Sirius: Distributing and Coordinating Application Reference Data

Michael Bevilacqua-Linn, Maulan Byron, Peter Cline, Jon Moore, and Steve Muir, *Comcast Cable*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/bevilacqua-linn

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Sirius: Distributing and Coordinating Application Reference Data

Michael Bevilacqua-Linn, Maulan Byron, Peter Cline, Jon Moore, and Steve Muir

{Michael_Bevilacqua-Linn,Peter_Cline2,Jon_Moore}@comcast.com,
{Maulan_Byron,Steve_Muir}@cable.comcast.com
Comcast Cable

Abstract

The main memory of a typical application server is now large enough to hold many interesting *reference datasets* which the application must access frequently but for which it is not the system of record. However, application architectures have not evolved to take proper advantage. Common solutions based on caching data from a separate persistence tier lead to error-prone I/O code that is still subject to cache miss latencies. We present an alternative library-based architecture that provides developers access to in-memory, native data structures they control while neatly handling replication and persistence. Our open-source library *Sirius* can thus give developers access to their reference data in single-node programming style while enjoying the scaling and robustness of a distributed system.

1 Introduction

Many applications need to use *reference data*—information that is accessed frequently but not necessarily updated in-band by the application itself. For example, a TV guide application may need to know the titles, seasons, and descriptions of various TV shows. Furthermore, many reference datasets now fit comfortably in memory, especially as the exponential progress of Moore's Law has outstripped these datasets' growth rates. To illustrate, the total number of feature films listed in the Internet Movie Database (IMDb) grew 40% from 1998 to 2013 [11], whereas commodity server RAM grew by 12,000% in the same period.[1]

In addition, common "*n*-tier" service architectures tend to separate the management of reference data from the application code that must use it. In order to maintain low latency, application developers maintain caching layers, either built into a supporting library such as an object-relational mapper (ORM) or managed explicitly with external caches like *memcached* [6] or *Redis* [17]. These schemes may be difficult to use correctly [18], force developers to deal with I/O, and are still subject to cache misses which drive up upper-percentile latencies. This raises the question at hand:

> How can we keep this reference data entirely in RAM, while ensuring it gets updated as needed and is easily accessible to developers?

In this paper, we describe an alternative architecture to the classic *n*-tier approach for managing reference data, based on a distributed system library called *Sirius*. Sirius offers the following combination of properties:

- Simple and semantically transparent interface and freedom to use arbitrary, native data structures for the in-memory representation of the reference data (Section 4).
- Eventually consistent, near real-time replication of reference data updates across datacenters connected by wide-area networks (WANs) and designed for robustness to certain common types of server and network failures. (Section 5).
- Persistence management and automated recovery after application server restarts (Section 6).
- Adequate write throughput to support reference data updates (Section 7).

We have been using Sirius for over 15 months in production to power applications serving tens of millions of clients. We discuss the operational aspects of this architecture in Section 8.

2 Background

We will begin with a description of *reference data* to establish the context for this paper. As a motivating example, we will use the domain of professionally-produced

[1] 1998 Sun Ultra 2 server with 256 MB vs. 2013 Elastic Compute Cloud (EC2) "m3.2xlarge" instance with 30 GB.

television and movie metadata—the primary use case that motivated the design and implementation of Sirius. Examples of this metadata include facts such as the year *Casablanca* was released, how many episodes were in Season 7 of *Seinfeld*, or when the next episode of *The Voice* will be airing (and on which channel). Such reference datasets have certain distinguishing characteristics:

Small. The overall dataset is not "big data" by any stretch of the imagination, and in fact can fit in main memory of modern commodity servers. For example, the entertainment metadata we use is in the low tens of gigabytes (GB) in size.

Very high read/write ratio. Overwhelmingly, this data is write-once, read-many—for example, the original *Casablanca* likely won't get a different release date, *Seinfeld* won't suddenly get new Season 7 episodes, and *The Voice* will probably air as scheduled. However, this data is central to almost every piece of functionality and user experience in relevant applications—and those applications may have tens of millions of users.

Asynchronously updated. End users largely have a read-only view of this data and hence are not directly exposed to the latency of updates. For example, an editorial change to correct a misspelling of "*Cassablanca*" may take a while to propagate to all the application servers, with end users seeing the update occur but without being able to distinguish whether it was accomplished in minutes or milliseconds. There *are* thresholds of acceptable latency, however: if a presidental press conference is suddenly called, schedules may need to be updated within minutes rather than hours.

Such reference datasets are relatively common: a download of the U.S. Federal Reserve's data on foreign exchange rates is 1.2 MB compressed [4], the entire *Encylopædia Britannica* is 4.2 GB [27], and the collection of global daily weather measurements from 1929–2009 is only 20 GB [2]. In fact, the most recent versions of all the English Wikipedia articles total around 43 GB in uncompressed XML format as of December 2, 2013, which already fits in the 88 GB of RAM on a "cr1.8xlarge" EC2 instance. In other words, many reference datasets are already "small," and more will become so as server memory sizes continue to grow.

2.1 Operational Environment

We would like to access our reference data in the context of providing modern interactive, consumer-facing Web and mobile application services. This imposes some important constraints and design considerations on a potential solution. In particular, these services must support:

Multiple datacenters. We expect our services to run in multiple locations both to minimize latency to geographically disparate clients but also to protect against datacenter failures caused by the proverbial backhoe or regional power outages. For example, AWS has experienced multiple total-region failures but no multi-region or global failures to date.

Low access latency. Because we are building interactive applications where actual humans are waiting for responses from our application servers, we must have fast access to our reference data. Service latencies directly impact usage and revenue [9].

Continuous delivery. Our services will be powering products that are constantly being evolved. We expect our application developers to spend a lot of time modifying the code that interacts with their reference data, and we expect to be able to deploy code updates to our production servers multiple times per day. In order to do this safely, we prefer approaches that support easy and rapid automated testing.

