The FuzzyLog: A Partially Ordered Shared Log

Joshua Lockerman
Yale University

†Jose M. Faleiro
UC Berkeley

†Juno Kim
UC San Diego

†Soham Sankaran
Cornell University

Daniel J. Abadi
University of Maryland, College Park

James Aspnes
Yale University

Siddhartha Sen
Microsoft Research

†Mahesh Balakrishnan
Yale University / Facebook

† *Work done while authors were at Yale University*

Abstract

The FuzzyLog is a partially ordered shared log abstraction. Distributed applications can concurrently append to the partial order and play it back. FuzzyLog applications obtain the benefits of an underlying shared log – extracting strong consistency, durability, and failure atomicity in simple ways – without suffering from its drawbacks. By exposing a partial order, the FuzzyLog enables three key capabilities for applications: linear scaling for throughput and capacity (without sacrificing atomicity), weaker consistency guarantees, and tolerance to network partitions. We present Dapple, a distributed implementation of the FuzzyLog abstraction that stores the partial order compactly and supports efficient appends / playback via a new ordering protocol. We implement several data structures and applications over the FuzzyLog, including several map variants as well as a ZooKeeper implementation. Our evaluation shows that these applications are compact, fast, and flexible: they retain the simplicity (100s of lines of code) and strong semantics (durability and failure atomicity) of a shared log design while exploiting the partial order of the FuzzyLog for linear scalability, flexible consistency guarantees (e.g., causal+ consistency), and network partition tolerance. On a 6-node Dapple deployment, our FuzzyLog-based ZooKeeper supports 3M/sec single-key writes, and 150K/sec atomic cross-shard renames.

1 Introduction

Large-scale data center 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 such as durability, high availability, and scalability are 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, while custom solutions can require melding application code with a medley of distributed protocols such as Paxos [29] and Two-Phase Commit (2PC) [21], which are individually complex, slow/inefficient when layered, and difficult to merge [40, 60].

A recently proposed class of designs centers on the *shared log abstraction*, funneling all updates through a globally shared log to enable fault-tolerant databases [9–11, 19, 51], metadata and coordination services [8, 12], key-value and object stores [3, 41, 57], and filesystem namespaces [50, 56]. Services built over a shared log are simple, compact layers that map a high-level API to append/read operations on the shared log, which acts as the source of strong consistency, durability, failure atomicity, and transactional isolation. For example, a shared log version of ZooKeeper uses 1K lines of code, an order of magnitude lower than the original system [8].

Unfortunately, the simplicity of a shared log requires imposing a system-wide total order that is *expensive*, often *impossible*, and typically *unnecessary*. Previous work showed 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) [7, 8]. 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 geographical 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 to disjoint data (e.g., different keys in a map) do not need to be ordered, while updates that touch the same data may commute because the application requires weak consistency guarantees (e.g., causal consistency [5]). In this paper, we explore the following question: *can we provide the sim-*

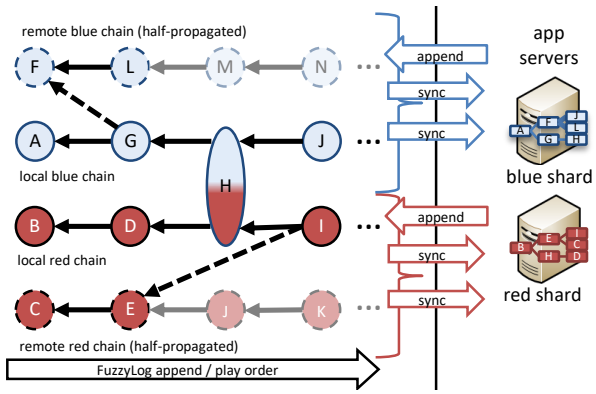

Figure 1: *In the FuzzyLog, each color contains updates to a data shard, while each chain contains updates from a geographical region.*

plicity of a shared log without imposing a total order?

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). 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. Internally, 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 region and lazily synchronized, so that each region has the latest copy of its own chain, but some stale prefix of the chains of other regions. Figure 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 cross-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 forward new nodes in 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, 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; it then plays forward 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; the FuzzyLog degenerates to a totally ordered shared log, and the simple protocol described above provides linearizability [23], 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 [5] within a shard by using a color across regions; and toggle between strong and weak consistency when the network partitions and heals 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 on 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 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 a map that provides Red-Blue consistency [32], as well as a transactional CRDT [6].

