MUSIC: Multi-Site Critical Sections over Geo-Distributed State

Bharath Balasubramanian*, Pamela Zave†, Richard Schlichting‡, Mohammad Salehe§,
Shankaranarayanan Puzhavakath Narayanan*, Seyed Hossein Mortazavi§,
Eyal De Lara§, Matti Hiltunen*, Kaustubh Joshi* and Gueyoung Jung*
*AT&T Labs Research, †Princeton University, ‡U.S. Naval Academy, §University of Toronto

Abstract— **A crucial requirement for many multi-site production services operating at global scale is the need for exclusive access to latest state. Here, a novel approach to address these requirements through the abstraction of a *critical section over geo-distributed state* is proposed. This abstraction is realized in a key-value store called MUSIC, which provides critical sections with novel semantics suitable for geo-distributed state referred to as *entry consistency under failures (ECF)*. The semantics of ECF in MUSIC, its formal verification, and its implementation are presented, along with details of how MUSIC has been used to realize various fundamental geo-distributed structuring paradigms. MUSIC has been deployed in production geo-distributed services at AT&T as part of the Open Network Automation Platform (ONAP). Our evaluation of MUSIC shows that, despite providing additional properties, MUSIC has higher throughput (∼1.4-17.17 times) than Zookeeper for larger critical section sizes and outperforms (∼2-4 times) similar structures in which state updates use Paxos or CockroachDB transactions.**

Keywords-**Distributed key-value stores, Distributed fault Tolerance, Geo-distributed services.**

I. Introduction

Geo-distributed services are critical for realizing new functionality that requires global scale. As such services become more complex, however, the demands on the store that manages service state across replicas also become more complex, with an increasing need to provide abstractions and semantics more tailored to the service requirements. While designing multi-site[1] use-cases for job schedulers and active replication in AT&T's next generation network control plane, we identified two crucial requirements. First, each client request must be processed *exclusively* by one service replica and only that replica can update service state corresponding to that request. Second, if a service replica fails while processing a client request, another service replica should continue to process the request from the *latest state*. We address these requirements through a key-value store called MUSIC (MUlti-SIte Critical Sections) that provides a familiar abstraction—critical sections over shared keys—but with novel underlying semantics referred to as *entry consistency under failures (ECF)*.

The requirements outlined above were first identified from a production use case for a multi-site job scheduler in the *homing service* of AT&T's network control plane. The role of this service is to determine suitable sites ("homes") on which to deploy complex virtual network functions (VNFs) [1] using optimization techniques. While a homing request (job) is submitted to the closest scheduler replica (worker), the actual homing may be performed by any idle replica. Homing is a complex and time-consuming process. For example, analysis of a week of production logs show that the mean and 99th percentile latency just to query geo-distributed cloud controllers and identify candidate cloud sites was 7 and 15 minutes, respectively. Hence, to minimize work duplication, each request should be processed from its latest state exclusively by one scheduler replica, despite failures.

Two fundamental challenges make it difficult to fulfill the above requirements. First, a replica that is presumed failed might try to update the state of a request that is now being processed by another service replica. This is a result of imperfect failure detection, which is not uncommon in geo-distributed services given the enhanced chance of network partitions [2], [3]. Second, a new replica that takes over request processing from a failed replica may not have access to the latest state. This can often happen when state is not updated at all replicas due to wide-area-network (WAN) latencies on the order of hundreds of milliseconds.

MUSIC addresses these challenges by providing the abstraction of a critical section with novel ECF semantics. Entry consistency, as originally defined for *failure-free* shared memory systems [4], [5], specifies that data shared among multiple processors becomes sequentially consistent at a processor only when this processor acquires a synchronization object (e.g., a lock) that guards the data. We significantly re-purpose these semantics to provide a replicated key-value store, where to modify the value of keys, a client has to acquire first a unique lock to the keys. On acquiring a lock to a set of keys, the lockholder enters a *critical section* and can read the *latest value* of the keys and perform *exclusive, sequentially consistent updates* to the keys. Crucially, MUSIC provides these guarantees despite: (a) lockholders that can fail in the middle of a write operation, and (b) prior lockholders—detected as failed incorrectly—that may attempt to modify shared keys.

The guarantees provided by MUSIC enable service developers to write programs for geo-distributed services with the familiarity with which they write multi-threaded concurrent programs, *despite* the complex failure modes associated with geo-distribution. In §VII, we describe how MUSIC is used to realize the VNF homing service described above and to

[1] A site is a data center at a physical location connected with other sites through a WAN.

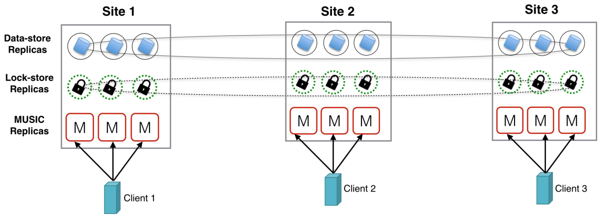


Fig. 1: The MUSIC architecture, where clients access a nearby MUSIC replica that in turns communicates with data/lock-store replicas. In our production deployment, since we use Cassandra for both the lock- and data-store, we have a 9-replica MUSIC cluster interacting with a 9-node Cassandra cluster with redundancy both within a site and across sites.

provide active replication with failover for a Management Portal Service [6]. MUSIC and these services are based on contributions from multiple companies, and are part of the open-source Open Network Automation Platform (ONAP) [7]. They are currently running in AT&T production deployments.

MUSIC is designed as a combination of a data store that maintains client key-value pairs and a lock store that is used for locking primitives (Figure 1). Through the novel use of vector timestamps and selective state synchronization, we guarantee ECF semantics in a performant manner. While the use of distributed consensus for reads and writes in a critical section would have made it easier to prove the correctness of MUSIC algorithms, we show that ECF can be guaranteed even when implemented through quorum operations in an eventually-consistent store. While both options have similar message complexity, this choice makes MUSIC more efficient in practice (see §VIII). Given the subtle reasoning required due to failures and more importantly imperfect failure detection, we model MUSIC semantics in Alloy [8] and verify its algorithms using the Alloy Analyzer [9].

MUSIC is implemented as a layer on top of Cassandra [10], which is used for both the data- and lock-store. We make novel use of Cassandra's light-weight transactions or LWTs (Paxos-based compare-and-set [11]) to implement locking primitives. MUSIC's evaluation across WAN latency profiles shows that, despite its additional properties (see §II), MUSIC outperforms ($\sim$1.4-17.17 times) Zookeeper for larger critical section sizes. Moreover, MUSIC outperforms ($\sim$2-4 times) critical sections with identical guarantees that use LWTs or CockroachDB [12] transactions for each state update.

In summary, this paper presents these contributions:

- The MUSIC key-value store, which provides novel ECF semantics presented to programmers as a critical section abstraction suitable for geo-distributed services (§III).
- A formally verified algorithm design (§IV – §V).
- An implementation [13] that is being used in AT&T's production network and is open-sourced through ONAP (§VI,§VII).
- Experiments validating MUSIC's effectiveness (§VIII).

This work builds on a brief announcement that focused only on the basic ideas and abstractions [14].

II. Related Work

Entry consistency has been widely used in shared-memory, multi-core systems [4], [5], where updates to memory and cache are tracked in an efficient manner so that a process acquiring a lock to, say, a page of memory has access to the latest version of the data in that page. Core failures are not usually considered an issue in such systems. In this paper, we re-purpose these semantics for geo-distributed services with failures and imperfect failure detection. Re-purposing similar models such as release consistency [5] for geo-distributed services is an avenue of future work.

Existing tools either do not provide the right abstractions to address our requirements or lead to prohibitively expensive designs. Zookeeper, Consul and etcd [15]–[17] facilitate implementation of the replicated state machine approach [18] by providing sequentially-consistent reads and writes to file system nodes/keys. Further, these tools use protocols like Zab [19] or multi-Paxos to optimize performance by electing a stable leader. MUSIC provides the higher-level abstraction of critical sections, where lockholders are not only guaranteed sequential consistency, but are also guaranteed to read the latest value since no other client can be writing a new value while the lockholder is reading the current value. In addition, by using the lockholder as a stable leader, MUSIC amortizes the cost of multiple operations in a critical section (see §IV).

