v3.4 docs

These docs cover everything from setting up and running an etcd cluster to using etcd in applications.

- Quickstart
- Overview
- Demo
- Install
- FAQ
- Libraries and tools
- Metrics
- Reporting bugs
- Tuning
- Discovery service protocol
- Logging conventions
- Learning
- Developer guide
- Operations guide
- Benchmarks
- Upgrading
- Platforms
- Triage

Last modified September 12, 2024: [v3.4] Update installation version to latest tag (v3.4.34) (5afe4d7)

Quickstart

Get etcd up and running in less than 5 minutes!

Follow these instructions to locally install, run, and test a single-member cluster of etcd:

1. Install etcd from pre-built binaries or from source. For details, see Install.

> **Important**: Ensure that you perform the last step of the installation instructions to verify that `etcd` is in your path.

2. Launch `etcd`:

```
$ etcd --logger=zap
{"level":"info","ts":"2021-09-20T08:19:31.340-0400","caller":"etcdmain/etcd.go:110",
⋮
```

> **Note**: The output produced by `etcd` are logs — info-level logs can be ignored.

3. From **another terminal**, use `etcdctl` to set a key:

```
$ etcdctl put greeting "Hello, etcd"
OK
```

4. From the same terminal, retrieve the key:

```
$ etcdctl get greeting
greeting
Hello, etcd
```

What's next?

Learn about more ways to configure and use etcd from the following pages:

- Explore the gRPC API.
- Set up a multi-machine cluster.
- Learn how to configure etcd.
- Find language bindings and tools.
- Use TLS to secure an etcd cluster.
- Tune etcd.

Last modified September 20, 2021: Quickstart v3.4: backport of v3.5 changes (#488) (fe845c7)

Overview

etcd API design principles

The etcd v3 API is designed to give users a more efficient and cleaner abstraction compared to etcd v2. There are a number of semantic and protocol changes in this new API.

To prove out the design of the v3 API the team has also built a number of example recipes, there is a video discussing these recipes too.

Design

1. Flatten binary key-value space
2. Keep the event history until compaction
    - access to old version of keys
    - user controlled history compaction
3. Support range query
    - Pagination support with limit argument
    - Support consistency guarantee across multiple range queries
4. Replace TTL key with Lease
    - more efficient/ low cost keep alive
    - a logical group of TTL keys
5. Replace CAS/CAD with multi-object Txn
    - MUCH MORE powerful and flexible
6. Support efficient watching with multiple ranges
7. RPC API supports the completed set of APIs.
    - more efficient than JSON/HTTP
    - additional txn/lease support
8. HTTP API supports a subset of APIs.
    - easy for people to try out etcd

- easy for people to write simple etcd application

Notes

Request Size Limitation

The max request size is around 1MB. Since etcd replicates requests in a streaming fashion, a very large request might block other requests for a long time. The use case for etcd is to store small configuration values, so we prevent user from submitting large requests. This also applies to Txn requests. We might loosen the size in the future a little bit or make it configurable.

Protobuf Defined API

[api protobuf]

[kv protobuf]

Examples

Put a key (foo=bar)

```
// A put is always successful
Put( PutRequest { key = foo, value = bar } )

PutResponse {
    cluster_id = 0x1000,
    member_id = 0x1,
    revision = 1,
    raft_term = 0x1,
}
```

Get a key (assume we have foo=bar)

```
Get ( RangeRequest { key = foo } )

RangeResponse {
    cluster_id = 0x1000,
    member_id = 0x1,
    revision = 1,
    raft_term = 0x1,
```

```
    kvs = {
      {
          key = foo,
          value = bar,
          create_revision = 1,
          mod_revision = 1,
          version = 1;
      },
    },
}
```

Range over a key space (assume we have foo0=bar0… foo100=bar100)

```
Range ( RangeRequest { key = foo, end_key = foo80, limit = 30  } )

RangeResponse {
    cluster_id = 0x1000,
    member_id = 0x1,
    revision = 100,
    raft_term = 0x1,
    kvs = {
      {
          key = foo0,
          value = bar0,
          create_revision = 1,
          mod_revision = 1,
          version = 1;
      },
         ...,
      {
          key = foo30,
          value = bar30,
          create_revision = 30,
          mod_revision = 30,
          version = 1;
      },
    },
}
```

Finish a txn (assume we have foo0=bar0, foo1=bar1)

```
Txn(TxnRequest {
    // mod_revision of foo0 is equal to 1, mod_revision of foo1 is greater than 1
    compare = {
```

```
        {compareType = equal, key = foo0, mod_revision = 1},
        {compareType = greater, key = foo1, mod_revision = 1}}
    },
    // if the comparison succeeds, put foo2 = bar2
    success = {PutRequest { key = foo2, value = success }},
    // if the comparison fails, put foo2=fail
    failure = {PutRequest { key = foo2, value = failure }},
)

TxnResponse {
    cluster_id = 0x1000,
    member_id = 0x1,
    revision = 3,
    raft_term = 0x1,
    succeeded = true,
    responses = {
      // response of PUT foo2=success
      {
            cluster_id = 0x1000,
            member_id = 0x1,
            revision = 3,
            raft_term = 0x1,
        }
    }
}
```

Watch on a key/range

```
Watch( WatchRequest{
           key = foo,
           end_key = fop, // prefix foo
           start_revision = 20,
           end_revision = 10000,
           // server decided notification frequency
           progress_notification = true,
       }
       … // this can be a watch request stream
      )

// put (foo0=bar0) event at 3
WatchResponse {
    cluster_id = 0x1000,
    member_id = 0x1,
    revision = 3,
    raft_term = 0x1,
    event_type = put,
    kv = {
```

```
                key = foo0,
                value = bar0,
                create_revision = 1,
                mod_revision = 1,
                version = 1;
            },
      }
      …

      // a notification at 2000
      WatchResponse {
          cluster_id = 0x1000,
          member_id = 0x1,
          revision = 2000,
          raft_term = 0x1,
          // nil event as notification
      }

      …

      // put (foo0=bar3000) event at 3000
      WatchResponse {
          cluster_id = 0x1000,
          member_id = 0x1,
          revision = 3000,
          raft_term = 0x1,
          event_type = put,
          kv = {
                  key = foo0,
                  value = bar3000,
                  create_revision = 1,
                  mod_revision = 3000,
                  version = 2;
            },
      }
      …
```

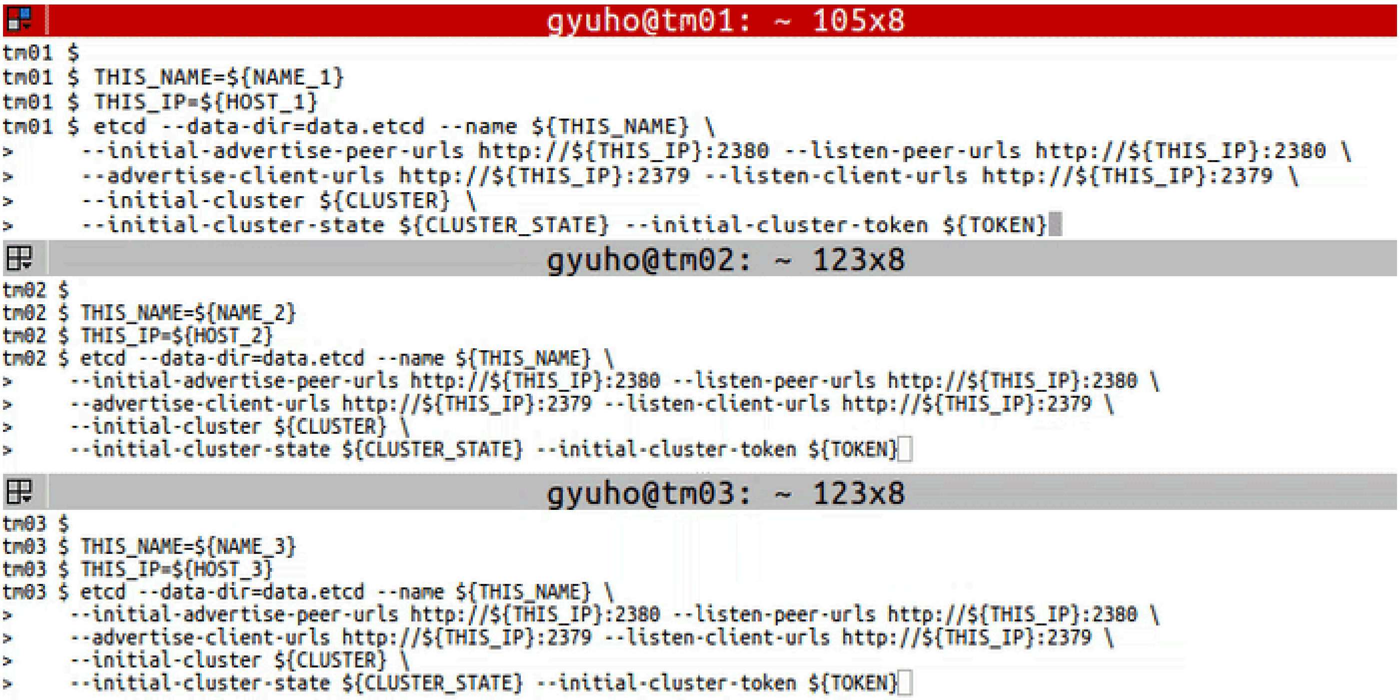
Last modified May 21, 2021: Removing link to v3api overview video as it is no longer available (#297) (2be16f6)

Demo

Procedures for working with an etcd cluster

This series of examples shows the basic procedures for working with an etcd cluster.