Robustness. Since we will be supporting large deployments, we expect to experience a variety of failure conditions, including application server crashes, network partitions, and failures of our own service dependencies. We would like our overall system to continue operating—although perhaps with degraded functionality—in the face of these failures.

Operational friendliness. Any system of sufficient complexity will exhibit emergent (unpredictable) behavior, which will likely have to be managed by operational staff. We would like our approach to have a simple *operational interface*: it should be easy to understand "how it works," things should fail in obvious but safe ways, it should be easy to observe system health and metrics, and there should be "levers" to pull with predictable effects to facilitate manual interventions.

3 Approach

As we have seen, we can fit our reference data comfortably in RAM, so we start with the idea that we will simply keep a full mirror of the data on each application server, stored in-process as native data structures. This offers them ultimate convenience:

- No I/O calls are needed to external resources to access the data. Correspondingly, there is no connection pool tuning required nor is there a need to handle network I/O exceptions.

- Automated tests involving the reference data can be vanilla unit tests that neither perform I/O nor require extensive use of mock objects or test doubles.

- Profilers are actually useful for finding and fixing application bottlenecks since behavior is isolated from external dependencies. By contrast, com-

mon "*n*-tier" application servers are often I/O bound rather than CPU bound.

- Developers have full freedom to choose data structures directly suited to the application's use cases.
- There are no "cache misses," as the entire dataset is present; access is fast and predictable.

Of course, this approach raises several important questions in practice. How do we keep the mirror up to date? How do we run multiple servers while ensuring each gets every update? How do we restore the mirrors after application server restarts?

3.1 Update publishing

We assume there are external systems responsible for curating the reference data set(s), and that these system will *publish* updates to our server rather than having our server poll the system of record looking for updates. This event-driven approach is straightforward to implement in our application; we update the native data structures in our mirror and continue serving client requests. We model this interface after HTTP as a series of PUTs and DELETEs against various URL keys.

Furthermore, this introduces a separation of operational concerns: should the external system of record become unavailable, our application simply stops receiving updates without needing the error handling code that would be required by a polling approach. Indeed, our application cannot (and does not need to) distinguish a failed source system from one with no fresh updates.

Finally, this is a parsimonious way to process updates where much of the reference data does not change from moment to moment, but where there *is* a regular trickle of updates. For example, our production datasets experience a nominal update rate of no more than 150 updates per second. This incremental approach allows us to avoid re-downloading and re-parsing a full mirror and then either calculating diffs ourselves or swapping out a mirror entirely for an updated version.

3.2 Replication and Consistency

To run a cluster of application servers, we need to apply the updates at every server. To isolate the system of record from needing to know how many application servers are deployed, we route updates through a load balancer, then rely on the servers to replicate the updates to each other. Thus, a source system can publish each update once. This also isolates the system of record from individual server failures.

Because we will be operating a distributed system across a WAN, we know we will experience frequent network partitions due to unusually high latency, route flaps, or other failures. Therefore, the CAP theorem dictates we will need to decide between availability and consistency during these times. We need read access to the reference data at all times and will have to tolerate some windows of inconsistency. That said, we want to preserve at least *eventual consistency* to retain operational sanity, and we can tolerate some unavailability of *writes* during a partition, as our reference data updates are asynchronous from the point of view of our clients.

To achieve this, our cluster uses a variant of the Multi-Paxos [5] protocol to agree on a consistent total ordering of updates, and then have each server apply the updates in order. This means that the system of record cannot publish updates without a quorum of available servers, but it also allows servers to read the reference data without coordination. A formal consensus protocol also allows us to consistently order updates from multiple systems of record. We provide more detail in Section 5.

Finally, we use a *catchup protocol* for filling in lost or missing updates from peers. This protocol can also be used to set up dependent servers that do not participate in Paxos but nonetheless receive and apply the stream of updates. As a result, we can set up very flexible replication topologies to scale out read access to the dataset.

3.3 Persistence

As with many Paxos implementations, each server provides persistence by maintaining a local transaction log on disk of the committed updates. When a server instance starts up, it replays this transaction log to rebuild its mirror from scratch, then rejoins the replication protocol described above, which includes "catch up" facilities for acquiring any missing updates.

Because updates are modeled as idempotent PUT and DELETE operations, we can compact this log to retain only the most recent updates for active keys, without requiring the application to implement checkpointing. Our live log compaction scheme is described in Section 6.

3.4 Library Structure

Finally, we have structured the overall system around a library called Sirius that handles the Paxos implementation, persistence, log compaction, and replay. The hosting application is then conceptually separated into two pieces: its external interface and business logic, and its mirror, as shown in Figure 1. There are then two different data paths used for processing reference data updates and client requests.

A server receiving a reference data update hands it off as a PUT or DELETE to Sirius, which runs it through the Paxos protocol and writes it to the persistent transaction