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 (§3): a durable, iterable DAG of colored nodes representing the partial order of updates in a distributed system. We argue that this abstraction is *useful* (§4), 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 *practically feasible* (§5), describing and evaluating a scalable, fault-tolerant implementation called Dapple.

2 Motivation

The shared log approach makes distributed services simple to build by deriving properties such as durability, consistency, failure atomicity, and concurrency control via simple append/read operations on a shared log abstraction. We describe the pros and cons of this approach.

2.1 The simplicity of a shared log

In the shared log approach, application state resides in the form of in-memory objects backed by a durable, fault-tolerant shared log. In effect, an object exists in two forms: an ordered sequence of updates stored durably in the shared log; and any number of views, which are full or partial copies of the data structure in its conventional form – such as a tree or a map – stored in DRAM on clients (i.e., application servers). Importantly, views constitute soft state and are instantiated, reconstructed, and updated on clients as required by playing the shared log forward. A client modifies an object by appending a new update to the log; it accesses the object by first synchronizing its local view with the log.

As described in prior work [7, 8], this design simplifies the construction of distributed systems by extracting key properties via simple appends/reads on the shared log, obviating the need for complex distributed protocols. Specifically, the shared log is the source of consistency: clients implement state machine replication [46] by funneling writes through the shared log and synchronizing their views with it on reads. The shared log also provides durability: clients can recover views after crashes simply by replaying the shared log. It acts as a source of failure atomicity and isolation for transactions: the shared log is literally the serializable order of transactions.

2.2 The drawbacks of a total order

The shared log approach achieves a total order over all updates in a distributed system. We argue that such a total order can be *expensive* or *impossible* to achieve when services scale beyond the confines of a small cluster.

Total ordering is expensive. The traditional way to impose a total order is via a leader that receives updates from clients and sequences them; however, this limits the throughput of the system at the I/O bandwidth of a single machine [16]. CORFU [7] uses an off-path sequencer – instead of a leader – that issues tokens or contiguous positions in an address space to clients. To append data, a client first obtains a token from the sequencer – effectively reserving an address in the address space – and then writes the payload directly to a stripe of storage servers responsible for storing that address. This allows clients to totally order updates to a cluster of storage servers without pushing all I/O through a single machine; instead, the aggregate throughput of the system is limited by the speed at which the sequencer can update a counter and hand out tokens (roughly 600K ops/sec in CORFU [8]). To leverage the total order without requiring all clients to play back every entry, runtimes built over CORFU such as Tango [8] and vCorfu [57] support selective playback via streams and materialized streams, respectively. This requires sequencer state to be more complex than a single counter (e.g., per-stream backpointers [8] or additional stream-specific counters [57]).