Set up a cluster



On each etcd node, specify the cluster members:

```
TOKEN=token-01
CLUSTER_STATE=new
NAME_1=machine-1
NAME_2=machine-2
NAME_3=machine-3
HOST_1=10.240.0.17
HOST_2=10.240.0.18
HOST_3=10.240.0.19
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_3}=http:/
```

Run this on each machine:

```
# For machine 1
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For machine 2
THIS_NAME=${NAME_2}
THIS_IP=${HOST_2}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For machine 3
THIS_NAME=${NAME_3}
THIS_IP=${HOST_3}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

Or use our public discovery service:

```
curl 'https://discovery.etcd.io/new?size=3'
https://discovery.etcd.io/a81b5818e67a6ea83e9d4daea5ecbc92

# grab this token
TOKEN=token-01
CLUSTER_STATE=new
NAME_1=machine-1
NAME_2=machine-2
NAME_3=machine-3
HOST_1=10.240.0.17
HOST_2=10.240.0.18
HOST_3=10.240.0.19
DISCOVERY=https://discovery.etcd.io/a81b5818e67a6ea83e9d4daea5ecbc92
```

```
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --discovery ${DISCOVERY} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

THIS_NAME=${NAME_2}
THIS_IP=${HOST_2}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --discovery ${DISCOVERY} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

THIS_NAME=${NAME_3}
THIS_IP=${HOST_3}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://${THI
    --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://${THIS_IP
    --discovery ${DISCOVERY} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

Now etcd is ready! To connect to etcd with etcdctl:

```
export ETCDCTL_API=3
HOST_1=10.240.0.17
HOST_2=10.240.0.18
HOST_3=10.240.0.19
ENDPOINTS=$HOST_1:2379,$HOST_2:2379,$HOST_3:2379

etcdctl --endpoints=$ENDPOINTS member list
```

Access etcd

```
tx01 $ etcdctl --endpoints=$ENDPOINTS
```

`put` command to write:

```
etcdctl --endpoints=$ENDPOINTS put foo "Hello World!"
```

`get` to read from etcd:

```
etcdctl --endpoints=$ENDPOINTS get foo
etcdctl --endpoints=$ENDPOINTS --write-out="json" get foo
```

Get by prefix

```
tx01 $ etcdctl --endpoints=$ENDPOINTS put web1 value1
OK
tx01 $ etcdctl --endpoints=$ENDPOINTS put web2 value2
OK
tx01 $ etcdctl --endpoints=$ENDPOINTS put web3 value3
OK
tx01 $ etcdctl --endpoints=$ENDPOINTS
```

```
etcdctl --endpoints=$ENDPOINTS put web1 value1
etcdctl --endpoints=$ENDPOINTS put web2 value2
etcdctl --endpoints=$ENDPOINTS put web3 value3

etcdctl --endpoints=$ENDPOINTS get web --prefix
```

Delete

```
tx01 $ etcdctl --endpoints=$ENDPOINTS put key myvalue
OK
tx01 $ etcdctl --endpoints=$ENDPOINTS de_
```

```
etcdctl --endpoints=$ENDPOINTS put key myvalue
etcdctl --endpoints=$ENDPOINTS del key

etcdctl --endpoints=$ENDPOINTS put k1 value1
etcdctl --endpoints=$ENDPOINTS put k2 value2
etcdctl --endpoints=$ENDPOINTS del k --prefix
```

Transactional write

`txn` to wrap multiple requests into one transaction:

```
tx01 $ etcdctl --endpoints=$ENDPOINTS put user1 bad
OK
tx01 $ etcdctl --endpoints=$ENDPOINTS _
```

```
etcdctl --endpoints=$ENDPOINTS put user1 bad
etcdctl --endpoints=$ENDPOINTS txn --interactive
```

```
compares:
value("user1") = "bad"

success requests (get, put, delete):
del user1

failure requests (get, put, delete):
put user1 good
```

Watch

`watch` to get notified of future changes:

```
tx01 $ etcdctl --endpoints=$ENDPOINTS
```

gyuho@tx01: ~ 67x6

```
tx01 $ etcdctl --endpoints=$ENDPOINTS _
```

```
etcdctl --endpoints=$ENDPOINTS watch stock1
etcdctl --endpoints=$ENDPOINTS put stock1 1000

etcdctl --endpoints=$ENDPOINTS watch stock --prefix
etcdctl --endpoints=$ENDPOINTS put stock1 10
etcdctl --endpoints=$ENDPOINTS put stock2 20
```

Lease

`lease` to write with TTL:

```
tx01 $ etcdctl --endpoints=$ENDPOINTS _
```

```
etcdctl --endpoints=$ENDPOINTS lease grant 300
# lease 2be7547fbc6a5afa granted with TTL(300s)

etcdctl --endpoints=$ENDPOINTS put sample value --lease=2be7547fbc6a5afa
etcdctl --endpoints=$ENDPOINTS get sample

etcdctl --endpoints=$ENDPOINTS lease keep-alive 2be7547fbc6a5afa
etcdctl --endpoints=$ENDPOINTS lease revoke 2be7547fbc6a5afa
# or after 300 seconds
etcdctl --endpoints=$ENDPOINTS get sample
```

Distributed locks

`lock` for distributed lock:

```
tx01 $ etcdctl --endpoints=$ENDPOINTS
```

gyuho@tx01: ~ 67x6

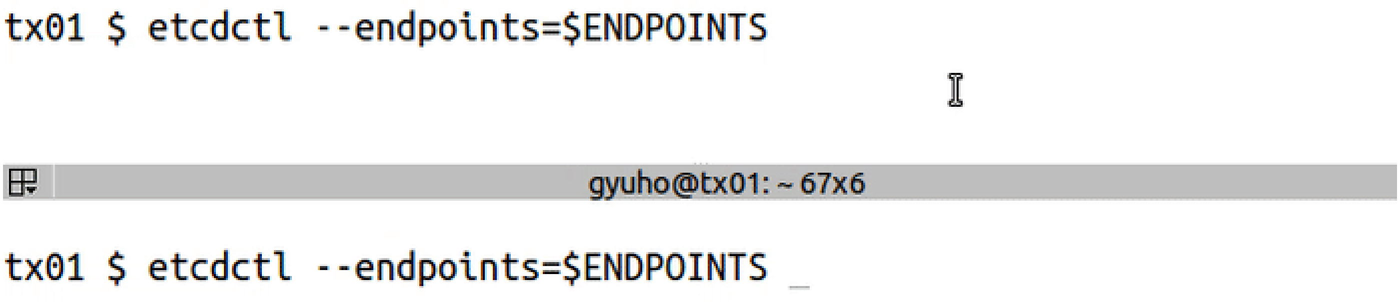
```
tx01 $ etcdctl --endpoints=$ENDPOINTS _
```

```
etcdctl --endpoints=$ENDPOINTS lock mutex1

# another client with the same name blocks
```

```
etcdctl --endpoints=$ENDPOINTS lock mutex1
```

Elections

`elect` for leader election:




```
etcdctl --endpoints=$ENDPOINTS elect one p1

# another client with the same name blocks
etcdctl --endpoints=$ENDPOINTS elect one p2
```

Cluster status

Specify the initial cluster configuration for each machine:

```
etcdctl --write-out=table --endpoints=$ENDPOINTS endpoint status

+-----------------+------------------+---------+---------+-----------+------------+----
|    ENDPOINT     |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAF
+-----------------+------------------+---------+---------+-----------+------------+----
| 10.240.0.17:2379 | 4917a7ab173fabe7 |  3.4.15 |   45 kB |      true |      false |
| 10.240.0.18:2379 | 59796ba9cd1bcd72 |  3.4.15 |   45 kB |     false |      false |
| 10.240.0.19:2379 | 94df724b66343e6c |  3.4.15 |   45 kB |     false |      false |
+-----------------+------------------+---------+---------+-----------+------------+----
```

```
etcdctl --endpoints=$ENDPOINTS endpoint health
```

```
10.240.0.17:2379 is healthy: successfully committed proposal: took = 3.345431ms
10.240.0.19:2379 is healthy: successfully committed proposal: took = 3.767967ms
10.240.0.18:2379 is healthy: successfully committed proposal: took = 4.025451ms
```

Snapshot

`snapshot` to save point-in-time snapshot of etcd database:

```
tx01 $ etcdctl --endpoints=$ENDPOINTS snapshot
```

