METRIC: A Middleware for Entry Transactional Database Clustering at the Edge

Enrique Saurez[1], Bharath Balasubramanian[2], Richard Schlichting[3], Brendan Tschaen[2], Zhe Huang[2], Shankaranarayanan Puzhavakath Narayanan[2], Umakishore Ramachandran[1]

Georgia Institute Of Technology[1], AT&T Labs Research[2], United States Naval Academy[3]
esaurez@gatech.edu,bharathb@research.att.com,schlicht@usna.edu,bt054f@att.com,
{zhehuang,snarayanan}@research.att.com,rama@gatech.edu

Abstract

A geo-distributed database for edge architectures spanning thousands of sites needs to assure efficient local updates while replicating sufficient state across sites to enable global management and support mobility, failover etc. To address this requirement, a new paradigm for database clustering that achieves a better balance than existing solutions between performance and strength of semantics called *entry transactionality* is introduced. Inspired by entry consistency in shared memory systems, entry transactionality guarantees that only a client that *owns* a range of keys in the database has a sequentially consistent value of the keys and can perform local and, hence, efficient transactions across these keys. Important use cases enabled by entry transactionality such as federated controllers and state management for edge applications are identified. The semantics of entry transactionality incorporating the complex failure modes in geo-distributed services are defined, and the difficult challenges in realizing these semantics are outlined. Then, a novel **M**iddleware for **E**ntry **Tr**ansact**i**onal **C**lustering (METRIC) that combines existing SQL databases with an underlying geo-distributed entry consistent store to realize entry transactionality is described. This paper provides initial findings from an on-going effort.

ACM Reference format:
Enrique Saurez[1], Bharath Balasubramanian[2], Richard Schlichting[3], Brendan Tschaen[2], Zhe Huang[2], Shankaranarayanan Puzhavakath Narayanan[2], Umakishore Ramachandran[1]. 2018. METRIC: A Middleware for Entry Transactional Database Clustering at the Edge. In *Proceedings of Middleware'18, Rennes, France, December 2018,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Cloud and network service providers are increasingly investing in infrastructure at the edge of the network. Examples of this trend include Microsoft's Azure Stack for the edge [1], large edge cloud infrastructures being built by AT&T and Verizon [2, 3], and the open-source Akraino edge stack [4] co-founded by AT&T and Intel. Akraino provides a blueprint for AT&T's edge software stack, which will be deployed on edge sites, and is expected to host various edge services like the virtualized RAN for 5G deployments and services for deep learning and augmented reality.

A geo-distributed database to manage client and service state for edge architectures that span multiple sites needs to satisfy two important requirements. First, processes need to update local state in an efficient and often transactional manner. Second, some or all of this state needs to be replicated consistently at other sites to enable global management and to support capabilities such as failover and mobility. For example, the control plane for AT&T edge services is envisaged to be a federation of regional controllers each managing hundreds of edge sites. Each edge site then has local controllers that typically perform transactions only on the state of their edge sites, while allowing the regional controllers to perform occasional transactions across the state of the edge sites in different regions to enforce global actions (described in Section 2). We address these requirements through a new paradigm for database clustering called *entry transactionality*, and describe METRIC, a **M**iddleware for **E**ntry **Tr**ansact**i**onal **C**lustering.

The first requirement is often addressed by having an instance of a SQL database running locally or at least close enough to assure fast access. While existing solutions for clustered databases offer a starting point for the second requirement, none of them achieve the right balance between strength of guarantees and performance, especially at the scale and geo-distribution of edge services deployed across thousands of sites. On the one hand, solutions like MariaDB with Gallera clustering [5], Spanner [6], and CockroachDB [7] provide full transactionality, but perform expensive 2-phase commit and rollback protocols across sites on which they are deployed. On the other hand, clustering in PostgreSQL guarantees transactionality only within each site with asynchronous replication across sites, thereby violating the requirement for consistent replication.