While an off-path sequencer works well for small clusters (e.g., 20 servers in two adjacent racks [8]), it does not scale along a number of key dimensions. One such dimension is *network diameter*: since the sequencer lives in a fixed point in the network, far-away clients must incur expensive round-trips on each append. A second dimension is *network bandwidth*; sequencers are not I/O-bound or easily parallelizable, and cannot keep pace with recent order-of-magnitude increases in I/O bandwidth. On 1 Gbps networks, a sequencer that runs at 600K ops/s can support a 20-server CORFU deployment (1 Gbps per server or 30K 4KB appends/sec); however, on a 40 Gbps network, supporting 20 servers will require the sequencer to run at 24M ops/s. A third dimension is *payload granularity*: shared log applications do not store large payloads (in the limit, these could be 64-bit pointers to bigger items stored in some external blob store). With 100-byte payloads, the same sequencer will now have to run at nearly 1 billion ops/s. A final dimension is *system size*:

```
// 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: *The FuzzyLog API.*

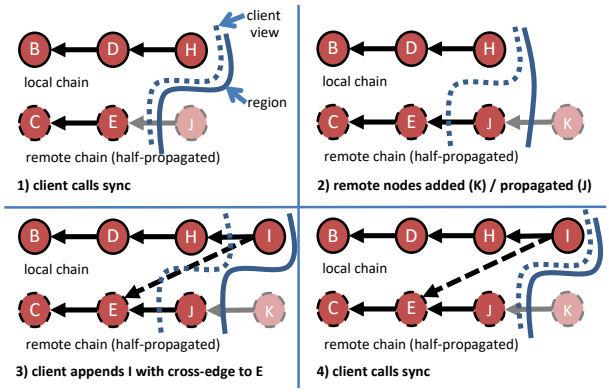

Figure 3: *The evolution of a single color.*

if we want to support 40 servers, we now need 2 billion ops/s from the sequencer.

Published numbers for sequencers in fully functional systems include: roughly 200K ops/sec (CORFU [7]), 250K ops/sec (NOPaxos [33]), and 600K ops/sec (Tango [8]). Stand-alone sequencers (i.e., simple counters without per-stream state) are faster; e.g., an RDMA-based counter runs at 122M ops/sec (80X faster than the next highest in the literature) [25]. Even at this speed, the largest cluster supported at 100 Gbps and a 512-byte payload would have just four servers.

Some approaches bypass the sequencer throughput cap at the cost of increasing append latency, either by aggressive batching [51] or writing out-of-order in the shared log address space and waiting for preceding holes to be filled [41]. The added append latency can be untenable for control plane services.

Total ordering is impossible. Regardless of how the total order is generated, it is fundamentally vulnerable to network partitions. Any protocol that provides a total order consistent with a linearizable order (i.e, if an update *B* starts in real time after another update *A* completes, then *B* occurs after *A* in the total order) is subject to unavailability during network partitions [14].

We find ourselves at a seeming impasse: a shared log enables simplicity and strong semantics for distributed systems, but imposes a total order that is expensive and sometimes impossible. We break this impasse with a partially ordered shared log abstraction.

3 The FuzzyLog Abstraction

The FuzzyLog addresses the ordering limitations in Section 2 via an expressive partial ordering API. The FuzzyLog's API captures two general patterns via which applications partially order operations. First, applications partition their state across logical data shards, such that updates against different shards are processed concurrently. Second, when deployed across geographical regions, applications weaken consistency to avoid synchronous cross-region coordination on the critical path of requests; as a result, updates across regions – even to the same logical data partition – can occur concurrently.

A FuzzyLog is a type of directed acyclic graph (DAG) that can be constructed and traversed concurrently by multiple clients. For clarity, we use the term 'node' exclusively to refer to nodes in the FuzzyLog DAG. Each node in the DAG is tagged with one or more colors. Colors divide an application's state into logical shards; nodes tagged with a particular color correspond to updates against the corresponding logical shard.

Each color is a set of totally ordered chains, one per region, with cross-edges between them that 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 interact only with their own region's local copy of the DAG; they can modify this copy by appending to their own region's chain for a color.

Figure 2 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. It can `append` a node to a set of colors. We first describe the operation of these calls in a FuzzyLog deployment with a single color (i.e., an application with a single data shard).

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 since the last `sync` invocation. Once all new nodes have been provided to the application via a passed-in callback, the `sync` returns with an opaque ID describing the snapshot. The nodes are seen in a reverse topological sort order of the DAG. Nodes in each chain are seen in the reverse order of edges in the chain. 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. Note that each node effectively describes a list of nodes – via its position in a totally ordered chain, and via

