ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-TUBPL05

RecSyncETCD: A FAULT-TOLERANT SERVICE FOR EPICS PV CONFIGURATION DATA*

T. Ashwarya, E.T. Berryman, M.G. Konrad
Facility for Rare Isotope Beams, Michigan State University, East Lansing, USA

Abstract

Record Synchronizer (RecSync) is comprised of a client module called RecCaster and a server module called RecCeiver. Together they work to make PV (or record) metadata residing in the Input-Output Controller's (IOC) database available to client applications like ChannelFinder. Currently, the server module RecCeiver is a custom-built Python application that cannot be run as a cluster, hence it does not provide fault-tolerance. Further, the existing RecSync does not implement any authentication or security feature that controls the access to reads and writes to only verified clients and servers. In this paper, we explore an alternative to the current RecCeiver called ETCD which is an open-source off-the-shelf distributed key-value storage known for its high-availability storage and retrieval abilities. It provides a role-based authentication feature along with CA certificates-based and TLS-based security feature to make client-server communication encrypted and verified. It also provides useful features like conditional atomic transaction operations, live-watching of keys in its storage and ability to view historical changes to a key.

RECORD SYNCHRONIZER

The existing RecSync [1] consists of two modules: RecCaster which is a client application that runs as part of an IOC and RecCeiver which is a stand-alone server application written in Python using the Twisted networking library. Their aim is to make the metadata related to a PV being hosted on an IOC available to clients like ChannelFinder [2]. The PV metadata that is sent from RecCaster to RecCeiver consists of EPICS base version, a whitelisted set of environment variables, name & type of all records and any info tags associated with the records. ChannelFinder further provides RESTful APIs that various other applications use in order to read the PV metadata.

Theory of Operation

RecCaster is a client application written in C and works as an EPICS support module that spawns a thread on start-up to upload all the IOC records to a server. After spawning its thread, it waits for an announcement from RecCeiver. Once it discovers a live and ready RecCeiver, it exchanges handshake messages with the server. After the exchange of initial handshaking, greeting messages are sent between the client and server and then the uploading of the records to the server begins. Once the records are uploaded successfully to the RecCeiver, client and server exchange periodic heartbeat messages indefinitely to signal they are alive. In case the client dies and does not respond to the server, the server closes its connection to the client.

RecCeiver on a successful upload of data from RecCeiver pushes all the data to ChannelFinder. In case of a disconnected IOC, it signals to ChannelFinder to mark the PVs from a disconnected IOC as inactive. RecCeiver can be additionally configured to write the data it receives from RecCaster to a SQL database or print it to screen or logs.

RecCeiver is a standalone application that cannot be run as a distributed service on multiple nodes in a cluster. This makes RecCeiver a possible single point of failure if the node running the service goes down or gets disconnected from network. At this point, until the RecCeiver service is restored, the data available in clients like ChannelFinder will be stale and will not accurately reflect an IOC's state. Also, the existing RecSync does not provide any authentication feature that restricts the transfer of the IOC record metadata to only verified servers and clients. These shortcomings can be overcome using ETCD as a replacement for the existing Python-based RecCeiver.

ETCD

In this section, we talk about the data model, operations and useful features of the proposed alternative to the existing RecCeiver i.e. ETCD [3]. ETCD is a distributed, open-source key-value storage providing full replication of the key-value store on a cluster of servers. It is commonly used for distributed system coordination and metadata storage like Zookeeper and Consul which are two other popular alternatives. ETCD uses Raft [4] to perform cluster operations like election of the cluster leader and getting a majority quorum before the data is committed and written to the disk. Raft protocol is a distributed consensus protocol run on each member of a cluster to maintain a replicated state machine. Raft provides the leader election algorithm to elect a leader node from among the cluster nodes. A cluster-wide log called replicated log is used to keep the same state among cluster members. Leader node is responsible for writing the data to the replicated log and distributing it to follower nodes to maintain an in-sync state within the cluster. In case the leader node dies or gets disconnected, a re-election is held to choose a new leader from the remaining nodes.