This paper describes a clustering middleware called METRIC that achieves the right balance between performance and replication semantics by realizing *entry transactionality*, a concept inspired by entry consistency in shared memory systems [8]. Entry transactionality guarantees that only a database client that *owns* a range of keys: (a) has a sequentially consistent value of the keys at the time ownership is granted, and (b) can perform local and efficient transactions across these keys. Non-owners can read the state of the keys, but with no consistency guarantees. We define the precise semantics of METRIC in Section 3.

While sharding is used internally in databases for horizontal scaling and performance [6, 7, 9, 10], by elevating the concept of key-ownership to an abstraction, rather than just in the underlying database implementation, METRIC enables service designers to *construct* services in a manner suited to geo-distribution (examples in Section 2). Further, since consistent replication and the complex

Middleware'18, Rennes, France

DOI: 10.1145/nnnnnnn.nnnnnnn

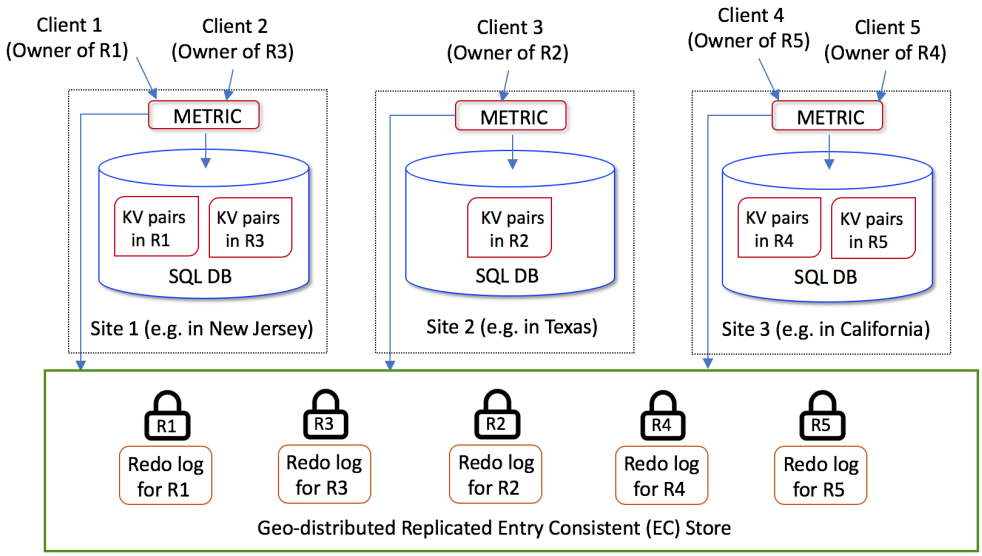

Figure 1. An architectural overview (details in Section 4) of a multi-site deployment of METRIC, where the actual key-value pairs for each range (Ri) are maintained in the SQL database associated with each METRIC process, and the locks for each range and the redo log are maintained in a geo-distributed EC store. A *site* is a data center at a physical location connected with other sites through a WAN.

failure modes of geo-distributed services are incorporated in the METRIC semantics, the service designer only has to understand how state ownership transitions to suit the needs of the service.

Realizing entry transactionality in a performant and correct manner involves addressing a challenging conflict between *efficiency* and *geo-replication*. In particular, while the owner of a key range should be able to perform transactions efficiently, enough transaction state also has to be geo-replicated to ensure that a potential new owner has access to consistent state despite failures. We address this conflict in METRIC through a lightweight clustering middleware over instances of a standard SQL database, as shown in Figure 1. A client can contact any METRIC process deployed on a nearby site to acquire ownership of keys and perform transactions on the SQL database connected to that METRIC process. The SQL database serves as an efficient site-level cache for each METRIC process and crucially, it can be chosen according to production preferences for specific databases like MariaDB or PostgreSQL, with their in-built clustering mechanisms (if any) turned off.

