Sirius

Distributing and Coordinating Application Reference Data

MICHAEL BEVILACQUA-LINN, MAULAN BYRON, PETER CLINE, JON MOORE, AND STEVE MUIR

Michael Bevilacqua-Linn is a Distinguished Engineer at Comcast. He's worked on their next generation IP video delivery systems as an architect and engineer for the past several years, and is interested in functional programming and distributed systems. He currently resides in Philadelphia.
Michael_Bevilacqua-Linn@comcast.com

Maulan Byron is passionate about making software scalable, fast, and robust. He has spent the majority of his career building and delivering mid-size to large-scale projects in the telecommunication and financial industries. His interests are in making things scale and perform better and finding solutions that make it easier to operationalize software products.
Maulan_Byron@cable.comcast.com

Peter Cline is a senior software engineer at Comcast. In the past two years there, his work has focused on distributed systems, hypermedia API design, and automated testing. Prior to Comcast, he worked in search and digital curation at the University of Pennsylvania.
Peter_Cline2@comcast.com

Sirius is an open-source library that provides developers of applications that require reference data with a simple in-memory object model while transparently managing cluster replication and consistency. We describe the design and implementation of Sirius in the context of TV and movie metadata, but Sirius has since been used in other applications at Comcast, and it is intended to support a broad class of applications and associated reference data.

Many applications need to use *reference data*-information that is accessed frequently but not necessarily updated in-band by the application itself. Such reference data sets now fit comfortably in memory, especially as the exponential progress of Moore's Law has outstripped these data sets' growth rates: For example, the total number of feature films listed in the Internet Movie Database grew 40% from 1998 to 2013, whereas commodity server RAM grew by two orders of magnitude in the same period.

Consider the type of reference data that drove the design and implementation of *Sirius*: metadata associated with television shows and movies. 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). This data set has certain distinguishing characteristics common to reference data:

It is **small**. Our data set is a few tens of gigabytes in size, fitting comfortably in main memory of modern commodity servers.

It is relatively static, with a **very high read/write ratio.** Overwhelmingly, this data is write-once, read-frequently: *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.

It is **asynchronously updated.** End users are not directly exposed to the latency of updates, and some propagation delay is generally tolerable, e.g., in correcting a misspelling of "*Cassablanca.*" However, if a presidential press conference is suddenly called, schedules may need to be updated within minutes rather than hours.

Common service architectures separate the management of reference data from the application code that must use it, typically leveraging some form of caching to maintain low latency access. Such schemes force developers to handle complex interfaces, and thus may be difficult to use correctly.

Sirius keeps reference data entirely in RAM, providing simple access by the application, while ensuring consistency of updates in a distributed manner. Persistent logging in conjunction with consensus and "catch up" protocols provides resilience to common failure modes and automatic recovery.

Sirius has been used in production for almost two years, and it supports a number of cloud services that deliver video to Comcast customers on a variety of platforms. These services must support:

Jon Moore, a technical fellow at Comcast Corporation, runs the Core Application Platforms group, which focuses on building scalable, performant, robust software components for the company's varied software product development groups. His current interests include distributed systems, hypermedia APIs, and fault tolerance. Jon received his PhD in computer and information science from the University of Pennsylvania and currently resides in West Philadelphia.
Jonathan_Moore@comcast.com

Steve Muir is a senior director, Software Architecture, at Comcast. He works on a broad range of cloud infrastructure projects, with a specific focus on system architectures and environments that support highly scalable service delivery. He has a broad background in operating systems, networking, and telecommunications, and holds a PhD in computer science from the University of Pennsylvania.
Steve_Muir@cable.comcast.com

Multiple datacenters. We expect our services to run in multiple locations, for both geo-locality of access and resilience to datacenter failures.

Low access latency. Interactive, consumer-facing applications must have fast access to our reference data: service latencies directly impact usage and revenue [3].

Continuous delivery. Our services will be powering products that are constantly evolving. Application interactions with reference data change, and we aim to be able to rapidly deploy code updates to our production servers. Hence, easy and rapid automated testing is essential.

Robustness. In production we expect to experience a variety of failure conditions: server crashes, network partitions, and failures of our own service dependencies. The application service must continue operating—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. Sirius must 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.