explicit pointers for cross-edges – that must be seen before it. Figure 3 shows the client synchronizing with the region in panel 1; trailing behind in panels 2 and 3; and synchronizing once again in panel 4. Snapshot IDs returned by `sync` calls at different clients can be compared to check if one subsumes the other.

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. The local region chain imposes a total order over all updates generated at that region. Further, outgoing cross-edges are added from the new node 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 seen by the client. For example, in Figure 3 (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` on a snapshot ID to indicate that the nodes in it are no longer required (e.g., because the client stored the corresponding materialized view in some 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 over a single color, the FuzzyLog supports appending to multiple colors. An `append` to a set of colors atomically appends the entry to the local chains for each color. The new node is 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.

Semantics: Operations to a single color across regions are causally consistent. 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 to 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.

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.

Discussion: Designing the FuzzyLog API required balancing the power of the API against its simplicity and the feasibility of implementing it. In earlier candidates for the API, we directly exposed chains to programmers and allowed `append`/`sync` on any subset of them with a choice of consistency guarantees. This API rendered a scalable implementation much more difficult; for example, guaranteeing a topological sort order for nodes in a subset of chains required us to potentially traverse every chain in the system. In addition, the consistency choices required programmers to reason about the performance and availability of different combinations (e.g., strongly consistent multi-appends on chains in different regions can block due to network partitions). We were able to drastically simplify the API once we realized the equivalence between colors and shards: for example, it makes sense for clients to play a single color since doing otherwise negates the scaling benefit of sharding; and to obtain causal consistency within a color since it is geo-replicated across regions that can partition.

4 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 and applies updates to keep its local view up-to-date. A `get` operation at the server simply waits for a `sync` to complete that started after it was issued, before accessing the local view and returning; this ensures that any updates that were appended to the FuzzyLog before the `get` was issued are reflected in the local view, 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 [7] over a conventional shared log. However, its reliance 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

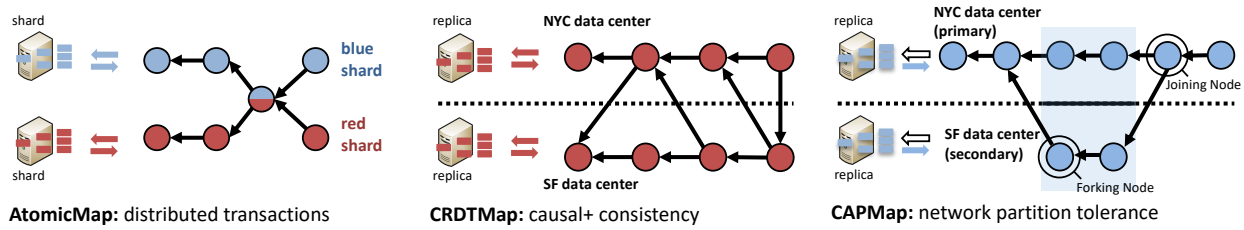

Figure 4: *FuzzyLog capabilities: AtomicMap, CRDTMap, and CAPMap.*

use the FuzzyLog to circumvent each of these limitations.

4.1 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 as nodes of the corresponding color to the FuzzyLog; each server syncs its local state with the color of its shard. This simple change to LogMap – requiring just the color parameter to be 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 a simple change to append an update to a set of colors instead of a single one, corresponding to the shards it modifies. AtomicMap (201 LOC, Figure 4 (Left)) realizes this design. One subtle point is that since FuzzyLog multi-color appends are serializable, AtomicMap is also serializable, not linearizable or strictly serializable.

To implement read/write transactions with stronger isolation levels, we use a protocol identical to the one used by Tango [8]. In TXMap (417 LOC), each server executes read-write transactions speculatively [8, 10], tracking read-sets and buffering write-sets. To commit, the server appends a speculative 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 is playing, it appends a second node with a yes/no decision 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 read-write conflicts when providing strict serializability). 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.

Interestingly, this protocol provides strict serializability even though the FuzzyLog itself is only serializable. Intuitively, within a single color, if a client waits after appending an intention for a transaction T until it plays the node, it is guaranteed to have seen all transactions that could appear before T in the serial order. 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 all transactions in all the colors involved that could appear before T have been seen. A decision node conveys two things: that all such transactions in a color have been seen; and whether they conflict with the transaction. As in Tango [8], our protocol requires at least one application server to be available for each shard in order to generate decision records.

