Tango: Distributed Data Structures over a Shared Log

Mahesh Balakrishnan[*], Dahlia Malkhi[*], Ted Wobber[*], Ming Wu[‡], Vijayan Prabhakaran[*]
Michael Wei[§], John D. Davis[*], Sriram Rao[†], Tao Zou[¶], Aviad Zuck[‖]

[*]Microsoft Research Silicon Valley [‡]Microsoft Research Asia [†]Microsoft
[§]University of California, San Diego [¶]Cornell University [‖]Tel-Aviv University

Abstract

Distributed systems are easier to build than ever with the emergence of new, data-centric abstractions for storing and computing over massive datasets. However, similar abstractions do not exist for storing and accessing metadata. To fill this gap, Tango provides developers with the abstraction of a replicated, in-memory data structure (such as a map or a tree) backed by a shared log. Tango objects are easy to build and use, replicating state via simple append and read operations on the shared log instead of complex distributed protocols; in the process, they obtain properties such as linearizability, persistence and high availability from the shared log. Tango also leverages the shared log to enable fast transactions across different objects, allowing applications to partition state across machines and scale to the limits of the underlying log without sacrificing consistency.

1 Introduction

Cloud platforms have democratized the development of scalable applications in recent years by providing simple, data-centric interfaces for partitioned storage (such as Amazon S3 [1] or Azure Blob Store [8]) and parallelizable computation (such as MapReduce [19] and Dryad [28]). Developers can use these abstractions to easily build certain classes of large-scale applications – such as user-facing Internet services or back-end machine learning algorithms – without having to reason about the underlying distributed machinery.

SOSP'13, Nov. 3–6, 2013, Farmington, PA, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522732

However, current cloud platforms provide applications with little support for storing and accessing metadata. Application metadata typically exists in the form of data structures such as maps, trees, counters, queues, or graphs; real-world examples include filesystem hierarchies [5], resource allocation tables [7], job assignments [3], network topologies [35], deduplication indices [20] and provenance graphs [36]. Updates to metadata usually consist of multi-operation transactions that span different data structures – or arbitrary subsets of a single data structure – while requiring atomicity and isolation; for example, moving a node from a free list to an allocation table, or moving a file from one portion of a namespace to another. At the same time, application metadata is required to be highly available and persistent in the face of faults.

Existing solutions for storing metadata do not provide transactional access to arbitrary data structures with persistence and high availability. Cloud storage services (e.g., SimpleDB [2]) and coordination services (e.g., ZooKeeper [27] and Chubby [14]) provide persistence, high availability, and strong consistency. However, each system does so for a specific data structure, and with limited or no support for transactions that span multiple operations, items, or data structures. Conventional databases support transactions, but with limited scalability and not over arbitrary data structures.

In this paper, we introduce Tango, a system for building highly available metadata services where the key abstraction is a Tango object, a class of in-memory data structures built over a durable, fault-tolerant shared log. As shown in Figure 1, the state of a Tango object exists in two forms: a *history*, which is 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 RAM on clients (i.e., application servers). In Tango, *the shared log is the object*; views constitute soft state and are instantiated, reconstructed, and updated on clients as required by playing the shared history forward. A client modifies a Tango object by appending a new

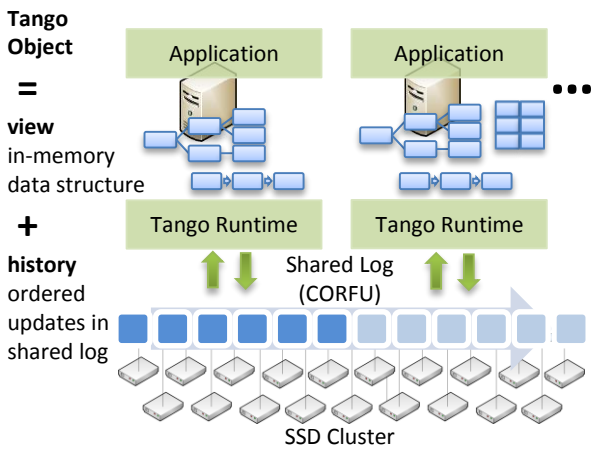

Figure 1: A Tango object is a replicated in-memory data structure layered over a persistent shared log.

update to the history; it accesses the object by first synchronizing its local view with the history.

Tango objects simplify the construction of metadata services by delegating key pieces of functionality to the underlying shared log. First, the shared log is the source of ***consistency*** for the Tango object: clients implement state machine replication by funneling writes through the shared history and synchronizing with it on reads, providing linearizability for single operations. Second, the shared log is the source of ***durability***: clients can recover views after crashes simply by playing back the history stored in the shared log. In addition, views can contain pointers to data stored in the shared log, effectively acting as indices over log-structured storage. Third, the shared log is the source of ***history***: clients can access previous states of the Tango object by instantiating a new view from a prefix of the history. Finally, the shared log enables ***elasticity***: the aggregate throughput of linearizable reads to the Tango object can be scaled simply by adding new views, without slowing down write throughput. In extracting these properties from a shared log, Tango objects can be viewed as a synthesis of state machine replication [42], log-structured storage [39], and history-based systems [21].

In addition, Tango provides ***atomicity*** and ***isolation*** for transactions across different objects by storing them on a single shared log. These objects can be different components of the same application (e.g., a scheduler using a free list and an allocation table), different shards of the application (e.g., multiple subtrees of a filesystem namespace), or even components shared across applications (e.g., different job schedulers accessing the same free list). In all these use cases, the shared log establishes a global order across all updates, efficiently enabling transactions as well as other strongly consistent multi-object operations such as atomic updates, consistent snapshots, coordinated rollback, and consistent remote mirroring. Tango implements these operations by manipulating the shared log in simple ways, obviating the need for the complex distributed protocols typically associated with such functionality. Multiplexing Tango objects on a single shared log also simplifies deployment; new applications can be instantiated just by running new client-side code against the shared log, without requiring application-specific back-end servers.

The Tango design is enabled by the existence of fast, decentralized shared log implementations that can scale to millions of appends and reads per second; our implementation runs over a modified version of CORFU [10], a recently proposed protocol that utilizes a cluster of flash drives for this purpose. However, a key challenge for Tango is the playback bottleneck: even with an infinitely scalable shared log, any single client in the system can only consume the log – i.e., learn the total ordering – at the speed of its local NIC. In other words, a set of clients can extend Tango object histories at aggregate speeds of millions of appends per second, but each client can only read back and apply those updates to its local views at tens of thousands of operations per second.

To tackle the playback bottleneck, Tango implements a stream abstraction over the shared log. A stream provides a *readnext* interface over the address space of the shared log, allowing clients to selectively learn or consume the subsequence of updates that concern them while skipping over those that do not. Each Tango object is stored on its own stream; to instantiate the view for a Tango object, a client simply plays the associated stream. The result is *layered partitioning*, where an application can shard its state into multiple Tango objects, each instantiated on a different client, allowing the aggregate throughput of the system to scale until the underlying shared log is saturated. The global ordering imposed by the shared log enables fast cross-partition transactions, ensuring that the scaling benefit of layered partitioning does not come at the cost of consistency.

Tango is built in C++ with bindings for Java and C# applications. We've built a number of useful data structures with Tango, including ZooKeeper (TangoZK, 1K lines), BookKeeper (TangoBK, 300 lines), and implementations of the Java and C# Collections interfaces such as TreeSets and HashMaps (100 to 300 lines each). Our implementations of these interfaces are persistent, highly available, and elastic, providing linear scaling for linearizable reads against a fixed write load. Additionally, they support fast transactions within and across data structures; for example, applications can transactionally delete a TangoZK node while creating an entry in a TangoMap. Finally, we ran the HDFS namenode over the

Tango variants of ZooKeeper and BookKeeper, showing that our implementations offer full fidelity to the originals despite requiring an order of magnitude less code.

