2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)

Short Paper: Towards An Edge-Located Time-Series Database

Timothy Krentz, Abhishek Dubey, Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN, 37211

***Abstract*—Smart infrastructure demands resilient data storage, and emerging applications execute queries on this data over time. Typically, time-series databases serve these queries; however, cloud-based time-series storage can be prohibitively expensive. As smart devices proliferate, the amount of computing power and memory available in our connected infrastructure provides the opportunity to move resilient time-series data storage and analytics to the edge. This paper proposes time-series storage in a Distributed Hash Table (DHT), and a novel key-generation technique that provides time-indexed reads and writes for key-value pairs. Experimental results show this technique meets demands for smart infrastructure situations.**

I. Introduction

Emergence of the Edge: The proliferation of cloud computing has spawned edge computing [1], where software services are run on or in close proximity to the devices that need them. This has supported the development of the Internet of Things (IoT), each 'smart' thing communicating with and supporting each other. Infrastructure such as the smart grid is better served by the edge paradigm than the cloud [2], and this is a particularly exciting application area for IoT devices to supervise, optimize and support a critical infrastructure of modern society. Infrastructure with edge computing often uses dispersed sources of information from user terminals or sensors. For example, a smart power grid may use several Phase Measurement Units (PMUs) attached at various points on the grid to characterize loads and sources, and use their phase measurements to optimally manage power.

Extant Challenges: As with any physical phenomena, these measurements represent state at a particular point in time, and many applications require time-indexed data from sensors like PMUs. Devices joining the network or recovering from a fault may participate in an extant application and require the previous system state. Software platforms like RIAPS [3] provide completely distributed service discovery and programming paradigms for smart infrastructure applications. In [3], the authors describe a distributed traffic controller, where traffic-light timings respond to traffic density sensors and communicate their actions. Supposing a traffic-light controller was recovering from a faulty state, it would require a temporal history of traffic density to re-synchronize with the desired traffic flow. This type of query is usually supported by one of many Time-Series Databases (TSDB); however, many current TSDBs are designed to run in a centralized manner, in a data center, aligned with the cloud computing paradigm. Many do not support the time resolution necessary for some real-time applications, nor provide excessively rich indexing that could hurt performance. Cloud-based solutions aren't viable for some applications that require low latency and/or low operating cost, both of which are properties more accessible to edge computing.

Contributions: In order for us to continue development of smart infrastructure, a low-cost time-series database built for edge-computing is required. Distributed Hash Tables (DHTs) work well in peer-to-peer and edge computing applications, but don't support time-indexed retrieval for a stored key. In this paper we propose a novel key structure that enables a DHT to access and store time-series data and discuss how modifications of this structure affect performance properties of this system. Section II will review the current work in this area, Section III will introduce the time-factored key, Section IV will discuss the how modifications of this key affect DHT performance, Section V will showcase performance of the initial implementation, Section VI will discuss how to improve upon this work, and Section VII will conclude.

II. Related Work

The focus of this work is adding a dimension to key-value storage to implement a TSDB on edge computing hardware. There are many TSDB solutions already available that provide higher-dimensional storage and retrieval.

Cassandra [4] is a popular distributed datastore designed for high write throughput without sacrificing read efficiency. It uses a notion of ownership, relating data keys to the node keys on a circularly organized keyspace. However, each node caches the entire cluster's keys locally, thus reducing the lookup message count to 1. Cassandra further permits multiple keys per query, where the first key relates to a specific column, the rest used to access that data as structured according to the user. Cassandra is the underlying storage to KairosDB [5].

InfluxDB[6] is probably the best known centralized time-series database. Measurements are stored in a series, made up of timestamped points. Points may have both fields and tags; fields are required for every datapoint, whereas tags are optional. For example, a PMU datapoint would have its required timestamp and field (phase angle), but could be tagged with a bus ID. Subsequent queries for phase data on that bus would be retrieved by looking at points indexed by that tag. InfluxDB further supports retention policies for each series, allowing the configuration of how long it keeps data,

978-1-7281-0151-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ISORC.2019.00037

when it is transferred to long-term storage, and how many times data are replicated.

OpenTSDB [7], like InfluxDB, supports tags to increase the dimensionality of its interface. Its underlying storage is built upon HBase, but it does not support write-time analytics. Furthermore, it only supports millisecond resolution, which is insufficient for real-time application like power management.

Storacle [8] is a time-series storage scheme idealized for smart grid applications, proposing a tiered use of memory, disk, and cloud writes for historical data. Though technically edge storage, Storacle still relies on a centralized TSDB to be hosted, carrying cost and single-failure implications. Furthermore, it is not specified that distributed applications can easily find and retrieve historical data from other Storacle nodes.