4.2 Weaker consistency across regions

Applications can often tolerate weaker consistency guarantees. One example is causal consistency [5], 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, they can be seen in any order.

CRDTMap implements a causally consistent map. In Figure 4 (Middle), 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 and propagated asynchronously to the other region. 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 [47], which exploits commutativity to execute concurrent updates in conflicting orders without requiring rollback logic. The CRDT design achieves this by predicating deletions performed by a server on put operations that the server has already seen; accordingly, each delete node in the DAG lists the put operations that it subsumes.

4.3 Tolerating network partitions

While CRDTMap can provide availability during network partitions, it does so by sacrificing consistency even when there is no partition in the system. CAPMap (named after the CAP conjecture [14]) provides strong consistency in the absence of network partitions and causal consistency during them (see Figure 4 (Right)).

As with our other map designs, CAPMap appends entries on `put` operations and then syncs until it sees the appended node. Unlike them, 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 completing the `put`. As a result, 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). Updates that were in-flight during the network partition event may be re-appended to the local region, appearing in the DAG as identical nodes in the primary and secondary forks. When the network partition heals, servers at the secondary stop appending locally and resume routing appends through the proxy at the primary. 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.

The FuzzyLog explicitly captures the effects of a network partition, including concurrent activity in the regions and duplicate updates. As a result, CAPMap can relax and reimpose strong consistency via a simple playback policy over the FuzzyLog. 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 nodes, it buffers them until the next primary node (i.e., the joining node). All buffered nodes are then applied immediately before the joining node (ignoring duplicate updates), ensuring that all servers observe the same total order and converge to the same state.

Secondary servers that experience a network partition continue operating over the local fork, applying changes to a speculative copy of state. When the partition heals, each secondary server throws away its speculative changes after the forking node and replays the nodes in the forked region of the DAG, applying updates in the primary fork before re-applying the secondary fork. Our CAPMap implementation realizes this speculative copy by cloning state on a fork, and throwing away the clone when the partition heals; but more efficient copy-on-write mechanisms could be used as well.

As a result, we obtain causal+ consistency [35] during network partitions and linearizability otherwise. Importantly, CAPMap achieves these properties via simple append and playback policies over the structure and contents of the FuzzyLog.

4.4 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. By changing 80 LOC in CRDTMap, we can obtain a transactional CRDT that provides cross-shard failure atomicity [6] (or equivalently, an isolation guarantee similar to Parallel Snapshot Isolation [48]).

RedBlueMap: The FuzzyLog can support RedBlue consistency [32], 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 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 [35, 36]. 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.

4.5 Garbage collection

As with shared log systems, GC 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.

5 Dapple Design / 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 [54]. Our current implementation assumes for durability that storage servers are outfitted with battery-backed DRAM [17, 24]. We first describe operations against a single color on an unreplicated chainserver.

5.1 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 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. Clients in the region actively write to one of these (the local log), while the remaining are asynchronously replicated from other regions (we call these shadow logs). 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-reads`. 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 delivering them 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 chain. The choice of replication protocol is orthogonal to the system design: we could equally use Multi-Paxos.

5.2 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. [22]) to consistently order appends.

Skeen's original 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 as-

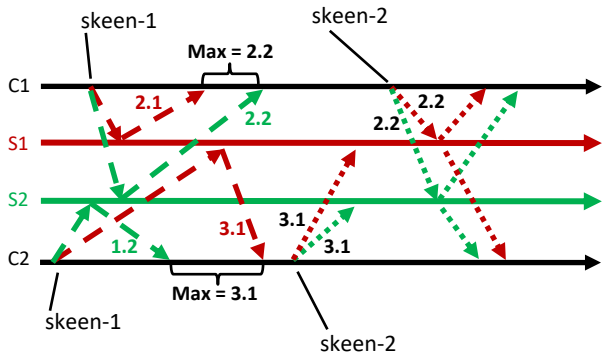

Figure 5: *Distributed ordering for multi-appends: servers return timestamps X.Y in phase 1 where X is a local logical clock and Y is a server-specific nonce.*

