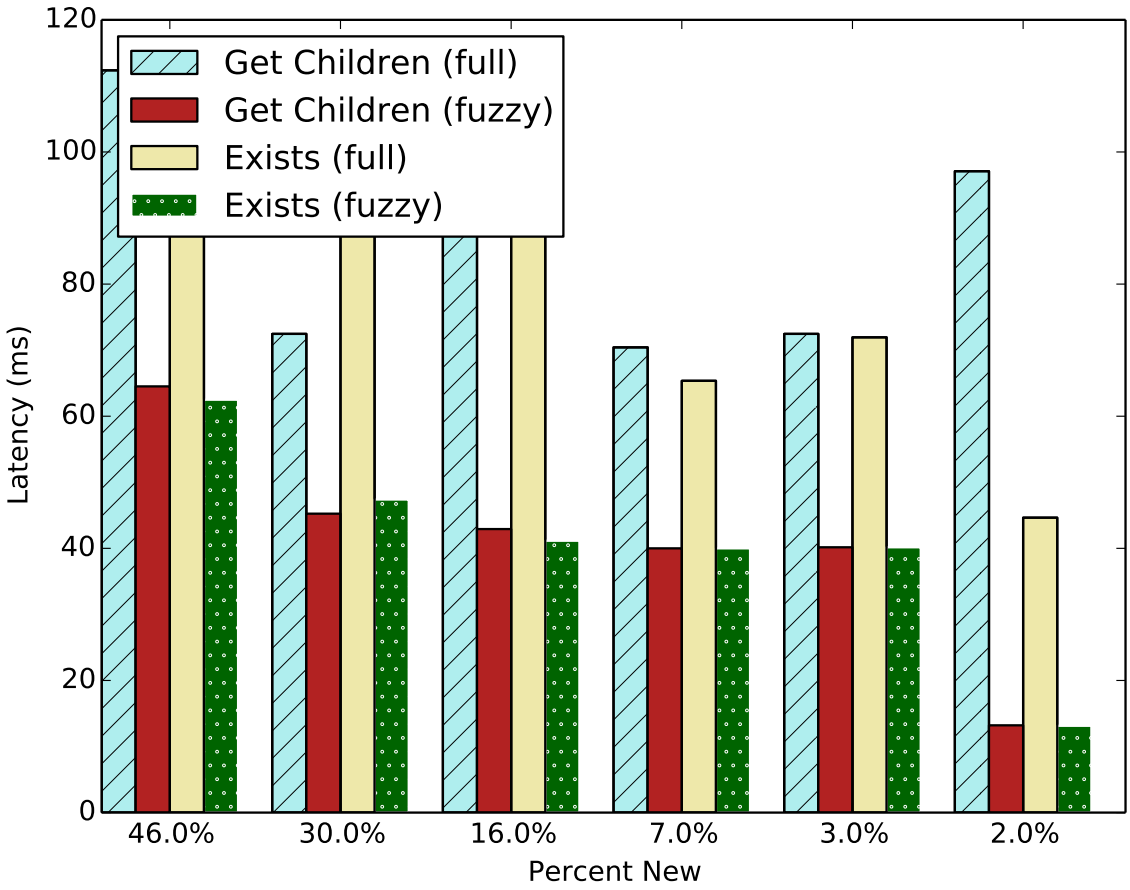


Figure 6.11: Zookeeper sync latency. The time to synchronize metadata based views increases as the rate of new inserts increases.

error increasing from new increments until the threshold is reached, at which point it resets to 0, in practice the observed error was uniformly distributed over the possible values since the sync timings was not coupled to the appends.

Zookeeper (function acceleration). For a more realistic example, we extended our Zookeeper with FuzzyViews for `exists` and `get_children`. These views used the same application code as our original Zookeeper implementation; the only changes needed were to add tagging at send time and to restrict our FuzzyViews' servers to only read the needed colors. These views were run against a work generator running at approximately 350 kHz for the synchronization latency times seen in Figure 6.11. Much as in the Key-set example above as the proportion of updates which mutate the metadata increases the FuzzyView's sync latency increases.