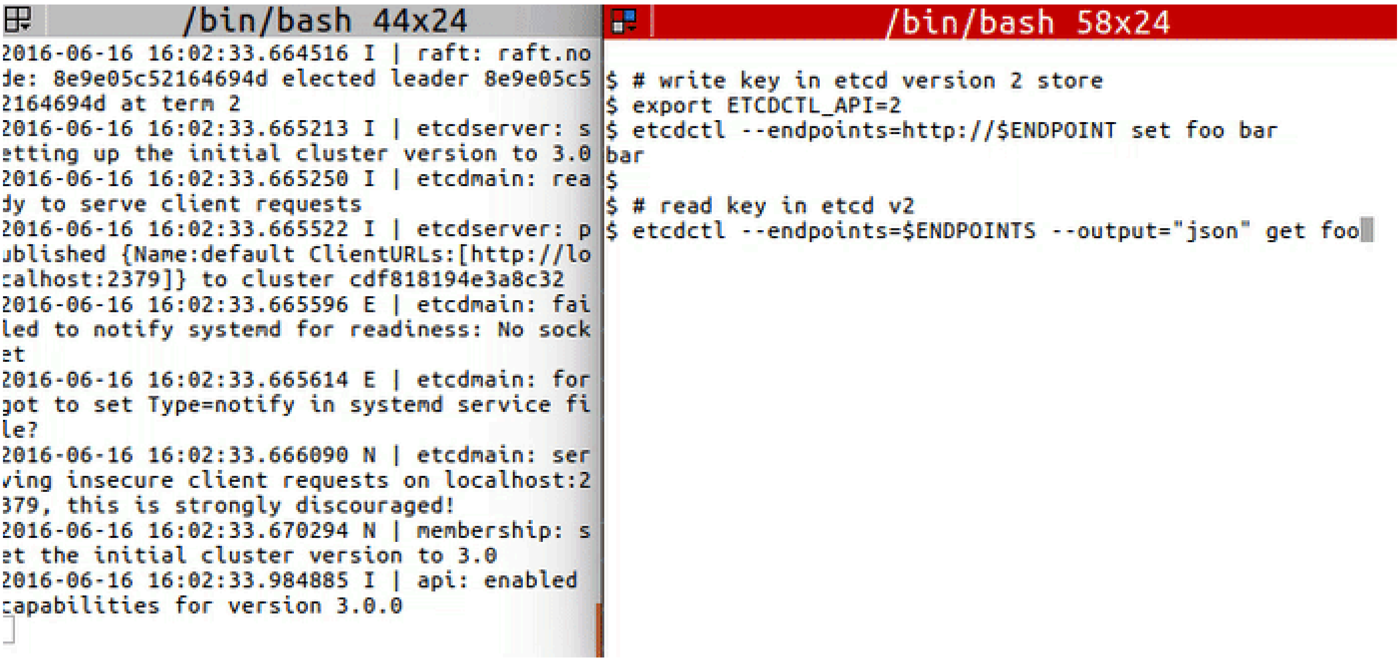
Snapshot can only be requested from one etcd node, so `--endpoints` flag should contain only one endpoint.

```
ENDPOINTS=$HOST_1:2379
etcdctl --endpoints=$ENDPOINTS snapshot save my.db

Snapshot saved at my.db
```

```
etcdctl --write-out=table --endpoints=$ENDPOINTS snapshot status my.db

+---------+----------+------------+------------+
|  HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+---------+----------+------------+------------+
| c55e8b8 |        9 |         13 | 25 kB      |
+---------+----------+------------+------------+
```

Migrate

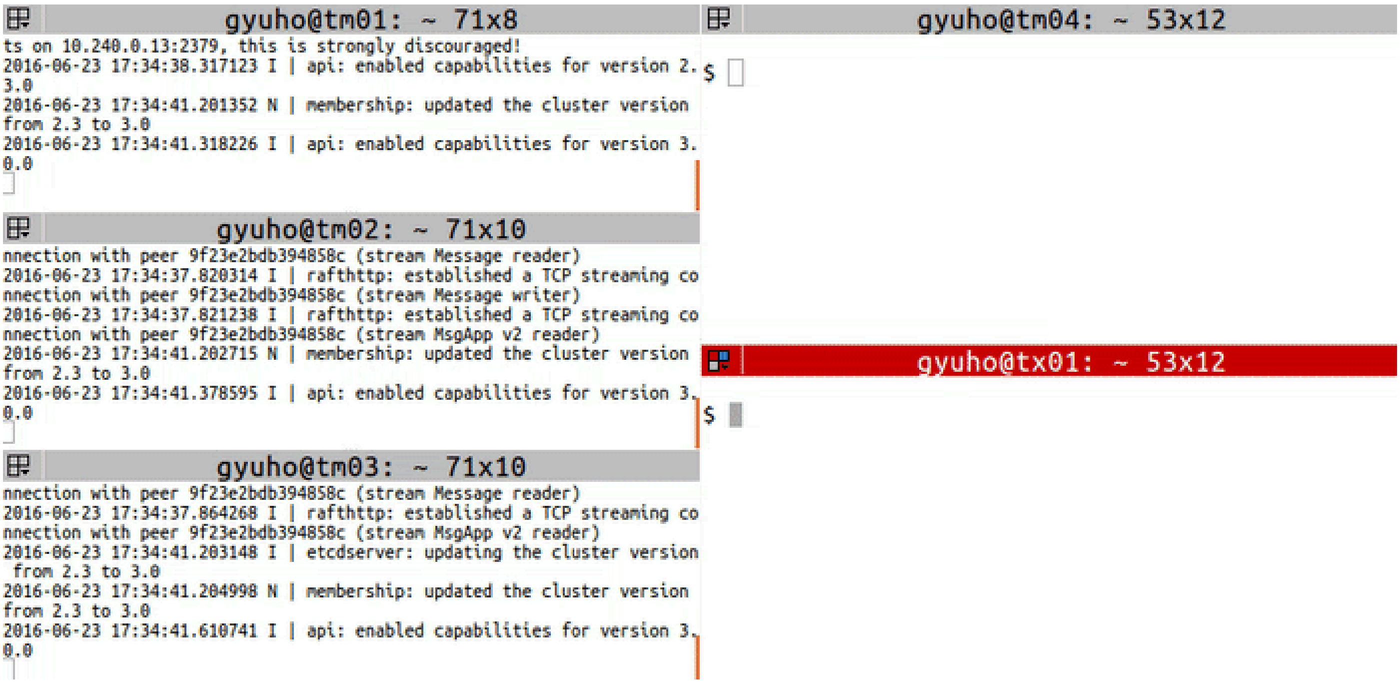
`migrate` to transform etcd v2 to v3 data:




```
# write key in etcd version 2 store
export ETCDCTL_API=2
etcdctl --endpoints=http://$ENDPOINT set foo bar

# read key in etcd v2
etcdctl --endpoints=$ENDPOINTS --output="json" get foo

# stop etcd node to migrate, one by one

# migrate v2 data
export ETCDCTL_API=3
etcdctl --endpoints=$ENDPOINT migrate --data-dir="default.etcd" --wal-dir="default.etcd/

# restart etcd node after migrate, one by one

# confirm that the key got migrated
etcdctl --endpoints=$ENDPOINTS get /foo
```

Member

`member` to add,remove,update membership:




```
# For each machine
TOKEN=my-etcd-token-1
CLUSTER_STATE=new
NAME_1=etcd-node-1
NAME_2=etcd-node-2
NAME_3=etcd-node-3
HOST_1=10.240.0.13
HOST_2=10.240.0.14
HOST_3=10.240.0.15
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_3}=http:/

# For node 1
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 \
    --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 \
    --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} \
    --initial-cluster-token ${TOKEN}

# For node 2
THIS_NAME=${NAME_2}
THIS_IP=${HOST_2}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 \
    --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 \
    --listen-client-urls http://${THIS_IP}:2379 \
```

```
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} \
    --initial-cluster-token ${TOKEN}

# For node 3
THIS_NAME=${NAME_3}
THIS_IP=${HOST_3}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 \
    --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 \
    --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} \
    --initial-cluster-token ${TOKEN}
```

Then replace a member with `member remove` and `member add` commands:

```
# get member ID
export ETCDCTL_API=3
HOST_1=10.240.0.13
HOST_2=10.240.0.14
HOST_3=10.240.0.15
etcdctl --endpoints=${HOST_1}:2379,${HOST_2}:2379,${HOST_3}:2379 member list

# remove the member
MEMBER_ID=278c654c9a6dfd3b
etcdctl --endpoints=${HOST_1}:2379,${HOST_2}:2379,${HOST_3}:2379 \
    member remove ${MEMBER_ID}

# add a new member (node 4)
export ETCDCTL_API=3
NAME_1=etcd-node-1
NAME_2=etcd-node-2
NAME_4=etcd-node-4
HOST_1=10.240.0.13
HOST_2=10.240.0.14
HOST_4=10.240.0.16 # new member
etcdctl --endpoints=${HOST_1}:2379,${HOST_2}:2379 \
    member add ${NAME_4} \
    --peer-urls=http://${HOST_4}:2380
```

Next, start the new member with `--initial-cluster-state existing` flag:

```
# [WARNING] If the new member starts from the same disk space,
# make sure to remove the data directory of the old member
#
# restart with 'existing' flag
TOKEN=my-etcd-token-1
CLUSTER_STATE=existing
NAME_1=etcd-node-1
NAME_2=etcd-node-2
NAME_4=etcd-node-4
HOST_1=10.240.0.13
HOST_2=10.240.0.14
HOST_4=10.240.0.16 # new member
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_4}=http:/

THIS_NAME=${NAME_4}
THIS_IP=${HOST_4}
etcd --data-dir=data.etcd --name ${THIS_NAME} \
    --initial-advertise-peer-urls http://${THIS_IP}:2380 \
    --listen-peer-urls http://${THIS_IP}:2380 \
    --advertise-client-urls http://${THIS_IP}:2379 \
    --listen-client-urls http://${THIS_IP}:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} \
    --initial-cluster-token ${TOKEN}
```

Auth

`auth`, `user`, `role` for authentication:

```
export ETCDCTL_API=3
ENDPOINTS=localhost:2379

etcdctl --endpoints=${ENDPOINTS} role add root
etcdctl --endpoints=${ENDPOINTS} role get root

etcdctl --endpoints=${ENDPOINTS} user add root
etcdctl --endpoints=${ENDPOINTS} user grant-role root root
etcdctl --endpoints=${ENDPOINTS} user get root

etcdctl --endpoints=${ENDPOINTS} role add role0
etcdctl --endpoints=${ENDPOINTS} role grant-permission role0 readwrite foo
etcdctl --endpoints=${ENDPOINTS} user add user0
etcdctl --endpoints=${ENDPOINTS} user grant-role user0 role0

etcdctl --endpoints=${ENDPOINTS} auth enable
# now all client requests go through auth
```

```
etcdctl --endpoints=${ENDPOINTS} --user=user0:123 put foo bar
etcdctl --endpoints=${ENDPOINTS} get foo
# permission denied, user name is empty because the request does not issue an authentica
etcdctl --endpoints=${ENDPOINTS} --user=user0:123 get foo
# user0 can read the key foo
etcdctl --endpoints=${ENDPOINTS} --user=user0:123 get foo1
```

Last modified August 25, 2021: [Remove the outdated gif screenshot and refine the behavior of auth (#446) (6d9af72)]

Install

Instructions for installing etcd from pre-built binaries or from source.