Consistent replication across sites is realized by storing data modified by each transaction in AT&T's multi-site entry consistent store (*EC store*) called MUSIC [11, 12]. This system implements the abstraction of a key-value store, with locking primitives that provide entry consistency, i.e., where sequential consistency is enforced only for the lock holder of a key. METRIC uses these abstractions to enforce ownership and to implement an entry consistent redo log, which offers better performance and availability across sites when compared with common geo-distributed databases that use a sequentially consistent commit/redo log across sites [6, 7].

In summary, then, this paper makes the following contributions:

- Identifying entry transactionality as an effective paradigm for the database needs of edge-oriented architectures, motivated by the use cases of federated state management and mobility/failover support.
- Introducing METRIC, a middleware that realizes entry transactionality, and defining the semantics of METRIC that take into account the complex failure modes of geo-distributed edge services.
- Providing an overview of the design and algorithms of METRIC that combines standard database solutions with a novel state replication protocol based on an underlying geo-distributed EC store.

Since our primary goal in this paper is to motivate METRIC and highlight key aspects, we only provide an overview of the overall effort, and leave details related to on-going work on the algorithms, implementation, and deployment for future papers.

2 Motivation

In this section, we describe two important use cases for entry transactionality that are relevant for edge services: *federated state management* and *ownership transition* across edge service replicas triggered by mobility, load-balancing, or failure. Both these use cases are motivated by the need for a geo-distributed database to manage the state of AT&T edge services.

Federation. A concrete example of the federation use case presented in the introduction is that of Virtual Network Function (VNF) deployment across edge sites (Figure 2). In this use case, each regional controller receives deployment specifications of VNFs from a global controller and needs to identify appropriate edge sites, i.e., the sites that satisfy the VNF's constraints on locality, specific hardware, etc., across which the virtual machines (VMs) of a VNF need to be deployed. The local controller at each edge site performs the actual role of placing the VMs of the VNF on the hosts of the edge site, and needs to manage the state of resources in a transactional database to ensure optimal and correct placement. The regional controller periodically reads the state of the resources across edge sites to track usage. For certain VNFs that require strict performance guarantees, the regional controller needs to reserve resources at the edge sites to ensure that these requirements are adequately satisfied at deployment time.

Entry transactionality is an effective way to address the state management requirements of this problem. Specifically, the regional and local controllers can all share a multi-site METRIC deployment, where the local controllers maintain their resource state in the SQL

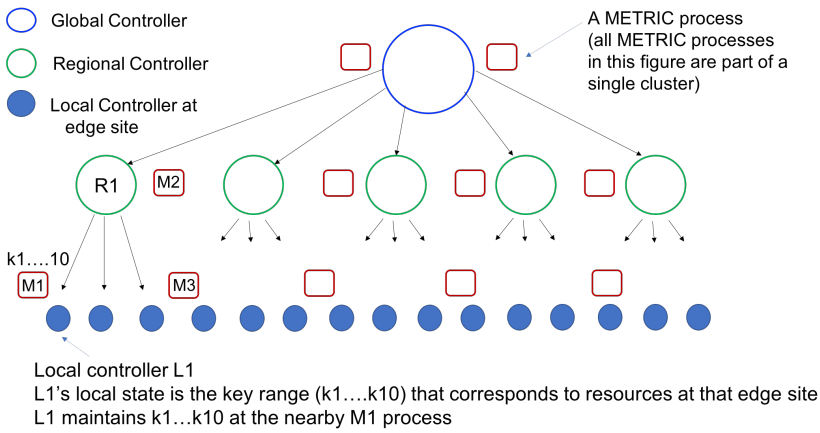

Figure 2. The use of METRIC to enable federated state management across the local and regional controllers.

database of a nearby METRIC process, acquire ownership of this state, and issue transactions over them during VM deployment. During regular operation, the regional controllers (non-owners) can read the state of these local resources from any METRIC process. To reserve resources, the use of METRIC enables a regional controller to simply acquire ownership of the state of all local controllers managed by it, read the sequentially consistent state of the resources, and then update the state in a transactional manner. This idea can be generalized to perform state management for various functions such as authentication and closed-loop control in a federated manner across local and regional controllers.