This argument with respect to sequentially-consistent tools naturally extends to tools that provide weaker consistency such as COPS for causal consistency [20], [21]. However, our lock-store design can benefit from any research that reduces the cost of sequential consistency through the use of mixed consistency [22], workload management [23] or WAN-aware designs [24], [25]. Other systems such as Atomix [26] extend Cassandra's per-key LWTs [11] to atomic maps, linked lists, and other data structures. While valuable, our experience is that it is useful to have a more general control structure such as critical sections for building production geo-distributed services. Note that this abstraction can then be used to build atomic data structures as needed.

Standalone locking services provided by Chubby, Curator, or Zookeeper recipes [27]–[29] implement distributed mutual exclusion [30], [31] and offer a lock abstraction similar to MUSIC. However, these services are oriented primarily towards providing coarse-grain locks for occasional synchronization, such as might be used for leader election. Moreover, while these tools provide a file system abstraction, they are intended for storing small amounts of metadata rather than as a general data store. As a result, these locking services are often used in conjunction with an external data store that is expected to enforce consistency as opposed to the locking service itself. MUSIC provides fine-grained locks as part of a general key-value store that can support arbitrary amounts of data storage. The locking exists to enforce specific consistency guarantees for that key-value store, and indeed, the primary goal of MUSIC locks is to serve as the object that triggers the synchronization needed to provide ECF.

Transactions with optimistic concurrency control [32]–[34] allow multiple processes to execute concurrent transactions on the same keys/table and use conflict resolution and/or multi-version concurrency control to ensure ACID semantics. Such transactions do not satisfy our *exclusivity* requirement. For example, in the homing use-case from §I, if each worker performs homing in an independent transaction, multiple workers may home the same job, which is prohibitively expensive. On the other hand if a worker performs *all* updates to service state in an ACID transaction with exclusivity guarantees, then replica failure during the transaction will rollback all the state updates, necessitating a complex rollback of the homing process and further, violating our *latest state* requirement.

While our requirements can be addressed by performing each state update in an exclusive transaction, this is prohibitively expensive, even for highly optimized geo-distributed databases (DB) like Spanner [35] or its open-source version, CockroachDB [12]. This solution requires distributed consensus to begin and end each transaction (see §4.2.1 in [35]), each of which involves a single local state update. MUSIC uses distributed consensus to enter and exit a critical section in which there are multiple quorum state updates. We show through basic cost analysis in §X-B4 and experiments (§VIII) that a MUSIC critical section executes ∼2-4 times faster than the solution above.

III. Semantics

In this section, we describe MUSIC's critical section abstraction and the guarantees it provides under failures. Our system model assumes distributed nodes that communicate using messages (that can be lost or re-ordered). To overcome the impossibility of distributed consensus in asynchronous systems [36], we assume partial synchrony [37], [38] where there are sufficient periods of communication synchrony with an upper bound on message delay. Nodes can suffer crash failures [39], which implies that other nodes cannot distinguish between a failed node and one that is slow to respond and/or unable to communicate. This is relatively common in geo-distributed systems where link failures [2], [3] can partition a node from some subset of other nodes.

MUSIC is implemented by a collection of replicas, where clients issue requests to the MUSIC replicas and the MUSIC replicas in turn issue requests to back-end *data-store* and *lock-store* replicas (see Figure 1). Clients use MUSIC by invoking the operations listed in Table I using a non-blocking request to a MUSIC replica of its choice.[2] The replica then executes a single-threaded sequence of steps, including requests to back-end stores, and reports success or failure to the client.

In this section, we first describe the ECF semantics as provided to MUSIC clients and then the semantics of the back-end stores as provided to MUSIC.

lockRef = createLockRef (key)	Enqueues a per-key unique increasing identifier that is good for one critical section.
result = acquireLock (key, lockRef)	Returns true if *lockRef* is first in the queue.
criticalPut (key, lockRef, value)	Writes the latest value of a key for the current lock holder.
value = criticalGet (key, lockRef)	Reads the latest value of a key for the current lock holder.
releaseLock (key, lockRef)	Removes *lockRef* from the queue and releases the lock.

TABLE I: MUSIC Operations.

A. MUSIC ECF Semantics

We present ECF semantics by describing the details of MUSIC operations, which are typically used as shown in Listing 1. Note that this code has been simplified by omitting the code for handing failures; these considerations are addressed in detail when we discuss failure semantics later in this section.

Listing 1: Example use of MUSIC by a client

```
lockRef = createLockRef(key);
while(acquireLock(key, lockRef)!=true) skip;
// enter critical section
v1=criticalGet(key, lockRef);
//v1 is guaranteed to be the true value of the key
v2=v1+1;
criticalPut (key, lockRef, v2);
//v2 is guaranteed to be the true value of the key
releaseLock(key, lockRef);
// exit critical section
```

Locking. To read or write to a key, a client must first acquire the lock to the key, which is granted fairly. Specifically, the client executes *createLockRef*, which returns a *lockRef*—a unique, increasing identifier for the key, used to authenticate the client as it makes its critical requests. A *lockRef* is good for a single execution of a critical section. For each key, MUSIC maintains a sequentially-consistent queue of pending *lockRefs*, in request order. To actually acquire the lock, the client polls by executing *acquireLock* until it returns true, meaning that this client's *lockRef* is first in the queue. Standard back-off mechanisms can be used to alleviate the cost of polling and contention across clients trying to acquire a lock to the same key. Our choice of separate operations of *createLockRef* and *acquireLock* is important—as we show in §IV, the former operation requires the use of distributed consensus while the latter, which is called several times by clients, is a purely local operation for most cases.

Critical section abstractions. On acquiring the lock, a client enters a critical section in which it can read and write the value of the key through a sequence of *criticalGet* and *criticalPut*[3]

[2] A blue font is used to highlight the subsequent use of MUSIC operations in the text.

[3] While the *criticalPut* has a corresponding delete function, we omit its description for simplicity.

operations, respectively.

ECF semantics as provided by MUSIC is a combination of the following two definitions and two properties.

Definition. *The* lockholding client *(or* lockholder*) is the client holding the* lockRef *that is first in the sequentially-consistent queue, if any.*

As described in §III-B, the first *lockRef* in the queue refers to the first *lockRef* at a quorum of the lock-store replicas. In this paper, a quorum is defined in the usual way as a majority of the replicas.

Exclusivity Property. *Only the* lockholder *can perform successful* criticalPut *or* criticalGet *operations on a key.*

Definition. *The* true value *of a key is the value of the most recent, successfully acknowledged,* criticalPut *performed by a lockholder.*

If a *criticalPut* succeeds in all respects except that the acknowledgment is lost on the way to the client, it has not succeeded. This is important because a non-failed client is obligated to retry until it does receive acknowledgment.

Latest-State Property. *If the lockholder requests a* criticalGet *operation and receives a value in response, the value it receives is the* true value *of the key.*

The paragraphs on lock release and failure resilience below explain the assumed behavior of clients when failures in MUSIC or its back-end stores occur, and the behavior of MUSIC replicas when client or back-end failures occur; these will entail a refinement of the definition of *true value.* These assumptions and refinements are encoded in the formal model (§V), which has been used to verify the above properties.

Lock release. When the lockholder is finished with a critical section, it should invoke the function *releaseLock* to make the lock available to other clients. Clearly, the client may fail while holding the lock. To fix this problem, any MUSIC replica can preempt the lock from a lockholder that appears to have failed, using time-outs for failure detection. This fix creates another problem, that of false failure detection (usually because of a network partition). If a lockholder is still alive but has had its lock preempted, its request for critical functions will be rejected by the MUSIC replica that receives the request with a notification that the client is no longer the lockholder. Further, the *Exclusivity Property* ensures that the *criticalPuts* of a preempted client will not compromise the data store.

Failure Semantics. A client can execute MUSIC operations on a key if it can reach any non-failed MUSIC replica. However, to perform a successful operation, that MUSIC replica must be able to reach a quorum of non-failed back-end replicas. If a MUSIC replica makes a request to back-end replicas and receives too many nacks or missing responses because a replica has failed or is slow in replying, then it returns a nack to the client. In this case, the client has to retry the function—usually at a different MUSIC replica—until the operation succeeds, the client fails, or the client is told it is no longer the lockholder. If the client does not receive a response after these retries, it must not attempt any other MUSIC operation on the key; in this case, the client can simply exit the code and attempt to modify the key in a new critical section if desired. We assume that there are enough replicas of each type, and that failures are infrequent enough that all proper requests eventually succeed. While omitted from Listing 1, each MUSIC call would incorporate these steps.