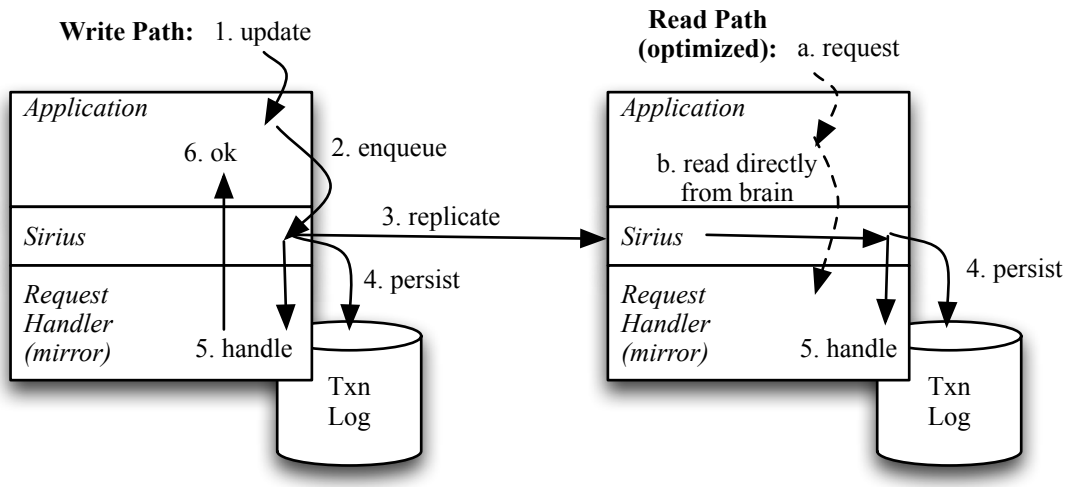

Figure 1: Architecture of a Sirius-based application.

log. The Sirius library on each server then hands the update off to its local mirror to be applied.

The application server can handle read requests in one of two ways. It can hand a request off to its local Sirius, which will serialize it with respect to outstanding updates and then route it as a GET to the mirror. This method would be most appropriate if the application developers wish to have Sirius provide a level of transactional isolation for read requests–ensuring that no updates are applied concurrently to those reads. However, it comes at the expense of serialized access, which may become a performance bottleneck.

In practice, though, our developers use concurrent data structures in the mirror and manage concurrency themselves, bypassing Sirius entirely on the read path.

4 Programming Interface

As we have just seen, a Sirius-based application is conceptually divided into two parts, with the Sirius library as an intermediary. The application proper provides its own interface, for example, exposing HTTP endpoints to receive the reference data publishing events. The application then routes reference data access through Sirius.

After taking care of serialization, replication, and persistence, Sirius invokes a corresponding callback to a *request handler* provided by the application. The request handler takes care of updating or accessing the in-memory representations of the reference data. The application developers are thus completely in control of the native, in-memory representation of this data.

The corresponding programming interfaces are shown in Figure 2; there is a clear correspondence between

```
public interface Sirius {
  Future<SiriusResult>
    enqueueGet(String key);
  Future<SiriusResult>
    enqueuePut(String key, byte[] body);
  Future<SiriusResult>
    enqueueDelete(String key);
  boolean isOnline();
}

public interface RequestHandler {
  SiriusResult handleGet(String key);
  SiriusResult handlePut(String key,
                         byte[] body);
  SiriusResult handleDelete(String key);
}
```

Figure 2: Sirius interfaces. A `SiriusResult` is a Scala case class representing either a captured exception or a successful return, either with or without a return value.

the Sirius-provided access methods and the application's own request handler. As such, it is easy to imagine a "null Sirius" implementation that would simply invoke the application's request handler directly. This semantic transparency makes it easy to reason functionally about the reference data itself.

The primary Sirius interface methods are all asynchronous; the contract is that the library itself takes care of scheduling the invocation of the request handlers at the right time to ensure enventual consistency across the

cluster nodes. The overall contract is:

- The request handlers for PUTs and DELETEs will be invoked serially and in a consistent order across all nodes.
- Enqueued asynchronous updates will not complete until successful replication has occurred.
- An enqueued GET will be routed locally only, but will be serialized with respect to pending updates.
- At startup time, Sirius will not report itself as "online" until it has completed replay of its transaction log, as indicated by the `isOnline` method.

Sirius does not provide facilities for consistent conditional updates (e.g., compare-and-swap); it merely guarantees consistent ordering of the updates. The low update rate coupled with Sirius not being the system of record mean that the simpler interface presented here has been sufficient in practice.

5 Replication

Updates passed to Sirius via `enqueuePut` or `enqueueDelete` are ordered and replicated via Multi-Paxos with each update being a *command* assigned to a *slot* by the protocol; the slot numbers are recorded as sequence numbers in the persistent log (Section 6). Our implementation fairly faithfully follows the description given by van Renesse [26]. However, there are some slight differences needed to produce a practical implementation and some optimizations made possible by our particular use case.

Stable leader. First, we use the common optimization that disallows continuous "Phase 1" weak leader elections; when a node is pre-empted by a peer with a higher ballot number, it *follows* that peer instead of trying to win the election again. While following, it pings the leader to check for liveness, and forwards any update requests. If the pings fail, a new Phase 1 election begins. This limits vote conflicts, and resultant chattiness, which in turn enhances throughput.

End-to-end retries. Second, because all of the updates are idempotent, we do not track unique request identifiers, as the updates can be retried by the external system of record if a publication attempt is not acknowledged. In turn, this assumption means that we do not need to write the internal process state of the Paxos protocol processes to stable storage to recover from crashes, beyond the persistent log recording decisions assigning updates to specific sequence/slot numbers. This may result in an in-flight update being lost in certain failure scenarios, like a cluster-wide power outage, but as the external client will not have had the write acknowledged, it will eventually time out and retry. This end-to-end design argument for retries allows us to work around the assumption from van Renesse that "a message sent by a non-faulty process to a non-faulty destination process is eventually received (at least once) by the destination process," *i.e.*, that the network is reliable.

Similarly, we bound some processes—notably the "Commander" process that attempts to get a quorum of cluster members to agree on the assignment of an update to a particular slot number—with timeouts and a limited number of retries before giving up on the command. As a practical example, during a long-lived network partition, a minority partition will not be able to make progress, and this prevents the buildup of an unbounded amount of protocol state for attempted but incomplete writes during the partition. In turn, this means Sirius degrades gracefully and does not slow the read path for those nodes, even though their reference datasets in memory may begin to become stale.