We make two contributions in this paper:

- We describe Tango, a system that provides applications with the novel abstraction of an in-memory data structure backed by a shared log. We show that Tango objects can achieve properties such as persistence, strong consistency, and high availability in tens of lines of code via the shared log, without requiring complex distributed protocols (Section 3). In our evaluation, a single Tango object running on 18 clients provides 180K linearizable reads/sec against a 10K/sec write load.

- We show that storing multiple Tango objects on the same shared log enables simple, efficient transactional techniques across objects (Section 4). To implement these techniques efficiently, we present a streaming interface that allows each object to selectively consume a subsequence of the shared log (Section 5). In our evaluation, a set of Tango objects runs at over 100K txes/sec when 16% of transactions span objects on different clients.

2 Background

In practice, metadata services are often implemented as centralized servers; high availability is typically a secondary goal to functionality. When the time comes to 'harden' these services, developers are faced with three choices. First, they can roll out their own custom fault-tolerance protocols; this is expensive, time-consuming, and difficult to get right. Second, they can implement state machine replication over a consensus protocol such as Paxos [31]; however, this requires the service to be structured as a state machine with all updates flowing through the Paxos engine, which often requires a drastic rewrite of the code.

The third option is to use an existing highly available data structure such as ZooKeeper, which provides a hierarchical namespace as an external service. However, such an approach forces developers to use a particular data structure (in the case of ZooKeeper, a tree) to store all critical application state, instead of allowing them to choose one or more data structures that best fit their needs (as an analogy, imagine if the C++ STL provided just a hash_map, or Java Collections came with just a TreeSet!). This is particularly problematic for high-performance metadata services which use highly tuned data structures tailored for specific workloads. For example, a membership service that stores server names in ZooKeeper would find it inefficient to implement common functionality such as searching the namespace on some index (e.g., CPU load), extracting the oldest/newest inserted name, or storing multi-MB logs per name.

In practice, developers are forced to cobble together various services, each solving part of the problem; for example, one of the existing, in-progress proposals for adding high availability to the HDFS namenode (i.e., metadata server) uses a combination of ZooKeeper, BookKeeper [30], and its own custom protocols [4]. Such an approach produces fragile systems that depend on multiple other systems, each with its own complex protocols and idiosyncratic failure modes. Often these underlying protocols are repetitive, re-implementing consensus and persistence in slightly different ways. The result is also a deployment nightmare, requiring multiple distributed systems to be independently configured and provisioned.

Can we provide developers with a wide range of data structures that are strongly consistent, persistent, and highly available, while using a single underlying abstraction? Importantly, can we make the development of such a data structure easy enough that developers can write new, application-specific data structures in tens of lines of code? The answer to these questions lies in the shared log abstraction.

2.1 The Shared Log Abstraction

Shared logs were first used in QuickSilver [25, 41] and Camelot [43] in the late 80s to implement fault-tolerance; since then, they have played roles such as packet recovery [26] and remote mirroring [29] in various distributed systems. Two problems have hampered the adoption of the shared log as a mainstream abstraction. First, any shared log implementation is subject to a highly random read workload, since the body of the log can be concurrently accessed by many clients over the network. If the underlying storage media is disk, these randomized reads can slow down other reads as well as reduce the append throughput of the log to a trickle. As Bernstein et al. observe [11], this concern has largely vanished with the advent of flash drives that can support thousands of concurrent read and write IOPS.

The second problem with the shared log abstraction relates to scalability; existing implementations typically require appends to the log to be serialized through a primary server, effectively limiting the append throughput of the log to the I/O bandwidth of a single machine. This problem is eliminated by the CORFU protocol [10], which scales the append throughput of the log to the speed at which a centralized sequencer can hand out new offsets in the log to clients.

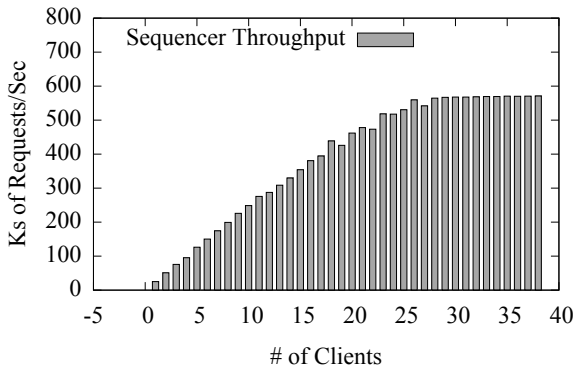

Figure 2: A centralized sequencer can scale to more than half a million operations/sec.

2.2 The CORFU Shared Log

The CORFU interface is simple, consisting of four basic calls. Clients can *append* entries to the shared log, obtaining an offset in return. They can *check* the current tail of the log. They can *read* the entry at a particular offset. The system provides linearizable semantics: a *read* or a *check* is guaranteed to see any completed *append* operations. Finally, clients can *trim* a particular offset in the log, indicating that it can be garbage collected.

Internally, CORFU organizes a cluster of storage nodes into multiple, disjoint replica sets; for example, a 12-node cluster might consist of 4 replica sets of size 3. Each individual storage node exposes a 64-bit write-once address space, mirrored across the replica set. Additionally, the cluster contains a dedicated sequencer node, which is essentially a networked counter storing the current tail of the shared log.

To append to the shared log, a client first contacts the sequencer and obtains the next free offset in the global address space of the shared log. It then maps this offset to a local offset on one of the replica sets using a simple deterministic mapping over the membership of the cluster. For example, offset 0 might be mapped to $A:0$ (i.e., page 0 on set A, which in turn consists of storage nodes A_0, A_1, and A_2), offset 1 to $B:0$, and so on until the function wraps back to $A:1$. The client then completes the append by directly issuing writes to the storage nodes in the replica set using a client-driven variant of Chain Replication [45].

Reads to an offset follow a similar process, minus the offset acquisition from the sequencer. Checking the tail of the log comes in two variants: a fast check (sub-millisecond) that contacts the sequencer, and a slow check (10s of milliseconds) that queries the storage nodes for their local tails and inverts the mapping function to obtain the global tail.

The CORFU design has some important properties:

The sequencer is not a single point of failure. The sequencer stores a small amount of soft state: a single 64-bit integer representing the tail of the log. When the sequencer goes down, any client can easily recover this state using the slow check operation. In addition, the sequencer is merely an optimization to find the tail of the log and not required for correctness; the Chain Replication variant used to write to the storage nodes guarantees that a single client will 'win' if multiple clients attempt to write to the same offset. As a result, the system can tolerate the existence of multiple sequencers, and can run without a sequencer (at much reduced throughput) by having clients probe for the location of the tail. A different failure mode involves clients crashing after obtaining offsets but before writing to the storage nodes, creating 'holes' in the log; to handle this case, CORFU provides applications with a fast, sub-millisecond *fill* primitive as described in [10] .

The sequencer is not a bottleneck for small clusters. In prior work on CORFU [10], we reported a user-space sequencer that ran at 200K appends/sec. To test the limits of the design, we subsequently built a faster CORFU sequencer using the new Registered I/O interfaces [9] in Windows Server 2012. Figure 2 shows the performance of the new sequencer: as we add clients to the system, sequencer throughput increases until it plateaus at around 570K requests/sec. We obtain this performance without any batching (beyond TCP/IP's default Nagling); with a batch size of 4, for example, the sequencer can run at over 2M requests/sec, but this will obviously affect the end-to-end latency of appends to the shared log. Our finding that a centralized server can be made to run at very high RPC rates matches recent observations by others; the Percolator system [38], for example, runs a centralized timestamp oracle with similar functionality at over 2M requests/sec with batching; Vasudevan et al. [46] report achieving 1.6M sub-millisecond 4-byte reads/sec on a single server with batching; Masstree [33] is a key-value server that provides 6M queries/sec with batching.