Requirements

Before installing etcd, see the following pages:

- Supported platforms
- Hardware recommendations

Install pre-built binaries

The easiest way to install etcd is from pre-built binaries:

1. Download the compressed archive file for your platform from Releases, choosing release v3.4.34 or later.

2. Unpack the archive file. This results in a directory containing the binaries.

3. Add the executable binaries to your path. For example, rename and/or move the binaries to a directory in your path (like `/usr/local/bin`), or add the directory created by the previous step to your path.

4. From a shell, test that `etcd` is in your path:

```
$ etcd --version
etcd Version: 3.4.34
...
```

Build from source

If you have Go version 1.2+, you can build etcd from source by following these steps:

1. Download the etcd repo as a zip file and unzip it, or clone the repo using the following command.

```
$ git clone -b v3.4.34 https://github.com/etcd-io/etcd.git
```

   To build from `main@HEAD`, omit the `-b v3.4.34` flag.

2. Change directory:

```
$ cd etcd
```

3. Run the build script:

```
$ ./build
```

   The binaries are under the `bin` directory.

4. Add the full path to the `bin` directory to your path, for example:

```
$ export PATH="$PATH:`pwd`/bin"
```

5. Test that `etcd` is in your path:

```
$ etcd --version
```

Installation check

For a slightly more involved sanity check of your installation, see Quickstart.

Last modified December 17, 2023: Fix required golang version to build from source. (ba02660)

FAQ

Frequently asked questions

etcd, general

Do clients have to send requests to the etcd leader?

Raft is leader-based; the leader handles all client requests which need cluster consensus. However, the client does not need to know which node is the leader. Any request that requires consensus sent to a follower is automatically forwarded to the leader. Requests that do not require consensus (e.g., serialized reads) can be processed by any cluster member.

Configuration

What is the difference between listen-<client,peer>-urls, advertise-client-urls or initial-advertise-peer-urls?

`listen-client-urls` and `listen-peer-urls` specify the local addresses etcd server binds to for accepting incoming connections. To listen on a port for all interfaces, specify `0.0.0.0` as the listen IP address.

`advertise-client-urls` and `initial-advertise-peer-urls` specify the addresses etcd clients or other etcd members should use to contact the etcd server. The advertise addresses must be reachable from the remote machines. Do not advertise addresses like `localhost` or `0.0.0.0` for a production setup since these addresses are unreachable from remote machines.

Why doesn't changing `--listen-peer-urls` or `--initial-advertise-peer-urls` update the advertised peer URLs in `etcdctl member list`?

A member's advertised peer URLs come from `--initial-advertise-peer-urls` on initial cluster boot. Changing the listen peer URLs or the initial advertise peers after booting the member won't affect the exported advertise peer URLs since changes must go through quorum to avoid membership configuration split brain. Use `etcdctl member update` to update a member's peer URLs.

Deployment

System requirements

Since etcd writes data to disk, its performance strongly depends on disk performance. For this reason, SSD is highly recommended. To assess whether a disk is fast enough for etcd, one possibility is using a disk benchmarking tool such as fio. For an example on how to do that, read here. To prevent performance degradation or unintentionally overloading the key-value store, etcd enforces a configurable storage size quota set to 2GB by default. To avoid swapping or running out of memory, the machine should have at least as much RAM to cover the quota. 8GB is a suggested maximum size for normal environments and etcd warns at startup if the configured value exceeds it. At CoreOS, an etcd cluster is usually deployed on dedicated CoreOS Container Linux machines with dual-core processors, 2GB of RAM, and 80GB of SSD *at the very least*. **Note that performance is intrinsically workload dependent; please test before production deployment**. See hardware for more recommendations.

Most stable production environment is Linux operating system with amd64 architecture; see supported platform for more.

Why an odd number of cluster members?

An etcd cluster needs a majority of nodes, a quorum, to agree on updates to the cluster state. For a cluster with n members, quorum is (n/2)+1. For any odd-sized cluster, adding one node will always increase the number of nodes necessary for quorum. Although adding a node to an odd-sized cluster appears better since there are more machines, the fault tolerance is worse since exactly the same number of nodes may fail without losing quorum but there are more nodes that can fail. If the cluster is in a state where it can't tolerate any more failures, adding a node before removing nodes is dangerous because if the new node fails to register with the cluster (e.g., the address is misconfigured), quorum will be permanently lost.

What is maximum cluster size?

Theoretically, there is no hard limit. However, an etcd cluster probably should have no more than seven nodes. Google Chubby lock service, similar to etcd and widely deployed within Google for many years, suggests running five nodes. A 5-member etcd cluster can tolerate two member failures, which is enough in most cases. Although larger clusters provide better fault tolerance, the write performance suffers because data must be replicated across more machines.

What is failure tolerance?

An etcd cluster operates so long as a member quorum can be established. If quorum is lost through transient network failures (e.g., partitions), etcd automatically and safely resumes once the network recovers and restores quorum; Raft enforces cluster consistency. For power loss, etcd persists the Raft log to disk; etcd replays the log to the point of failure and resumes cluster participation. For permanent hardware failure, the node may be removed from the cluster through runtime reconfiguration.

It is recommended to have an odd number of members in a cluster. An odd-size cluster tolerates the same number of failures as an even-size cluster but with fewer nodes. The difference can be seen by comparing even and odd sized clusters:

Cluster Size	Majority	Failure Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

Adding a member to bring the size of cluster up to an even number doesn't buy additional fault tolerance. Likewise, during a network partition, an odd number of members guarantees that there will always be a majority partition that can continue to operate and be the source of truth when the partition ends.

Does etcd work in cross-region or cross data center deployments?

Deploying etcd across regions improves etcd's fault tolerance since members are in separate failure domains. The cost is higher consensus request latency from crossing data center boundaries. Since etcd relies on a member quorum for consensus, the latency from crossing data centers will be somewhat pronounced because at least a majority of cluster members must respond to consensus requests. Additionally, cluster data must be replicated across all peers, so there will be bandwidth cost as well.

With longer latencies, the default etcd configuration may cause frequent elections or heartbeat timeouts. See tuning for adjusting timeouts for high latency deployments.

Operation

How to backup a etcd cluster?

etcdctl provides a `snapshot` command to create backups. See backup for more details.

Should I add a member before removing an unhealthy member?

When replacing an etcd node, it's important to remove the member first and then add its replacement.

etcd employs distributed consensus based on a quorum model; (n/2)+1 members, a majority, must agree on a proposal before it can be committed to the cluster. These proposals include key-value updates and membership changes. This model totally avoids any possibility of split brain inconsistency. The consequence of permanent quorum loss is catastrophic.

How this applies to membership: If a 3-member cluster has 1 downed member, it can still make forward progress because the quorum is 2 and 2 members are still live. However, adding a new member to a 3-member cluster will increase the quorum to 3 because 3 votes are required for a majority of 4 members. Since the quorum increased, this extra member buys nothing in terms of fault tolerance; the cluster is still one node failure away from being unrecoverable.

Additionally, that new member is risky because it may turn out to be misconfigured or incapable of joining the cluster. In that case, there's no way to recover quorum because the cluster has two members down and two members up, but needs three votes to change membership to undo the botched membership addition. etcd will by default reject member add attempts that could take down the cluster in this manner.

On the other hand, if the downed member is removed from cluster membership first, the number of members becomes 2 and the quorum remains at 2. Following that removal by adding a new member will also keep the quorum steady at 2. So, even if the new node can't be brought up, it's still possible to remove the new member through quorum on the remaining live members.

Why won't etcd accept my membership changes?

etcd sets `strict-reconfig-check` in order to reject reconfiguration requests that would cause quorum loss. Abandoning quorum is really risky (especially when the cluster is already unhealthy). Although it may be tempting to disable quorum checking if there's quorum loss to add a new member, this could lead to full fledged cluster inconsistency. For many applications, this will make the problem even worse ("disk geometry corruption" being a candidate for most terrifying).

Why does etcd lose its leader from disk latency spikes?

This is intentional; disk latency is part of leader liveness. Suppose the cluster leader takes a minute to fsync a raft log update to disk, but the etcd cluster has a one second election timeout. Even though the leader can process network messages within the election interval (e.g., send heartbeats), it's effectively unavailable because it can't commit any new proposals; it's waiting on the slow disk. If the cluster frequently loses its leader due to disk latencies, try tuning the disk settings or etcd time parameters.

What does the etcd warning "request ignored (cluster ID mismatch)" mean?

Every new etcd cluster generates a new cluster ID based on the initial cluster configuration and a user-provided unique `initial-cluster-token` value. By having unique cluster ID's, etcd is protected from cross-cluster interaction which could corrupt the cluster.

Usually this warning happens after tearing down an old cluster, then reusing some of the peer addresses for the new cluster. If any etcd process from the old cluster is still running it will try to contact the new cluster. The new cluster will recognize a cluster ID mismatch, then ignore the request and emit this warning. This warning is often cleared by ensuring peer addresses among distinct clusters are disjoint.

What does "mvcc: database space exceeded" mean and how do I fix it?

The multi-version concurrency control data model in etcd keeps an exact history of the keyspace. Without periodically compacting this history (e.g., by setting `--auto-compaction`), etcd will eventually exhaust its storage space. If etcd runs low on storage space, it raises a space quota alarm to protect the cluster from further writes. So long as the alarm is raised, etcd responds to write requests with the error `mvcc: database space exceeded`.

To recover from the low space quota alarm:

1. Compact etcd's history.
2. Defragment every etcd endpoint.
3. Disarm the alarm.

What does the etcd warning "etcdserver/api/v3rpc: transport: http2Server.HandleStreams failed to read frame: read tcp 127.0.0.1:2379->127.0.0.1:43020: read: connection reset by peer" mean?

This is gRPC-side warning when a server receives a TCP RST flag with client-side streams being prematurely closed. For example, a client closes its connection, while gRPC server has not yet processed all HTTP/2 frames in the TCP queue. Some data may have been lost in server side, but it is ok so long as client connection has already been closed.