Write behind. Finally, the Paxos replicas apply decisions in order by sequence number, buffering any out-of-order decisions as needed. We elected to write the decisions out to the log in sequence number order, as it simplifies log replay and the compaction activities described below in Section 6. We also acknowledge the write once a decision for it has been received, but without waiting for persistence or application to complete; this reduces system write latency and prevents "head-of-line" blocking.

On the other hand, this means that an external publishing client may not get "read your writes" consistency and that there is a window during which an acknowledged write can be lost without having been written to stable storage (*e.g.*, due to a power outage). In practice, neither of these is a problem, as the reference dataset updates are read-only from the point of view of the application's primary customers; similarly, Sirius is not the system of record for the reference dataset, so it is possible to reconstruct lost writes if needed by re-publishing the relevant updates from the upstream system.

State pruning. Once updates have been applied locally, a hint with the maximum sequence number is sent to the local Paxos replica; this allows the in-memory state of proposed updates and recorded decisions to be pruned. We additionally prune proposed but unfinished updates that are older than a given configured cutoff window we expect is longer than an external client's read timeout setting, again relying on the end-to-end retry mechanism to re-publish them.

5.1 Catch-Up Protocol

Because updates must be applied in the same order on all nodes and we also want to log updates to disk in

that order, nodes are particularly susceptible to lost decision messages; these delay updates with higher sequence numbers. Therefore, each node periodically selects a random peer and requests a range of updates starting from the lowest sequence number for which it does not have a decision.

The peer replies with all the decisions it has that fall within the given slot number range. Some of these may be returned from a small in-memory cache of updates kept by the peer, especially if the missing decision is a relatively recent one. However, the peer may need to consult the persistent log for older updates no longer in its cache (see Section 6).

If the peer replies with a full range of updates, the process continues: the node requests the next range from the same peer. Once a partial range is returned, the catchup protocol ceases until the next period begins with a new, random peer.

The catchup protocol also provides for dependent cluster members that do not participate in Paxos. Cluster configurations contain only primary members, which thereby know about each other and can participate in Paxos. Other cluster members periodically catch up via the primaries. In practice, this allows a primary "ingest" cluster to feed several dependent application clusters following along for updates—often within seconds of each other and across datacenters, depending upon configuration of the catchup range size and request interval.

In turn, this lets us keep write latencies to a minimum: Paxos only runs across local area networks (LANs). Different clusters can be activated as primaries by pushing a cluster configuration update, which the Sirius library processes without an application restart.

6 Persistence

As updates are ordered by Paxos, Sirius also writes them out to disk in an append-only file. Each record includes an individual record-level checksum, its Paxos sequence number, a timestamp (used for human-readable logging, not for ordering), an operation code (PUT or DELETE), and finally a key and possibly a body (PUTs only), along with related field framing information. This results in variable-sized records, which are not ordinarily a problem: the log is appended by normal write processing, and is normally only read at application startup, where it is scanned sequentially anyway.

There is one exception to this sequential access pattern: while responding to catchup requests, we need to find updates that have fallen out of cache, perhaps because of a node crash or long-lived network partition. In this case, we must find a particular log entry by its sequence number.

At system startup time, Sirius will read the log in order to stream the updates to the application's request handler. At the same time, it will construct a companion index file if one does not already exist. This index file consists of fixed-length records of the form *<seqnum, offset, checksum>*. Although the log file is guaranteed to be sorted in increasing sequence number order, compaction—as we will see shortly—may introduce gaps in the sequence. We perform a binary search on the index file itself to find a particular update number, then use the file offset to locate the update itself in the main log file. Because the index file is small, the operating system's filesystem cache can usually keep it in memory, allowing a quick binary search. When an update is found in the log, any further updates in the requested catchup range can then be streamed without reconsulting the index. In practice, the catchup protocol streams updates to other nodes at least as fast as new updates are written.

6.1 Compaction

Sirius can *compact* its log file: because the PUTs and DELETEs are idempotent, we can remove every log entry for a key except the one with the highest sequence number. Because the overall reference dataset does not grow dramatically in size over time, a compacted log is a relatively compact representation of it; we find that the reference dataset takes up more space in RAM than it does in the log once all the appropriate indices have been created in the application's mirror. This avoids the need for the application to participate in creating snapshots or checkpoints, as in other event-sourced systems [5].

The original Sirius-based application we deployed was under continuous development at the time and was redeployed with new code nearly every day. We took advantage of the application restarts to compact offline during deployments. A separate tool that understands log format would compact the log while a particular application instance was down; the application would then restart by replaying the compacted log. While workable, this was suboptimal: the offline compaction lengthened deployments, and we relied on frequent restarts to prevent the log from getting unwieldy.

Therefore, we developed a scheme for *live compaction* that the Sirius library manages itself in the background. In order to bound the resources (particularly memory) needed by the compaction scheme, we work incrementally by dividing the log into *segments* with a maximum number of entries in each, as in other log-based systems [23, 24]. Each segment is kept in its own directory, and contains a data file and an index file, as described above. Sirius adds new entries only to the highest-numbered segment; when that segment fills up,

its file is closed and a new segment is started.

The overall compaction process is as follows: we take a segment s_i and find the set of all keys mentioned in it, $K(s_i)$. Then, for each lowered-numbered segment s_j, we produce a new copy of its data file s'_j that only contains updates to keys not in $K(s_i)$. When s'_j is complete, we swap it into place for s_j atomically. Files are named and managed in a way that allows the compaction process to recover from an application restart; if it finds an existing s'_j file it assumes it is incomplete, removes it, and restarts the compaction of that segment.

After the compaction of the individual segments, we then check if any adjacent segments are small enough to be *merged* and still fit under the maximum segment size; this calculation can be done cheaply by checking the size of the relevant segment index files. The merge operation itself is simply a concatenation of the log and index files for the relevant segments.