sumed 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 three fault-tolerance mechanisms – 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 [28]. All client operations are predicated on relatively coarse-grain leases [20] (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 chainserver replica.

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 augmented with a server-specific unique nonce to break ties. Each chainserver inserts the multi-append operation into a pending queue along with the returned timestamp. For example, in Figure 5, origin client C1 contacts S1, which responds with 2.1, where the local clock value is 2 and the unique nonce is 1. 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, 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. For example, in Figure 5, client C1 sends back a max timestamp of 2.2 to servers S1 and S2. 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. In Figure 5, server S1 receives a phase 2 message with a max timestamp of 3.1 from client C2, but does not respond immediately since it previously responded to a phase 1 message from client C1 with a timestamp of 2.1. Once C1 sends a phase 2 message with a max timestamp of 2.2, S1 knows the ordering for both outstanding multi-appends and can respond to both C1 and C2.

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 multi-append.

Correctness: Skeen's protocol has been proven to generate a total order by others [22, 45]. To prove our recovery protocol correct, we wrote conventional proofs as well as a machine-checked proof in Coq. We omit the

full proof for lack of space. Informally, we prove that the test-and-set mechanism ensures that only one client is actively mutating the state of the system at any given point in time. Further, we show that each append can be modeled as a four-stage state machine (some servers in phase 1, some uninitiated; some in phase 1, some in phase 2; some in phase 2, some completed; all completed). Any recovery client finds the system in a particular state and advances it in a manner identical to the non-failing case.

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 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. In addition, this protocol is subject to FLP [18] and susceptible to livelock, since recovery clients can fence each other perpetually. 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.

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 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,

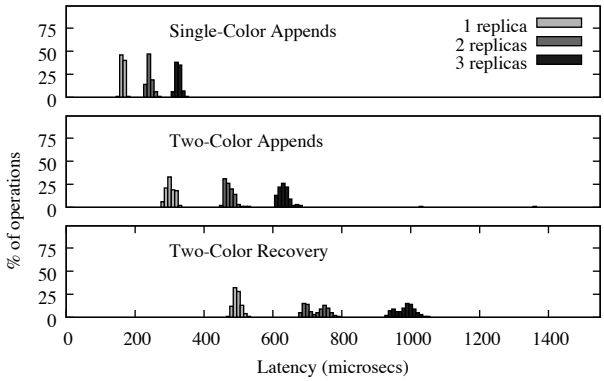

Figure 6: *Dapple executes single-color appends in one phase; multi-color appends in two phases; and recovers from crashed clients in three phases.*

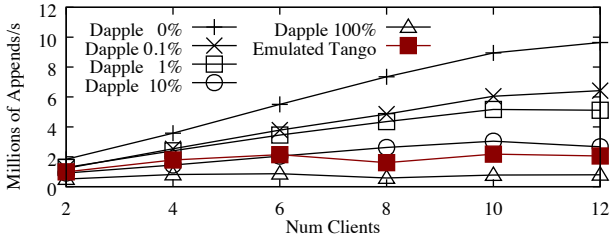

Figure 7: *Dapple scales with workload parallelism, but a centralized sequencer bottlenecks emulated Tango.*

while two-color appends execute in two phases 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 (§6.1). Second, we use the Map variants from §4 to show that Dapple provides linear scaling with atomicity (§6.2), weaker consistency guarantees (§6.3), and network partition tolerance (§6.4). Finally, we describe a ZooKeeper clone over Dapple (§6.5).

6.1 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 [8] and vCorfu [57] 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 us-

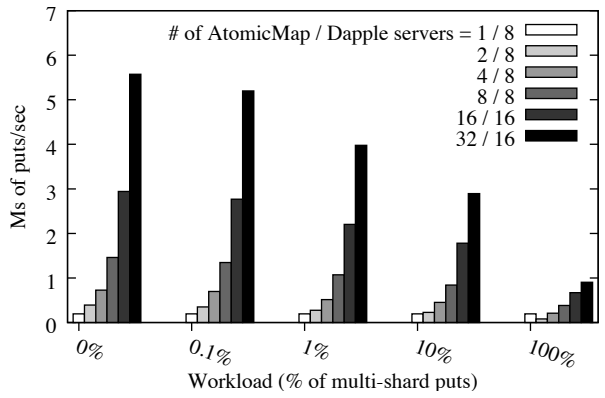

Figure 8: *AtomicMap scales throughput while supporting multi-shard transactions. Each bar labelled N / K shows throughput with N AtomicMap servers running against a K-server Dapple deployment.*

ing 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 7 shows average throughput over a 10-second run for workloads with different percentages of two-color appends. Emulated Tango cannot scale beyond four clients 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.

6.2 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 4.1), 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 8 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 8 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).

6.3 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 9, 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 put operations at a controlled aggregate rate (left y-axis), while the fifth is a reader issuing get operations 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 get operations at the reader (not shown in the graph), but instead manifests as staleness.

6.4 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

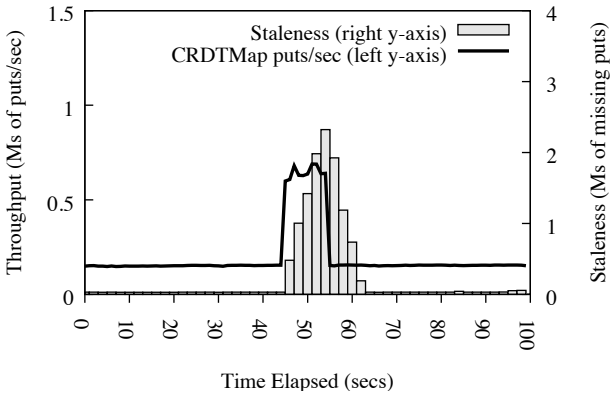

Figure 9: *CRDTMap provides a trade-off between throughput and staleness.*

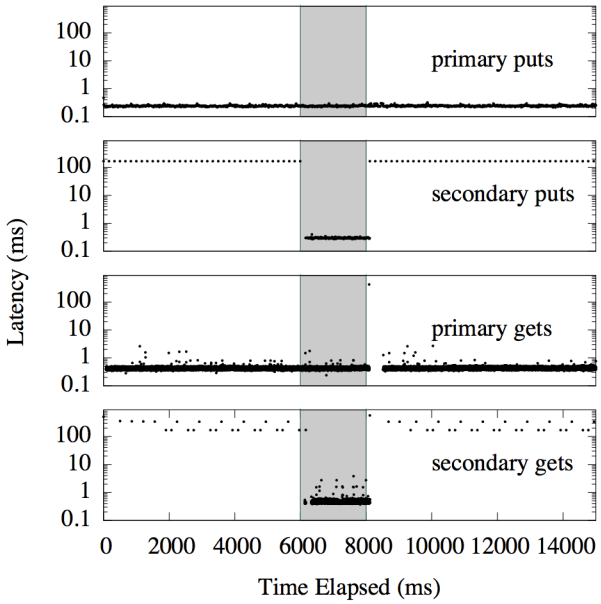

Figure 10: *CAPMap switches between linearizability and causal+ consistency during network partitions.*