What if a lockholder fails or is forcibly preempted while it is waiting for completion and acknowledgment of a *criticalPut*? This one case requires a refinement of the definition of *true value* above. In this case, the *true value* is either the value of the most recent acknowledged write (the definition above) or it is the value of the write being attempted by the previous lockholder when it was preempted. The choice of which is non-deterministic, but the system commits to a choice and ensures that the true value is present in at least a quorum of data-store replicas before the next lockholder enters its critical section.

We only consider MUSIC operations on a single key here since our use-cases do not currently require locks over multiple keys. The semantics can easily be extended by following the deadlock-avoidance rule that locks are always acquired in lexicographic order, and an *acquireLock* on multiple keys is successful only if it is individually successful for all the keys in the key set. More efficient mechanisms are an important avenue of future work.

B. Back-end Store Semantics

MUSIC uses a *data store* that maintains key-value pairs created by the client, with standard eventual consistency semantics [40], but with novel use of *vector timestamps* for ordering. Specifically, the value of a key in any replica of the data store has a (*lockRef, time*) vector timestamp associated with it that reflects the *lockRef* and real time of the last write received at that particular replica. This write could have been either part of a *criticalPut* in the *lockRef*'s critical section, or propagated by another data-store replica. The domains of both the *lockRef* and time are ordered, and *lockRef*s are more significant in the comparison. In the definition of *true value* in §III, "most recent" refers to a comparison of vector timestamps; we define the *true timestamp* as the winning "most recent" timestamp of the write. We rely on local clocks *only* to sequentialize multiple actions of a single client.

The data store provides to MUSIC a *dsPutQuorum (key, value, (lockRef, time))* function, which attempts to update the key and its timestamp at a quorum of the data-store replicas. The data store also provides a *dsGetQuorum (key)* function, which communicates with a quorum of data-store replicas and returns their latest value of the key. A write to a data-store replica eventually propagates to all other replicas, where it is accepted only if the vector timestamp of the write is greater than the current timestamp at that replica.

MUSIC uses a *lock store* with standard sequential consistency semantics [41] to create and store the queue of *lockRefs* for each key. The *lockRef* queue is updated through the functions *lsGenerateAndEnqueue (key)*, which atomically generates a per-key unique increasing identifier and enqueues it in the lock store, and *lsDequeue (key, lockRef)*. All writes to the queue are totally ordered, and the write order is determined by a consensus protocol [28], [42] that to succeed needs to update at least a quorum of lock-store replicas. All writes eventually propagate to all other replicas. There is also a function *lsPeek (key)* returning the *lockRef* that is first in the queue of the *local replica* of the lock store, i.e., any replica executing at the same geographic site.

IV. Algorithms

A. Execution in Failure-Free Scenarios

In this section we present the algorithms executed by each MUSIC replica to implement ECF semantics. We first explain how and why they work in failure-free scenarios. Subsequent sections will then explain the more complex scenarios. In each algorithm, we specify its dominating cost with respect to whether it is a quorum operation, or incurs distributed consensus, on large or small amounts of data. For initialization, we assume all keys and their lock queues have already been created in the data and lock store, respectively.

In *createLockRef*, MUSIC simply calls the lock-store function *lsGenerateAndEnqueue* to obtain a lock reference. Although *createLockRef* incurs the cost of distributed consensus, it is executed only once for each critical section.

```
createLockRef (key)  [cost: lockRef consensus write]
lockRef = lsGenerateAndEnqueue (key);
return (lockRef);
```

As described in §III, after enqueueing a *lockRef* the client invokes *acquireLock* until its *lockRef* is first in the queue. The code for *acquireLock* begins with a check of the local replica of the lockstore. If the *lockRef* is greater than (later than) the first entry in its queue, MUSIC returns failure to the client, and the client tries *acquireLock* again. The *lockRef* may in fact be first but the local replica has not been updated yet, in which case retry is also the right thing to do. Checking a local replica is efficient, which is important because there may be several executions of *acquireLock* for each critical section. In failure-free scenarios, if the *lockRef* is first in the queue, then the code directly skips to the last line and returns true.

```
acquireLock (key, lockRef)  [cost: synchFlag quorum read]
 if (lockRef > lsPeek (key))
     //lockRef not first yet, or local store not yet updated
     return (false);
 if (lockRef < lsPeek (key)) //lock forcibly released
     return (youAreNoLongerLockHolder);
 if (dsGetQuorum(synchFlag) == true)
     [cost, only after forced release: value quorum read and
            write, synchFlag quorum write]
     value = dsGetQuorum (key);
     dsPutQuorum (key, value, (lockRef, time));
     dsPutQuorum(synchFlag, false, (lockRef, time));
 //data store defined as true value for key
return (true);
```

To update the data store, a lockholding client requests *criticalPut*. In failure-free scenarios, only the last two statements of the algorithm matter. MUSIC writes the new value in a quorum of data-store replicas, then acknowledges to the client a successful put. The code for *criticalGet* is similar, with the quorum write replaced with a quorum read of the data-store.

```
criticalPut (key, lockRef, value)  [cost: value quorum write]
 if (lockRef > lsPeek (key))
     //lockRef not first yet, or local store not yet updated
     return (false);
 if (lockRef < lsPeek (key)) //lock forcibly released
     return (youAreNoLongerLockHolder);
dsPutQuorum (key, value, (lockRef, currentTime));
return (true);
```

Finally, when a client is finished with its critical section, it releases the lock. In failure-free scenarios, this simply removes its *lockRef* from the lock queue.

```
releaseLock (key, lockRef)  [cost: lockRef consensus write]
 if (lockRef < lsPeek (key))
     return (true); //lock has been forcibly released
lsDequeue (key, lockRef);
return (true);
```

The proof of the *Latest-State Property* (§III-A) is based on the following definition and invariant.

Definition. *The data store is* defined as value v *if fewer than a quorum of data-store replicas hold a value that is not* v.

This is similar to saying that at least a quorum of replicas have the value *v*, but more precise because it allows for replica failures. When a lockholding client is waiting for a *criticalPut* to succeed, waiting for a *criticalGet* to succeed, or in a critical section but not accessing the data store, it is in the states *Putting, Getting,* or *Critical*, respectively. We have verified that the following property is always true:

Critical-Section Invariant. *If the lockholding client is in a* Critical *or* Getting *state, then the data store is* defined as the true value.

It is easy to see from the invariant that the quorum read in *criticalGet* must return the true value.

From the algorithms presented in this section it is clear that, in failure-free scenarios, a MUSIC critical section essentially requires two consensus writes to a small *lockRef*, one quorum read of a small *synchFlag*, and a quorum operation for each state update using a *criticalPut*. §VIII shows that the common pattern of multiple state updates in a critical section effectively amortizes the cost of entry and exit.

B. Failures

a) Lockholder Failure: A MUSIC replica can diagnose failure of the current lockholder and preempt its lock by executing *forcedRelease*, which is an internal function not exposed to clients.

```
forcedRelease (key, lockRef)
[cost: lockRef consensus write, synchFlag quorum write]
 if (lockRef < lsPeek (key))
     return (true); // lock was previously released
dsPutQuorum (synchFlag, false, (lockRef+$\delta$, time));
lsDequeue (key, lockRef); // no−op if lockRef not in queue
return (true);
```

The lockholder may fail and be preempted while it is attempting a *criticalPut*. When this happens, the state of the data store is unknown; before a new lockholder can enter a critical section, the data store must be synchronized, meaning that it is once again defined as the true value. To do this, MUSIC uses a *synchFlag* for each key indicating when the data store must be synched (like a "dirty bit"). This flag is implemented in the data store with vector timestamps, and accessed through quorum reads/writes. We distinguish the cost of these quorum operations from the cost of quorum operations on the key's value, because the latter is usually far larger.

To synchronize, in *acquireLock*, MUSIC first does a quorum read of the data store. If there was an incomplete *criticalPut,* then this read may or may not catch the updated value. Either way, the result of the read is re-written into the data store with a new timestamp carrying the new lockholder's *lockRef.* This is how the nondeterminism in §III, requiring an exception in the definition of the true value, is resolved. After the quorum write, MUSIC resets the *synchFlag* with the new *lockRef* in its timestamp, eventually over-writing all values with past *lockRefs* in their timestamps.