Live compaction in Sirius is thus incremental and restartable and does not require a manual operational maintenance step with a separate tool. Thus, while operations staff for a Sirius-based application need to be aware of the segment directory, they do not have to actively manage it. Because all the log segments are regular files, they can be backed up normally, or zipped and copied to another machine. Since the logs are normally append-only, and compaction is incremental, these copies can be taken while the application is running without any special synchronization. We have taken advantage of this to bootstrap new nodes efficiently, especially when seeding a new datacenter, or to copy a production dataset elsewhere for debugging or testing.

7 Experimental Evaluation

In practice we have found Sirius has sufficient write throughput to support our reference data use cases. In this section, we analyze our Sirius implementation experimentally. The library is written in Scala, using the Akka actor library.

All experiments were run on Amazon Web Services (AWS) Elastic Computer Cluster (EC2) servers running a stock 64-bit Linux kernel [2] on "*m1.xlarge*" instances[3] with 4 virtual CPUs and 15 GB RAM. These instances have a 64-bit OpenJDK Java runtime[4] installed; Sirius-based tests use version 1.1.4 of the library.

7.1 Write throughput

Because the optimized read path for an application bypasses Sirius to read directly from the mirror, we are primarily interested in measuring Sirius' write throughput. For these tests, we embed Sirius in a reference web application[5] that exposes a simple key-value store interface via HTTP and uses Java's `ConcurrentHashMap` for its mirror. Load is generated from separate instances running JMeter[6] version 2.11. All requests generate PUTs with 179 byte values (the average object size we see in production use).

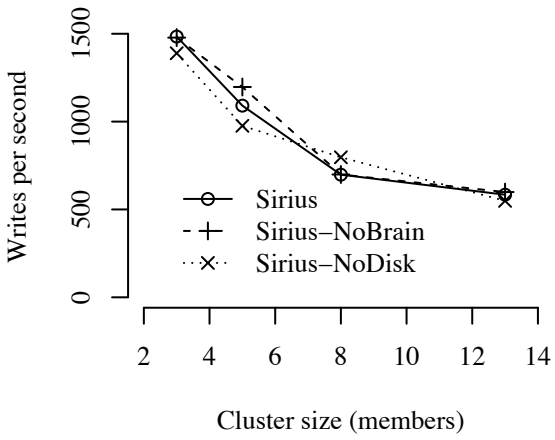

Figure 3: Sirius write throughput.

For these throughput tests, we begin by establishing a baseline under a light load that establishes latency with minimal queueing delay. We then increase load until we find the throughput at which average latency begins to increase; this establishes the maximum practical operating capacity. Our results are summarized in Figure 3.

This experiment shows write throughput for various cluster sizes; it was also repeated for a reference application with a "null" `RequestHandler` (Sirius-NoBrain) and one where disk persistence was turned off (Sirius-NoDisk). There are two main observations to make here:

1. Throughput degrades as cluster size increases. This is primarily due to the quorum-based voting that goes on in Paxos: the more cluster members there are, the more votes are needed for a quorum, and hence the greater chance that there are enough machines sporadically running "slow" (e.g., due to a garbage collection pause) to slow down the algorithm. This trend is consistent with those reported by Hunt *et al.* for ZooKeeper [10].

2. Throughput is not affected by disabling the persistence layer nor by eliminating `RequestHandler` work; we conclude that the Paxos algorithm (or our implementation of it) is the limiting factor.

[2] Kernel `3.4.73-64.112.amzn1.x86_64`.

[3] `http://aws.amazon.com/ec2/instance-types/`

[4] Java version `1.6.0_24`; OpenJDK release `IcedTea6 1.11.14`.

[5] `http://github.com/Comcast/sirius-reference-app`

[6] `http://jmeter.apache.org/`

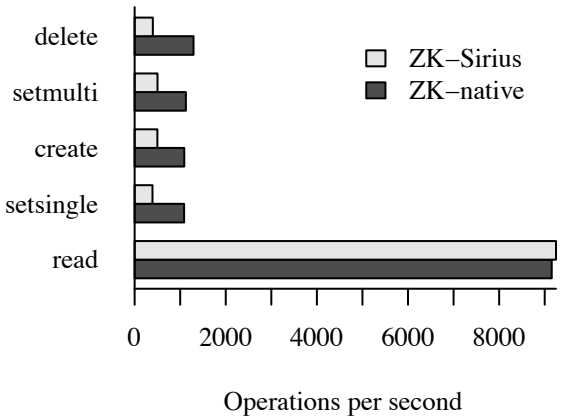

Figure 4: Comparison to ZooKeeper.

7.2 Comparison to ZooKeeper

One of the primary related technologies is ZooKeeper [10]. ZooKeeper offers similar semantics: consistent ordering of writes and eventually consistent reads. However, ZooKeeper also implements a broader set of functionality than Sirius provides, for example, allowing strongly consistent read operations.

We created a modified version of ZooKeeper where the atomic broadcast and persistence layers were replaced with Sirius. This "ZK-Sirius" only supports the core create, read, update, and delete primitives that the systems have in common, but not the full ZooKeeper API. This was sufficient to run benchmarks comparing ZK-Sirius with the native ZooKeeper using a third-party open source benchmarking tool.[7] Because we reused the ZooKeeper interface, we did not have to modify the benchmarking tool at all. The results from testing 5-node clusters are shown in Figure 4. Our prototype and all experiments were based on version 3.4.5 of ZooKeeper.

From the authors' description of the benchmark:

> "The benchmark exercises the ensemble's performance at handling znode reads, repeated writes to a single znode, znode creation, repeated writes to multiple znodes, and znode deletion.... The benchmark connects to each server in the ZooKeeper ensemble using one thread per server."

We conducted the test using the synchronous mode of operation, where each client makes a new request upon receiving the result of the previous one.