Mobility/Failover Support. An important reason for having sites closer to the edge of the network is to provide low-latency services to end users. For example, consider an object recognition service that performs deep learning at the edge. To provide service to users close to them, we envision that replicas of this object recognition service will be deployed at multiple edge sites across the country. Each service replica maintains various aspects of the state corresponding to the end user associated to that service replica—a catalog of objects, mobility history, session state, account details and so on. Sometimes a user associated with a service replica needs to transition to a new service replica. This event is triggered by the service to handle the failure of the old service replica, to load balance among replicas, or to account for the user moving from one location to another, necessitating association with a new service replica to satisfy latency requirements. The new service replica needs access to the latest state of the user and must continue modifying this state as it serves the user. This requirement can be addressed by maintaining user state in a METRIC deployment where each service replica acquires ownership of the state of the users associated with it, either at initialization or triggered by a transition.

The use cases described above can be addressed neither by a fully transactional MariaDB/CockroachDB deployment across the country due to the performance implications described in the introduction, nor by asynchronous PostgreSQL replication since these use cases often require stronger guarantees. Most systems address similar use cases by maintaining independent database clusters for different components—for example, a cluster for edge sites in a city and a cluster for each regional controller. To obtain a federated view of shared state or to allow for transition, these systems need complex handwritten code that migrates state between the database clusters. This not only increases system complexity, but is also prone to errors, especially in the face of the failures during migration. A solution based on entry transactionality completely hides such complexity from the system designer, who only needs to understand how state ownership transitions. The underlying data semantics are guaranteed by METRIC.

3 METRIC Semantics

In this section, we describe one of our core contributions, the semantics of METRIC, and explain them using the use cases in Section 2.

Our system model assumes a distributed system of processes that communicate using messages. To overcome the impossibility of distributed consensus in asynchronous systems [13], like most practical systems, we assume partial synchrony [14, 15] to achieve consensus, where there are sufficient periods of communication synchrony with an upper bound on message delay. Processes can suffer crash failures [16], which implies that a process cannot distinguish between a failed process and one that is slow to respond and/or unable to communicate. The latter is relatively common in geo-distributed systems where failures in communication links [17, 18] can render a process partitioned from some subset of other processes in the system.

METRIC offers its clients the abstraction of a replicated database with the additional notion of key ownership. The METRIC model consists of a set of METRIC processes, where clients issue requests to any METRIC process of its choice. While we support the full suite of SQL operations, for ease of exposition, we assume a database that maintains state in the form of an ordered range of unique keys each with its own value. The semantics of replication is that a key has a single correct value determined by the rule "last write wins," based on the timestamp associated with each write. The correct value eventually propagates to all replicas, assuming frequent enough communication.

Ownership. A client can acquire ownership to a contiguous range of keys, or a *key-range*, at a METRIC process, where it is guaranteed access to the sequentially consistent state of the keys in the key-range at that METRIC process. A single client can be the owner of multiple non-overlapping key-ranges. Any client can take ownership of a key-range belonging to another client at any time at any METRIC process. For example, in Figure 2, while $L1$ is usually the owner for the key range ($k1..k10$) that it maintains at $M1$, when $R1$ wants ownership for this range, it can communicate with a closer process $M2$ to obtain the latest values of the keys.

While it has no implications on correctness, for performance reasons we assume that transfer of ownership is relatively rare with minimal contention, i.e., typically there are only one or two clients that want ownership of a key-range. Hence, METRIC does not provide any arbitration mechanism across clients seeking ownership, and we assume explicit or implicit signaling among clients to decide which client should be the new owner. For example, in the federated controllers use case, the regional controller can explicitly send a message to the local controllers to coordinate, and then ask METRIC for ownership of their local state. In the edge mobility use case, when a user moves from one service replica to another, the new service replica requests and obtains ownership of the user's state with the implicit knowledge that the old service replica no longer needs ownership.