What if *acquireLock* and *forcedRelease* with the same *lockRef* are racing to write the value of the *synchFlag*? *forcedRelease* sets the flag using a value in the *lockRef* field of the timestamp that is strictly higher than the lock reference it is releasing by a small number δ. This δ needs to be greater than zero so that it over-writes the concurrent *synchFlag* reset with the same *lockRef*, but small enough to be over-written by a *synchFlag* reset with the next *lockRef* in the queue. (In our production deployment we chose δ to be 1 microsecond). Note that setting the *synchFlag* in *forcedRelease* cannot race with reading the *synchFlag* in the next successful *acquireLock*, because the quorum write is completed before the last *lockRef* is dequeued.

What if a lockholder releases the lock, and subsequently some MUSIC replica thinks its *lockRef* still holds the lock and executes *forcedRelease* on it? In this case the *synchFlag* might be erroneously true, but the only consequence of this error is that the next *acquireLock* will synchronize the data store when it is not necessary.

It may be that a client requests *createLockRef* and then dies before receiving the reply or before acquiring the lock. In this case there will be a *lockRef* in the lock queue with no client as owner. When the orphan *lockRef* becomes first in the queue, it will be removed by *forcedRelease*.

b) False Failure Detection: Due to delayed packets and/or network partitions, a MUSIC replica might execute a *forcedRelease* when the lockholding client is still alive and in its critical-section code. The client can continue to request *criticalPuts* concurrently with the critical section of a subsequent lockholder. However, before the subsequent lockholder entered its critical section, *acquireLock* synchronized the data store. So the true value in the data store now has a timestamp with the current *lockRef,* over-riding any timestamp with the preempted *lockRef.* Consequently, the preempted client's *criticalPuts* will have no effect on the data store, and the critical-section invariant will be preserved. As the code shows, as soon as its local lock store is updated, a MUSIC replica will know that the client's *lockRef* is out-of-date and return "youAreNoLongerLockHolder".

We can now address the proof of the *Exclusivity Property* (§III-A). Because of the guards on *criticalPut,* the only clients whose requests proceed to the quorum write are those with current or past *lockRefs.* If the *lockRef* is past and the client is still active, it has been preempted. For its write to change the value of the key, its *lockRef* must be later than or equal to the *lockRef* of the true timestamp. So this invariant applies:

SynchFlag Invariant. *If a client has a* lockRef *that is both past (released) and later than or equal to the* lockRef *of the* true timestamp*, then the* synchFlag *is true.*

This is a simplified statement because the preempted client may have failed, but there might still be traces of its requests as pending tasks in MUSIC replicas or as ongoing writes to the data store. If these traces are present, the *synchFlag* must be true as well. In these cases, before another client can enter its critical section, the data store will be synchronized, which will either obliterate the ongoing write or confirm it as belonging to the previous lockholder's critical section.

V. Formal Verification

In this section we provide an overview of MUSIC's formal verification – an effort that ensures correctness despite the complex failure modes of geo-distributed services.

A. Modeling the System

In choosing an approach to verification, we were motivated by the need to experiment and design iteratively; we chose Alloy [8] as a modeling language because it has fully-automated ("push-button") verification over bounded domains. The Alloy language is a smooth and versatile integration of first-order predicate logic, relational algebra, and object orientation.[4] The model takes the form of a state-transition system, which means that it consists of four parts: (i) a declaration of the structure and component types of each state of the system, (ii) an invariant, which is a large and complex predicate describing which system states are legal, (iii) a predicate describing the initial state of the system, and (iv) for each type of event, a precondition/postcondition pair. The pre/postconditions are predicates on the system states, describing when the event is enabled, and how it alters the system state, respectively.

Verification assumes that each event is atomic, but they are fine-grained events. With the exception of lock-store events

[4] A further discussion of this choice, including comparison to alternatives, can be found in [43].

(§V-C), the "biggest" atomic event is confined to one node, which can read a message from its input buffer, change its local state, and send an output message.

The model of MUSIC replicas is the specification that our implementation must satisfy. The models of the network, clients, lock-store replicas, and data-store replicas record assumptions about the behavior of these components, which interact with MUSIC replicas but are not controlled by them.

B. Verification

For a state-transition model, there is a conventional set of proof obligations based on the system invariant: (i) the initial state must satisfy the system invariant, and (ii) for every event and system state satisfying the system invariant, if the event is enabled in that state and executed from that state, the resulting state also satisfies the invariant. If the proof obligations are discharged, then every possible system state satisfies the system invariant, and all correctness conditions implied by the invariant are true. Thus, in this style of verification, every property we want to prove must be stated as an invariant, i.e., a conjunct of the system invariant.

§IV presented two important invariants, the *Critical-Section Invariant* and the *SynchFlag Invariant.* To formalize these properties, defined concepts such as the *true value* and *true timestamp* are put into the model state as *history variables*, which appear nowhere in the implementation. In state transitions they are updated to show what the "true" state of the system should be, so that invariants such as the *Critical-Section Invariant* can state that implementable state components are equal to history variables whenever appropriate. The *Latest-State Property* (§III) is turned into an invariant of the form "if a reply to a *criticalGet* request is waiting in the lockholding client's input queue, then it is carrying the key's true value."

The system invariant is large, complex, and most of its conjuncts say how the many components of a real system state—one that could have arisen during execution of the system—are consistent with one another. In total the formal model plus assertions consists of 1328 lines of Alloy code, including 49 invariants and 21 event types (see [44]).

The proof obligations are discharged automatically by running the Alloy Analyzer on the model; the Analyzer does exhaustive "model enumeration" over bounded scopes, which means that there are maximum sizes for basic sets (for example, our model is analyzed with 5 instances of each replicated node type). To check that an event type preserves the system invariant, it constructs (symbolically) all possible system states that satisfy both the invariant and the event precondition. It then computes the post-state for each possible pre-state, and checks that it satisfies the invariant. If a post state does not satisfy the invariant, the Analyzer presents this counterexample for debugging. The widely-accepted "small scope hypothesis" [8] says that most bugs are exposed by counterexamples in small scopes. In practice it is not difficult to choose scopes that find *all* the bugs, as far as can be determined by comparison with other forms of verification. It only takes a few seconds to a few minutes to check each assertion.

Our focus to date has been on formally verifying safety properties (something bad does not happen, e.g., the system does not return a wrong answer). Our reasoning about liveness (something good does happen) is based on three observations. First, every lockholder's local peek will eventually return true since the lock store eventually propagates updates to all replicas. Second, critical operations will eventually succeed since we assume there are enough back-end store replicas and that failures are infrequent enough. Finally, every aspiring client will eventually acquire the lock to a key since failed/non-responsive lockholders will be timed-out and preempted. In future work, we will formally verify these properties.

C. Modeling the Back-end Stores

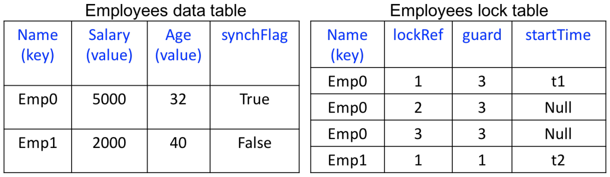
Atomic events on the lock store are larger-grained than other events in the model, because of the strong consistency properties of consensus protocols. The most innovative aspect of our verification effort is our approach to modeling eventually-consistent stores. Earlier versions of the MUSIC model, which included explicit representation of the data store with all its replication and failure cases, were too complex to comprehend fully. We solved this problem by modeling only certain properties of the data store that are directly observable by MUSIC replicas. The result is a weak and incomplete representation of behavior, but this does not matter, because the properties we do model are exactly the ones that MUSIC correctness relies on.

An attempted quorum write by a MUSIC replica is modeled by a *(MUSIC timestamp, value)* pair. The entire set of past attempted writes is kept in a history variable, partitioned into subsets *pending* and *succeeded.* A pair begins in *pending* and moves to *succeeded* when and only when the requesting MUSIC replica receives an acknowledgment message that the write has succeeded, which means that a quorum of replicas has been updated. Because this is a MUSIC view of the data store, if the write fails, or the data store completes a write but the MUSIC replica requesting it has timed out the request, or has died and does not receive the reply, then the attempted-write pair stays in *pending* forever.

The *true pair* is defined as the write pair with the latest timestamp. Thus all write pairs except the true pair should not affect the value returned by a quorum read. The data store is *defined* from MUSIC's view (§IV-A) when, and only when, the true pair is in *succeeded*. The crucial property of the data store is that if a quorum read operation is requested and replied to, and *the data store is continuously defined between the request and reply,* then the reply returns the true pair. This is a specification that any eventually-consistent data store should satisfy. The design of MUSIC ensures that *criticalGets* are requested by the lockholder when, and only when, the data store is continuously defined.