Only old versions of gRPC log this. etcd >=v3.2.13 by default log this with DEBUG level, thus only visible with `--debug` flag enabled.

Performance

How should I benchmark etcd?

Try the benchmark tool. Current benchmark results are available for comparison.

What does the etcd warning "apply entries took too long" mean?

After a majority of etcd members agree to commit a request, each etcd server applies the request to its data store and persists the result to disk. Even with a slow mechanical disk or a virtualized network disk, such as Amazon's EBS or Google's PD, applying a request should normally take fewer than 50 milliseconds. If the average apply duration exceeds 100 milliseconds, etcd will warn that entries are taking too long to apply.

Usually this issue is caused by a slow disk. The disk could be experiencing contention among etcd and other applications, or the disk is too simply slow (e.g., a shared virtualized disk). To rule out a slow disk from causing this warning, monitor backend_commit_duration_seconds (p99 duration should be less than 25ms) to confirm the disk is reasonably fast. If the disk is too slow, assigning a dedicated disk to etcd or using faster disk will typically solve the problem.

The second most common cause is CPU starvation. If monitoring of the machine's CPU usage shows heavy utilization, there may not be enough compute capacity for etcd. Moving etcd to dedicated machine, increasing process resource isolation cgroups, or renicing the etcd server process into a higher priority can usually solve the problem.

Expensive user requests which access too many keys (e.g., fetching the entire keyspace) can also cause long apply latencies. Accessing fewer than a several hundred keys per request, however, should always be performant.

If none of the above suggestions clear the warnings, please open an issue with detailed logging, monitoring, metrics and optionally workload information.

What does the etcd warning "failed to send out heartbeat on time" mean?

etcd uses a leader-based consensus protocol for consistent data replication and log execution. Cluster members elect a single leader, all other members become followers. The elected leader must periodically send heartbeats to its followers to maintain its leadership. Followers infer leader failure if no heartbeats are received within an election interval and trigger an election. If a leader doesn't send its heartbeats in time but is still running, the election is spurious and likely caused by insufficient resources. To catch these soft failures, if the leader skips two heartbeat intervals, etcd will warn it failed to send a heartbeat on time.

Usually this issue is caused by a slow disk. Before the leader sends heartbeats attached with metadata, it may need to persist the metadata to disk. The disk could be experiencing contention among etcd and other applications, or the disk is too simply slow (e.g., a shared virtualized disk). To rule out a slow disk from causing this warning, monitor wal_fsync_duration_seconds (p99 duration should be less than 10ms) to confirm the disk is reasonably fast. If the disk is too slow, assigning a dedicated disk to etcd or using faster disk will typically solve the problem. To tell whether a disk is fast enough for etcd, a benchmarking tool such as fio can be used. Read here for an example.

The second most common cause is CPU starvation. If monitoring of the machine's CPU usage shows heavy utilization, there may not be enough compute capacity for etcd. Moving etcd to dedicated machine, increasing process resource isolation with cgroups, or renicing the etcd server process into a higher priority can usually solve the problem.

A slow network can also cause this issue. If network metrics among the etcd machines shows long latencies or high drop rate, there may not be enough network capacity for etcd. Moving etcd members to a less congested network will typically solve the problem. However, if the etcd cluster is deployed across data centers, long latency between members is expected. For such deployments, tune the `heartbeat-interval` configuration to roughly match the round trip time between the machines, and the `election-timeout` configuration to be at least 5 * `heartbeat-interval`. See tuning documentation for detailed information.

If none of the above suggestions clear the warnings, please open an issue with detailed logging, monitoring, metrics and optionally workload information.

What does the etcd warning "snapshotting is taking more than x seconds to finish ..." mean?

etcd sends a snapshot of its complete key-value store to refresh slow followers and for backups. Slow snapshot transfer times increase MTTR; if the cluster is ingesting data with high throughput, slow followers may livelock by needing a new snapshot before finishing receiving a snapshot. To catch slow snapshot performance, etcd warns when sending a snapshot takes more than thirty seconds and exceeds the expected transfer time for a 1Gbps connection.

Last modified May 14, 2024: fix broken url in faq.md (8af244d)

Libraries and tools

A listing of etcd tools and client libraries

Note that third-party libraries and tools (not hosted on https://github.com/etcd-io) mentioned below are not tested or maintained by the etcd team. Before using them, users are recommended to read and investigate them.

Tools

- etcdctl - A command line client for etcd
- etcd-dump - Command line utility for dumping/restoring etcd.
- etcd-fs - FUSE filesystem for etcd
- etcddir - Realtime sync etcd and local directory. Work with windows and linux.
- etcd-browser - A web-based key/value editor for etcd using AngularJS
- etcd-lock - Master election & distributed r/w lock implementation using etcd - Supports v2
- etcd-console - A web-base key/value editor for etcd using PHP
- etcd-viewer - An etcd key-value store editor/viewer written in Java
- etcdtool - Export/Import/Edit etcd directory as JSON/YAML/TOML and Validate directory using JSON schema
- etcdloadtest - A command line load test client for etcd version 3.0 and above.
- lucas - A web-based key-value viewer for kubernetes etcd3.0+ cluster.
- etcd-manager - A modern, efficient, multi-platform and free etcd 3.x GUI & client tool. Available for Windows, Linux and Mac.
- etcd-backup-restore - Utility to periodically and incrementally backup and restore the etcd.
- etcd-druid - A Kubernetes operator to deploy etcd clusters and manage day-2 operations.
- etcdadm - A command-line tool for operating an etcd cluster.
- etcdhelper - An intellij platform plugin for etcd.

Go libraries

- etcd/client/v3 - the officially maintained Go client for v3
- etcd/client/v2 - the officially maintained Go client for v2

- go-etcd - the deprecated official client. May be useful for older (<2.0.0) versions of etcd.
- encWrapper - encWrapper is an encryption wrapper for the etcd client Keys API/KV.

Java libraries

- coreos/jetcd - Supports v3
- boonproject/etcd - Supports v2, Async/Sync and waits
- justinsb/jetcd
- diwakergupta/jetcd - Supports v2
- jurmous/etcd4j - Supports v2, Async/Sync, waits and SSL
- AdoHe/etcd4j - Supports v2 (enhance for real production cluster)
- cdancy/etcd-rest - Uses jclouds to provide a complete implementation of v2 API.

Scala libraries

- maciej/etcd-client - Supports v2. Akka HTTP-based fully async client
- eiipii/etcdhttpclient - Supports v2. Async HTTP client based on Netty and Scala Futures.
- mingchuno/etcd4s - Supports v3 using gRPC with optional Akka Stream support.

Perl libraries

- hexfusion/perl-net-etcd - Supports v3 grpc gateway HTTP API
- robn/p5-etcd - Supports v2

Python libraries

- kragniz/python-etcd3 - Client for v3
- jplana/python-etcd - Supports v2
- russellhaering/txetcd - a Twisted Python library
- cholcombe973/autodock - A docker deployment automation tool
- lisael/aioetcd - (Python 3.4+) Asyncio coroutines client (Supports v2)
- txaio-etcd - Asynchronous etcd v3-only client library for Twisted (today) and asyncio (future)
- dims/etcd3-gateway - etcd v3 API library using the HTTP grpc gateway
- aioetcd3 - (Python 3.6+) etcd v3 API for asyncio
- Revolution1/etcd3-py - (python2.7 and python3.5+) Python client for etcd v3, using gRPC-JSON-Gateway

Node libraries

- microsoft/etcd3 - Supports v3
- stianeikeland/node-etcd - Supports v2 (w Coffeescript)
- lavagetto/nodejs-etcd - Supports v2

- [deedubs/node-etcd-config] - Supports v2

Ruby libraries

- [iconara/etcd-rb]
- [jpfuentes2/etcd-ruby]
- [ranjib/etcd-ruby] - Supports v2
- [davissp14/etcdv3-ruby] - Supports v3

C libraries

- [apache/celix/etcdlib] - Supports v2
- [jdarcy/etcd-api] - Supports v2
- [shafreeck/cetcd] - Supports v2

C++ libraries

- [edwardcapriolo/etcdcpp] - Supports v2
- [suryanathan/etcdcpp] - Supports v2 (with waits)
- [nokia/etcd-cpp-api] - Supports v2
- [nokia/etcd-cpp-apiv3] - Supports v3

Clojure libraries

- [aterreno/etcd-clojure]
- [dwwoelfel/cetcd] - Supports v2
- [rthomas/clj-etcd] - Supports v2

Erlang libraries

- [marshall-lee/etcd.erl] - Supports v2
- [zhongwencool/eetcd] - Supports v3+ (GRPC only)

Elixir

- [team-telnyx/etcdex] - Supports v3+ (GRPC only)

.Net Libraries

- [wangjia184/etcdnet] - Supports v2
- [drusellers/etcetera]
- [shubhamranjan/dotnet-etcd] - Supports v3+ (GRPC only)
- [SimplifyNet/Etcd.Microsoft.Extensions.Configuration]

PHP Libraries

- linkorb/etcd-php
- activecollab/etcd
- ouqiang/etcd-php - Client for v3 gRPC gateway

Haskell libraries

- wereHamster/etcd-hs

R libraries

- ropensci/etseed

Nim libraries

- etcd_client

Tcl libraries

- efrecon/etcd-tcl - Supports v2, except wait.