Garbage collection is a red herring. System designers tend to view log-structured designs with suspicion, conditioned by decades of experience with garbage collection over hard drives. However, flash storage has sparked a recent resurgence in log-structured designs, due to the ability of the medium to provide contention-free random reads (and its need for sequential writes); every SSD on the market today traces its lineage to the original LFS design [39], implementing a log-structured storage system that can provide thousands of IOPS despite concurrent GC activity. In this context, a single CORFU storage node is an SSD with a custom interface (i.e., a write-once, 64-bit address space instead of a conventional LBA, where space is freed by explicit trims rather

than overwrites). Accordingly, its performance and endurance levels are similar to any commodity SSD. As with any commodity SSD, the flash lifetime of a CORFU node depends on the workload; sequential trims result in substantially less wear on the flash than random trims.

Finally, while the CORFU prototype we use works over commodity SSDs, the abstract design can work over any form of non-volatile RAM (including battery-backed DRAM and Phase Change Memory). The size of a single entry in the log (which has to be constant across entries) can be selected at deployment time to suit the underlying medium (e.g., 128 bytes for DRAM, or 4KB for NAND flash).

3 The Tango Architecture

A Tango application is typically a service running in a cloud environment as a part of a larger distributed system, managing metadata such as indices, namespaces, membership, locks, or resource lists. Application code executes on clients (which are compute nodes or application servers in a data center) and manipulates data stored in Tango objects, typically in response to networked requests from machines belonging to other services and subsystems. The local view of the object on each client interacts with a Tango runtime, which in turn provides persistence and consistency by issuing appends and reads to an underlying shared log. Importantly, Tango runtimes on different machines do not communicate with each other directly through message-passing; all interaction occurs via the shared log. Applications can use a standard set of objects provided by Tango, providing interfaces similar to the Java Collections library or the C++ STL; alternatively, application developers can roll their own Tango objects.

3.1 Anatomy of a Tango Object

As described earlier, a Tango object is a replicated in-memory data structure backed by a shared log. The Tango runtime simplifies the construction of such an object by providing the following API:

- *update_helper*: this accepts an opaque buffer from the object and appends it to the shared log.
- *query_helper*: this reads new entries from the shared log and provides them to the object via an *apply* upcall.

The code for the Tango object itself has three main components. First, it contains the view, which is an in-memory representation of the object in some form, such as a list or a map; in the example of a TangoRegister shown in Figure 3, this state is a single integer. Second,

```
class TangoRegister {
    int oid;
    TangoRuntime *T;
    int state;
    void apply(void *X) {
        state = *(int *)X;
    }
    void writeRegister(int newstate){
        T->update_helper(&newstate,
                    sizeof(int), oid);
    }
    int readRegister() {
        T->query_helper(oid);
        return state;
    }
}
```

Figure 3: TangoRegister: a linearizable, highly available and persistent register. Tango functions/upcalls in bold.

it implements the mandatory *apply* upcall which changes the view when the Tango runtime calls it with new entries from the log. The view must be modified only by the Tango runtime via this apply upcall, and not by application threads executing arbitrary methods of the object.

Finally, each object exposes an external interface consisting of object-specific mutator and accessor methods; for example, a TangoMap might expose *get*/*put* methods, while the TangoRegister in Figure 3 exposes *read*/*write* methods. The object's mutators do not directly change the in-memory state of the object, nor do the accessors immediately read its state. Instead, each mutator coalesces its parameters into an opaque buffer – an *update record* – and calls the *update_helper* function of the Tango runtime, which appends it to the shared log. Each accessor first calls *query_helper* before returning an arbitrary function over the state of the object; within the Tango runtime, *query_helper* plays new update records in the shared log until its current tail and applies them to the object via the *apply* upcall before returning.

We now explain how this simple design extracts important properties from the underlying shared log.

Consistency: Based on our description thus far, a Tango object is indistinguishable from a conventional SMR (state machine replication) object. As in SMR, different views of the object derive consistency by funneling all updates through a total ordering engine (in our case, the shared log). As in SMR, strongly consistent accessors are implemented by first placing a marker at the current position in the total order and then ensuring that the view has seen all updates until that marker. In conventional SMR this is usually done by injecting a read

operation into the total order, or by directing the read request through the leader [13]; in our case we leverage the *check* function of the log. Accordingly, a Tango object with multiple views on different machines provides linearizable semantics for invocations of its mutators and accessors.

Durability: A Tango object is trivially persistent; the state of the object can be reconstructed by simply creating a new instance and calling *query_helper* on Tango. A more subtle point is that the in-memory data structure of the object can contain pointers to values stored in the shared log, effectively turning the data structure into an index over log-structured storage. To facilitate this, each Tango object is given direct, read-only access to its underlying shared log, and the *apply* upcall optionally provides the offset in the log of the new update. For example, a TangoMap can update its internal hash-map with the offset rather than the value on each *apply* upcall; on a subsequent *get*, it can consult the hash-map to locate the offset and then directly issue a random read to the shared log.

History: Since all updates are stored in the shared log, the state of the object can be rolled back to any point in its history simply by creating a new instance and syncing with the appropriate prefix of the log. To enable this, the Tango *query_helper* interface takes an optional parameter that specifies the offset at which to stop syncing. To optimize this process in cases where the view is small (e.g., a single integer in TangoRegister), the Tango object can create checkpoints and provide them to Tango via a *checkpoint* call. Internally, Tango stores these checkpoints on the shared log and accesses them when required on *query_helper* calls. Additionally, the object can forgo the ability to roll back (or index into the log) before a checkpoint with a *forget* call, which allows Tango to trim the log and reclaim storage capacity.

The Tango design enables other useful properties. Strongly consistent reads can be scaled simply by instantiating more views of the object on new clients. More reads translate into more *check* and *read* operations on the shared log, and scale linearly until the log is saturated. Additionally, objects with different in-memory data structures can share the same data on the log. For example, a namespace can be represented by different trees, one ordered on the filename and the other on a directory hierarchy, allowing applications to perform two types of queries efficiently (i.e., "list all files starting with the letter B" vs. "list all files in this directory").

3.2 Multiple Objects in Tango

We now substantiate our earlier claim that storing multiple objects on a single shared log enables strongly consistent operations across them without requiring complex distributed protocols. The Tango runtime on each client can multiplex the log across objects by storing and checking a unique object ID (OID) on each entry; such a scheme has the drawback that every client has to play every entry in the shared log. For now, we make the assumption that each client hosts views for all objects in the system. Later in the paper, we describe layered partitioning, which enables strongly consistent operations across objects without requiring each object to be hosted by each client, and without requiring each client to consume the entire shared log.

Many strongly consistent operations that are difficult to achieve in conventional distributed systems are trivial over a shared log. Applications can perform coordinated rollbacks or take consistent snapshots across many objects simply by creating views of each object synced up to the same offset in the shared log. This can be a key capability if a system has to be restored to an earlier state after a cascading corruption event. Another trivially achieved capability is remote mirroring; application state can be asynchronously mirrored to remote data centers by having a process at the remote site play the log and copy its contents. Since log order is maintained, the mirror is guaranteed to represent a consistent, system-wide snapshot of the primary at some point in the past. In Tango, all these operations are implemented via simple appends and reads on the shared log.

Tango goes one step further and leverages the shared log to provide transactions within and across objects. It implements optimistic concurrency control by appending speculative transaction *commit records* to the shared log. Commit records ensure atomicity, since they determine a point in the persistent total ordering at which the changes that occur in a transaction can be made visible at all clients. To provide isolation, each commit record contains a read set: a list of objects read by the transaction along with their versions, where the version is simply the last offset in the shared log that modified the object. A transaction only succeeds if none of its reads are stale when the commit record is encountered (i.e., the objects have not changed since they were read). As a result, Tango provides the same isolation guarantee as 2-phase locking, which is at least as strong as strict serializability [12], and is identical to the guarantee provided by the recent Spanner [18] system.