As expected, the Sirius-backed version of ZooKeeper achieves essentially identical read performance to the standard ZooKeeper implementation. Throughput on write operations—SETSINGLE, CREATE, SETMULTI and DELETE—is measured to be approximately 40–50% of standard ZooKeeper throughput. Given the read-heavy nature of our reference data workloads, this is a practical tradeoff in order to get the key benefit Sirius provides: namely, in-memory access to the data via arbitrary, native datastructures.

Our point is not to suggest an alternative implementation for ZooKeeper—in particular, Sirius does not support the synchronous reads that are possible with ZooKeeper—but rather to illustrate that our relatively untuned Sirius is in the neighborhood of a highly tuned and production-hardened system like ZooKeeper. This exercise also illustrates two major points:

1. The Sirius programming interface is simple and flexible enough to easily integrate it into ZooKeeper's internals.

2. Distributed system semantics similar to ZooKeeper's—consistently ordered writes with eventually consistent local reads—are available in the form of arbitrary, native data structures.

8 Operational Concerns

In addition to providing a convenient programming interface, we designed Sirius to be operationally friendly. This means that major errors, when they occur, should be obvious and noticeable, but also means that the system should degrade gracefully and preserve as much functionality as possible. Errors and faults are expected, and by and large Sirius manages recovery on its own, although operations staff may intervene if needed. Finally, Sirius has relatively few moving parts that nonetheless provide a lot of operational flexibility.

8.1 Bootstrapping

When a server is first brought online, either as a new cluster member or after recovering from a failure, it may be far behind its active peers. It may have a partial or empty log. While these scenarios do not happen often, they are among the most obvious (and inescapable). With Sirius, there are two fairly straightforward ways to bring such a server back up to date.

First, as mentioned in Section 6, the log is just a collection of regular files that can simply be copied from an existing peer. Second, we can spin up a server with an empty log, and it will use the catch-up protocol to fetch the entire log from a neighbor. Anecdotally, we have bootstrapped several gigabytes of log in minutes this way.

[7] `https://github.com/brownsys/zookeeper-benchmark`

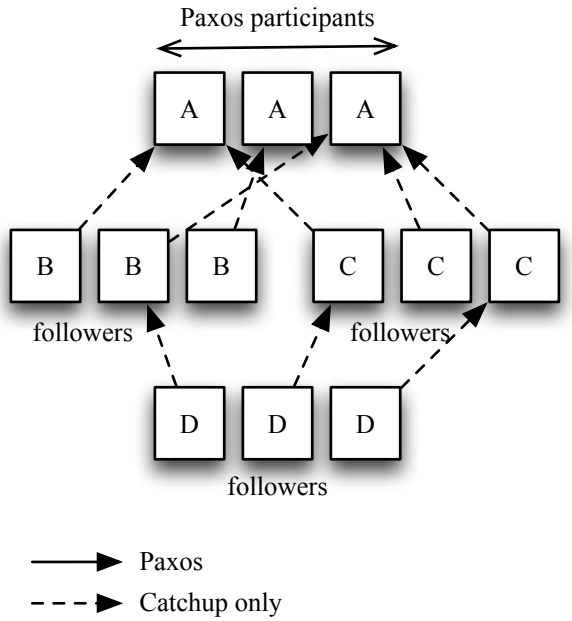

Figure 5: Flexible replication topologies with Sirius.

8.2 Follower Topology

As we described in Section 5, cluster membership is managed with a simple configuration file, with non-Paxos members using the catch-up protocol to "follow" along the primary ingest cluster. This leads to a large amount of topology flexibility; we can, and have, set up small "ingest-only" clusters to process the reference data updates, followed by multiple, larger follower clusters that are scaled out to serve reads. In fact, by specifying different membership files for different clusters, it is possible to follow multiple, redundant clusters and to set up chains of following clusters if desired.

Consider the example topology shown in Figure 5 with four clusters labeled A–D. Clusters A, B, and C share a common configuration that lists the servers of cluster A as primary members; this causes cluster A to participate in Paxos, and clusters B and C to follow the servers in cluster A. Cluster D has a configuration that lists the members of clusters B and C as primaries; this causes cluster D to catch up from randomly selected members of both clusters, even though neither of them are the primary Paxos cluster.

8.3 waltool

To support debugging and operations, we distribute the command-line *waltool* along with Sirius. Waltool primarily allows for manipulation of the Sirius log, and provides the following functionality:

- log format conversion (*e.g.*, from the original unsegmented version to the segmented version and back)
- print the last few entries in the log in human-readable format (similar to the Unix *tail* command)
- log filtering on keys via regular expression—similar to *grep*—to produce a new log that contains or excludes keys matching the expression
- offline compaction
- replay PUT/DELETEs for each update in the log as equivalent HTTP requests sent to a specified server

8.4 Error Handling

Each update in both the index and data files are checksummed. Eventually, a bit will flip, a checksum will fail, and Sirius will refuse to start. Currently, recovery is manual, albeit straightforward: Sirius reports the point at which the problematic record begins. An operator can truncate the log at this point or delete a corrupted index, and Sirius can take care of the rest, rebuilding the index or retrieving the missing updates as needed.

As mentioned above, the liveness of a server's Paxos subsystem is tied to its cluster membership. While rare, we have seen some cases of a cluster failing to make progress. Generally, these resolve themselves (temporary network partitions, DNS flaps), but occasionally they require manual intervention, such as when Paxos has livelocked due to a bug in implementation. We have found that these can almost always be addressed by "power-cycling" Paxos: that is, removing all nodes from the cluster, then putting them back, effectively restarting with a blank slate. This is done without restarting any nodes, simply by changing the monitored configuration file. Nevertheless, situations needing this kind of intervention are rare, and are becoming more rare as we hunt down the last of the bizarre edge cases.