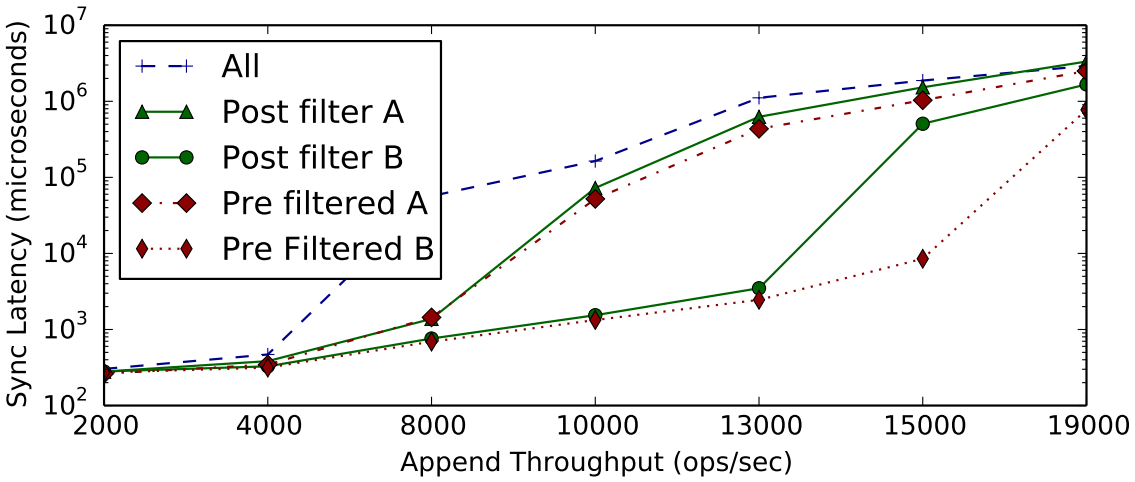


Figure 6.12: ML sync latency vs Append rate. In the ensemble learning scenario learning takes a significant amount of time, thus post-hoc filtering is partially effective, though not as effective as pre-hoc. In each ensemble (A and B) the pre-filtered version is still faster than the post-filtered version (Note log scale)

Ensemble Learning. The ensemble learning example, Figure 6.12, represents something of a worse case for SMR, with unusually large updates and a very significant amount of work per update. This example is dominated by the difference in number of updates between cohorts A and B, with a majority of updates falling in A. Due to the significant amount of time it takes to learn a new example, post-hoc filtering is useful, providing much of the speedup of pre-hoc filtering.

Chapter 7

Conclusion and Future Work

The shared log provides a compelling model for building distributed systems: it offloads much of the complexity in building such systems—in particular, consistency and durability—from the application layer to the system layer. However, while the totally ordered shared log is a simple abstraction, it is also restrictive, tightly bounding the scalability and consistency of the resulting software; totally ordered logs only work for applications which can handle centralization of control. The FuzzyLog expands upon the shared log approach with *partially ordered* logs. This weakening of log semantics enables the usage of the shared logs in domains that were previously inaccessible. By partially ordering the log we are able to order subsets of the logs independently from each other. Within a single datacenter this allows the logs to scale-out, since independent data shards no longer need to coordinate through a centralized sequencer. Even more significantly, this lack of centralization enables shared-log-based systems to span geographic regions in a fault-tolerant manner, since portions of the log can continue to make progress even when network connectivity is lost. Crucially, applications can achieve these capabilities in hundreds of lines of code via simple, data-centric operations on the FuzzyLog, retaining the core simplicity of the shared log approach. Dapple, our realization of the FuzzyLog, implements the abstraction in a performant manner, built upon a classic message-ordering protocol which we extended with the ability to handle client and server failures. Compared to traditional SMR, the FuzzyLog abstraction, and its implementation in Dapple, provides a significantly expanded set of capabilities, with additional complexity only when necessitated by the semantics of

these features.

We further extend Dapple by filtering out events within a single, totally ordered, sub-history. Choosing only the events significant to the problem at hand, objects built from such sub-histories can be constructed more efficiently, vastly reducing the latency on queries. Removing sensitive events from an objects' sub-history permits objects to be built which only contain information that is needed for the specific use case at hand, reducing the risk that private information will be leaked. The objects created by this technique, which we call FuzzyViews, retain all the benefits of the underlying shared log, and are every bit as serializable and durable as the full object. FuzzyViews are flexible, and can be used in tandem with other FuzzyViews, or even full instantiations of the object, without compromising these benefits; since both the FuzzyView and the full object are backed by the same log, updates to the log are seen by both of them. This enables the usage of a combination of FuzzyViews and full objects as needed by an application. For instance, one can use a FuzzyView to accelerate particularly common queries, while falling back to the full object for those queries which the FuzzyView cannot handle.