This article describes how the design and implementation of Sirius satisfies these requirements. Further technical details, additional performance evaluation results, and in-depth consideration of related work are covered in the associated full-length paper [1].

Approach

As we have seen, our reference data set fits comfortably in RAM, so we take the approach of keeping a complete copy of the data (or a subset) on each application server, stored in-process as native data structures. This offers developers ultimate convenience:

- No I/O calls are needed to access externally stored data, and thus there is no need to handle network I/O exceptions.
- Automated testing and profiling involving the reference data only requires direct interaction with "plain old Java objects."
- Developers have full freedom to choose data structures directly suited to the application's use cases.
- There are no "cache misses" since the entire data set is present; access is fast and predictable.

Of course, this approach raises several important questions in practice. How do we keep each mirror up-to-date? How do we restore the mirrors after an application server restarts or fails?

Update Publishing

We assume that external systems manage the reference data set and push updates to our server, rather than having our server poll the system of record for updates. This event-driven approach is straightforward to implement in our application; we update the native data structures in our mirror while continually serving client requests. We model this interface after HTTP as a series of PUTs and DELETEs against various URL keys.

Replication and Consistency

To run a cluster of application servers, we need to apply the updates at every server. The system of record pushes updates through a load balancer, primarily to isolate it from individual server failures, and members of the cluster are responsible for disseminating those updates to their peers in a consistent manner.

The CAP theorem dictates that in the event of a network partition, we will need to decide between availability and consistency. 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 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 [2] protocol to agree on a consistent total ordering of updates and then have each server apply the updates in order. A general consensus protocol also allows us to consistently order updates from multiple systems of record. We provide more detail in the section on Replication.

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.

Library Structure

Finally, Sirius is structured as a library that handles the Paxos implementation, persistence, log compaction, and replay. The hosting application is then conceptually separated into two pieces (Figure 1); its external interface and business logic, and its mirror of the reference data. This approach allows full flexibility over data structures while still offering an easy-to-understand interface to the developer.

Programming Interface

As we just described, Sirius' library structure divides an application into two parts, with Sirius as an intermediary. The application provides its own interface: for example, exposing HTTP endpoints to receive the reference data updates. 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 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.

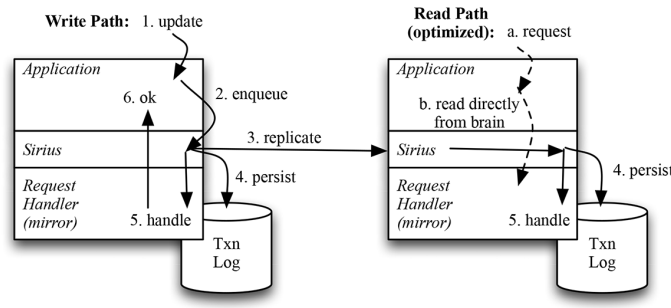

Figure 1: Architecture of a Sirius-based application

This semantic transparency makes it easy to reason functionally about the reference data itself.

The primary Sirius interface methods are all asynchronous; the Sirius library invokes request handlers in such a way as to provide eventual 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 accept new updates or 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. Indeed, in practice, many applications do not use Sirius on their read path, instead reading directly from concurrent data structures in the mirror.

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. Our implementation fairly faithfully follows the description given by van Renesse [9], with some slight differences:

Stable leader. First, we use the common optimization that disallows continuous "weak" leader elections; 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 if not acknowledged. In turn, that assumption means that we do not need to store and recover the internal Paxos state on failures.

```
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.

Similarly, we bound some processes, specifically achieving a quorum on the assignment of an update to a particular slot number, with timeouts and limited retries. During a long-lived network partition, a minority partition will not be able to make progress; this limits the amount of state accumulated for incomplete writes. Sirius thus degrades gracefully, with no impact on read operations for those nodes, even though their reference data sets in memory may begin to become stale.

Write behind. Nodes apply updates ("decisions" in Paxos terminology) in order by sequence number, buffering any out-of-order decisions as needed. Updates are acknowledged once a decision has been received, but without waiting for persistence or application to complete; this reduces system write latency and prevents "head-of-line" blocking.

However, this means that there is a window during which an acknowledged write can be lost without having been written to stable storage. In practice, since Sirius is not the system of record for the reference data set, it is possible to reconstruct lost writes by republishing the relevant updates.

Catch-up Protocol

Because updates must be applied in the same order on all nodes, and updates are logged to disk in that order, nodes are particularly susceptible to lost decision messages, which 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 the section on Persistence). This process continues until no further updates need to be transmitted.

The catch-up protocol also supports auxiliary cluster members that do not participate in Paxos. Primary cluster members know about each other and participate in the consensus protocol for updates. Secondary cluster members periodically receive updates from primary nodes using the catch-up protocol. In practice, this allows a primary "ingest" cluster to disseminate update to dependent application clusters, often within seconds of each other and across datacenters.

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.

This leads to a large amount of topology flexibility: Figure 3 shows how four clusters A–D can be given different configuration files in order to control distribution of updates. Only A participates in the Paxos algorithm, while B and C directly follow A, and D follows both B and C.

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). This creates 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.

However, there is one exception to this sequential access pattern: While responding to catch-up requests, we need to find updates no longer cached, 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.

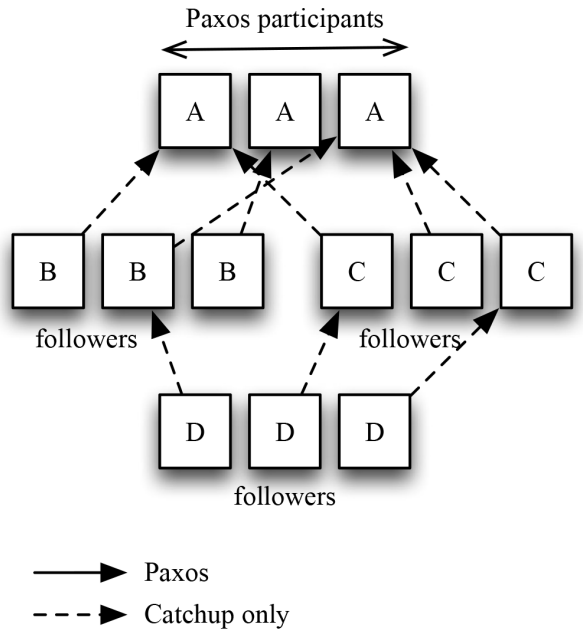

Figure 3: Flexible replication topologies with Sirius

This is accomplished by creating an index structure during the process of reading the update log at startup time. The index is small enough to be stored in memory and can thus be randomly accessed in an efficient manner, permitting use of binary search to locate a particular update by sequence number.

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 data set does not grow dramatically in size over time, a compacted log is a relatively compact representation of it; we find that the reference data set 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 [2].

Early production deployments of Sirius took advantage of rolling application restarts as part of continuous development to incorporate offline compaction of the persistent log. However, frequent restarts were required to prevent the log from getting unwieldy.

Therefore, we developed a scheme for live compaction that Sirius manages in the background. The log is divided into segments with a bounded number of entries, as in other log-based systems [7, 8]. Sirius appends updates to the highest-numbered segment; when that segment fills up, its file is closed and a new segment is started.

Compaction is accomplished by using the most recent log segment to create a set of "live" key-value pairs and deleted keys. Prior log segments can then be pruned by removing updates that would be subsequently invalidated, while updates to other keys are added to the live set. After compaction of an individual segment, the system combines adjacent segments when doing so does not exceed the maximum segment size.

Live compaction in Sirius is thus incremental and restartable and does not require a manual operational maintenance step with a separate tool. Since the logs are normal append-only files, and compaction is incremental, 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 data set elsewhere for debugging or testing.

Experimental Evaluation