These systems are all successful but very general; they support very flexible access methods like tags, which incur a significant performance hit. Some also do not support the time resolution necessary for real-time phenomena that operate on the sub-millisecond scale, such as the power grid. For this application, a simpler solution was created. The Berkeley Tree Database (BTrDB) [9] is a centralized time-series storage solution designed for storing simple data (timestamps and scalars) at very high throughput. BTrDB sets the bar for storage of PMU data. Part of its success is due to supporting only very simple data, a timestamp and a float. Its underlying structure is a time-partitioned tree; traversing the tree, each vertex contains summary statistics of its subtrees, until eventually the leaves are the raw data points. By maintaining running statistics at each vertex, read requests that would otherwise calculate a statistic from the raw data can retrieve it from a vertex of the desired resolution directly.

BTrDB is an excellent solution for storage and analytics on high-throughput real-time data like phase measurements. However, it was not designed to be distributed, which is a fundamental paradigm of some applications. Therein lies the motivation to create a simple TSDB that operates in a highly distributed manner, which carries the benefits of being resilient to node failures and tolerant of changes in topology.

III. Time-Factored DHT Keys

Distributed Hash Tables are lightweight, highly distributed datastores that provide key-value storage and retrieval for clients. Keys are structures (often strings) that clients provide to the DHT along with data they want to store, and again when they want to retrieve said data. The keys are hashed into a DHT ID such that requests of a certain key can be efficiently routed to some subset of all nodes participating in the DHT whose IDs are "closest" to the key. That subset can store the value and reply to requests to read from that ID. DHTs are highly resilient to failures as the values are stored on multiple nodes, so if the node closest to an ID fails, the value is preserved on some subset of peers and can be migrated to a new node to maintain the system's replication factor. Both data keys stored and the DHT nodes themselves have unique IDs, all residing in an abstract space.

This paper implements a DHT similar to Kademlia [10], using 160-bit IDs and the XOR of a key ID and a node's ID interpreted as a number as a metric for "closeness". Typically, this ID is the SHA-1 hash of the key a client application uses to keep track of the value. As such, the bits within the ID are meaningless to a Kademlia DHT as anything other than some location in an abstract space. The client, however, sees the provided key in whatever manner it likes; for example, the name of a Smart Grid PMU can be the key to its phase measurements, hashed to find the DHT ID, and provided to the DHT to retrieve a value for said PMU. A PMU could write its phase measurements to the DHT, where the closest nodes would store its timestamped data in a local time-indexed container. Unfortunately, for any one measurement this ultimately resembles a centralized model of storage, albeit with some replication. It still suffers a potential traffic bottleneck at this subset of nodes, and reduces the number of failures a particular value can tolerate. A better solution for routing queries for time-indexed data in a DHT is needed.

To better utilize the entire network to store and access time-series data, we propose dividing the ID for a key into two parts, a "time quantum" in which the current value was acquired, and the hashed key. This time quantum represents a constant-length segment of physical time, where all time quanta combined represent all of continuous time. Any specific time in which the value of a key-value pair was created, e.g. the timestamp of a measurement from a PMU, should only exist in a single time quantum. Finally, each time quantum should have a unique identifier which is hashed to create the second part of the DHT ID. This paper maintains the Kademlia standard of using 160-bit keys and routing them based on the XOR distance metric, but the first and last 80 bits are separately generated from the relevant time quanta's identifier and the key provided by the application. The identifier is the UNIX time (number of seconds since Jan 01, 1970) defining the beginning of that time quantum. In this paper, the time-quantum length is 10 seconds, so any UNIX timestamp rounded down to the nearest multiple of 10 results in said timestamp's time quantum identifier. Concerns regarding SHA-1 collisions can be alleviated by simply routing based off longer hashed keys, but for the scale and longevity of the experiments herein this was considered unlikely. By structuring keys this way, the entire time-series for a key is sharded into blocks of values within their respective time-quanta.

IV. Behavior of Time-Factored Keys

By factoring an ID into time-key subsections, several beneficial properties emerge; that said, as Kademlia's notion of distance is an XOR of IDs interpreted as a number, the order of the hashed time quanta identifier and hashed key portions of the ID have a significant effect on how a measurement is stored in the DHT, thus changing the emergent properties. These properties will be discussed separately for Quanta-First IDs and Key-First IDs.