One of the main challenges involved in constructing our implementation of the FuzzyLog, Dapple, was ensuring that the distributed message ordering protocol was both correct and as performant as possible. Two proofs will be of particular interest to a general audience: we proved the correctness of our failure handling protocol, which can be used by other systems in need of message ordering; we also discovered lower and upper bounds on the latency of scalable serializable and strictly serializable transactions. In short, there is a tradeoff between throughput and latency: if one wishes to construct a strictly serializable transaction protocol that communicate with the minimum number of servers, it will have a worst-case time at least linear in the number of independent servers. This is particularly notable since the worst case latency for such protocols which are merely serializable is constant. To our knowledge this is the first such latency bound, and the performance degradation inherent in strict serializability is important to note for those choosing which semantics a distributed system should implement. This tradeoff becomes more subtle since we demonstrate that there are general techniques to transform a serializable protocol into a strictly serializable one, implying that it is possible, and may be beneficial, to implement

only the former at the system layer while leaving the latter to the application.

Both the FuzzyLog and FuzzyViews provide novel ways in which the inherent ordering of events in a system, or lack thereof, can be exploited to enhance the efficiency and power of distributed systems, and our implementation in Dapple demonstrates that their abstractions can be realized in an effective manner.

Bibliography

[1] F. B. Schneider. The state machine approach: A tutorial. In *Fault-tolerant distributed computing*, pages 18–41. Springer, 1990.

[2] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[3] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[4] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, pages 517–532, 2016.

[5] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.

[6] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–12, 2011.

[7] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1295–1309, New York, NY, USA, 2015. ACM.

[8] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. 2015.

[9] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[10] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: an architecture for scale-out of olxp workloads. *Proceedings of the VLDB Endowment*, 8(12):1716–1727, 2015.

[11] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, New York, NY, USA, 2013. ACM.

[12] M. Bevilacqua-Linn, M. Byron, P. Cline, J. Moore, and S. Muir. Sirius: Distributing and coordinating application reference data. In *USENIX Annual Technical Conference*, pages 293–304, 2014.

[13] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Chariots: A scalable shared log for data management in multi-datacenter cloud environments.

[14] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, et al. vcorfu: A cloud-scale object store on a shared log.

[15] Zlog transactional key-value store. http://noahdesu.github.io/2016/08/02/zlog-kvstore-intro.html.

[16] M. Wei, C. Rossbach, I. Abraham, U. Wieder, S. Swanson, D. Malkhi, and A. Tai. Silver: a scalable, distributed, multi-versioning, always growing (ag) file system. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[17] A. Thomson and D. J. Abadi. Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems. In *FAST*, pages 1–14, 2015.

[18] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[19] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, volume 12, pages 265–278, 2012.

[23] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, pages 405–414. IEEE, 2016.

[24] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM.

[25] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[26] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[27] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[28] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42, 2015.

[29] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[31] L. Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[32] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *OSDI*, volume 12, pages 237–250, 2012.

[33] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[34] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 56–65. IEEE, 1994.

[35] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[36] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 279–294. ACM, 2015.

[37] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *OSDI*, volume 14, pages 495–509, 2014.

[38] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, volume 14, pages 479–494, 2014.

[39] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

[40] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[41] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[43] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[44] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. *Flexible update propagation for weakly consistent replication*, volume 31. ACM, 1997.

[45] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1615–1628. ACM, 2016.

[46] L. Benmouffok, J. Busca, J. M. Marquès, M. Shapiro, P. Sutra, and G. Tsoukalas. Telex: Principled system support for write-sharing in collaborative applications. *CoRR*, abs/0805.4680, 2008.

[47] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, pages 331–345, Berlin, Heidelberg, 2005. Springer-Verlag.

[48] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 295–310. ACM, 2015.

[49] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[50] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[51] E. A. Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[52] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400. ACM, 2011.

[53] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*. USENIX Association, 2000.

[54] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[55] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[56] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[57] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag.

[58] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[59] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, 2014.

[60] J. Langford and T. Zhang. The epoch-greedy algorithm for contextual multi-armed bandits. In *Proceedings of the 21st Advances in Neural Information Processing Systems (NIPS)*, 2007.

[61] M. C. L. H. J. L. S. L. J. L. D. M. G. O. O. R. S. S. A. S. Alekh Agarwal, Sarah Bird. A multiworld testing decision service. *CoRR*, abs/1606.03966, 2016.

[62] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[63] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[64] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 135–150, Berkeley, CA, USA, 2016. USENIX Association.

[65] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Cambridge, MA, USA, 1999. AAI0800775.

[66] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 578–585. IEEE, 1997.

[67] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[68] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 54–70. ACM, 2015.

[69] A. Kalia, M. Kaminsky, and D. 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 (OSDI 16)*, pages 185–201. USENIX Association, 2016.

[70] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[71] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. volume 23. ACM, 1989.

[72] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*, pages 840–847. IEEE, 1998.