Rust libraries

- jimmycuadra/rust-etcd - Supports v2

Gradle Plugins

- gradle-etcd-rest-plugin - Supports v2

Chef Integration

- coderanger/etcd-chef

Chef Cookbook

- spheromak/etcd-cookbook

BOSH Releases

- cloudfoundry-community/etcd-boshrelease
- cloudfoundry/cf-release

Projects using etcd

- etcd Raft users - projects using etcd's raft library implementation.
- apache/celix - an implementation of the OSGi specification adapted to C and C++
- binocarlos/yoda - etcd + ZeroMQ
- blox/blox - a collection of open source projects for container management and orchestration with AWS ECS
- calavera/active-proxy - HTTP Proxy configured with etcd

- chain/chain - software designed to operate and connect to highly scalable permissioned blockchain networks
- derekchiang/etcdplus - A set of distributed synchronization primitives built upon etcd
- go-discover - service discovery in Go
- gleicon/goreman - Branch of the Go Foreman clone with etcd support
- garethr/hiera-etcd - Puppet hiera backend using etcd
- mattn/etcd-vim - SET and GET keys from inside vim
- mattn/etcdenv - "env" shebang with etcd integration
- kelseyhightower/confd - Manage local app config files using templates and data from etcd
- configdb - A REST relational abstraction on top of arbitrary database backends, aimed at storing configs and inventories.
- kubernetes/kubernetes - Container cluster manager introduced by Google.
- mailgun/vulcand - HTTP proxy that uses etcd as a configuration backend.
- duedil-ltd/discodns - Simple DNS nameserver using etcd as a database for names and records.
- skynetservices/skydns - RFC compliant DNS server
- xordataexchange/crypt - Securely store values in etcd using GPG encryption
- spf13/viper - Go configuration library, reads values from ENV, pflags, files, and etcd with optional encryption
- lytics/metafora - Go distributed task library
- ryandoyle/nss-etcd - A GNU libc NSS module for resolving names from etcd.
- Gru - Orchestration made easy with Go
- Vitess - Vitess is a database clustering system for horizontal scaling of MySQL.
- lclarkmichalek/etcdhcp - DHCP server that uses etcd for persistence and coordination.
- openstack/networking-vpp - A networking driver that programs the FD.io VPP dataplane to provide OpenStack cloud virtual networking
- OpenStack - OpenStack services can rely on etcd as a base service.
- CoreDNS - CoreDNS is a DNS server that chains plugins, part of CNCF and Kubernetes
- Uber M3 - M3: Uber's Open Source, Large-scale Metrics Platform for Prometheus
- Rook - Storage Orchestration for Kubernetes
- Patroni - A template for PostgreSQL High Availability with ZooKeeper, etcd, or Consul
- Trillian - Trillian implements a Merkle tree whose contents are served from a data storage layer, to allow scalability to extremely large trees.

Last modified May 25, 2024: [add] SimplifyNet/Etcd.Microsoft.Extensions.Configuration .NET library link (c7e4a1a)

Metrics

Metrics for real-time monitoring and debugging

etcd uses Prometheus for metrics reporting. The metrics can be used for real-time monitoring and debugging. etcd does not persist its metrics; if a member restarts, the metrics will be reset.

The simplest way to see the available metrics is to cURL the metrics endpoint `/metrics`. The format is described here.

Follow the Prometheus getting started doc to spin up a Prometheus server to collect etcd metrics.

The naming of metrics follows the suggested Prometheus best practices. A metric name has an `etcd` or `etcd_debugging` prefix as its namespace and a subsystem prefix (for example `wal` and `etcdserver`).

etcd namespace metrics

The metrics under the `etcd` prefix are for monitoring and alerting. They are stable high level metrics. If there is any change of these metrics, it will be included in release notes.

Metrics that are etcd2 related are documented v2 metrics guide.

Server

These metrics describe the status of the etcd server. In order to detect outages or problems for troubleshooting, the server metrics of every production etcd cluster should be closely monitored.

All these metrics are prefixed with `etcd_server_`

Name	Description	Type
has_leader	Whether or not a leader exists. 1 is existence, 0 is not.	Gauge

Name	Description	Type
leader_changes_seen_total	The number of leader changes seen.	Counter
proposals_committed_total	The total number of consensus proposals committed.	Gauge
proposals_applied_total	The total number of consensus proposals applied.	Gauge
proposals_pending	The current number of pending proposals.	Gauge
proposals_failed_total	The total number of failed proposals seen.	Counter

`has_leader` indicates whether the member has a leader. If a member does not have a leader, it is totally unavailable. If all the members in the cluster do not have any leader, the entire cluster is totally unavailable.

`leader_changes_seen_total` counts the number of leader changes the member has seen since its start. Rapid leadership changes impact the performance of etcd significantly. It also signals that the leader is unstable, perhaps due to network connectivity issues or excessive load hitting the etcd cluster.

`proposals_committed_total` records the total number of consensus proposals committed. This gauge should increase over time if the cluster is healthy. Several healthy members of an etcd cluster may have different total committed proposals at once. This discrepancy may be due to recovering from peers after starting, lagging behind the leader, or being the leader and therefore having the most commits. It is important to monitor this metric across all the members in the cluster; a consistently large lag between a single member and its leader indicates that member is slow or unhealthy.

`proposals_applied_total` records the total number of consensus proposals applied. The etcd server applies every committed proposal asynchronously. The difference between `proposals_committed_total` and `proposals_applied_total` should usually be small (within a few thousands even under high load). If the difference between them continues to rise, it indicates that the etcd server is overloaded. This might happen when applying expensive queries like heavy range queries or large txn operations.

`proposals_pending` indicates how many proposals are queued to commit. Rising pending proposals suggests there is a high client load or the member cannot commit proposals.

`proposals_failed_total` are normally related to two issues: temporary failures related to a leader election or longer downtime caused by a loss of quorum in the cluster.

Disk

These metrics describe the status of the disk operations.

All these metrics are prefixed with `etcd_disk_` .

Name	Description	Type
wal_fsync_duration_seconds	The latency distributions of fsync called by wal	Histogram
backend_commit_duration_seconds	The latency distributions of commit called by backend.	Histogram

A `wal_fsync` is called when etcd persists its log entries to disk before applying them.

A `backend_commit` is called when etcd commits an incremental snapshot of its most recent changes to disk.

High disk operation latencies (`wal_fsync_duration_seconds` or `backend_commit_duration_seconds`) often indicate disk issues. It may cause high request latency or make the cluster unstable.

Network

These metrics describe the status of the network.

All these metrics are prefixed with `etcd_network_`

Name	Description	Type
peer_sent_bytes_total	The total number of bytes sent to the peer with ID `To` .	Counter(To)
peer_received_bytes_total	The total number of bytes received from the peer with ID `From` .	Counter(From)
peer_sent_failures_total	The total number of send failures from the peer with ID `To` .	Counter(To)
peer_received_failures_total	The total number of receive failures from the peer with ID `From` .	Counter(From)
peer_round_trip_time_seconds	Round-Trip-Time histogram between peers.	Histogram(To)
client_grpc_sent_bytes_total	The total number of bytes sent to grpc clients.	Counter

Name	Description	Type
client_grpc_received_bytes_total	The total number of bytes received to grpc clients.	Counter

`peer_sent_bytes_total` counts the total number of bytes sent to a specific peer. Usually the leader member sends more data than other members since it is responsible for transmitting replicated data.

`peer_received_bytes_total` counts the total number of bytes received from a specific peer. Usually follower members receive data only from the leader member.

gRPC requests

These metrics are exposed via go-grpc-prometheus.

etcd_debugging namespace metrics

The metrics under the `etcd_debugging` prefix are for debugging. They are very implementation dependent and volatile. They might be changed or removed without any warning in new etcd releases. Some of the metrics might be moved to the `etcd` prefix when they become more stable.

Snapshot

Name	Description	Type
snapshot_save_total_duration_seconds	The total latency distributions of save called by snapshot	Histogram

Abnormally high snapshot duration (`snapshot_save_total_duration_seconds`) indicates disk issues and might cause the cluster to be unstable.

Prometheus supplied metrics

The Prometheus client library provides a number of metrics under the `go` and `process` namespaces. There are a few that are particularly interesting.

Name	Description	Type
process_open_fds	Number of open file descriptors.	Gauge

Name	Description	Type
process_max_fds	Maximum number of open file descriptors.	Gauge

Note: The process metrics, such as `process_open_fds` and `process_max_fds`, are not supported on Darwin (macOS) systems at this time.

Heavy file descriptor (`process_open_fds`) usage (i.e., near the process's file descriptor limit, `process_max_fds`) indicates a potential file descriptor exhaustion issue. If the file descriptors are exhausted, etcd may panic because it cannot create new WAL files.

Generated list of metrics

v3.4.0	v3.4.1	v3.4.10	v3.4.11	v3.4.12	v3.4.13
v3.4.2	v3.4.3	v3.4.4	v3.4.5	v3.4.6	v3.4.7
v3.4.8	v3.4.9				

Last modified March 23, 2024: website: add note on lack of process metrics support for Darwin (8adf69f)

Reporting bugs

How to file issue reports for the etcd project

If any part of the etcd project has bugs or documentation mistakes, please let us know by opening an issue. We treat bugs and mistakes very seriously and believe no issue is too small. Before creating a bug report, please check that an issue reporting the same problem does not already exist.

To make the bug report accurate and easy to understand, please try to create bug reports that are:

- Specific. Include as much details as possible: which version, what environment, what configuration, etc. If the bug is related to running the etcd server, please attach the etcd log (the starting log with etcd configuration is especially important).
- Reproducible. Include the steps to reproduce the problem. We understand some issues might be hard to reproduce, please includes the steps that might lead to the problem. If possible, please attach the affected etcd data dir and stack strace to the bug report.
- Isolated. Please try to isolate and reproduce the bug with minimum dependencies. It would significantly slow down the speed to fix a bug if too many dependencies are involved in a bug report. Debugging external systems that rely on etcd is out of scope, but we are happy to provide guidance in the right direction or help with using etcd itself.
- Unique. Do not duplicate existing bug report.
- Scoped. One bug per report. Do not follow up with another bug inside one report.

It may be worthwhile to read Elika Etemad's article on filing good bug reports before creating a bug report.

We might ask for further information to locate a bug. A duplicated bug report will be closed.