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 10 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 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 region. As part of this joining request, the secondary provides a snapshot ID reflecting the last node it appended to its local chain. The proxy at the primary waits until the nodes in the snapshot are replicated to the primary region 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.

6.5 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. Create operations are restricted to a single DappleZK shard. Each rename atomically moves a file from one DappleZK shard to another via the distributed transaction protocol described in Section 4.1.

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 parti-

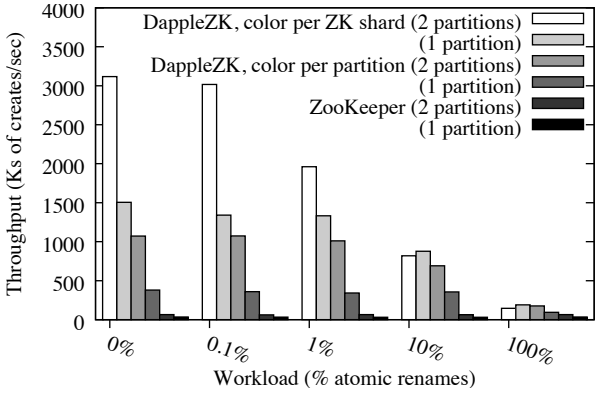

Figure 11: *DappleZK exploits Dapple's partial ordering to implement a scalable version of the ZooKeeper API.*

tion 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 DRAM to enable a fair comparison. 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 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 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.

7 Related Work

Abstractions for ordering updates in a distributed system have a long history. Examples include Virtual Synchrony [13, 53], State Machine Replication [46], Viewstamp Replication [42], Multi-Paxos [52], and newer approaches such as Raft [43]. Most of these impose a total order on updates; the exceptions track particular partial orders imposed by operation commutativity (pessimistically [30, 37] and optimistically [26]), causal consistency (as in Virtual Synchrony and Lazy Replication [27]), or network partitions (as in Extended Virtual Synchrony [38]). In contrast, the FuzzyLog expresses the partial orders relating to both causality and data sharding within a single ordering abstraction.

FuzzyLog designs for providing weaker consistency are informed by a number of systems: COPS [35] and Eiger [36] provide causal consistency in a partitioned store, while Bayou allows for disconnected updates and eventual reconciliation [44, 49]. TARDiS [15] exposes *branch-on-conflict* as an abstraction in a fully replicated, multi-master store. In contrast to the TARDiS DAG, the FuzzyLog allows applications to construct a wider range of partial orders (e.g., CAPMap branches on network partitions rather than conflicts), and enables distributed transactions via color-based partitioning.

A number of systems provide distributed transactions over addresses or objects [4, 34]. Recent systems leverage modern networks such as RDMA and Infiniband to enable high-speed transactions [17, 31]. FuzzyLog provides a lower layer of abstraction, which in turn supports general-purpose transactions using shared log techniques [8, 10]. There has also been recent interest in improving distributed transaction throughput and latency via techniques such as transaction chopping [39, 58, 59, 61]. These mechanisms could be employed by transactional FuzzyLog applications.

Finally, the FuzzyLog is heavily inspired by shared log designs from research [7, 8, 10] and industry [1, 2, 55].

8 Conclusion

The shared log approach simplifies the construction of control plane services, but tightly bounds the scalability and consistency of the resulting systems. The FuzzyLog abstraction – and its implementation in Dapple – extends the shared log approach to partial orders, allowing applications to scale linearly without sacrificing transactional guarantees, obtain a range of consistency guarantees, and switch seamlessly between these guarantees when the network partitions and heals. 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.

Acknowledgments

This work was funded primarily by an NSF AitF grant (CCF-1637385), and partly by NSF grants CCF-1650596 and IIS-1718581. We thank Zhong Shao for his significant input from the beginning of the project. We also thank Luis Rodrigues and Yair Amir for feedback on the ideas behind the FuzzyLog. Vijayan Prabhakaran and Hakim Weatherspoon provided valuable comments on early drafts of this paper. Finally, we would like to thank Kang Chen for shepherding the paper, as well as the anonymous reviewers for their insightful reviews.

References

[1] Facebook logdevice. https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/.

[2] VMware CorfuDB. https://github.com/CorfuDB/CorfuDB.

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

[4] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SOSP 2007*.

[5] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995), 37–49.

[6] AKKOORATH, D. D., TOMSIC, A. Z., BRAVO, M., LI, Z., CRAIN, T., BIENIUSA, A., PREGUIÇA, N., AND SHAPIRO, M. Cure: Strong semantics meets high availability and low latency. In *IEEE ICDCS 2016*.

[7] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. Corfu: A shared log design for flash clusters. In *USENIX NSDI 2012*.

[8] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed Data Structures over a Shared Log. In *ACM SOSP 2013*.

[9] BERNSTEIN, P. A., AND DAS, S. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Eng. Bull. 38*, 1 (2015), 32–49.