Operations. Only the owner of a key-range can write to the keys in a key-range. The write operation includes addition, deletion and SQL-style joins of keys in the range. For both reads and writes the owner sends the request to the METRIC process that granted ownership to the key-range. The owner is guaranteed ACID transactional semantics with serializable isolation for all reads and writes to the keys in the key-range. Hence, in Figure 2, if $L1$ obtains ownership of ($k1..k10$) at $M1$, then it has to issue transactions to this key-range at $M1$. Non-owners can read (potentially inconsistent) values of any key at any METRIC process.

Failures. If a client does not receive a response to an ownership request at a METRIC process, it assumes the latter has failed and simply sends the request to some other METRIC process. If the owner of a key-range does not receive a response for an operation within a transaction at a METRIC process, it assumes the latter has failed, re-requests ownership of the key-range at some other METRIC process, and retries the aborted transaction at that process. For example, in Figure 2, if $L1$ does not receive a response for a certain query in a transaction to a key in ($k1..k10$) at $M1$, it needs to acquire ownership of this range at another METRIC process, say $M3$, and re-try the entire transaction.

We assume that client failures are detected using timeouts by other clients. If a client detects the failure of another client, it can acquire ownership of some or all of the keys owned by the failed client and perform transactions on them. This enables the mobility/-failover use case described in Section 2. If a client loses ownership of its keys to another client due to erroneous failure detection, then the operations of the original owner will fail, thereby informing it that ownership has transitioned. Note that if a client loses ownership of even a single key in a key-range, it loses ownership to all the keys in that key-range.

METRIC may use other backend stores internally, and we assume that at least a quorum of the processes of these backend stores are always available. For example, in our design of METRIC, we maintain client key-value pairs in the MUSIC geo-distributed EC store, which guarantees that data is replicated for both fault tolerance and availability in at least a quorum of its processes.

Table 1 summarizes the main abstractions provided by METRIC and Pseudocode 1 illustrates their use through a simple example of a client accessing METRIC to own a key-range and perform transactions on it.

A client that intends to perform transactions across a range of keys in METRIC first needs to identify a METRIC process, *proc*, typically located at a nearby site for performance reasons. The client then uses the *own* function to acquire ownership over a range of keys. The function returns a globally unique identifier *ownId* (e.g., a UUID) that is used by METRIC to identify the owner for a range of keys. On acquiring ownership, the client is guaranteed the latest value of the keys in the range at *proc* and can now perform transactions within that range using the standard interface to SQL transactions provided by METRIC to begin, execute and commit queries during a transaction. The one added requirement is that the client also needs to include its *ownerId* so that METRIC can verify that it is indeed the owner of the key range. Note that one of the features of the METRIC abstractions is that the client can use them to acquire ownership of a key range from another client, either voluntarily or on detecting failure of the client, in a manner identical to that shown in Pseudocode 1.

Abstraction	Description
ownerId = own (key-range)	Returns a unique identifier that is good for one request for ownership.
txId = beginTransaction (ownerId)	Begins a transaction across the keys in the key-range of the owner.
executeQuery (ownerId, txId, query)	Performs SQL queries for the owner in a transaction including joins within the key-range.
commitTransaction (ownerId, txId)	Commits a transaction. If a non-owner attempts this, METRIC rejects the commit.

Table 1. METRIC abstractions that enhance standard SQL operations with new abstractions for key-ownership.