VI. Implementation

The MUSIC algorithms in §IV are implemented in Java 1.8 with approximately 12k LOC [13]. Functionality is provided as a Java library that clients can import and as a multi-site



Employees data table

Name (key)	Salary (value)	Age (value)	synchFlag
Emp0	5000	32	True
Emp1	2000	40	False

Employees lock table

Name (key)	lockRef	guard	startTime
Emp0	1	3	t1
Emp0	2	3	Null
Emp0	3	3	Null
Emp1	1	1	t2

Fig. 2: Example of a MUSIC data store and lock store table in Cassandra.

REST web service, the latter as shown in Figure 1. Due to its proven multi-site performance and production support through Datastax [45] (more details in §X-A1), our goal was to use an *unmodified* version of Cassandra to realize both the lock store and data store. Each MUSIC replica can query any Cassandra node for lock and data store operations. In this section, we describe how we make novel use of Cassandra's light-weight transactions (LWT) [11] to ensure that the lock store is updated in a sequentially-consistent manner while the data store is updated using the relatively efficient quorum operations (§III-B).

To realize the MUSIC data and lock stores, we use Cassandra's SQL-like (CQL) model with keyspaces that contain data tables and lock tables. As shown in Figure 2, for each row in a data table, the primary key/index corresponds to a MUSIC key, while the (value) columns collectively correspond to the MUSIC value of the key. Each row in a lock table is indexed by the pair *(key, lock reference)* to maintain a queue of lock references sorted in ascending order for each key. Other columns in Figure 2 are explained below. The data in these tables can be replicated and sharded across Cassandra nodes as required. Clients send key-value pairs for these tables in JSON format, which are then converted to CQL queries.

ECF Functions. While Cassandra uses the time-of-write (*time*) to order all values for a key, MUSIC's data store (unlike the lock store) requires vector timestamps (*lockRef, time*) as described in §III-B. To address this challenge, we implement a function that maps vector to scalar timestamps, $v2s(lockRef, time) = lockRef \cdot T + time$, where T is a configurable parameter that restricts the maximum time for which a lock-holder can be in a critical section. In §X-A2, we prove that this mapping preserves the ordering of vector timestamps.

The *criticalPut* operation is implemented as follows. First, we perform a lock store peek, which is implemented as a Cassandra *eventual read* query on the lock table that reads from one replica to obtain the topmost lock reference for a key. Then, we ensure that the critical section duration is not exceeded by maintaining a *startTime* variable for each lock reference in the lock-store that is initialized in *acquireLock* just before returning *true* for this lock reference when access is granted. *criticalPut* can then simply reject any operations with $(time - startTime) \geq T$. In *dsPutQuorum* we convert *(lockRef, time)* into a scalar timestamp $v2s(lockRef, time)$ and perform a Cassandra *quorum update* operation to update the value of the key and its timestamp at a quorum of replicas. The only difference in *criticalGet* is the use of *dsGetQuorum*, which is implemented as a Cassandra *quorum select* query that returns the latest value from a quorum of replicas.

For *createLockRef* the most difficult challenge was generating per-key increasing lock references using just one consensus operation. We show in §X-A3 that a simple choice of 128-bit time-based UUIDs for lock references can cause overflows in the data table. To avoid this, we use a 64-bit static integer variable called *guard* whose value is constant across rows of a key (see Figure 2). Each call to *createLockRef* then uses Cassandra's *batch* operation to atomically increment this variable and enqueue its value in the lock table using one LWT. In *forcedRelease*, we implement *lsDequeue* using the LWT delete, which atomically removes the entire key-lock reference row from the lock table. Finally, all the functions used internally in *acquireLock* have been described above. In §X-A4, we provide pseudo-code for several of these functions.

Additional Functions. To enhance MUSIC's usability (see §VII), we provide a *get(key)* and a *put(key, value)* function that are implemented as an eventual read and write on any Cassandra replica respectively. Clients should use these functions only on keys for which no ECF guarantees are expected.

VII. MUSIC in Production

In this section we describe MUSIC's use in realizing the VNF Homing Service (from §I) and a Management Portal Service [6] with active replication. These services have been deployed since September 2017 across AT&T sites as shown in Figure 1, with the three sites in San Diego, Kansas City, and North Carolina, respectively. The deployment is associated with a controlled introduction, so usage is expected to ramp up from here. As it does, we will collect data for more extensive analysis. We are also currently designing MUSIC-based solutions for ONAP's Application and SDN Controller Services. Note that our pseudo-code here does not include failure handling for simplicity; refer to §III for guidelines on how this would be added.

a) VNF Homing: This use-case is an example of the job-scheduler paradigm where workers (scheduler replicas) vie for jobs (homing requests). A homing request contains a list of VNF service chains to be placed, together with associated constraints such as hardware requirements, distance between VNFs, bandwidth between VNFs, etc. that determine the placement of the VNFs on the cloud sites. Each worker chooses the appropriate sites by solving for the constraints. Figure 3(a) shows the schematic of a Homing Service consisting of worker pools replicated across multiple sites. Incoming homing requests are load-balanced across the homing service front-end (Client API), which inserts it into a pool of pending homing requests. Any free worker can select and solve a homing request, moving it across the different execution states of the homing process as shown in Figure 3(b).

A Client API replica receives an incoming homing request or job with a unique identifier (*jobId*), and creates a key in MUSIC with this *jobId*. The value of the key is a combination of the dynamic *job execution state* and a static *job description* that provides enough information for a worker to resolve this request. This information is placed in MUSIC by the receiving

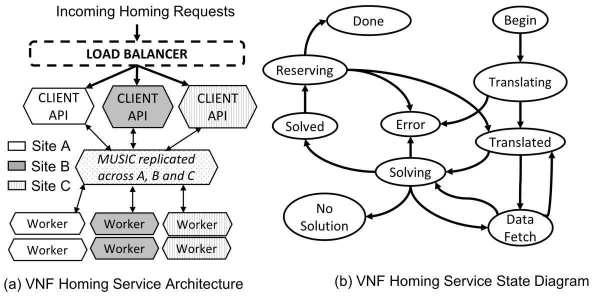


Fig. 3: Schematic of a VNF Homing service.

Client API replica using the *put* operation without any locks. It then periodically checks to see if the job has been completed (its execution state is "DONE") using the *get* function in order to delete completed jobs from MUSIC.

```
At each worker, in a periodic loop do:
 Queue currentJobIds = getAllKeys();
 while (currentJobIds is not empty){
   jobId = currentJobIds.pop();
   //get job (e.g. homing request) details.
   {jobState, jobDesc} = get (jobId);
   if (jobState != DONE)
      lockRef = createLockRef (jobId);
      if (acquireLock (jobId, lockRef) == true)
         executeJobInCriticalSection (jobId, lockRef);
         releaseLock (jobId, lockRef); // exit critical section
      else
         removeLockReference (jobId, lockRef);
 //wait x seconds for new jobs to be added
```

```
executeJobInCriticalSection (jobId, lockRef)
 //get latest job state
 {jobState, jobDesc} = criticalGet (jobId, lockRef);
 while (jobState != DONE)
   //execute job and progress it to its next state
   jobState = next state from current job state;
   // for the homing use−case refer to Figure 3(b)
   value = {jobState, jobDescription};
   criticalPut (jobId, lockRef, value);
```

Each worker (pseudo-code above) iterates through all jobs in MUSIC according to their time of submission, using *getAllKeys* and *pop* helper functions. Since these functions do not use locks, the values may be stale but that has no impact on the correctness of the job scheduler. When a worker finds an incomplete job, it attempts to acquire exclusive access to the job using a MUSIC lock over the *jobId*. Workers that fail to acquire a lock to a job use a *removeLockReference* function to evict their lock reference from MUSIC and ensure timely garbage collection. The worker executes the job in a critical section from its latest state and regularly updates the state of the job in MUSIC using *criticalPut*. As a result, if this worker fails, another worker can execute the job from its latest state.