8.5 MBean Server

If supplied with a JMX MBean server[8], Sirius will automatically register many metrics and status markers into it. These include the perceived liveness of the neighboring cluster members, the identity and ballot of the currently elected leader, the number of out-of-order events buffered for persistence, the rolling average time of persisting to disk, the duration of the latest compaction, among others.

[8] See `http://en.wikipedia.org/wiki/Java_Management_Extensions`.

9 Related Work

> "A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible." –Freeman Dyson

9.1 Consensus

As we described in Section 5, our Paxos [16] implementation follows the one described by van Renesse [26] closely—in fact, our first implementation was a fairly straightforward translation of his pseudocode into Scala/Akka. Similarly, Chandra *et al.* [5] implemented their own state machine description language in C++ in order to get a concise description of the protocol. We also use *master leases* as they describe to optimize throughput of the Multi-Paxos algorithm.

Finally, there are alternative consensus protocols such as RAFT [20], Egalitarian Paxos [19], or ZAB [10] we could have used instead of Paxos without changing the overall application architecture; we will discuss some of these in Section 10. Initially, however, we were attracted to the deep coverage of classic Paxos in the literature to guide our implementation.

9.2 Persistence

Transaction logs are a well-known mechanism for crash recovery in databases, and the Sirius log functions primarily as the *commit log* for Paxos. Unlike more general Paxos implementations like Chubby [5], we do not have to implement application state snapshots or *checkpoints* in Sirius, as the semantics encoded in the log allow us to successfully compact it without assistance from the client application. We also take advantage of the append-only nature of the log in the common-case write path to minimize write latencies to spinning disks, as in journaling file systems [22].

As in log-structured filesystems (LFS) [23], the Sirius log is the primary storage location. We similarly use *segments* as a way to partition the work of compaction, although in our case it is more about enabling faster replay at startup time and bounding the resources needed for continuous compaction than it is about reclaiming free space. Finally, we also create indices mapping sequence numbers to locations within the log, although these are kept separate from the log segments themselves and are essentially hints that can be rebuilt when needed.

Bitcask [24] is a log-structured, persistent hashtable similar in style to the Sirius write-ahead log: it segments the log, periodically merges old segments, and builds *hint files* alongside the segments to provide file addresses for the data bound to particular keys. Indeed, Sirius's log file format is very similar to Bitcask's, although we explicitly differentiate PUTs from DELETEs and also have to record our Paxos sequence numbers. Unlike Bitcask, though, we do not have to build the in-memory *keydir* version of the hint files, as Sirius does not have to provide random read access to the latest update for a particular key. Our index files facilitate, rather, finding updates by sequence number to support catchup.

Finally, LevelDB [1] is a persistent, ordered hashtable that might have handled compaction for us by tracking the most recent updates to particular keys, except that we really have a need for two indices into the set of updates: one by sequence number to support replay and catchup and one by key to support compaction.

9.3 Replicated data structures

There are libraries that provide replicated data structures–typically hashtables–such as Hazelcast [8], Gemfire [21], and Terracotta [25]. While these cover a number of important and practical use cases, they do not permit the use of arbitrary data structures for the replicated data. Although reference datasets are represented as a stream of key-value pairs with Sirius, our applications construct more complex representations in their in-memory mirrors. In particular, the reference data describing television schedules required custom data structures to support all of our use cases efficiently.

9.4 Shared External Logs

Tango [3], like Sirius, provides in-memory *views* that are backed by a shared, persistent log. However, this requires a specialized array of SSD nodes, whereas we needed Sirius to be able to run in a commodity cloud environment. In addition, Tango requires checkpointing support from the client application in order to truncate its logs. Finally, the SSD array, while highly redundant itself, is a single point of failure from the point of view of the client applications, whereas Sirius-related failures are localized to individual cluster nodes.

9.5 External storage

Another option would have been to keep the reference datasets in another system external to the application, such as memcached [6], Redis [17], or ZooKeeper [10]. However, systems like these bring a rich yet ultimately limited set of data structures and require I/O for reads, whether directly by the application programmer or via library calls. In either case, the programmer is still on the hook to provide error handling and proper I/O configuration, something that is difficult for many—if not most—developers to do correctly.

9.6 Event sourcing and message buses

The LMAX architecture [7] builds memory-resident data structures in a single master system while asynchronously replicating the master's transaction log to secondary and tertiary spares. Unlike Sirius, however, this architecture requires the application to participate in producing checkpoints as a way of eventually truncating the log. While the LMAX system posted truly impressive throughput for a single node, ultimately we needed a system that would enable us to scale read operations by adding more servers.

Another approach might have been to use a distributed message bus like Kafka [14] to distribute the updates out to all the cluster nodes. However, Kafka does make some consistency tradeoffs—including the possibility of acknowledged writes being lost [12]—to achieve high throughput. For our reference data use cases, however, we preferred the opposite tradeoff. Kafka also has a limited historical window, which means we would have had to implement checkpointing and replay for our applications anyway.

9.7 In-memory data distribution

Koszewnik [13] describes an entirely different approach to distributing reference data updates, where a single master machine periodically pulls the updates and applies them to an in-memory model. A serialization library, Zeno, then emits optimized deltas of each incremental run to well-known server locations, as well as periodically generating a full checkpoint. This allows downstream clients to individually poll for and apply deltas, and allows new server instances to bootstrap from the most recent checkpoint.

This avoids the need for running a complex consensus algorithm like Paxos, at the expense of having to manually restart the master process if it fails. On the other hand, it dictates the in-memory representation used by all the cluster members, whereas Sirius permits different applications to be part of the same Sirius cluster, sharing the same update stream while building their own customized in-memory representations of the data.