Data Model

From a logical perspective, ETCD's data store is a flat-binary key space where byte string keys are lexically sorted indices. The entire key store have multiple revisions that monotonically increment over the cluster's lifetime and older revisions of a key are available for fetch and read. From a physical point of view, ETCD's data store is a persistent b+ tree that is ordered by key in lexical byte order

* Work supported by the U.S. Department of Energy Office of Science under Cooperative Agreement DE-SC0000661

[5]. ETCD maintains a secondary in-memory B-tree index to speed up read queries [6]. This index contains pointers to the persistent B+ tree.

ETCD Operations

Operations that change the cluster state like writes and deletes are handled by the cluster leader as they require consensus from a majority of the cluster nodes. The leader node on receiving a new change replicates the information to the follower nodes. After it receives a receipt of acceptance from a majority of alive follower nodes, it writes the change to the disk. Read operations do not need a consensus and hence can be performed by any alive node of the cluster.

The writes to ETCD can be supplemented with a lease specifying a certain period of time for which this key should be kept in the store and once the lease time is expired or revoked, the key gets deleted from the ETCD store. Leases are generally used to detect liveness of ETCD clients and are refreshed periodically by clients to show that they are alive. Apart from read, write and delete operations, ETCD provides a watch operation where keys can be asynchronously monitored for modifications. An ETCD watch continuously watches a key for changes and then sends these changes to clients over a stream of event messages that contain information about the change made to the key in an ordered, reliable and atomic way. ETCD also provides transaction operations which means multiple requests can be grouped together as a single request and be processed atomically. Requests are grouped in a then/else manner and executed if a certain condition exists in the key-value store. All comparisons are atomic and if all comparisons are true, transaction is successful otherwise it is considered a fail and else (or failure) block is applied.

ETCD uses alerts to notify a user of any issues that requires the user's attention. Some of these alert situations include a failed member resulting in an unavailable cluster, ETCD instance with no leader, high number of leader changes, more than 1% of requests failing within last 5 minutes, slow requests, file descriptors getting near to exhaustion, ETCD cluster communication being too slow, disk latency being too high and failing ETCD proposals. An ETCD server provides its local metrics that can be used to monitor the health of the node and for debugging purposes. These metrics can be fed to monitoring tools like Prometheus and Grafana (see Fig. 1) to continuously monitor and adjust the health of ETCD cluster [7, 8].

Multi-Version Concurrency Control

Concurrency control is required by any data storage to provide concurrent access to a piece of data to multiple readers and writers. Locks are used to restrict reading of a piece of data while the write operation is being performed. During the locking period, readers get to read a previous revision of data hence multiple revisions of the same data have to be maintained to provide concurrency. To implement data store revisions, ETCD maintains a 64-bit cluster wide counter called the 'store revision' which is incremented anytime a change is applied to the key-value store to help sequentially order all updates. It also retains the past revisions of a key up to a certain time period which is configured by the cluster administrator. A compaction process is run periodically at the configured setting and deletes the older revisions of a key.