Figure 4 shows an example of the transactional interface provided by Tango to application developers, where calls to object accessors and mutators can be bracketed by *BeginTX* and *EndTX* calls. *BeginTX* creates a transaction context in thread-local storage. *EndTX* appends a commit record to the shared log, plays the log forward until the commit point, and then makes a commit/abort decision. Each client that encounters the commit record decides – independently but deterministically – whether

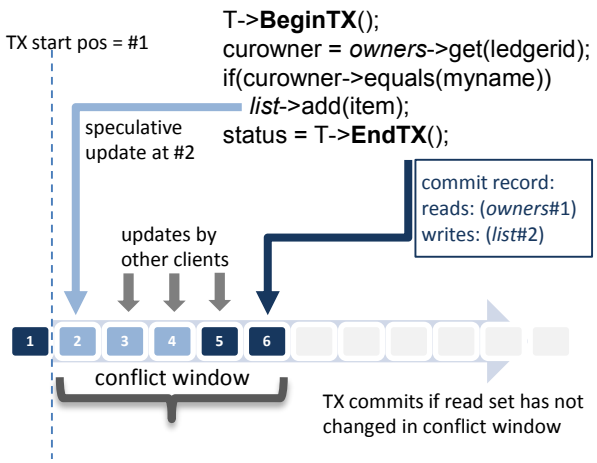

Figure 4: Example of a single-writer list built with transactions over a TangoMap and a TangoList.

it should commit or abort by comparing the versions in the read set with the current versions of the objects. If none of the read objects have changed since they were read, the transaction commits and the objects in the write set are updated with the *apply* upcall. In the example in Figure 4, this happens if the committed transactions in the conflict window do not modify the 'owners' data structure.

To support transactional access, Tango object code requires absolutely no modification (for example, the TangoRegister code in Figure 3 supports transactions); instead, the Tango runtime merely substitutes different implementations for the *update_helper* and *query_helper* functions if an operation is running within a transactional context. The *update_helper* call now buffers updates instead of writing them immediately to the shared log; when a log entry's worth of updates have been accumulated, it flushes them to the log as speculative writes, not to be made visible by other clients playing the log until the commit record is encountered. The *EndTX* call flushes any buffered updates to the shared log before appending the commit record to make them visible. Correspondingly, the *query_helper* call does not play the log forward when invoked in a transactional context; instead, it updates the read set of the transaction with the OID of the object and its current version.

Naming: To assign unique OIDs to each object, Tango maintains a directory from human-readable strings (e.g. "FreeNodeList" or "WidgetAllocationMap") to unique integers. This directory is itself a Tango object with a hard-coded OID. Tango also uses the directory for safely implementing the *forget* garbage collection interface in the presence of multiple objects; this is complicated by the fact that entries can contain commit records impacting multiple objects. The directory tracks the *forget* offset for each object (below which its entries can be reclaimed), and Tango only trims the shared log below the minimum such offset across all objects.

Versioning: While using a single version number per object works well for fine-grained objects such as registers or counters, it can result in an unnecessarily high abort rate for large data structures such as maps, trees or tables, where transactions should ideally be allowed to concurrently modify unrelated parts of the data structure. Accordingly, objects can optionally pass in opaque key parameters to the *update_helper* and *query_helper* calls, specifying which disjoint sub-region of the data structure is being accessed and thus allowing for fine-grained versioning within the object. Internally, Tango then tracks the latest version of each key within an object. For data structures that are not statically divisible into sub-regions (such as queues or trees), the object can use its own key scheme and provide upcalls to the Tango runtime that are invoked to check and update versions.

Read-only transactions: For these, the *EndTX* call does not insert a commit record into the shared log; instead, it just plays the log forward until its current tail before making the commit/abort decision. If there's no write activity in the system (and consequently no new updates to play forward), a read-only transaction only requires checking the tail of the shared log; in CORFU, this is a single round-trip to the sequencer. Tango also supports fast read-only transactions from stale snapshots by having *EndTX* make the commit/abort decision locally, without interacting with the log. Write-only transactions require an append on the shared log but can commit immediately without playing the log forward.

Failure Handing: A notable aspect of Tango objects (as described thus far) is the simplicity of failure handling, a direct consequence of using a fault-tolerant shared log. A Tango client that crashes in the middle of a transaction can leave behind orphaned data in the log without a corresponding commit record; other clients can complete the transaction by inserting a dummy commit record designed to abort. In the context of CORFU, a crashed Tango client can also leave holes in the shared log; any client that encounters these uses the CORFU *fill* operation on them after a tunable time-out (100ms by default). Beyond these, the crash of a Tango client has no unpleasant side-effects.

4 Layered Partitions

In the previous section, we showed how a shared log enables strongly consistent operations such as transactions across multiple Tango objects. In our description, we assumed that each client in the system played the

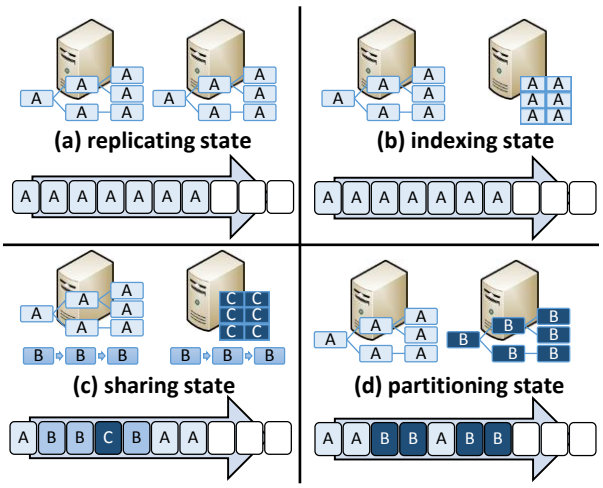

Figure 5: Use cases for Tango objects: application state can be distributed in different ways across objects.

entire shared log, with the Tango runtime multiplexing the updates in the log across different Tango objects. Such a design is adequate if every client in the system hosts a view of every object in the system, which is the case when the application is a large, fully replicated service (as in example (a) in Figure 5). For example, a job scheduling service that runs on multiple application servers for high availability can be constructed using a TangoMap (mapping jobs to compute nodes), a TangoList (storing free compute nodes) and a TangoCounter (for new job IDs). In this case, each of the application servers (i.e. Tango clients) runs a full replica of the job scheduler service and hence needs to access all three objects. Requiring each node to play back the entire log is also adequate if different objects share the same history, as described earlier; in example (b) in Figure 5, a service hosts a tree-based map and a hash-based map over the same data to optimize for specific access patterns.

However, for other use cases, different clients in the system may want to host views of different subsets of objects, due to different services or components sharing common state (see example (c) in Figure 5). For instance, let's say that the job scheduler above coexists with a backup service that periodically takes nodes in the free list offline, backs them up, and then returns them to the free list. This backup service runs on a different set of application servers and is composed from a different set of Tango objects, but requires access to the same TangoList as the job scheduler. In this scenario, forcing each application server to play the entire shared log is wasteful; the backup service does not care about the state of the objects that compose the job scheduler (other than the free list), and vice versa. Additionally, different clients may want to host views of disjoint subsets of objects (as in example (d) in Figure 5), scaling the system for operations within a partition while still using the underlying shared log for consistency across partitions.