10 Conclusions and Future Work

Overall, we have been happy with Sirius; we will have been using Sirius-powered services in production for almost two years at the time of this paper's publication with few, if any, operational problems. The library's simple and transparent interface, coupled with the ease and control of using native data structures, have led multiple independent teams within Comcast to incorporate Sirius into their services, all to positive effect. Nevertheless, we have identified some opportunities for improvements.

10.1 Paxos Improvements

As with most Multi-Paxos systems, overall write throughput in Sirius is limited by the throughput of the currently-elected leader, and we do experience periodic "bulk load" events where this becomes a bottleneck, albeit a tolerable one to date. Alternative protocols such as Egalitarian Paxos [19] could alleviate this bottleneck with little change to the overall application architecture.

In addition, our cluster configuration is currently statically configured, although our implementation periodically polls its configuration file to watch for updates. Technically, this leaves a window open for inconsistency because cluster membership is not synchronized with the consensus protocol. In practice, however, we are able to pause and buffer the writes into the cluster, switch the configuration, and then resume writing. Consensus protocols like RAFT [20] that integrate cluster membership with consensus could ease our operations.

10.2 Replication

As we described earlier, our WAN replication currently piggybacks on our Paxos catch-up mechanism. Therefore, every member of our downstream non-ingest clusters pulls a copy every update across the WAN. In practice, again, this does not result in a problematic amount of bandwidth, but it is clearly inefficient. Allowing for topology-aware configuration and replication such as those found in Cassandra [15] could allow us to pull fewer (perhaps one) copy of each update across the WAN, before then further replicating locally.

10.3 Replay

In practice, since our reference datasets fit in memory, so do their representations in our write-ahead logs. This means the system read-ahead caches do a good job at the I/O required for the linear scans necessary for replay at system startup time. Still, there is a bottleneck where Sirius passes the updates synchronously and serially to the application's `RequestHandler`; an alternative mechanism for safely processing some updates in parallel would be desirable.

10.4 Conclusions

In this paper, we have described a novel architectural approach for handling application reference data centered around a new distributed system library, Sirius. A Sirius-based architecture allows for:

- in-memory, local access to the reference data in arbitrary data structures;
- eventually consistent replication of updates;
- local persistence and replay of updates;
- with a semantically-transparent library interface.

The Sirius library is available under the Apache 2 License from: `http://github.com/Comcast/sirius`.

Acknowledgments

We would like to thank the anonymous reviewers for their comments; the paper is clearer and more concise for your efforts. We would additionally like to thank the CIM Platform/API team at Comcast, Sirius' first users and development collaborators; Sirius would not have been possible without your help and hard work.

References

[1] Leveldb website. `https://code.google.com/p/leveldb/`. [Online; accessed 16-Jan-2014].

[2] AMAZON WEB SERVICES, INC. Daily Global Weather Measurements, 1929-2009 (NCDC, GSOD). `http://aws.amazon.com/datasets/2759`, 2013. [Online; accessed 8-January-2014].

[3] 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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 325–340.

[4] BOARD OF GOVERNORS OF THE FEDERAL RESERVE SYSTEM. FRB H10: Data download program. `http://www.federalreserve.gov/datadownload/Choose.aspx?rel=H10`, January 2014. [Online; accessed 8-January-2014].

[5] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[6] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004).

[7] FOWLER, M. The LMAX architecture. `http://martinfowler.com/articles/lmax.html`, July 2011. [Online; accessed 17-Jan-2014].

[8] HAZELCAST, INC. `http://hazelcast.org`, 2014. [Online; accessed 23-January-2014].

[9] HOFF, T. Latency is everywhere and it costs you sales: How to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, July 2009. *High Scalability*, blog. [Online; accessed 20-January-2014].

[10] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[11] IMDB.COM, INC. Advanced title search. `http://www.imdb.com/search/title`. [Online; accessed 24-January-2014].

[12] KINGSBURY, K. Call me maybe: Kakfa. `http://aphyr.com/posts/293-call-me-maybe-kafka`, September 2013. [Online; accessed 17-Jan-2014].

[13] KOSZEWNIK, D. Announcing Zeno—Netflix's in-memory data distribution framework. `http://techblog.netflix.com/2013/12/announcing-zeno-netflixs-in-memory-data.html`, December 2013. Netflix Tech Blog. [Online; accessed 20-January-2014].

[14] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: a distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases* (New York, NY, USA, June 2011), NetDB 2011, ACM.

[15] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

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

[17] LERNER, R. M. At the forge: Redis. *Linux Journal 2010*, 197 (Sept. 2010).

[18] MILLER, A. Hibernate query cache considered harmful? `http://tech.puredanger.com/2009/07/10/hibernate-query-cache/`, July 2009. *Pure Danger Tech*, blog. [Online; accessed 8-January-2014].

[19] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[20] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. `https://ramcloud.stanford.edu/raft.pdf`, October 2013. [Online; accessed 13-Jan-2014].

[21] PIVOTAL SOFTWARE, INC. Pivotal gemfire. `http://gopivotal.com/products/pivotal-gemfire`. [Online; accessed 16-Jan-2014].

[22] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 8–8.

[23] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[24] SHEEHY, J., AND SMITH, D. Bitcask: A log-structured hash table for fast key/value data. `http://downloads.basho.com/papers/bitcask-intro.pdf`. [Online; accessed 16-Jan-2014].

[25] TERRACOTTA, INC. *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, Berkeley, CA, USA, 2008.

[26] VAN RENESSE, R. Paxos made moderately complex. `http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf`, March 2011. [Online; accessed 13-Jan-2014].

[27] WIKIPEDIA. Encyclopædia Britannica Ultimate Reference Suite — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Encyclop%C3%A6dia_Britannica_Ultimate_Reference_Suite`, 2013. [Online; accessed 8-January-2014].