b) Management Portal: The Portal Service provides the front-end through which clients—users and administrators—can use ONAP. Portal is deployed as client-facing REST front-end replicas communicating with back-end replicas each processing client requests. Administrators use Portal to change the roles of users, which changes their privileges over different ONAP projects. To ensure consistent role updates, each request has to be processed from its latest state by exactly one back-end replica. Unlike VNF homing, a request to change a user role involves *just one state update*, specifically to a key-value pair *(userId-role)* maintained in MUSIC.

```
write (userID, role) at Portal REST front end:
  owner = get(userID−owner)→owner; //cache locally
  if (owner == empty) //only on initialization
     owner = closest back end replica;
  while (owner→write (userID, role) != SUCCESS)
     owner = next closest replica; //repeat RETRY times
```

```
write (userID, role) at Portal back end P:
  ownerDetails = get(userID−owner); //cache locally
  if (ownerDetails→owner == empty) //only on initialization
     own (userID); // first owner
  if (ownerDetails→owner != P) //only on previous owner failure
     forcedRelease(userID, ownerDetails→lockRef);
     own (userID); // new owner
  criticalPut(userID, ownerDetails→lockRef, role);
```

```
own (userID) at Portal back end P: // called infrequently
  lockRef = createLockRef(user);
  while(acquireLock(userID, lockRef)!=true) skip;
  put(userID−owner,(P, lockRef)); //no locks needed
```

A front-end replica (above) routes each request to the user's owner, also maintained in MUSIC. If no owner has been assigned or if the current owner fails to respond, the request is retried at other available back-end replicas, sorted according to latency. A back-end replica on receiving such a request becomes that user's owner by forcibly releasing the lock, acquiring a new lock to the user, and updating the ownership and lock reference details. This lock reference is used for this and subsequent requests to perform critical operations. Since the cost of stale ownership information is just unnecessary ownership transition with no implication on correctness, this information can be cached at each replica. If each back-end replica executes requests in a single thread, MUSIC ensures exclusive, latest reads and writes. Crucially, a critical section is interrupted and ownership transitions *only* on back-end replica failure. By reducing the number of calls to create and release lock references, both of which require distributed consensus across the WAN, we reduce the chance of failed operations, amortize the locking cost across multiple *criticalPuts* for a given user, and reduce the time taken to execute each *write*.

VIII. Evaluation

In this section, we address the following questions:

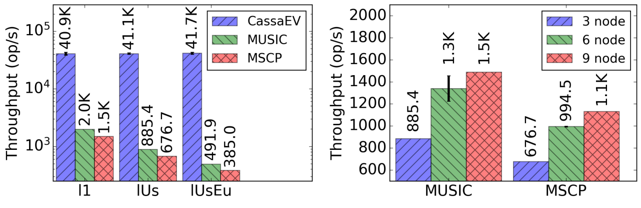
- How does MUSIC perform (throughput and latency) under different latency profiles and cluster sizes?
- Does MUSIC significantly outperform a tool with identical guarantees in which critical operations use Cassandra's LWTs as opposed to just quorum puts?
- Do the additional properties provided by MUSIC over a sequentially-consistent system like Zookeeper come at a high cost?

Profile	Site 1	Site 2	Site 3	RTT in ms
l1	Ohio	Ohio	N. Virginia	0.2, 15.14, 15.14
lUs	Ohio	N. Calif.	Oregon	53.79, 72.14, 24.2
lUsEu	Ohio	N. Calif.	Frankfurt	53.79, 100.56, 150.74

TABLE II: Latency profiles used for 3-site deployments.

- Does MUSIC significantly outperform a solution with identical guarantees implemented using a highly optimized geo-distributed database like CockroachDB?

a) Methodology: We use Proliant SE1101 servers running Ubuntu 18.04 with 16GB RAM, 256GB SSD disk and eight 2.5GHz cores. We partition the servers into 3 logical sites and emulate WAN latencies between sites using NetEm [46] based on latency profiles that closely reflect our target deployments as shown in Table II. The RTT between sites is based on AWS measurements [47] and is presented in the order Site 1-Site 2, Site 1-Site 3, Site 2-Site3. The same RTT is used for both directions. While l1 is within one AWS region, lUs and lUsEu are across regions. We run a Cassandra 3.11.3 cluster, where by default we have one Cassandra node per site. For all our experiments, we maintain one copy of each key-value pair on each site. We use a load-generator on each site that uses the MUSIC library, which communicates with Cassandra. We measure the *peak throughput* by saturating the servers with multiple threads, and *mean latency* using a single thread of operation. To prevent collision-induced variability, each thread updates non-overlapping key ranges. Keys are updated using a default *data size* of 10 bytes. Each experiment is performed at least five times, with each run taking 5 minutes on average. For all results, we present the mean and standard deviation (when greater than 5%). We only present results for writes since the results for reads are similar. In our experiments we do not introduce any failures since we assume they are relatively infrequent and our goal is to measure performance during normal operation.



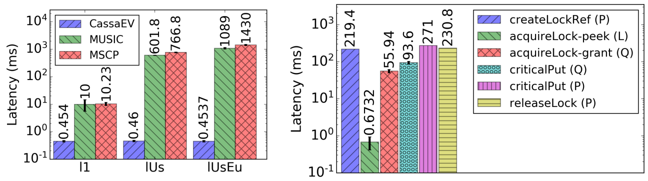
(a)

(b)

Fig. 4: MUSIC has ∼ 30% higher throughput than MSCP, in which critical puts use LWTs as opposed to just quorum puts. Both solutions scale as we increase the number of nodes from 3 to 9 in a fully sharded deployment.

b) Microbenchmarks: In our experiments, we compare three operations: (i) a write in a critical section where the code creates a lock reference, acquires a lock, performs a critical (quorum) put, and then releases the lock (*MUSIC*); (ii) a Cassandra local write with eventual consistency, which provides an upper bound for performance (*CassaEV*); and (iii) a write in a MUSIC critical section using a sequentially-consistent (SC) LWT put rather than a quorum put (*MSCP*). The MSCP operation serves as a lower bound on performance and illustrates the high cost of LWTs for critical puts.

Figure 4(a) shows the throughput of CassaEV, MUSIC, and MSCP across the three latency profiles. CassaEV has throughput of nearly 41K op/s, which is comparable to the throughput published for a three node cluster by Datastax [48]. MUSIC's throughput is much less than CassaEV since the former is a multi-step operation that uses both consensus and quorum operations. However, MUSIC outperforms MSCP by ∼30% across all the latency profiles due to the latter's use of an LWT put. For the lUS latency profile, as the total number of Cassandra nodes is increased from 3 to 9 with 3 replicas of each key sharded across these nodes, MUSIC outperforms MSCP (∼30-36%) and the throughput of the solutions improves (Figure 4 (b)). Since similar results were seen in experiments across AWS data-centers, we do not present them.



(a) Mean latency

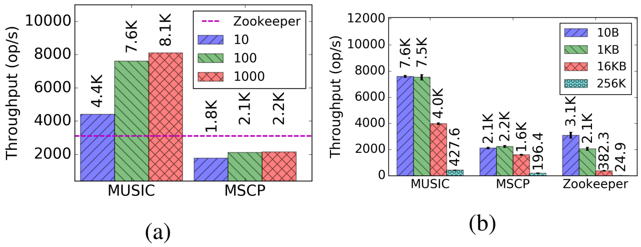
(b) Latency breakdown for lUs

Fig. 5: MUSIC's ∼30% lower latency than MSCP for the cross-region profiles (lUs, lUsEu) is due to the higher cost of the latter's LWT critical put.

Figure 5(a) shows the mean latency of CassaEV, MUSIC and MSCP across all the latency profiles. As expected, the latency of CassaEV is similar across all the latency profiles. However, the latency of MUSIC is approximately 30% less than MSCP for the cross-region profiles (lUs, lUsEu), which translates to the higher throughput observed earlier; similar results are seen in the CDF (see §X-B1). For the remaining experiments in this section, we only present results for the lUs profile since the results are similar for lUsEu and l1 does not capture cross-country WAN latencies.

Figure 5(b) shows a fine-grained latency breakdown of the main MUSIC operations (see §VI). Since MSCP is identical to MUSIC except for its LWT *criticalPut* (marked 'P' for Paxos), we just show the latency for that operation. Both *createLockRef* and *releaseLock* use LWTs and require 219-230 ms, which corresponds to 4 RTTs across Ohio-N.California (RTT = 53.79 ms). For clarity, we break *acquireLock* into the peek, which is a ∼0.67 ms local function (marked 'L') that can be called multiple times by clients awaiting a lock, and the grant, which is a ∼55 ms quorum read (marked 'Q') of the *synchFlag* across Ohio-N.California called only for the next lockholder. While the MUSIC *criticalPut* is a ∼93 ms quorum write across Ohio-Oregon (RTT = 72.14 ms), MSCP's *criticalPut* is a ∼270 ms LWT write.[5]

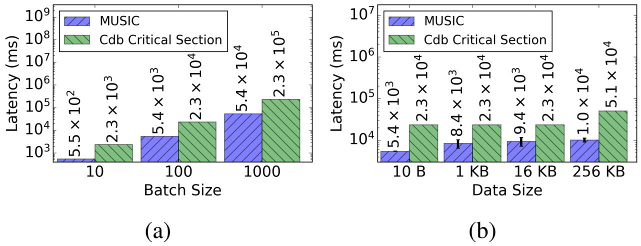
[5] In §X-B2, we present similar results with YCSB [49] loads, where we allow lock collisions among threads.