We call this *layered partitioning*: each client hosts a (possibly overlapping) partition of the global state of the system, but this partitioning scheme is layered over a single shared log. To efficiently implement layered partitions without requiring each client to play the entire shared log, Tango maps each object to a *stream* over the shared log. A stream augments the conventional shared log interface (*append* and random *read*) with a streaming *readnext* call. Many streams can co-exist on a single shared log; calling *readnext* on a stream returns the next entry belonging to that stream in the shared log, skipping over entries belonging to other streams. With this interface, clients can selectively consume the shared log by playing the streams of interest to them (i.e., the streams of objects hosted by them). Importantly, streams are not necessarily disjoint; a *multiappend* call allows a physical entry in the log to belong to multiple streams, a capability we use to implement transactions across objects.

Accordingly, each client now plays the streams belonging to the objects in its layered partition. How does this compare with conventionally partitioned or sharded systems? As in sharding, each client hosting a layered partition only sees a fraction of the traffic in the system, allowing throughput to scale linearly with the number of partitions (assuming these don't overlap). Unlike sharding, applications now have the ability to efficiently perform strongly consistent operations such as transactions across layered partitions, since the shared log imposes a global ordering across partitions. In exchange for this new capability, there's now a cap on aggregate throughput across all partitions; once the shared log is saturated, adding more layered partitions does not increase throughput.

In the remainder of this section, we describe how Tango uses streams over a shared log to enable fast transactions without requiring all clients to play the entire log. In the next section, we describe our implementation of streams within the CORFU shared log.

4.1 Transactions over Streams

Tango uses streams in an obvious way: each Tango object is assigned its own dedicated stream. If transactions never cross object boundaries, no further changes are required to the Tango runtime. When transactions cross object boundaries, Tango changes the behavior of its *EndTX* call to *multiappend* the commit record to all the streams involved in the write set. This scheme ensures two important properties required for atomicity and isolation. First, a transaction that affects multiple objects

occupies a single position in the global ordering; in other words, there is only one commit record per transaction in the raw shared log. Second, a client hosting an object sees every transaction that impacts the object, even if it hosts no other objects.

When a commit record is appended to multiple streams, each Tango runtime can encounter it multiple times, once in each stream it plays (under the hood, the streaming layer fetches the entry once from the shared log and caches it). The first time it encounters the record at a position X, it plays all the streams involved until position X, ensuring that it has a consistent snapshot of all the objects touched by the transaction as of X. It then checks for read conflicts (as in the single-object case) and determines the commit/abort decision.

When each client does not host a view for every object in the system, writes or reads can involve objects that are not locally hosted at the client that generates the commit record or the client that encounters it. We examine each of these cases:

A. Remote writes at the generating client: The generating client – i.e., the client that executed the transaction and created the commit record – may want to write to a remote object (i.e., an object for which it does not host a local view). This case is easy to handle; as we describe later, a client does not need to play a stream to append to it, and hence the generating client can simply append the commit record to the stream of the remote object.

B. Remote writes at the consuming client: A client may encounter commit records generated by other clients that involve writes to objects it does not host; in this case, it simply updates its local objects while ignoring updates to the remote objects.

Remote-write transactions are an important capability. Applications that partition their state across multiple objects can now consistently move items from one partition to the other. For example, in our implementation of ZooKeeper as a Tango object, we can partition the namespace by running multiple instances of the object, and move keys from one namespace to the other using remote-write transactions. Another example is a producer-consumer queue; with remote-write transactions, the producer can add new items to the queue without having to locally host it and see all its updates.

C. Remote reads at the consuming client: Here, a client encounters commit records generated by other clients that involve reads to objects it does not host; in this case, it does not have the information required to make a commit/abort decision since it has no local copy of the object to check the read version against. To resolve this problem, we add an extra round to the conflict resolution process, as shown in Figure 6. The client that generates and appends the commit record (App1 in the figure) immediately plays the log forward until the

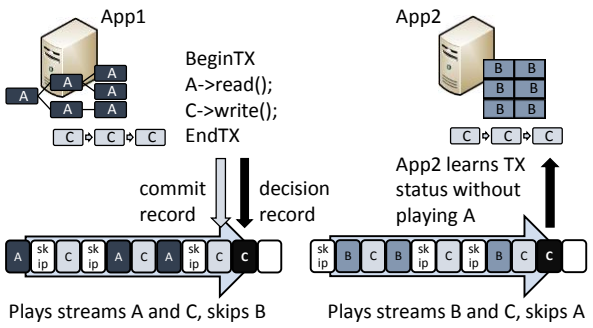

Figure 6: Transactions over streams: decision records allow clients to learn about a transaction's status without hosting its read set.

commit point, makes a commit/abort decision for the record it just appended, and then appends an extra *decision record* to the same set of streams. Other clients that encounter the commit record (App2 in the figure) but do not have locally hosted copies of the objects in its read set can now wait for this decision record to arrive. Significantly, the extra phase adds latency to the transaction but does not increase the abort rate, since the conflict window for the transaction is still the span in the shared log between the reads and the commit record.

Concretely, a client executing a transaction must insert a decision record for a transaction if there's some other client in the system that hosts an object in its write set but not all the objects in its read set. In our current implementation, we require developers to mark objects as requiring decision records; for example, in Figure 6, App1 marks object *A* but not object *C*. This solution is simple but conservative and static; a more dynamic scheme might involve tracking the set of objects hosted by each client.

D. Remote reads at the generating client: Tango does not currently allow a client to execute transactions and generate commit records involving remote reads. Calling an accessor on an object that does not have a local view is problematic, since the data does not exist locally; possible solutions involve invoking an RPC to a different client with a view of the object, if one exists, or recreating the view locally at the beginning of the transaction, which can be too expensive. If we do issue RPCs to other clients, conflict resolution becomes problematic; the node that generated the commit record does not have local views of the objects read by it and hence cannot check their latest versions to find conflicts. As a result, conflict resolution requires a more complex, collaborative protocol involving multiple clients sharing partial, local commit/abort decisions via the shared log; we plan to explore this as future work.

A second limitation is that a single transaction can

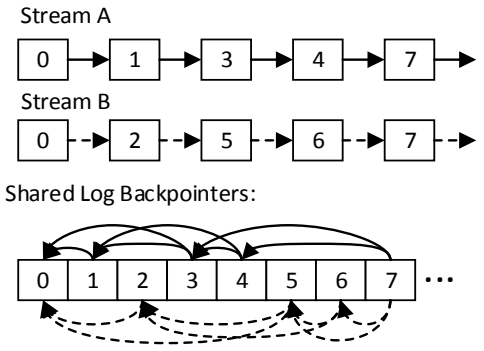

Figure 7: Streams are stored on the shared log with redundant backpointers.

only write to a fixed number of Tango objects. The *multiappend* call places a limit on the number of streams to which a single entry can be appended. As we will see in the next section, this limit is set at deployment time and translates to storage overhead within each log entry, with each extra stream requiring 12 bytes of space in a 4KB log entry.

Failure Handling: The decision record mechanism described above adds a new failure mode to Tango: a client can crash after appending a commit record but before appending the corresponding decision record. A key point to note, however, is that the extra decision phase is merely an optimization; the shared log already contains all the information required to make the commit/abort decision. Any other client that hosts the read set of the transaction can insert a decision record after a time-out if it encounters an orphaned commit record. If no such client exists and a larger time-out expires, any client in the system can reconstruct local views of each object in the read set synced up to the commit offset and then check for conflicts.

5 Streaming CORFU

In this section, we describe our addition of a streaming interface to the CORFU shared log implementation.

As we described in Section 2, CORFU consists of three components: a client-side library that exposes an *append*/*read*/*check*/*trim* interface to clients; storage servers that each expose a 64-bit, write-once address space over flash storage; and a sequencer that hands out 64-bit counter values.