The optimized read path for an application bypasses Sirius to access reference data directly, so we are primarily interested in measuring write performance. In practice Sirius provides sufficient write throughput to support our reference data use cases, but here we present experimental analysis.

The Sirius 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 on *m1.xlarge* instances, each with four virtual CPUs and 15 GB RAM. These instances have a 64-bit OpenJDK Java runtime installed; Sirius-based tests use version 1.1.4 of the library.

Write Throughput

For these tests, we embed Sirius in a reference Web application 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 version 2.11. All requests generate PUTs with 179 byte values (the average object size we see in production use).

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 4.

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:

Throughput degrades as cluster size increases, primarily due to the quorum-based voting that goes on in Paxos: In larger clusters there is a greater chance that enough machines are sporadically running "slow" (e.g., due to a garbage collection pause) to slow

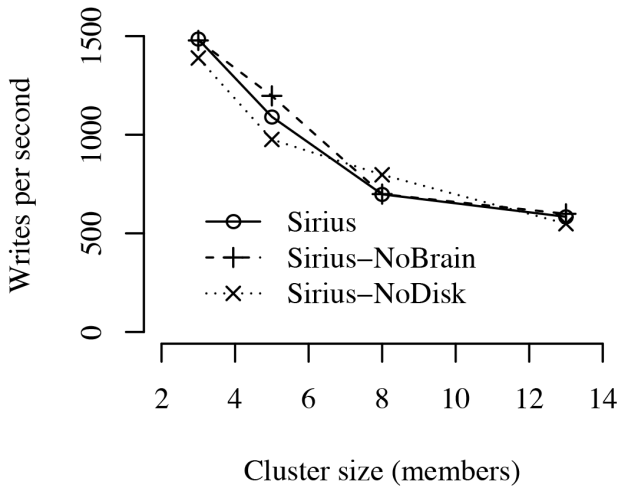

Figure 4: Sirius write throughput

down the consensus algorithm, as reported by Hunt et al. for ZooKeeper [4].

Throughput is not affected by disabling the persistence layer or by eliminating RequestHandler work; we conclude that the Paxos algorithm (or our implementation of it) is the limiting factor.

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 it 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; however, operations staff may intervene if needed.

For example, operational intervention is helpful but not required in bringing a server online, either as a new cluster member or after recovering from a failure. The server may be far behind its active peers, and may have a partial or empty log. The catch-up protocol can be used to fetch the entire log, if necessary, from a peer, which can be accomplished in a few minutes for several gigabytes of log. However, operators can accelerate the process by copying an active node's log files, thus "seeding" the new server's state.

To support debugging and operations, we distribute a command-line tool along with Sirius. This tool reads and manipulates the Sirius log, providing functionality, including: format conversion, pretty printing log entries, searching via regular expression, and offline compaction. It also allows updates to be replayed as HTTP requests sent to a specific server.

Updates in the index and data files are checksummed. When corruption occurs and is detected 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.

Conclusions and Future Work

Sirius has been deployed in support of production services for approximately two years, with very few operational issues. 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.

- The current Multi-Paxos-based consensus protocol limits write throughput to the capacity of the current leader; this could be alleviated by an alternative protocol, such as Egalitarian Paxos [5].
- Cluster membership updates are not synchronized with the consensus protocol. Consensus protocols like RAFT [6] that integrate cluster membership with consensus could simplify operations.
- WAN replication currently piggybacks on our cluster catch-up protocol, requiring one-to-one transfers of updates. A topology-aware distribution of updates could be beneficial in reducing bandwidth usage.
- Replaying write-ahead logs synchronously and serially at startup takes a significant time; hence a mechanism for safely processing some updates in parallel is desirable.

All things considered, the design and implementation of Sirius has been very successful. We would like to acknowledge 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.

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

References

[1] M. Bevilacqua-Linn, M. Byron, P. Cline, J. Moore, and S. Muir, "Sirius: Distributing and Coordinating Application Reference Data," in *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX Association.

[2] T. D. Chandra, R. Griesemer, and J. Redstone, "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.

[3] T. Hoff, "Latency Is Everywhere and It Costs You Sales—How to Crush It," *High Scalability* blog, July 2009: http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, accessed January 20, 2014.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association, pp. 145–158.

[5] I. Moraru, D. G. Andersen, and M. Kaminsky, "There Is More Consensus in Egalitarian Parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358-372.

[6] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," October 2013: https://ramcloud.stanford.edu/raft.pdf, accessed January 13, 2014.

[7] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.,* vol. 10, no. 1 (Feb. 1992), 26–52.

[8] J. Sheehy, and D. Smith, "Bitcask: A Log-Structured Hash Table for Fast Key/Value Data": http://downloads.basho.com/papers/bitcask-intro.pdf, accessed January 16, 2014.

[9] R. van Renesse, "Paxos Made Moderately Complex": http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf, March 2011, accessed January 13, 2014..