Pseudocode 1. Example use of METRIC abstractions.

```
# code snippet at client
# identify METRE process, "proc", at nearby site,
# e.g., a TCP end-point
ownerId = proc.own(key-range);
# client is now guaranteed sequentially consistent state
#  of the keys in key-range at proc
# client has to use proc to perform transactions to this
# key-range
txId = proc.beginTransaction(ownerId);
# keys in query that belong to key-range
query1 = "select * from table, where k1 = ...";
rowSet = proc.executeQuery(ownerId, txId, query1);
query2 = "update table T, set .., where k2 = ..."
proc.executeQuery(ownerId, txId, query2)
proc.commitTransaction(ownerId, txId);
```

4 Design and Algorithms

4.1 Overview

Realizing entry transactionality is a challenging problem. On the one hand, the owner of a key range should be able to perform transactions efficiently at a METRIC process. On the other hand, enough transaction state also has to be geo-replicated to ensure that a new owner potentially accessing the keys on an entirely different site has access to sequentially consistent state despite the failures inherent in geo-distributed systems. This challenge is exacerbated by our goal of being a drop-in replacement for the clustering mechanisms in existing databases such as MySQL and PostgreSQL to support production preferences.

While geo-distributed sharded databases also need to replicate data consistently across sites, Spanner [6] relies on specialized hardware for atomic clocks, while CockroachDB [7] incurs high penalties for read operations. Both of these solutions use a sequentially consistent cross-site commit log, where each operation to the log incurs the cost of distributed consensus across sites. Calvin [19],

FoundationDB [20], and TiDB [21] rely on a global sequencer to order their operations, which incurs a heavy penalty across sites as well. Clearly, all these solutions are incompatible with the need to provide efficient operations for the owner of a key range in entry transactionality.

In this section, we present an overview of how METRIC addresses this challenge in a *performant and correct manner*. The key insights behind the solution can be summarized as follows:

- Combining the rich and time-tested features of a SQL database to provide transactionality to an owner of a key range as a local cache, with the novel geo-distributed clustering solution provided by METRIC.
- Using an Entry consistent (EC) store in a novel way both to ensure exclusive access to the owner of a key-range and to maintain a geo-distributed redo log that is replicated across sites with entry-consistent semantics.
- Limiting the use of distributed consensus across sites to the case of ownership transition.
- Ensuring that each query in a METRIC transaction is a local SQL database operation by updating transaction state only when committing a transaction using quorum operations to the redo log in the EC store.
- Dealing with failures as essentially ownership transition, thereby combining two crucial aspects of entry transactionality and making it far easier to reason about correctness.

4.2 Design

Here, we describe the three main components of a METRIC deployment, as shown in Figure 1: METRIC processes, the SQL database associated with each METRIC process, and the geo-distributed EC store.

METRIC processes are deployed across multiple sites for locality, availability, and fault-tolerance. They provide an interface to the clients that can be accessed through advertised end-points (e.g., a TCP/REST endpoint) and implement the abstractions shown in Table 1. A METRIC process is stateless in that it maintains all state in the SQL database and the EC store.

Each METRIC process is strictly associated with an instance of a SQL database (DB) that it uses to execute transactions for clients that own key ranges at that METRIC process. We assume that the DB supports ACID transactions with serializable level of isolation, an assumption satisfied by most common databases like MySQL and PostgreSQL. Moreover, the DB is assumed to be purely local to each METRIC process and not clustered *across sites*. However, to load balance and account for failures, the DB can be clustered within a site, as long as it provides the abstraction of a single transactional database to the METRIC process with which it is associated. For example, a METRIC process can be connected to a three-node MariaDB-Gallera [5] cluster deployed *within a site*, as opposed to a single MariaDB instance.

One of our key insights is the use of an EC store to maintain a geo-distributed redo log (and associated data structures) to ensure that on ownership transition, the new owner has access to the sequentially consistent state of its key range at the SQL DB of the METRIC process that granted it ownership. Specifically, we use the MUSIC (MUlti-SIte entry Consistency) EC key-value store described in [11], which is now running in AT&T's production systems. MUSIC provides the abstraction of a replicated key-value store where clients can acquire a lock to a key in MUSIC and be guaranteed a sequentially consistent value of this key. When the lock holder performs reads and writes to the locked key, i.e., in a critical section, other writers are excluded and the operations are sequentially consistent so that all reads and writes are totally ordered.