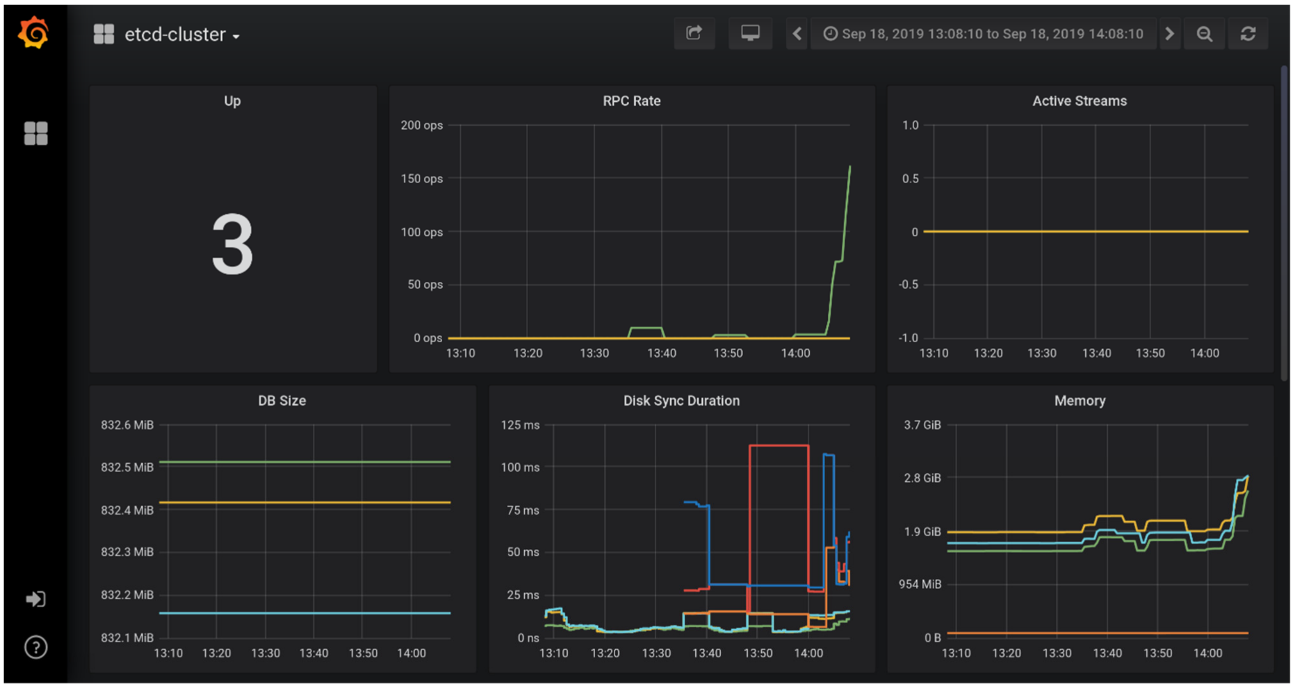

Figure 1: Grafana View of a 3-node ETCD cluster at FRIB facility displaying various ETCD metrics like DB size, Disk sync duration, memory, traffic etc.

Apart from the global counter 'store revision', ETCD maintains 'create_revision', 'mod_revision' and 'version' for each key. 'Create_revision' is the store revision when the key was created, 'mod_revision' is the store revision when the key was last modified and 'version' is a local counter that increments at every modification of the key.

Authentication and Security

ETCD provides authentication as a role-based access control to key-value storage. Various roles can be created and assigned to usernames such that only certain roles have access to modify a certain key or key range thus limiting the key access to only certain usernames. Access to keys can be granted for read, write or both.

To provide security, ETCD uses automatic Transport Layer Security (TLS) and certificates for server-client and peer communication. ETCD server is configured with a CA certificate and signed key pair to provide ability to clients to verify the server identity. Clients can further provide their own certificates to server in order for server to verify an authorized client. Peer communication among cluster members can be encrypted and authenticated by exchanging signed certificates among peers.

ETCD APIs

ETCD uses gRPC remote procedure call based APIs to provide the following six types of functionality [9].

- KV – APIs for creating, updating, retrieving and deleting key-value pairs from ETCD server.
- Watch – APIs for monitoring changes to a key.
- Lease – APIs for granting, revoking and displaying key leases.
- Authentication – APIs for setting authentication mechanism between server and clients.
- Cluster – APIs for configuring membership to the ETCD cluster and for viewing cluster state.
- Maintenance – APIs for performing maintenance operations like taking data snapshots, defragmenting and compacting storage.

KV APIs that are used for reads, writes and transactions provide the following five API guarantees.