(a) (b)

Fig. 6: While Zookeeper's throughput (∼3k) is greater than MUSIC for batch size of 1 (885.4 in Figure 4(a)), as the batch and data size increases, the cost of locking is amortized and MUSIC outperforms not only MSCP (∼2-3.5 times) but surprisingly, also Zookeeper (∼1.4-17.17 times).

c) Comparisons with Zookeeper: In Figure 6, we compare MUSIC with a 3-node (1 node per site), fully replicated Zookeeper 3.4.13 deployment on the same infrastructure. To illustrate the amortization effects of MUSIC we vary the number of writes (*batch size*) in a critical section from 10 to 1000, and vary the data size from 10B to 256KB for a fixed batch size of 100. As the number of writes increases (Figure 6(a)), MUSIC's peak throughput nearly doubles as the cost of locking gets amortized across the quorum puts. This structure is reflective of our most common use-cases. Despite the additional properties provided by MUSIC (see §II), MUSIC outperforms Zookeeper (∼1.4-2.3 times). This is surprising, as a MUSIC critical put (a quorum write) and a Zookeeper Zab write [19] both require just one RTT. The improvement is more pronounced (∼2.45-17.17 times) when we increase the *data size* (6 (b)). Since we observed a stable consensus leader in Zookeeper, these performance differences are perhaps due to the queuing effects of consensus writes. As expected, MUSIC outperforms MSCP (∼2-3.5 times) in both experiments. Moreover, our comparisons with Zookeeper suggest that even if we replaced the use of LWTs for critical puts in MSCP with the more efficient Zab operation, MUSIC will still outperform MSCP.



(a) (b)

Fig. 7: MUSIC executes faster (∼2-4 times) than a critical section with identical guarantees implemented on CockroachDB.

d) Comparisons with CockroachDB: In Figure 7, we compare MUSIC with a 3-node (1 node per site), fully replicated CockroachDB (Cdb) 3.4.13 deployment on the same infrastructure for the 1Us profile. To provide the same properties as a MUSIC critical section with multiple state updates, each state update in Cdb needs to be done in an exclusive transaction (see §II). Since Cdb transactions use optimistic concurrency control by default, we used their locking primitives to build a Cdb critical section (pseudo-code in §X-B3). The main purpose of this comparison is to validate the qualitative analysis in §X-B4 that shows that a MUSIC-based critical section is nearly two times faster than one based on CockroachDB. We do not attempt an in-depth comparison because clearly a transactional DB like CockroachDB offers different semantics than MUSIC.

Our workload consists of a single thread executing one critical section in both these tools. As expected, as we increase the batch size (Figure 7 (a)) or increase the data size for a fixed batch size of 100 (Figure 7 (b)) MUSIC has much lower mean latency (∼2-4 times) than the Cdb critical section.

IX. Conclusions

Critical sections that provide exclusive access to the latest state have been a fundamental building block of concurrent systems for decades. Here, we argue that such an abstraction is also invaluable for building geo-distributed services, but is challenging to realize using existing solutions for sequential consistency, locking services, or geo-distributed databases due to more complex failure modes and high WAN latencies. We address this challenge through a formally verified key-value store, MUSIC, with novel ECF semantics, presented to programmers as a critical section abstraction suitable for geo-distributed services. MUSIC is being used in production deployments and is part of the open-source ONAP project for multi-site services. Our evaluation of MUSIC demonstrates its effectiveness in multi-site settings.

Future work will focus on providing richer locking abstractions and new design patterns to address various additional use-cases. We are also building a hierarchical version of MUSIC that will scale better across the WAN.

References

[1] "HAS," https://wiki.onap.org/pages/viewpage.action?pageId=16005528.

[2] R. Potharaju and N. Jain, "An empirical analysis of intra- and inter-datacenter network failures for geo-distributed services," in *SIGMETRICS '13 Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. ACM, June 2013, pp. 335–336.

[3] "The network is reliable: An informal survey of real-world communications failures," http://www.bailis.org/papers/partitions-queue2014.pdf.

[4] B. N. Bershad *et al.*, "The midway distributed shared memory system," in *Digest of Papers. Compcon Spring*, Feb 1993, pp. 528–537.

[5] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[6] "Portal," https://wiki.onap.org/display/DW/ONAP+Portal, 2019.

[7] "Open network automation platform (onap)," https://www.onap.org/.

[8] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006, 2012.

[9] "Alloy," http://alloytools.org.

[10] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[11] "LWTs," https://docs.datastax.com/en/drivers/python/3.2/lwt.html.

[12] "CockroachDB," www.cockroachlabs.com.

[13] "MUSIC gerrit repository," https://gerrit.onap.org/r/gitweb?p=music.git.

[14] B. Balasubramanian *et al.*, "Brief announcement: Music: Multi-site entry consistency for geo-distributed services," in *ACM PODC 2018*.

[15] P. Hunt *et al.*, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[16] "Consul by Hashicorp," https://www.consul.io/, 2019.

[17] "A distributed, reliable key-value store for the most critical data of a distributed system." https://etcd.io/, 2019.

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[19] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *ACM LADIS 2008*.

[20] W. Lloyd *et al.*, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *ACM SOSP 2011*, pp. 401–416.

[21] D. D. Akkoorath *et al.*, "Cure: Strong semantics meets high availability and low latency," in *IEEE ICDCS*, 2016, pp. 405–414.

[22] C. Li *et al.*, "Making geo-replicated systems fast as possible, consistent when necessary," in *USENIX OSDI 2012*.

[23] Z. Wu *et al.*, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *ACM SOSP 2013*.

[24] K. Lev-Ari *et al.*, "Modular composition of coordination services," in *USENIX Annual Technical Conference (ATC)*, 2016.

[25] A. Ailijiang *et al.*, "Efficient distributed coordination at wan-scale," in *IEEE ICDCS 2017*.

[26] "Atomix," https://atomix.io/, 2019.

[27] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *USENIX OSDI 2006*, pp. 335–350.

[28] T. D. Chandra *et al.*, "Paxos made live: an engineering perspective," in *ACM PODC 2007*.

[29] "Curator recipes," https://curator.apache.org/curator-recipes/index.html.

[30] D. Agrawal and A. El Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 1–20, Feb. 1991.

[31] A. Bar-Noy *et al.*, "Fault-tolerant critical section management in asynchronous environments," *Inf. Comput.*, vol. 95, no. 1, pp. 1–20, Nov. 1991.

[32] A. Shraer *et al.*, "Cloudkit: Structured storage for mobile applications," *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 540–552, Jan. 2018.

[33] S. Das *et al.*, "G-store: A scalable data store for transactional multi key access in the cloud," in *ACM SOCC 2010*.

[34] I. Zhang *et al.*, "Building consistent transactions with inconsistent replication," in *SOSP 2015*.

[35] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM TOCS 2013*, vol. 31, no. 3, p. 8.

[36] M. J. Fischer *et al.*, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[37] D. Dolev *et al.*, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, pp. 77–97, Jan. 1987.

[38] C. Dwork *et al.*, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[39] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[40] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[41] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers C-28*, vol. 9, pp. 690–691, September 1979.

[42] ——, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, October 2006.

[43] P. Zave, "A practical comparison of Alloy and Spin," *Formal Aspects of Computing*, 2014, available at Springer via http://dx.doi.org/10.1007/s00165-014-0302-2.

[44] "Music alloy," https://wiki.onap.org/display/DW/MUSIC+Alloy+Code.

[45] "Datastax," https://www.datastax.com/, 2019.

[46] S. Hemminger, "Network emulation with NetEm," in *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*, M. Pool, Ed., Linux Australia. Sydney NSW, Australia: Linux Australia, Apr. 2005.

[47] "Aws inter-region latency monitoring," https://www.cloudping.co/.

[48] "Apache cassandra nosql performance benchmarks," https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks.

[49] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with ycsb," in *ACM SOCC 2010*.