A. Quanta-First IDs

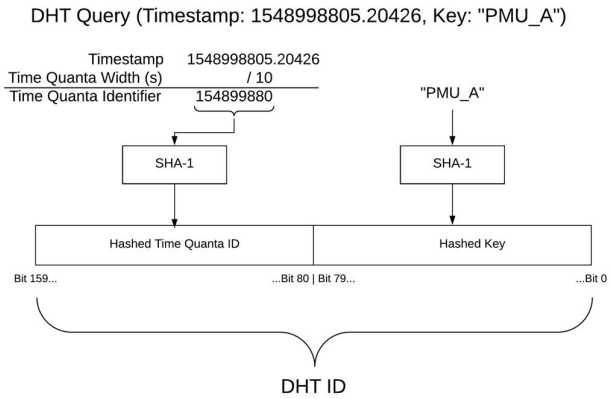

Fig. 1. Querying a time-indexed key using the Quanta-First ID Format

In Quanta-First IDs (QFIs), the first 80 bits of the DHT ID are the hash of the relevant time quanta's identifier, and the last 80 are the hash of the key provided by the application, as in Figure 1. Though the identifier only changes by one digit every time quanta, SHA-1 produces highly-variable hashes for similar inputs, so shards should disperse very evenly throughout the network. This is a result of Kademlia's interpretation of distance; the most significant factor affecting how close a node is to a measurement's ID is the hash of whatever time quanta that measurement occurred in. As shards are dispersed evenly throughout the network, the bulk of any measurement's history spanning some non-trivial number of time quanta is highly resilient to single node failure. As such, the system could stand reducing the DHT's replication factor for applications that can tolerate some gaps in the measurement's history, improving the underlying DHT's performance. Should an application need to retrieve a specific time-series for a key, it needs only construct a set of IDs to lookup by hashing the identifiers for the time quanta covering the timespan of the request, and prepend the first 80 bits of each hashed quanta identifier to the first 80 bits of the hashed key. The resulting set of IDs can be used in parallel read requests, so retrieval times should be independent of the history length requested.

B. Key-First IDs

In Key-First IDs (KFIs), the first half of the ID comes from the value hash, and the second from the time quanta identifier hash. As the key should remain constant for the data series, the changing time quanta identified has minimal effect on which nodes the query is routed to, because Kademlia's distance metric will first consider bitwise differences between node IDs and hashed key portion of the queried ID. The resulting behaviour here is that shards are most likely stored on the same subset of nodes whose IDs are closest to the hashed key, resulting in the more centralized appearance of data storage as described in Section III. As only a subset of nodes owns the key, there is greater likelihood of node failures jeopardizing the value history. This is resolved with replication in the underlying DHT, so ultimately Key-First IDs closely represent regular DHT usage. KFIs will still allow time-indexed lookups, but the parallel requests will likely all go to the same subset of nodes. TSDB functionality will be present, but localizing the data to a specific subset of node means a query of arbitrary history length should ultimately occur with one request, rather than a request for each shard covering the history.

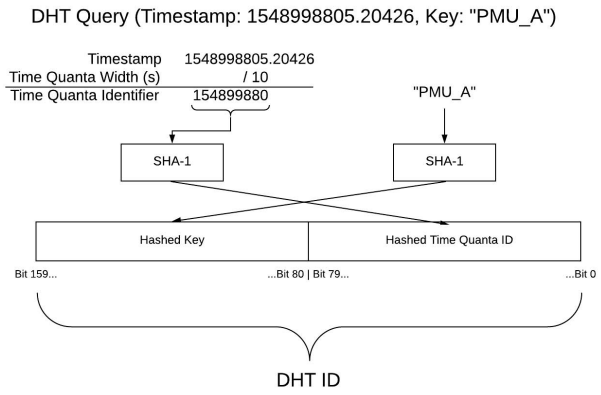

Fig. 2. Querying a time-indexed key using the Key-First ID Format

V. Evaluation

A simple DHT was built in Go [11], borrowing the 160-bit IDs and XOR distance metric from Kademlia. All data was stored in memory in a hashmap, mapping the ID to a slice of timestamped measurements. As per Kademlia, all communication happens through Remote Procedure Calls (RPCs). This implementation runs RPCs over HTTP, using Go's `net/rpc` package, while serialization was done with `encoding/gob`. The experiment was deployed to an 18-node cluster of BeagleBone Blacks (BBB), running Ubuntu 18.04 on a single-core ARM-A8 32-bit processor. The BBBs were connected in a flat network topology of unmanaged switches and a single router. The experimental DHT runs multiple goroutines[1] to handle concurrent requests and protect the local storage.

To evaluate the performance of time-factored key writes, it is assumed that the DHT has been running for some time. As DHTs typically offer request-reply interfaces, data must be stored with aggregated writes, rather than streams. It is assumed that the time to establish and execute a write-RPC is significantly more than the time it takes to serialize the data to be written. Therefore, to establish a lower bound on write latency, the experiment has a single node execute 10,000 single measurement (timestamp, value) store-requests and measures total completion time. This was repeated for replication factors of 1 (no data replication), 4, and 7.