Frequently asked questions

How to get a stack trace

```
$ kill -QUIT $PID
```

How to get etcd version

```
$ etcd --version
```

How to get etcd configuration and log when it runs as systemd service ‘etcd2.service’

```
$ sudo systemctl cat etcd2
$ sudo journalctl -u etcd2
```

Due to an upstream systemd bug, journald may miss the last few log lines when its processes exit. If journalctl says etcd stopped without fatal or panic message, try `sudo journalctl -f -t etcd2` to get full log.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Tuning

When to update the heartbeat interval and election timeout settings

The default settings in etcd should work well for installations on a local network where the average network latency is low. However, when using etcd across multiple data centers or over networks with high latency, the heartbeat interval and election timeout settings may need tuning.

The network isn't the only source of latency. Each request and response may be impacted by slow disks on both the leader and follower. Each of these timeouts represents the total time from request to successful response from the other machine.

Time parameters

The underlying distributed consensus protocol relies on two separate time parameters to ensure that nodes can handoff leadership if one stalls or goes offline. The first parameter is called the *Heartbeat Interval*. This is the frequency with which the leader will notify followers that it is still the leader. For best practices, the parameter should be set around round-trip time between members. By default, etcd uses a `100ms` heartbeat interval.

The second parameter is the *Election Timeout*. This timeout is how long a follower node will go without hearing a heartbeat before attempting to become leader itself. By default, etcd uses a `1000ms` election timeout.

Adjusting these values is a trade off. The value of heartbeat interval is recommended to be around the maximum of average round-trip time (RTT) between members, normally around 0.5-1.5x the round-trip time. If heartbeat interval is too low, etcd will send unnecessary messages that increase the usage of CPU and network resources. On the other side, a too high heartbeat interval leads to high election timeout. Higher election timeout takes longer time to detect a leader failure. The easiest way to measure round-trip time (RTT) is to use PING utility.

The election timeout should be set based on the heartbeat interval and average round-trip time between members. Election timeouts must be at least 10 times the round-trip time so it can account for variance in the network. For example, if the round-trip time between members is 10ms then the election timeout should be at least 100ms.

The upper limit of election timeout is 50000ms (50s), which should only be used when deploying a globally-distributed etcd cluster. A reasonable round-trip time for the continental United States is 130ms, and the time between US and Japan is around 350-400ms. If the network has uneven performance or regular packet delays/loss then it is possible that a couple of retries may be necessary to successfully send a packet. So 5s is a safe upper limit of global round-trip time. As the election timeout should be an order of magnitude bigger than broadcast time, in the case of ~5s for a globally distributed cluster, then 50 seconds becomes a reasonable maximum.

The heartbeat interval and election timeout value should be the same for all members in one cluster. Setting different values for etcd members may disrupt cluster stability.

The default values can be overridden on the command line:

```
# Command line arguments:
$ etcd --heartbeat-interval=100 --election-timeout=500

# Environment variables:
$ ETCD_HEARTBEAT_INTERVAL=100 ETCD_ELECTION_TIMEOUT=500 etcd
```

The values are specified in milliseconds.

Snapshots

etcd appends all key changes to a log file. This log grows forever and is a complete linear history of every change made to the keys. A complete history works well for lightly used clusters but clusters that are heavily used would carry around a large log.

To avoid having a huge log etcd makes periodic snapshots. These snapshots provide a way for etcd to compact the log by saving the current state of the system and removing old logs.

Snapshot tuning

Creating snapshots with the V2 backend can be expensive, so snapshots are only created after a given number of changes to etcd. By default, snapshots will be made after every 10,000 changes. If etcd's memory usage and disk usage are too high, try lowering the snapshot threshold by setting the following on the command line:

```
# Command line arguments:
$ etcd --snapshot-count=5000
```

```
# Environment variables:
$ ETCD_SNAPSHOT_COUNT=5000 etcd
```

Disk

An etcd cluster is very sensitive to disk latencies. Since etcd must persist proposals to its log, disk activity from other processes may cause long `fsync` latencies. The upshot is etcd may miss heartbeats, causing request timeouts and temporary leader loss. An etcd server can sometimes stably run alongside these processes when given a high disk priority.

On Linux, etcd's disk priority can be configured with `ionice`:

```
# best effort, highest priority
$ sudo ionice -c2 -n0 -p `pgrep etcd`
```

Network

If the etcd leader serves a large number of concurrent client requests, it may delay processing follower peer requests due to network congestion. This manifests as send buffer error messages on the follower nodes:

```
dropped MsgProp to 247ae21ff9436b2d since streamMsg's sending buffer is full
dropped MsgAppResp to 247ae21ff9436b2d since streamMsg's sending buffer is full
```

These errors may be resolved by prioritizing etcd's peer traffic over its client traffic. On Linux, peer traffic can be prioritized by using the traffic control mechanism:

```
tc qdisc add dev eth0 root handle 1: prio bands 3
tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 2380 0xffff flowi
tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip dport 2380 0xffff flowi
tc filter add dev eth0 parent 1: protocol ip prio 2 u32 match ip sport 2379 0xffff flowi
tc filter add dev eth0 parent 1: protocol ip prio 2 u32 match ip dport 2379 0xffff flowi
```

To cancel `tc`, execute:

```
tc qdisc del dev eth0 root
```

Last modified August 6, 2021: Add tc cancel command. (#428) (29b874f)

Discovery service protocol

Discover other etcd members in a cluster bootstrap phase

Discovery service protocol helps new etcd member to discover all other members in cluster bootstrap phase using a shared discovery URL.

Discovery service protocol is *only* used in cluster bootstrap phase, and cannot be used for runtime reconfiguration or cluster monitoring.

The protocol uses a new discovery token to bootstrap one *unique* etcd cluster. Remember that one discovery token can represent only one etcd cluster. As long as discovery protocol on this token starts, even if it fails halfway, it must not be used to bootstrap another etcd cluster.

The rest of this article will walk through the discovery process with examples that correspond to a self-hosted discovery cluster. The public discovery service, discovery.etcd.io, functions the same way, but with a layer of polish to abstract away ugly URLs, generate UUIDs automatically, and provide some protections against excessive requests. At its core, the public discovery service still uses an etcd cluster as the data store as described in this document.

Protocol workflow

The idea of discovery protocol is to use an internal etcd cluster to coordinate bootstrap of a new cluster. First, all new members interact with discovery service and help to generate the expected member list. Then each new member bootstraps its server using this list, which performs the same functionality as -initial-cluster flag.

In the following example workflow, we will list each step of protocol in curl format for ease of understanding.

By convention the etcd discovery protocol uses the key prefix `_etcd/registry`. If `http://example.com` hosts an etcd cluster for discovery service, a full URL to discovery keyspace will be `http://example.com/v2/keys/_etcd/registry`. We will use this as the URL prefix in the example.

Creating a new discovery token

Generate a unique token that will identify the new cluster. This will be used as a unique prefix in discovery keyspace in the following steps. An easy way to do this is to use `uuidgen`:

```
UUID=$(uuidgen)
```

Specifying the expected cluster size

The discovery token expects a cluster size that must be specified. The size is used by the discovery service to know when it has found all members that will initially form the cluster.

```
curl -X PUT http://example.com/v2/keys/_etcd/registry/${UUID}/_config/size -d value=${clu
```

Usually the cluster size is 3, 5 or 7. Check optimal cluster size for more details.

Bringing up etcd processes

Given the discovery URL, use it as `-discovery` flag and bring up etcd processes. Every etcd process will follow this next few steps internally if given a `-discovery` flag.

Registering itself

The first thing for etcd process is to register itself into the discovery URL as a member. This is done by creating member ID as a key in the discovery URL.

```
curl -X PUT http://example.com/v2/keys/_etcd/registry/${UUID}/${member_id}?prevExist=fals
```

Checking the status

It checks the expected cluster size and registration status in discovery URL, and decides what the next action is.

```
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}/_config/size
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}
```

If registered members are still not enough, it will wait for left members to appear.

If the number of registered members is bigger than the expected size N, it treats the first N registered members as the member list for the cluster. If the member itself is in the member list, the discovery procedure succeeds and it fetches all peers through the member list. If it is not in the member list, the discovery procedure finishes with the failure that the cluster has been full.

In etcd implementation, the member may check the cluster status even before registering itself. So it could fail quickly if the cluster has been full.

Waiting for all members

The wait process is described in detail in the etcd API documentation.

```
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}?wait=true&waitIndex=${curre
```

It keeps waiting until finding all members.

Public discovery service

CoreOS Inc. hosts a public discovery service at https://discovery.etcd.io/ , which provides some nice features for ease of use.

Mask key prefix

Public discovery service will redirect `https://discovery.etcd.io/${UUID}` to etcd cluster behind for the key at `/v2/keys/_etcd/registry`. It masks register key prefix for short and readable discovery url.

Get new token

```
GET /new

Sent query:
	size=${cluster_size}
Possible status codes:
	200 OK
	400 Bad Request
200 Body:
	generated discovery url
```

The generation process in the service follows the steps from Creating a New Discovery Token to Specifying the Expected Cluster Size.

Check discovery status

```
GET /${UUID}
```

The status for this discovery token, including the machines that have been registered, can be checked by requesting the value of the UUID.