- Atomicity – KV APIs are atomic which means it either completes entirely or not at all.
- Consistency – KV APIs are consistent which means the order in which requests are made to the cluster is kept the same across all members of the cluster.
- Isolation – KV APIs are serially isolated which means one cannot read intermediate data from ETCD.
- Durability – KV APIs are durable which means completed operations, written and read data are all durable.
- Linearizability – KV APIs are linearized which means reads from ETCD return the most current value. However, watch operations are not linearized and users are responsible to verify the revisions in watch messages to check for the order.

There is a command line utility tool called 'etcdctl' also available to communicate with the ETCD cluster through gRPC. Additionally, there exists a JSON-gRPC gateway that translates HTTP/JSON requests into gRPC request calls to ETCD and gRPC responses from ETCD back to HTTP format.

Fault-Tolerance in ETCD

Faults or failures can occur in any deployment of servers due to hardware/software malfunctions, power issues or network disconnection/partition. An ETCD server can automatically recover from temporary failures like system reboots without any loss of existing data stored on its disk. There are multiple failure scenarios discussed below that ETCD cluster can handle using its fault-tolerance techniques to recover from permanent failures of its cluster members.

- Failure of follower nodes – ETCD cluster can tolerate up to (N-1)/2 follower nodes' failure in a cluster of N nodes.
- Failure of the leader node – On leader failure, election happens after election timeout period within the cluster to elect a new leader. Write requests are queued till the re-election is complete. The new leader refreshes all existing leases.
- Network Partition – During network partition, cluster gets divided into majority (with more member nodes) and minority sides (with lesser member nodes). In case leader is in majority partition, minority side becomes unavailable leaving majority as the new available cluster. In case leader is on minority side, a new leader is elected on the majority side.
- Bootstrapping Failure – This kind of failure happens during the bootstrapping period of cluster when required members cannot successfully start. This is handled by removing data directories and re-bootstrapping with a new cluster token.

RecSyncETCD

The new RecSyncETCD comprises of two modules: RecCaster-ETCD which is a C-based client application that runs on IOC and ETCD server which is an off-the-shelf distributed key-value storage as the new RecCeiver. RecCaster-ETCD reads records metadata from IOC database and writes it to ETCD as key-value pairs. Each record from IOC database has one key in the ETCD data store and the corresponding value reflects the record's metadata. The value of a non-alias record's key comprises of the following three parts.

1. ENV – This consists of record's environment information like hostname, priority, port, time-to-live in ETCD, EPICS base version, top directory, architecture, IOC name, EPICS CA related parameters, EPICS host architecture, current working directory, IOC engineer, IOC location etc.
2. PROP – This consists of information like record type and if record has any aliases.

3. INFO – This consists of info tags attached to a record in a key-value format.

For an alias record, the value in ETCD comprises of two parameters: a flag denoting that this record is an alias and the name of this alias record's parent record. RecCaster-ETCD uses a queue-based data buffer to store write tasks for each record. On starting, RecCaster-ETCD spawns a producer thread and multiple consumer threads to increase performance by sending parallel write requests to ETCD server. The number of required consumer threads is configurable using variable *recMaxNumThreads* which has a default value of 5. The producer thread writes each key-value pair to this queue and each consumer thread reads the key-value pairs from the queue and performs the write task to ETCD server.

Theory of Operation

Before the RecCaster-ETCD is started, the configuration for the ETCD server has to be provided using an EPICS environment variable ETCD_SERVER. The main thread (also the producer thread) of RecCaster-ETCD checks for ETCD_SERVER environment variable and throws error if this variable is missing or in wrong format. It also checks for connectivity with the provided ETCD server and retries the checking of connection indefinitely till it establishes a connection.

On establishing a connection with ETCD server, it asks ETCD server to grant it a lease for a certain time period *recEtcdLeasePeriod*. This parameter is configurable and has a default value of 15 minutes. In case granting of lease is unsuccessful, the main thread keeps re-trying to get a lease granted. After the lease is granted, the main (or producer) thread prepares a write task for each IOC record by forming its key-value pair with an associated lease. These write tasks are pushed to the data queue where the consumer threads retrieve these tasks and execute them.