To implement streaming, we changed the client-side library to allow the creation and playback of streams. Internally, the library stores stream metadata as a linked list of offsets on the address space of the shared log, along with an iterator. When the application calls *readnext* on a stream, the library issues a conventional CORFU random read to the offset pointed to by the iterator, and moves the iterator forward.

To enable the client-side library to efficiently construct this linked list, each entry in the shared log now has a small stream header. This header includes a stream ID as well as backpointers to the last K entries in the shared log belonging to the same stream. When the client-side library starts up, the application provides it with the list of stream IDs of interest to it. For each such stream, the library finds the last entry in the shared log belonging to that stream (we'll shortly describe how it does so efficiently). The K backpointers in this entry allow it to construct a K-sized suffix of the linked list of offsets comprising the stream. It then issues a read to the offset pointed at by the Kth backpointer to obtain the previous K offsets in the linked list. In this manner, the library can construct the linked list by striding backward on the log, issuing $\frac{N}{K}$ reads to build the list for a stream with N entries. A higher redundancy factor K for the backpointers translates into a longer stride length and allows for faster construction of the linked list.

By default, the stream header stores the K backpointers using 2-byte deltas relative to the current offset, which overflow if the distance to the previous entry in the stream is larger than 64K entries. To handle the case where all K deltas overflow, the header uses an alternative format where it stores $\frac{K}{4}$ backpointers as 64-bit absolute offsets which can index any location in the shared log's address space. Each header now has an extra bit that indicates the backpointer format used (relative or absolute), and a list of either K 2-byte relative backpointers or $\frac{K}{4}$ 8-byte absolute backpointers. In practice, we use a 31-bit stream ID and use the remaining bit to store the format indicator. If $K = 4$, which is the minimum required for this scheme, the header uses 12 bytes. To allow each entry to belong to multiple streams, we store a fixed number of such headers on the entry. The number of headers we store is equal to the number of streams the entry can belong to, which in turn translates to the number of objects that a single transaction can write to.

Appending to a set of streams requires the client to acquire a new offset by calling increment on the sequencer (as in conventional CORFU). However, the sequencer now accepts a set of stream IDs in the client's request, and maintains the last K offsets it has issued for each stream ID. Using this information, the sequencer returns a set of stream headers in response to the increment request, along with the new offset. Having obtained the new offset, the client-side library prepends the stream headers to the application payload and writes the entry using the conventional CORFU protocol to update the storage nodes. The sequencer also supports an inter-

face to return this information without incrementing the counter, allowing clients to efficiently find the last entry for a stream on startup or otherwise.

Updating the metadata for a stream at the client-side library (i.e., the linked list of offsets) is similar to populating it on startup; the library contacts the sequencer to find the last entry in the shared log belonging to the stream and backtracks until it finds entries it already knows about. The operation of bringing the linked list for a stream up-to-date can be triggered at various points. It can happen reactively when *readnext* is called; but this can result in very high latencies for the *readnext* operation if the application issues reads burstily and infrequently. It can happen proactively on appends, but this is wasteful for applications that append to the stream but never read from it, since the linked list is never consulted in this case and does not have to be kept up-to-date. To avoid second-guessing the application, we add an additional *sync* call to the modified library which brings the linked list up-to-date and returns the last offset in the list to the application. The application is required to call this *sync* function before issuing *readnext* calls to ensure linearizable semantics for the stream, but can also make periodic, proactive *sync* calls to amortize the cost of keeping the linked list updated.

Failure Handling: Our modification of CORFU has a fault-tolerance implication; unlike the original protocol, we can no longer tolerate the existence of multiple sequencers, since this can result in clients obtaining and storing different, conflicting sets of backpointers for the same stream. To ensure that this case cannot occur, we modified reconfiguration in CORFU to include the sequencer as a first-class member of the 'projection' or membership view. When the sequencer fails, the system is reconfigured to a new view with a different sequencer, using the same protocol used by CORFU to eject failed storage nodes. Any client attempting to write to a storage node after obtaining an offset from the old sequencer will receive an error message, forcing it to update its view and switch to the new sequencer. In an 18-node deployment, we are able to replace a failed sequencer within 10 ms. Once a new sequencer comes up, it has to reconstruct its backpointer state; in the current implementation, this is done by scanning backward on the shared log, but we plan on expediting this by having the sequencer store periodic checkpoints in the log. The total state at the sequencer is quite manageable; with $K = 4$ backpointers per stream, the space required is $4 * 8$ bytes per stream, or 32MB for 1M streams.

In addition, crashed clients can create holes in the log if they obtain an offset from the sequencer and then fail before writing to the storage units. In conventional CORFU, any client can use the *fill* call to patch a hole with a junk value. Junk values are problematic for streaming CORFU since they do not contain backpointers. When the client-side library strides through the backpointers to populate or update its metadata for a stream, it has to stop if all K relative backpointers from a particular offset lead to junk entries (or all $\frac{K}{4}$ backpointers in the absolute backpointer format). In our current implementation, a client in this situation resorts to scanning the log backwards to find an earlier valid entry for the stream.

6 Evaluation

Our experimental testbed consists of 36 8-core machines in two racks, with gigabit NICs on each node and 20 Gbps between the top-of-rack switches. Half the nodes (evenly divided across racks) are equipped with two Intel X25V SSDs each. In all the experiments, we run an 18-node CORFU deployment on these nodes in a 9X2 configuration (i.e., 9 sets of 2 replicas each), such that each entry is mirrored across racks. The CORFU sequencer runs on a powerful, 32-core machine in a separate rack. The other 18 nodes are used as clients in our experiments, running applications and benchmarks that operate on Tango objects; we don't model the external clients of these applications and instead generate load locally. We use 4KB entries in the CORFU log, with a batch size of 4 at each client (i.e., the Tango runtime stores a batch of 4 commit records in each log entry).

6.1 Single Object Linearizability

We claimed that Tango objects can provide persistence, high availability and elasticity with high performance. To demonstrate this, Figure 8 shows the performance of a single Tango object with a varying number of views, corresponding to the use case in Figure 5(a), where the application uses Tango to persist and replicate state. The code we run is identical to the TangoRegister code in Figure 3. Figure 8 (Left) shows the latency / throughput trade-off on a single view; we can provide 135K sub-millisecond reads/sec on a read-only workload and 38K writes/sec under 2 ms on a write-only workload. Each line on this graph is obtained by doubling the window size of outstanding operations at the client from 8 (left-most point) to 256 (right-most point).

Figure 8 (Middle) shows performance for a 'primary/backup' scenario where two nodes host views of the same object, with all writes directed to one node and all reads to the other. Overall throughput falls sharply as writes are introduced, and then stays constant at around 40K ops/sec as the workload mix changes; however, average read latency goes up as writes dominate, reflecting the extra work the read-only 'backup' node has to

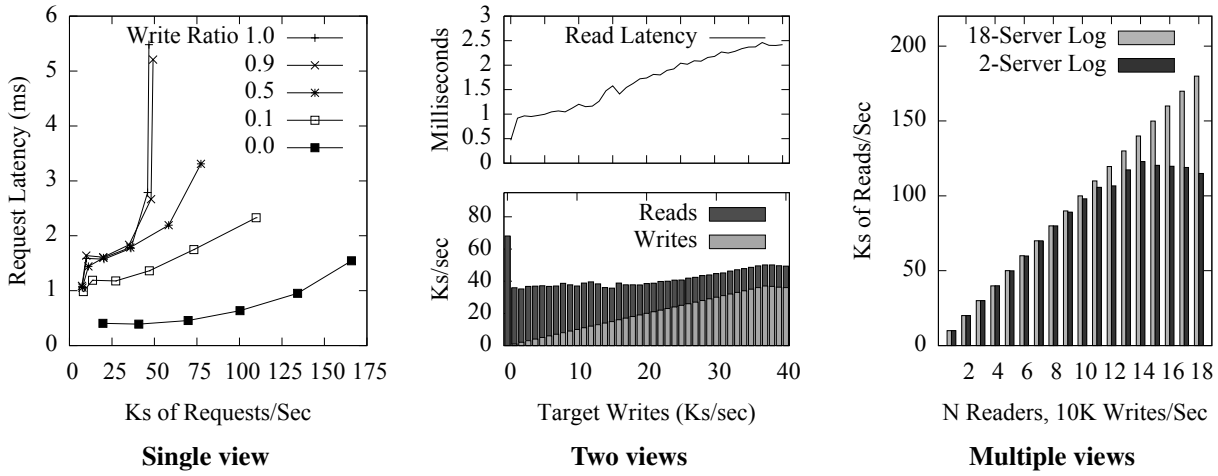

Figure 8: A TangoRegister is persistent with one view; highly available with two views; and elastic with N views.

perform to catch up with the 'primary'. Note that either node can serve reads or writes, effectively enabling immediate fail-over if one fails. This graph shows that Tango can be used to support a highly available, high-throughput service.