For time-factored key reads, we evaluate whether the length of time being requested (and therefore the number of time

[1]Goroutines are lightweight threads in the Go language.

quantas) has a significant effect on read time. In our experiment, we wrote 10,000 PMU measurements at 60Hz, then allowed a different node to request that PMU's data for a range of elapsed time. The time to complete this request - including computing the relevant IDs, making requests for each time-quanta, and aggregating the results - was measured for time periods of 10, 80, and 150 seconds.

Table I summarizes the write performance for Quanta-First and Key-First IDs. For this implementation, writing with Quanta-First IDs induces a minimum of ~4.5ms write time, which grows as replication is increased. The results are fast enough to keep pace with a PMU writing individual datapoints at 60Hz, as even with 7-way replication writes complete in ~4ms on average - less than the 16.6ms write period.

Table II summarizes the read performance for Quanta-First and Key-First IDs. Requesting a single time quanta (10 seconds) produces much faster read times than groups of time-quanta (80 and 150 seconds), even though read queries for separate time quanta were parallelized. The drop in read-time from ~4 seconds for 80 seconds of history to ~3 seconds for 150 seconds of history can be explained by very high variance in the data collected. Read latencies like these - on the order of seconds - would be usable in the RIAPS traffic-controller case, as we can naturally assume second-length latencies are tolerable compared to the rate at which traffic changes.

TABLE I
WRITE PERFORMANCE (MS/WRITE) VS REPLICATION FACTOR

Key Format	Replication Factor		
	1	4	7
QFI	4.54	10.9	14.1
KFI	5.05	10.1	13.4

TABLE II
READ PERFORMANCE (S/READ) VS REQUEST TIMESPAN

Key Format	Time-Series Length(s)		
	10	80	150
QFI	0.176	4.33	3.54
KFI	0.081	3.354	2.45

VI. FUTURE WORK

This work currently supports time-indexed storage and retrieval of data, but these are only part of modern use cases for TSDBs. Many applications using TSDBs also require real-time analytics of the data stored, so an addition to be made is a method for tracking and providing analytics of data series as they are written into the DHT. This might be done at the time-quanta level, and where a read-request simply retrieves the summary from that quanta, similar to how BTrDB reads analytics from tree vertices as opposed to the raw data. As metadata such as this is much smaller than the raw data itself, metadata could be stored and accessed using KFI keying; the resulting property of single nodes holding all the metadata allows them operate upon and track metadata without any additional DHT lookups. Time-series data is also frequently provided as a stream. This is dissonant to classical DHT client-server requests, so a modification is necessary to allow datastreams to reach their desired DHT node. Finally, though clients of this TSDB can request any continuous time-interval of data, they receive that interval padded with the data of the enclosing time quanta. An improved method for storing values within time quanta that allows trimming around a specific timestamp is needed, in order to prevent giving the client more data than they asked for.

VII. DISCUSSION AND CONCLUSION

This paper presented a novel strategy for using distributed hash tables to store and access time-series data. By considering datapoints to exist within a unique time quanta, and using that quanta as part of DHT routing and ownership, historical data can be written and retrieved for arbitrary lengths of time by aggregating separate requests of specific time-quanta. These results show that time-factored keys provide time-indexed queries to data stored in a DHT. These queries can meet demand for Smart Grid PMU storage and traffic density monitoring.

Acknowledgment: This work was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000666. The views and opinions of authors expressed herein do not necessarily state or reflect those of the US Government or any agency thereof.

REFERENCES

[1] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2831347.2831354

[2] EPRI, "Transforming smart grid devices into open application platforms," Electric Power Research Institute Report 3002002859, July 2014. [Online]. Available: http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?productid=000000003002002859

[3] S. Eisele, I. Mardari, A. Dubey, and G. Karsai, "Riaps: Resilient information architecture platform for decentralized smart systems," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2017, pp. 125–132.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[5] "Kairosdb." [Online]. Available: http://kairosdb.github.io/

[6] "Influxdata (influxdb) | time series database monitoring & analytics." [Online]. Available: http://www.influxdata.com/

[7] "Opentsdb - a distributed, scalable monitoring system." [Online]. Available: http://opentsdb.net/overview.html

[8] S. Cejka, R. Mosshammer, and A. Einfalt, "Java embedded storage for time series and meta data in Smart Grids," *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015*, pp. 434–439, 2016.

[9] M. P. Anderson and D. E. Culler, "Btrdb: Optimizing storage system design for timeseries processing," *Fast'16*, pp. 39–52, 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen

[10] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65.

[11] "The go programming language." [Online]. Available: https://golang.org/