[10] BERNSTEIN, P. A., DAS, S., DING, B., AND PILMAN, M. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *ACM SIGMOD 2015*.

[11] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR 2011*.

[12] BEVILACQUA-LINN, M., BYRON, M., CLINE, P., MOORE, J., AND MUIR, S. Sirius: Distributing and Coordinating Application Reference Data. In *USENIX ATC 2014*.

[13] BIRMAN, K. P., AND JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS) 5*, 1 (1987), 47–76.

[14] BREWER, E. A. Towards robust distributed systems. In *PODC 2000*.

[15] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. Tardis: A branch-and-merge approach to weak consistency. In *ACM SIGMOD 2016*.

[16] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR) 36*, 4 (2004), 372–421.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP 2015*.

[18] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[19] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. In *VLDB 2015*.

[20] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM SOSP 1989*.

[21] GRAY, J. N. Notes on data base operating systems. In *Operating Systems*. Springer, 1978, pp. 393–481.

[22] GUERRAOUI, R., AND SCHIPER, A. Total order multicast to multiple groups. In *IEEE ICDCS 1997*.

[23] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX OSDI 2016*.

[25] KAMINSKY, A. K. M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *USENIX ATC 2016*.

[26] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., DAHLIN, M., ET AL. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *USENIX OSDI 2012*.

[27] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS) 10*, 4 (1992), 360–391.

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

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

[30] LAMPORT, L. Generalized consensus and paxos. Tech. rep., Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[31] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *ACM SOSP 2015*.

[32] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N. M., AND RODRIGUES, R. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX OSDI 2012*.

[33] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI 2016*.

[34] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. Providing persistent objects in distributed systems. In *ECOOP 1999*.

[35] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *USENIX NSDI 2013*.

[36] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *USENIX NSDI 2013*.

[37] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *ACM SOSP 2013*.

[38] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended virtual synchrony. In *IEEE ICDCS 1994*.

[39] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distributed Transactions. In *USENIX OSDI 2014*.

[40] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *USENIX OSDI 2016*.

[41] NAWAB, F., ARORA, V., AGRAWAL, D., AND EL ABBADI, A. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. In *EDBT* (2015), pp. 13–24.

[42] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *ACM PODC 1988*.

[43] ONGARO, D., AND OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In *USENIX ATC 2014*.

[44] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *ACM SOSP 1997*.

[45] RODRIGUES, L., GUERRAOUI, R., AND SCHIPER, A. Scalable atomic multicast. In *IEEE ICCCN 1998*.

[46] SCHNEIDER, F. B. The state machine approach: A tutorial. In *Fault-tolerant distributed computing* (1990), Springer, pp. 18–41.

[47] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[48] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *ACM SOSP 2011*.

[49] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM SOSP 1995*.

[50] THOMSON, A., AND ABADI, D. J. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *USENIX FAST 2015*.

[51] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *ACM SIGMOD 2012*.

[52] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos made moderately complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 42.

[53] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A flexible group communication system. *Communications of the ACM 39*, 4 (1996), 76–83.

[54] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *USENIX OSDI 2004*.

[55] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *ACM SIGMOD 2017*.

[56] WEI, M., ROSSBACH, C., ABRAHAM, I., WIEDER, U., SWANSON, S., MALKHI, D., AND TAI, A. Silver: a scalable, distributed, multi-versioning, always growing (Ag) file system. In *USENIX HotStorage 2016*.

[57] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., ET AL. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *In USENIX NSDI 2017*.

[58] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining ACID and BASE in a Distributed Database. In *USENIX OSDI 2014*.

[59] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via modular concurrency control. In *ACM SOSP 2015*.

[60] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. Building consistent transactions with inconsistent replication. In *ACM SOSP 2015*.

[61] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP 2013*.