Figure 8 (Right) shows the elasticity of linearizable read throughput; we scale read throughput to a Tango object by adding more read-only views, each of which issues 10K reads/sec, while keeping the write workload constant at 10K writes/sec. Reads scale linearly until the underlying shared log is saturated; to illustrate this point, we show performance on a smaller 2-server log which bottlenecks at around 120K reads/sec, as well as the default 18-server log which scales to 180K reads/sec with 18 clients. Adding more read-only views does not impact read latency; with the 18-server log, we obtain 1 ms reads (corresponding to the point on the previous graph for a 10K writes/sec workload).

6.2 Transactions

We now show that Tango provides transactional support within and across objects. We first focus on single-object transactions. Figure 9 shows transaction throughput and goodput (i.e., committed transactions) on a single TangoMap object as we vary the degree of contention (by increasing the number of keys within the map) and increase the number of nodes hosting views of the object. Each node performs transactions, and each transaction reads three keys and writes three other keys to the map. Figure 9 (Top) chooses keys using a highly skewed zipf distribution (corresponding to workload 'a' of the Yahoo! Cloud Serving Benchmark [17]). Figure 9 (Bottom) chooses keys using a uniform distribution. For 3 nodes, transaction goodput is low with tens

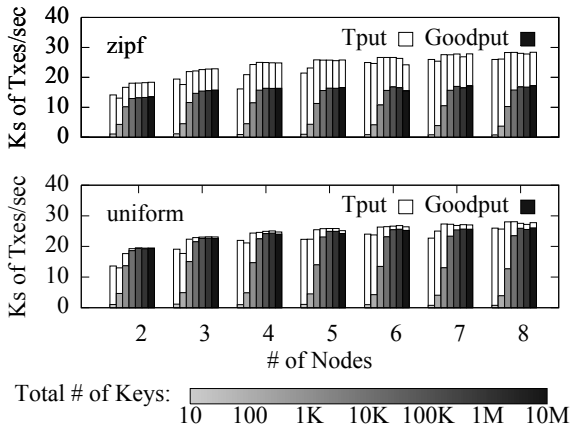

Figure 9: Transactions on a fully replicated TangoMap offer high, stable goodput.

or hundreds of keys but reaches 99% of throughput in the uniform case and 70% in the zipf case with 10K keys or higher. Transaction throughput hits a maximum with three nodes and stays constant as more nodes are added; this illustrates the playback bottleneck, where system-wide throughput is limited by the speed at which a single client can play the log. Transaction latency (not shown in the graph) averages at 6.5 ms with 2 nodes and 100K keys. Next, we look at how layered partitioning alleviates the playback bottleneck.

Figure 10 (Left) substantiates our claim that layered partitioning can provide linear scaling until the underlying shared log is saturated. Here, each node hosts the view for a different TangoMap and performs single-object transactions (with three reads and three writes) over it. We use both a small shared log with 6 servers as well as the default 18-server one. As expected, throughput scales linearly with the number of nodes until it satu-

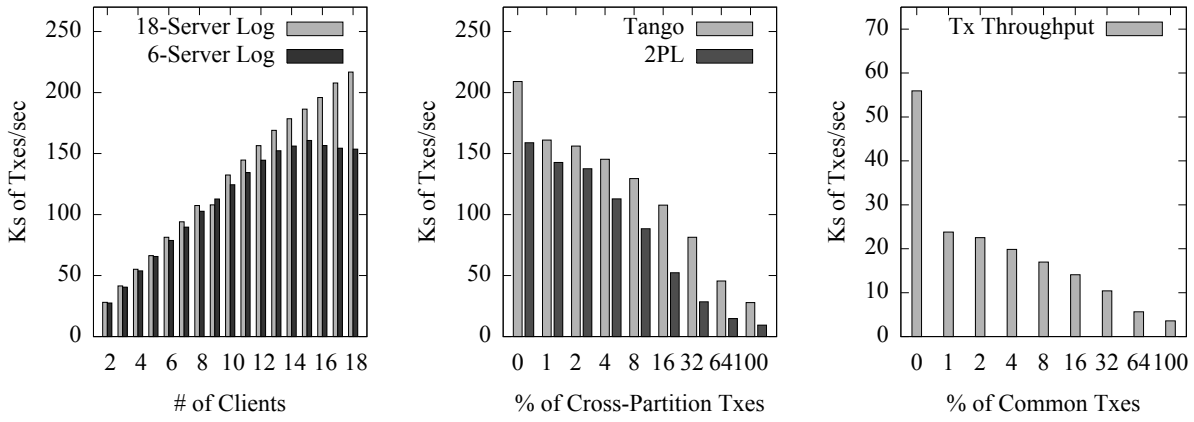

Figure 10: Tango allows services to scale via partitioning (Left), while still providing fast transactions across partitions (Middle) and on shared objects (Right).

rates the shared log on the 6-server deployment at around 150K txes/sec. With an 18-server shared log, throughput scales to 200K txes/sec and we do not encounter the throughput ceiling imposed by the shared log.

We stated that the underlying shared log enables fast transactions across objects. We now look at two types of transactions across different objects. First, in Figure 10 (Middle), we consider the partitioned setup from the previous experiment with 18 nodes running at 200K txes/sec, where each node hosts a view for a different TangoMap with 100K keys. We introduce cross-partition transactions that read the local object but write to both the local as well as a remote object (this corresponds to an operation that moves a key from one map to another).

To provide a comparison point, we modified the Tango runtime's *EndTX* call to implement a simple, distributed 2-phase locking (2PL) protocol instead of accessing the shared log; this protocol is similar to that used by Percolator [38], except that it implements serializability instead of snapshot isolation for a more direct comparison with Tango. On *EndTX-2PL*, a client first acquires a timestamp from a centralized server (corresponding to the Percolator timestamp server; we use our sequencer instead); this is the version of the current transaction. It then locks the items in the read set. If any item has changed since it was read, the transaction is aborted; if not, the client then contacts the other clients in the write set to obtain a lock on each item being modified as well as the latest version of that item. If any of the returned versions are higher than the current transaction's version (i.e., a write-write conflict) or a lock cannot be obtained, the transaction unlocks all items and retries with a new sequence number. Otherwise, it sends a commit to all the clients involved, updating the items and their versions and unlocking them.

As Figure 10 (Middle) shows, throughput degrades gracefully for both Tango and 2PL as we double the percentage of cross-partition transactions. We don't show goodput in this graph, which is at around 99% for both protocols with uniform key selection. Our aim is to show that Tango has scaling characteristics similar to a conventional distributed protocol while suffering from none of the fault-tolerance problems endemic to such protocols, such as deadlocks, crashed coordinators, etc.