Crucially, MUSIC limits the use of distributed consensus to entry and exit of a critical section, and implements reads and writes using more efficient quorum operations across an underlying eventually consistent store. So, while commonly used geo-distributed databases [6, 7] implement a commit/redo log on top of a sequentially-consistent store, METRIC implements a redo log on top of an entry-consistent store, which promises much better performance for geo-distributed sites due to its limited use of consensus.

4.3 Algorithm Overview

own (key-range): Every range with an owner in METRIC is represented by a unique *range-owner-key* in MUSIC, with the value pointing to the MUSIC table that is the redo log for that range/owner combination. Each range-owner-key is associated with a lock, where the unique id of that lock is the *ownerId*. When a client requests ownership of a range from a METRIC process, that process first releases all locks that have key ranges that overlap with the requested range. It then creates a new range-key corresponding to the requested range and acquires a lock to this range, creating a new ownerId.

At this point, the METRIC process is guaranteed that no other client can modify the keys in the requested range by virtue of MUSIC's locking semantics. It can read the redo logs of all the keys in the requested range and populate the DB associated with it, thereby providing the owner of this range with the sequentially consistent value of these keys at the METRIC process. Note that this function is used by clients both to acquire ownership during initialization and during a transition, either voluntarily or on detecting a client failure.

To grant ownership and populate the consistent value of the keys in the DB, METRIC uses both distributed consensus for lock transitions in MUSIC and quorum operations (to read the latest redo log state) across sites. Hence, this is a relatively expensive operation and should not be invoked very frequently. But as illustrated in Section 2, there are many important use cases where ownership transition is the exception rather than the norm. Additionally, the cost can be reduced and amortized by prefetching the corresponding data into nodes where there is an expectation of ownership, an operation which is supported by both MUSIC and METRIC architecture.

beginTransaction (ownerId): The METRIC process first ensures that ownerId is indeed the owner for the key range and if yes, then it creates a unique transaction id for this transaction. Further, it creates a shadow table in the DB to track the operations in this transaction.

executeQuery (ownerId, txId, query): After confirming that the ownerId of the query is the owner, the METRIC process executes this query locally at the DB and creates an entry in the shadow table with the old and new value of the key after this query executes. Note that, since an owner of a key range issues queries to the same METRIC process and hence the same DB, it is guaranteed ACID semantics. Further, any joins, which are only supported within the

key range of the owner, do not involve cross-site operations since all the queries are executed locally.

commitTransaction (ownerId, txId): The METRIC process first creates a concise digest of all the old and new values modified by this transaction (physical logging), which was regularly maintained in the shadow table of the DB during query execution. It then appends this digest to the redo log in MUSIC created specifically for this owner using cross-site quorum operations.

5 Discussion

In this paper, we have motivated the need for entry transactionality and provided the complete semantics, abstractions, and design of METRIC. Here, we describe the status of other aspects of this work and elaborate on future directions.

Algorithms. Due to space constraints we could only provide an overview of the basic algorithms. In future papers, we intend to specify fully detailed algorithms with thorough arguments for correctness. Due to the imperfect failure detection inherent in geo-distributed services, it is often impossible to distinguish between a failed process and one that is slow to respond. Hence, there can be many subtle cases that we need to reason about. For example, an owner, evicted due to suspicion of failure may corrupt the state of the new owner. While the use of an entry consistent store guarantees exclusive access to the most recent state to the lock holder, those apply only to a single key, whereas we are using it to enforce ACID transactional guarantees across key ranges.

Implementation. We have a `JAVA` implementation of our system in which we have implemented most aspects of the algorithms except for failover of ownership [22]. We provide a `JAVA jdbc` driver with additional APIs for ownership of key ranges that clients can ingest and use to access METRIC. We use MariaDB for the SQL database of each METRIC process and, as already noted, we use MUSIC [12] as the EC store. We are currently in the process of testing our implementation, and we will include relevant details in future papers.