[50] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC 2014*.

X. Appendix

A. Additional Implementation Details

1) Why Cassandra?: Since Cassandra's LWTs provide a compare-and-set primitive, it requires 4 RTTs across replicas to complete [11]. While we could have satisfied our requirements through basic consensus writes with implementations requiring only 1 RTT [50], we chose Cassandra because: (1) since we need it for our data store, it is far easier to maintain and deploy just one tool in production, (2) Datastax does not provide support for modified versions of Cassandra, so we could not optimize its default LWT solution, (3) as shown in §VIII, the cost of consensus is sufficiently amortized even for modest critical section sizes, and (4) Cassandra's fully sharded implementation of LWTs makes it seamless to add new nodes and scale out. Integrating new, efficient consensus primitives into the open-source code of Cassandra is an avenue of future work.

2) v2s Preserves Ordering:

Lemma. *$v2s$ preserves the ordering of vector timestamps.*

Proof: Consider two vector timestamps $t1 = (lockref1, time1)$ and $t2 = (lockref2, time2)$. If *t1* = *t2*, then trivially, $v2s(t1) = v2s(t2)$.

If $t1 < t2$ such that $(lockref1 = lockRef2)$ and $(time1 < time2)$, i.e., timestamps typical of operations *within* a critical section, then,
$v2s(t1) = lockref1 + time1 < lockRef1 + time2 = lockRef2 + time2 = v2s(t2)$ and hence, $v2s(t1) < v2s(t2)$.

If $t1 < t2$ such that $(lockref1 < lockRef2)$, i.e., timestamp $t1$ belongs to an earlier critical section than $t2$, then,
$v2s(t1) - v2s(t2)$
$= (lockRef1 - lockRef2) \cdot T + (time1 - time2)$
$< T - (time2 - time1)$ {lock references are positive integers and $lockRef1 < lockRef2$}
< 0 { $(time2 - time1) < T$ since each critical section can last for only T seconds}

The same argument can be extended to show that $(t1 > t2) \Rightarrow (v2s(t1) > v2s(t2))$. ■

3) Timestamp Overflow Analysis: Since Cassandra timestamps are signed 64-bit integers, to prevent overflow, the following inequality has to hold: $(lockRef \cdot T + time) \leq 2^{63}$. We note that $time$ here is only used to order writes within a critical section and hence, $v2s$ can be implemented as $(lockRef \cdot T + time - startTime)$ where, $startTime$ is the time at which the critical section for $lockRef$ began. The inequality then simplifies to $(lockRef \cdot T) \leq 2^{63}$ since $(time - startTime) < T$. With this inequality and time captured in milliseconds, clearly we can support nearly 10 million lock references as long as $T < 29$ years, which is more than sufficient for our needs. The problem with using 128-bit UUIDs as lock references is that we cannot control which among the 2^{128} UUIDs will be generated and from the inequality this can easily cause an overflow.

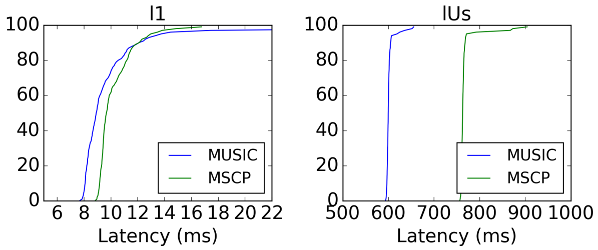


Fig. 8: The latency CDFs for MUSIC and MSCP.

4) Simplified CQL code: In this section, we provide a simplified version of the CQL code for some of the functions to illustrate how the algorithms translate to code that reads and writes to Cassandra.

```
createLockRef (key)
  prevGuard = SELECT guard FROM lock_table WHERE
        lockName = key; //consistency = eventual
  lockRef = 1;
  if  (guard != null )
    lockRef = prevGuard + 1;
  BEGIN BATCH; //will be retried configurable  number of times
   UPDATE lock_table SET guard = {lockRef} WHERE key =
        {key} IF guard = prevGuard;
   INSERT INTO lock_table (key, lockRef) VALUE (key,lockRef) IF
        NOT EXISTS;
  APPLY BATCH; //entire batch is one LWT operation
  return  lockRef;
```

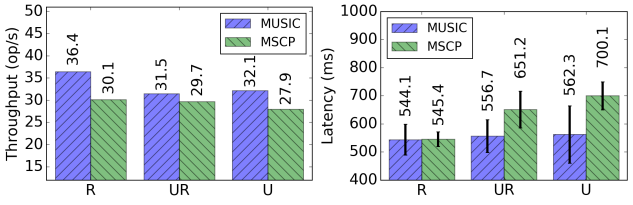
```
criticalPut (key, lockRef, value)
 (topLockRef, startTime) = SELECT (lockRef, startTime) FROM
        lock_table WHERE lockName = key; //consistency = eventual
 if  (lockRef > topLockRef)
    return  ( false ) ;
 if  (lockRef < topLockRef)
     return  (youAreNotLockHolder);
 if  ((currentTime − startTime) > T)
    return  (exceededDuration);
 dsPutQuorum (key, lockRef, value, startTime);
 return  ( true ) ;
```

```
dsPutQuorum (key, lockRef, value, startTime)
 scalarTime = lockRef∗T + (currentTime − startTime);
 UPDATE data_table USING TIMESTAMP = scalarTime SET value
        = {lockRef} WHERE key = {key} ; //consistency = quorum
```

B. Additional Evaluation Details

1) Latency CDFs: In Figure 8, we show the latency CDFs for MUSIC and MSCP. While for the l1 profile the latency is similar, for the cross-region lUS profile, MUSIC outperforms MSCP by ∼30%.

2) YCSB Benchmarks: In this section we compare the performance of MUSIC and MSCP for the workloads provided by Yahoo!'s YCSB benchmark [49], the de-facto benchmarking tool for NoSQL databases. We implemented our own MUSIC adapter for YCSB such that the YCSB reads/writes/inserts were converted to MUSIC and MSCP operations respectively. We ran three different workloads for a three node MUSIC cluster with the lUs latency profile: R only has READs, UR is composed of 50% READS and 50% UPDATES, and U only has updates. In each scenario tuples are selected randomly with a Zipfian distribution.



(a) Throughput (Mean)

(b) Latency (Mean)

Fig. 9: MUSIC and MSCP comparison using the YCSB benchmark

In Figure 9, we see that MUSIC consistently outperforms MSCP in terms of both throughput (∼6-20%) and latencies ($\sim 0-20\%$). In these workloads, among the 10,000 operations performed in each of the loads there were ∼5.5 % collisions, i.e., contention for locks.

3) CockroachDB RW Transactions: Pseudo-code for a critical section in CockroachDB that provides the same guarantees as a MUSIC critical section:

```
do batch size times:
  BEGIN TRANSACTION;
    SELECT ∗ FROM t WHERE k=lock; //check for lock
    UPSERT INTO t (k,v) VALUES (lock,ME) RETURNING
          NOTHING; //critical section entry using Raft consensus
  END TRANSACTION;
  UPSERT INTO t (k,v) VALUES (k,random−string−of−size−10
        bytes) RETURNING NOTHING; //local state update
  UPSERT INTO t (k,v) VALUES (lock,NONE) RETURNING
        NOTHING; //critical section exit using Raft consensus
  COMMIT;
```

4) Qualitative Comparison with Spanner RW Transaction: In this section, we compare the cost of a MUSIC critical section with x *criticalPuts* to shared state with a solution with similar guarantees implemented in Spanner/CockroachDB where each update to shared state is performed in a separate transaction (see §II). This corresponds to Read-Write transactions in Spanner (Section 4.2.1 in [35]), where each transaction involves one consensus (C) operation for locking the transaction, one local operation for the update (negligible cost), and one consensus operation for the commit of the transaction, leading to a cost of 2C/transaction. Since this has to be performed x times, the total cost of this solution is $2 \cdot x \cdot C$. Consider the cost of doing the above in MUSIC (§IV): one consensus (C) operation for creating a lock reference, one quorum (Q) look up of the *synchFlag* to acquire the lock, x quorum writes in the critical section and one consensus operation for releasing the lock, leading to a total cost of $2C + (x+1) \cdot Q$. Assuming, generously, that a good implementation of consensus will only cost as much as a quorum operation, the above translates to $(3+x) \cdot C \approx$ x$\cdot C$ when $x >> 3$. Hence, the MUSIC-based solution is nearly two times faster.