Consumer threads use a gRPC-based API from the ETCDv3 C++ client library [10] to place write requests of the record's key-value pair to ETCD. In case any of these calls fail or return an error value, consumer threads stop sending any new requests and signal the main thread to start a fresh upload of records to ETCD with a new lease identifier. This transaction-like write operation makes sure that ETCD always has either all IOC records or none to avoid reflecting any intermediate IOC state to clients like ChannelFinder.

After all records have been successfully uploaded, consumer threads enter a suspended state and the main thread periodically sends requests to refresh the lease to ETCD server to keep the records alive in ETCD. At this point, if a user exits the IOC, the main thread sends a request to ETCD server to revoke the lease for this IOC which further deletes all the IOC records from ETCD server. Debugging logs for RecCaster-ETCD can be configured to be turned on by setting the variable *recEtcdDebugMode* to either 1, 2 or 3 which represent the different logging levels. The default value of this variable is 0 (or turned off). Figure 2 summarizes the communication between RecCaster-ETCD client and ETCD server.

Failure Handling

There are multiple failure scenarios possible when either the ETCD client (i.e. RecCaster-ETCD) or the ETCD server itself shuts down during operation and takes some time to restore their state. Here we discuss three such scenarios and how RecCaster-ETCD handles each of these.

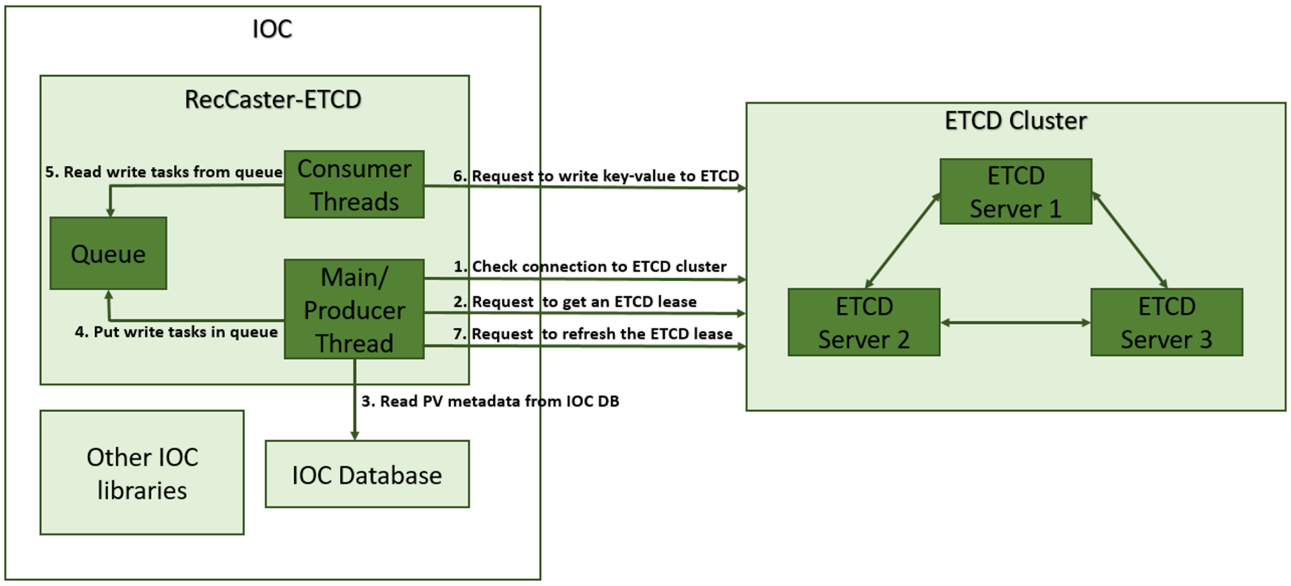
IOC
RecCaster-ETCD
5. Read write tasks from queue
Consumer Threads
Queue
Main/ Producer Thread
4. Put write tasks in queue
6. Request to write key-value to ETCD
1. Check connection to ETCD cluster
2. Request to get an ETCD lease
7. Request to refresh the ETCD lease
3. Read PV metadata from IOC DB
Other IOC libraries
IOC Database
ETCD Cluster
ETCD Server 1
ETCD Server 2
ETCD Server 3

Figure 2: Communication between RecCaster-ETCD and a 3-node ETCD cluster – Each interaction is numbered 1-7 in a chronological order.