Next, we look at a different type of transaction in Figure 10 (Right), where each node in a 4-node setup hosts a view of a different TangoMap as in the previous experiment, but also hosts a view for a common TangoMap shared across all the nodes (corresponding to the use case in Figure 5(d)). Each map has 100K keys. For some percentage of transactions, the node reads and writes both its own object as well as the shared object; we double this percentage on the x-axis, and throughput falls sharply going from 0% to 1%, after which it degrades gracefully. Goodput (not shown in the graph) drops marginally from 99% of throughput to 98% of throughput with uniform key selection.

6.3 Other Data Structures

To show that Tango can support arbitrary, real-world data structures, we implemented the ZooKeeper interface over Tango in less than 1000 lines of Java code, compared to over 13K lines for the original [15] (however, our code count does not include ancillary code used to maintain interface compatibility, such as various Exceptions and application callback interfaces, and does not include support for ACLs). The performance of the resulting implementation is very similar to the TangoMap numbers in Figure 10; for example, with 18 clients running independent namespaces, we obtain around 200K txes/sec if transactions do not span names-

paces, and nearly 20K txes/sec for transactions that atomically move a file from one namespace to another. The capability to move files across different instances does not exist in ZooKeeper, which supports a limited form of transaction within a single instance (i.e., a *multi-op* call that atomically executes a batch of operations).

We also implemented the single-writer ledger abstraction of BookKeeper [30] in around 300 lines of Java code (again, not counting Exceptions and callback interfaces). Ledger writes directly translate into stream appends (with some metadata added to enforce the single-writer property), and hence run at the speed of the underlying shared log; we were able to generate over 200K 4KB writes/sec using an 18-node shared log. To verify that our versions of ZooKeeper and BookKeeper were full-fledged implementations, we ran the HDFS namenode over them (modifying it only to instantiate our classes instead of the originals) and successfully demonstrated recovery from a namenode reboot as well as failover to a backup namenode.

7 Related Work

Tango fits within the SMR [42] paradigm, replicating state by imposing a total ordering over all updates; in the vocabulary of SMR, Tango clients can be seen as learners of the total ordering, whereas the storage nodes comprising the shared log play the role of acceptors. A key difference is that the shared log interface is a superset of the traditional SMR upcall-based interface, providing persistence and history in addition to consistency.

Tango also fits into the recent trend towards enabling strong consistency across sharded systems via a source of global ordering; for example, Percolator [38] uses a central server to issue non-contiguous timestamps to transactions, Spanner [18] uses real time as an ordering mechanism via synchronized clocks, and Calvin [44] uses a distributed agreement protocol to order batches of input transactions. In this context, the Tango shared log can be seen as a more powerful ordering mechanism, since it allows any client to iterate over the global ordering and examine any subsequence of it.

Multiple systems have aimed to augment objects with strong consistency, persistence and fault-tolerance properties. Thor [32] provided applications with persistent objects stored on backend servers. More recently, Live Objects [37] layer object interfaces over virtual synchrony, OpenReplica [6] transparently replicates Python objects over a Paxos implementation, while Tempest [34] implements fault-tolerant Java objects over reliable broadcast and gossip. Tango objects are similar to the Distributed Data Structures proposed in Ninja [22, 23], in that they provide fault-tolerant, strongly consistent data structures, but differ by providing transactions across items, operations, and different data structures.

Tango is also related to log-structured storage systems such as LFS [39] and Zebra [24]. A key difference is that Tango assumes a shared log with an infinite address space and a *trim* API; internally, the shared log implementation we use implements garbage collection techniques similar to those found in modern SSDs.

The transaction protocol described in Section 3 is inspired by Hyder [11], which implemented optimistic concurrency control for a fully replicated database over a shared log; we extend the basic technique to work in a partitioned setting over multiple objects, as described in Section 4. In addition, the Hyder paper included only simulation results; our evaluation provides the first implementation numbers for transactions over a shared log. The original CORFU paper implemented atomic multi-puts on a shared log, but did not focus on arbitrary data structures or full transactions over a shared log.

A number of recent projects have looked at new programming abstractions for non-volatile RAM [16]; some of these provide transactional interfaces over commodity SSDs [40] or byte-addressable NV-RAM [47]. Tango has similar goals to these projects but is targeted at a distributed setting, where fault-tolerance and consistency are as important as persistence.

8 Conclusion

In the rush to produce better tools for distributed programming, metadata services have been left behind; it is arguably as hard to build a highly available, persistent and strongly consistent metadata service today as it was a decade earlier. Tango fills this gap with the abstraction of a data structure backed by a shared log. Tango objects are simple to build and use, relying on simple append and read operations on the shared log rather than complex messaging protocols. By leveraging the shared log to provide key properties – such as consistency, persistence, elasticity, atomicity and isolation – Tango makes metadata services as easy to write as a MapReduce job or a photo-sharing website.

Acknowledgments

We'd like to thank Dave Andersen for shepherding the paper. We thank Phil Bernstein for making us think about shared log designs, Marcos Aguilera for valuable feedback on the system design and guarantees, Subramaniam Krishnan for help with the HDFS tests, Paul Barham for assistance with the sequencer implementation, and Shobana Balakrishnan for her input in the early stages of the project.

References

[1] Amazon S3. http://aws.amazon.com/s3/.

[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[3] Apache Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] HDFS 1623: High Availability Framework for HDFS NN. https://issues.apache.org/jira/browse/HDFS-1623.

[5] HDFS Name Node. http://wiki.apache.org/hadoop/NameNode.

[6] OpenReplica.org. http://www.openreplica.org.

[7] RAMCloud Coordinator. https://ramcloud.stanford.edu/wiki/display/ramcloud/Coordinator+-+Design+Discussions.

[8] Windows Azure. http://www.windowsazure.com.

[9] Windows Registered I/O. http://msdn.microsoft.com/en-us/library/windows/desktop/ms740642(v=vs.85).aspx.

[10] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *NSDI 2012*.

[11] P. Bernstein, C. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR 2011*, 2011.

[12] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *Software Engineering, IEEE Transactions on*, (3):203–216, 1979.

[13] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-performance Data Store. In *NSDI 2011*.

[14] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI 2006*.

[15] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *SOSP 2009*.

[16] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 105–118. ACM, 2011.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[18] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-Distributed Database. In *OSDI 2012*.

[19] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *USENIX ATC 2010*.

[21] R. S. Finlayson and D. R. Cheriton. Log Files: An Extended File Service Exploiting Write-Once Storage. In *SOSP 1987*.

[22] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *OSDI 2002*.

[23] S. D. Gribble, M. Welsh, R. Von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[24] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[25] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM TOCS*, 6(1):82–108, 1988.

[26] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM CCR*, 25(4):328–341, 1995.

[27] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC 2010*.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[29] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: remote mirroring done write. In *USENIX ATC 2003*.

[30] F. P. Junqueira, I. Kelly, and B. Reed. Durability with BookKeeper. *ACM SIGOPS Operating Systems Review*, 47(1):9–15, 2013.

[31] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[32] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD Record*, vol-

ume 25, pages 318–329. ACM, 1996.

[33] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys 2012*.

[34] T. Marian, M. Balakrishnan, K. Birman, and R. Van Renesse. Tempest: Soft state replication in the service tier. In *DSN 2008*.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[36] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *FAST 2010*.

[37] K. Ostrowski, K. Birman, and D. Dolev. Live distributed objects: Enabling the active web. *Internet Computing, IEEE*, 11(6):72–78, 2007.

[38] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[39] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1), Feb. 1992.

[40] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant. Hathi: durable transactions for memory using flash. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 33–38. ACM, 2012.

[41] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *ACM SIGOPS OSR*, volume 25. ACM, 1991.

[42] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[43] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Compcon Spring'88*.

[44] 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*, pages 1–12. ACM, 2012.

[45] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.

[46] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012.

[47] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.