Production Deployment. While METRIC is best motivated by edge architectures, it was initially designed to provide drop-in geo-distributed clustering for components within AT&T's ONAP virtualized network control plane [23] that require the use of a SQL database. We expect to integrate METRIC with ONAP Portal [24], which provides common management services and connectivity to clients of ONAP, and deploy it in production in early 2019. This deployment is also expected to serve as a blueprint for the future federation of ONAP components to serve 5G and edge use cases.

Evaluation. We intend to micro-benchmark our system using the JAVA-compatible benchmark suite provided in [25], and compare METRIC's throughput and latency with other clustered database solutions like MariaDB with Gallera, CockroachDB, and PostgreSQL for multi-site deployments with varying WAN latencies between sites. We plan to perform the same comparison in our production deployment with the ONAP Portal as the client. Further, we intend to present findings from our deployment, including details on ownership transition frequency, the ratio of reads to writes, and other relevant metrics.

6 Conclusions

The scale and geo-distribution of edge architectures necessitate a rethinking of state-management solutions to find the right balance between strength of guarantees and performance. In this paper, we introduced a new paradigm for database clustering, called entry transactionality, that achieves a good balance by providing transactionality only to the owner of a range of keys in the database. We presented a middleware for entry transactional database clustering called METRIC that provides a geo-distributed clustering solution over instances of a SQL database. METRIC's abstractions of a clustered database with the notion of key ownership has several important applications such as state management for AT&T's federated control plane and support for edge mobility/failover use cases. We addressed the hard challenge of defining meaningful METRIC semantics while incorporating the complex failure modes of geo-distributed services. Finally, we presented an overview of how the METRIC solution addresses the tension between efficiency and geo-replication inherent to entry transactionality. We believe METRIC will have a significant impact on AT&T's edge services and elsewhere. This first paper has described the motivation and core ideas, and sets the stage for future papers to present the complete details of a fully realized METRIC system.

References

[1] "Microsoft's Azure Stack." https://azure.microsoft.com/en-us/overview/azure-stack.

[2] "ATT is Reinventing the Cloud Through Edge Computing." http://about.att.com/story/reinventing_the_cloud_through_edge_computing.html.

[3] "Verizon's cloud-in-a-box pushes the edge with OpenStack." https://siliconangle.com/blog/2017/07/17/verizons-cloud-box-pushes-edges-openstack-openstacksummit.

[4] "Akraino Edge Stack." https://www.akraino.org/.

[5] F. Razzoli, *Mastering MariaDB.* Packt Publishing, 2014.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[7] "CockroachDB." www.cockroachlabs.com.

[8] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," in *Digest of Papers. Compcon Spring*, pp. 528–537, Feb 1993.

[9] "Google Vitess." https://vitess.io/.

[10] "Citus Data." https://www.citusdata.com/.

[11] B. Balasubramanian, R. D. Schlichting, and P. Zave, "Brief announcement: MUSIC: multi-site entry consistencyfor geo-distributed services," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018* (C. Newport and I. Keidar, eds.), pp. 281–284, ACM, 2018.

[12] "MUSIC Code." https://gerrit.onap.org/r/gitweb?p=music.git.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[14] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, pp. 77–97, Jan. 1987.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, pp. 288–323, Apr. 1988.

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

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

[18] P. Deutsch, "The eight fallacies of distributed computing," *URL: http://today. java. net/jag/Fallacies. html*, 2004.

[19] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12, ACM, 2012.

[20] "FoundationDB." https://apple.github.io/foundationdb/.

[21] "Pingcap TiDB." https://www.pingcap.com/en/.

[22] "METRIC Seed Code." https://github.com/esaurez/ETDB.

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

[24] "ONAP Portal." https://wiki.onap.org/display/DW/ONAP+Portal.

[25] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, pp. 277–288, Dec. 2013.