- When an IOC goes down unexpectedly - If the IOC running RecCaster-ETCD reboots, then it restarts the RecCaster-ETCD and consequently a new ETCD lease is granted and all the records are written back to ETCD server with the new lease identifier. During the time period an IOC is down, records from that IOC remain in the ETCD server till the lease runs out. Once the lease expires, the records get deleted and have to be rewritten by re-starting the IOC.
- When an IOC disconnects from ETCD server – If there is a network interruption between the IOC and ETCD server, RecCaster-ETCD will not be able to refresh the ETCD lease for this IOC. If the network interruption gets corrected within the remaining lease time, the lease will get refreshed in the next try. In case the lease has expired when the network interruption is resolved, RecCaster-ETCD gets a new lease granted and re-uploads all records to ETCD. If the network interruption occurs while the writes are in progress, RecCaster-ETCD waits for the network issues to be resolved and then re-uploads all the records.
- When ETCD server goes down – the ETCD server can handle temporary reboots by saving all records and leases on its hard disk. After the server is restored, the leases can be refreshed successfully by the RecCaster-ETCD and no records will be lost. In case the server goes down while the writes are in progress, RecCaster-ETCD tries indefinitely to connect back to ETCD server and re-uploads all records on a successful re-connection.

SUMMARY

ETCD has been found to be a fault-tolerant, secure and off-the-shelf alternative to the existing custom-built RecCeiver. Its exhaustive list of APIs provide an easy way for the RecCaster-ETCD clients to use them in order to save the record metadata to ETCD storage and keep them alive for the time period the IOC is alive. RecCaster-ETCD implements a transaction-style upload of all record metadata of an IOC by which the IOC state in ETCD is never intermediate but always either complete or absent. ETCD leases are a useful feature that RecCaster-ETCD uses to keep data alive in ETCD store and to automatically delete an IOC's data from ETCD when the IOC gets disconnected. ETCD, when used in a multi-node cluster provides the fault-tolerance benefits where the RecCeiver service will be up till the majority of ETCD nodes are up. With ETCD's role-based authentication, it is possible to have only verified IOCs upload their record metadata to a verified ETCD server using secure TLS protocol.

FUTURE WORK

The future work for RecSyncETCD involves the development of a new ChannelFinder which reads data from ETCD and provides this data as RESTful APIs to other clients. This new ChannelFinder will use data-pull method to get its data from the ETCD server instead of the existing data-push method where the RecCeiver pushes all its data to the ChannelFinder. Another important future work activity involves adding the authentication and security feature between ETCD server and the RecCaster-ETCD client. This will include configuring ETCD server with role-based authentication to provide access to only the permitted IOCs to write their records to ETCD. Also, both the ETCD server and RecCaster-ETCD will exchange and verify CA certificates to make sure communication happens only among whitelisted servers and clients.

REFERENCES

[1] https://github.com/ChannelFinder/recsync

[2] https://github.com/ChannelFinder/ChannelFinderService

[3] ETCD, https://etcd.io/

[4] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm", USENIX, 2014.

[5] https://en.wikipedia.org/wiki/B%2B_tree

[6] https://en.wikipedia.org/wiki/B-tree

[7] Prometheus, https://prometheus.io/

[8] Grafana, https://grafana.com/

[9] gRPC, https://grpc.io/docs/

[10] https://github.com/nokia/etcd-cpp-apiv3