Open-source repository

The repository is located at https://github.com/coreos/discovery.etcd.io. It could be used to build a custom discovery service.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Logging conventions

Logging level categories

etcd uses the capnslog library for logging application output categorized into *levels*. A log message's level is determined according to these conventions:

- Error: Data has been lost, a request has failed for a bad reason, or a required resource has been lost
  - Examples:
    - A failure to allocate disk space for WAL
- Warning: (Hopefully) Temporary conditions that may cause errors, but may work fine. A replica disappearing (that may reconnect) is a warning.
  - Examples:
    - Failure to send raft message to a remote peer
    - Failure to receive heartbeat message within the configured election timeout
- Notice: Normal, but important (uncommon) log information.
  - Examples:
    - Add a new node into the cluster
    - Add a new user into auth subsystem
- Info: Normal, working log information, everything is fine, but helpful notices for auditing or common operations.
  - Examples:
    - Startup configuration
    - Start to do snapshot
- Debug: Everything is still fine, but even common operations may be logged, and less helpful but more quantity of notices.
  - Examples:
    - Send a normal message to a remote peer
    - Write a log entry to disk

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Learning

Learning resources

Data model
etcd data storage methodologies

etcd client design
Client architectural decisions & their implementation details

etcd learner design
Mitigating common challenges with membership reconfiguration

etcd v3 authentication design
etcd v3 authentication

etcd3 API
etcd3 API central design overview

etcd API guarantees
API guarantees made by etcd

etcd versus other key-value stores
History and use of etcd & comparison with other tools

Glossary
Terms used in etcd documentation, command line, and source code

Last modified April 26, 2021: [Docsy theme (#244) (86b070b)]

Data model

etcd data storage methodologies

etcd is designed to reliably store infrequently updated data and provide reliable watch queries. etcd exposes previous versions of key-value pairs to support inexpensive snapshots and watch history events (“time travel queries”). A persistent, multi-version, concurrency-control data model is a good fit for these use cases.

etcd stores data in a multiversion persistent key-value store. The persistent key-value store preserves the previous version of a key-value pair when its value is superseded with new data. The key-value store is effectively immutable; its operations do not update the structure in-place, but instead always generate a new updated structure. All past versions of keys are still accessible and watchable after modification. To prevent the data store from growing indefinitely over time and from maintaining old versions, the store may be compacted to shed the oldest versions of superseded data.

Logical view

The store’s logical view is a flat binary key space. The key space has a lexically sorted index on byte string keys so range queries are inexpensive.

The key space maintains multiple **revisions**. When the store is created, the initial revision is 1. Each atomic mutative operation (e.g., a transaction operation may contain multiple operations) creates a new revision on the key space. All data held by previous revisions remains unchanged. Old versions of key can still be accessed through previous revisions. Likewise, revisions are indexed as well; ranging over revisions with watchers is efficient. If the store is compacted to save space, revisions before the compact revision will be removed. Revisions are monotonically increasing over the lifetime of a cluster.

A key’s life spans a generation, from creation to deletion. Each key may have one or multiple generations. Creating a key increments the **version** of that key, starting at 1 if the key does not exist at the current revision. Deleting a key generates a key tombstone, concluding the key’s current generation by resetting its version to 0. Each modification of a key increments its version; so, versions are monotonically increasing within a key’s generation. Once a compaction happens, any generation ended before the compaction revision will be removed, and values set before the compaction revision except the latest one will be removed.

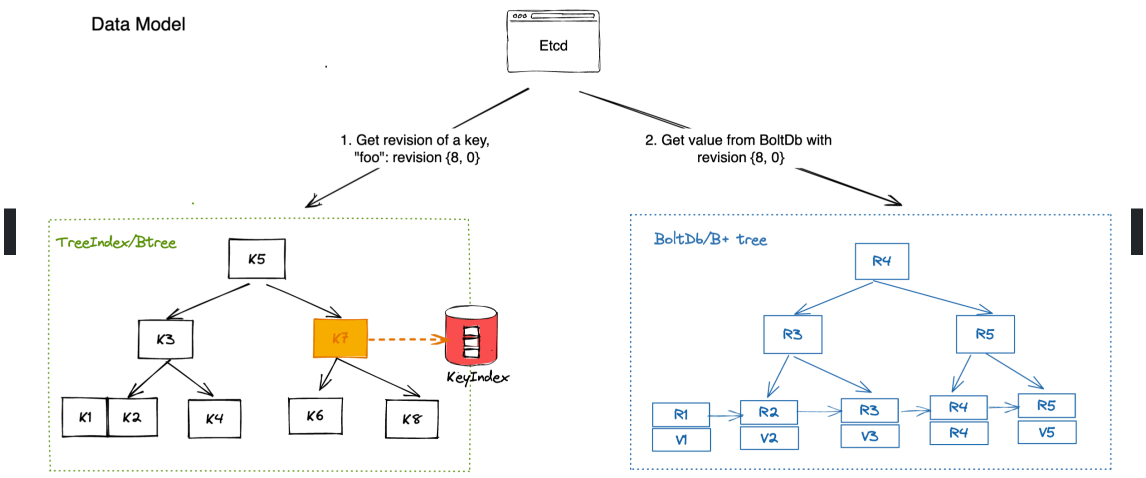
Physical view

etcd stores the physical data as key-value pairs in a persistent [b+tree](). Each revision of the store's state only contains the delta from its previous revision to be efficient. A single revision may correspond to multiple keys in the tree.

The key of key-value pair is a 3-tuple (major, sub, type). Major is the store revision holding the key. Sub differentiates among keys within the same revision. Type is an optional suffix for special value (e.g., `t` if the value contains a tombstone). The value of the key-value pair contains the modification from previous revision, thus one delta from previous revision. The b+tree is ordered by key in lexical byte-order. Ranged lookups over revision deltas are fast; this enables quickly finding modifications from one specific revision to another. Compaction removes out-of-date keys-value pairs.

etcd also keeps a secondary in-memory btree index to speed up range queries over keys. The keys in the btree index are the keys of the store exposed to user. The value is a pointer to the modification of the persistent b+tree. Compaction removes dead pointers.

Overall, etcd gets the revision information from btree and then uses the revision as key to fetch value from b+tree(As shown below).



Last modified February 26, 2023: Update data_model.md (009fe6b)

etcd client design

Client architectural decisions & their implementation details

etcd Client Design

Gyuho Lee (github.com/gyuho, Amazon Web Services, Inc.), Joe Betz (github.com/jpbetz, Google Inc.)

Introduction

etcd server has proven its robustness with years of failure injection testing. Most complex application logic is already handled by etcd server and its data stores (e.g. cluster membership is transparent to clients, with Raft-layer forwarding proposals to leader). Although server components are correct, its composition with client requires a different set of intricate protocols to guarantee its correctness and high availability under faulty conditions. Ideally, etcd server provides one logical cluster view of many physical machines, and client implements automatic failover between replicas. This documents client architectural decisions and its implementation details.

Glossary

clientv3: etcd Official Go client for etcd v3 API.

clientv3-grpc1.0: Official client implementation, with `grpc-go v1.0.x`, which is used in latest etcd v3.1.

clientv3-grpc1.7: Official client implementation, with `grpc-go v1.7.x`, which is used in latest etcd v3.2 and v3.3.

clientv3-grpc1.23: Official client implementation, with `grpc-go v1.23.x`, which is used in latest etcd v3.4.

Balancer: etcd client load balancer that implements retry and failover mechanism. etcd client should automatically balance loads between multiple endpoints.

Endpoints: A list of etcd server endpoints that clients can connect to. Typically, 3 or 5 client URLs of an etcd cluster.

Pinned endpoint: When configured with multiple endpoints, <= v3.3 client balancer chooses only one endpoint to establish a TCP connection, in order to conserve total open connections to etcd cluster. In v3.4, balancer round-robins pinned endpoints for every request, thus distributing loads more evenly.

Client Connection: TCP connection that has been established to an etcd server, via gRPC Dial.

Sub Connection: gRPC SubConn interface. Each sub-connection contains a list of addresses. Balancer creates a SubConn from a list of resolved addresses. gRPC ClientConn can map to multiple SubConn (e.g. example.com resolves to `10.10.10.1` and `10.10.10.2` of two sub-connections). etcd v3.4 balancer employs internal resolver to establish one sub-connection for each endpoint.

Transient disconnect: When gRPC server returns a status error of `code Unavailable`.

Client Requirements

Correctness. Requests may fail in the presence of server faults. However, it never violates consistency guarantees: global ordering properties, never write corrupted data, at-most once semantics for mutable operations, watch never observes partial events, and so on.

Liveness. Servers may fail or disconnect briefly. Clients should make progress in either way. Clients should never deadlock waiting for a server to come back from offline, unless configured to do so. Ideally, clients detect unavailable servers with HTTP/2 ping and failover to other nodes with clear error messages.

Effectiveness. Clients should operate effectively with minimum resources: previous TCP connections should be gracefully closed after endpoint switch. Failover mechanism should effectively predict the next replica to connect, without wastefully retrying on failed nodes.

Portability. Official client should be clearly documented and its implementation be applicable to other language bindings. Error handling between different language bindings should be consistent. Since etcd is fully committed to gRPC, implementation should be closely aligned with gRPC long-term design goals (e.g. pluggable retry policy should be compatible with gRPC retry). Upgrades between two client versions should be non-disruptive.

Client Overview

etcd client implements the following components:

- balancer that establishes gRPC connections to an etcd cluster,
- API client that sends RPCs to an etcd server, and
- error handler that decides whether to retry a failed request or switch endpoints.

Languages may differ in how to establish an initial connection (e.g. configure TLS), how to encode and send Protocol Buffer messages to server, how to handle stream RPCs, and so on.

However, errors returned from etcd server will be the same. So should be error handling and retry policy.

For example, etcd server may return `"rpc error: code = Unavailable desc = etcdserver: request timed out"`, which is transient error that expects retries. Or return `rpc error: code = InvalidArgument desc = etcdserver: key is not provided`, which means request was invalid and should not be retried. Go client can parse errors with `google.golang.org/grpc/status.FromError`, and Java client with `io.grpc.Status.fromThrowable`.

clientv3-grpc1.0: Balancer Overview

`clientv3-grpc1.0` maintains multiple TCP connections when configured with multiple etcd endpoints. Then pick one address and use it to send all client requests. The pinned address is maintained until the client object is closed (see *Figure 1*). When the client receives an error, it randomly picks another and retries.

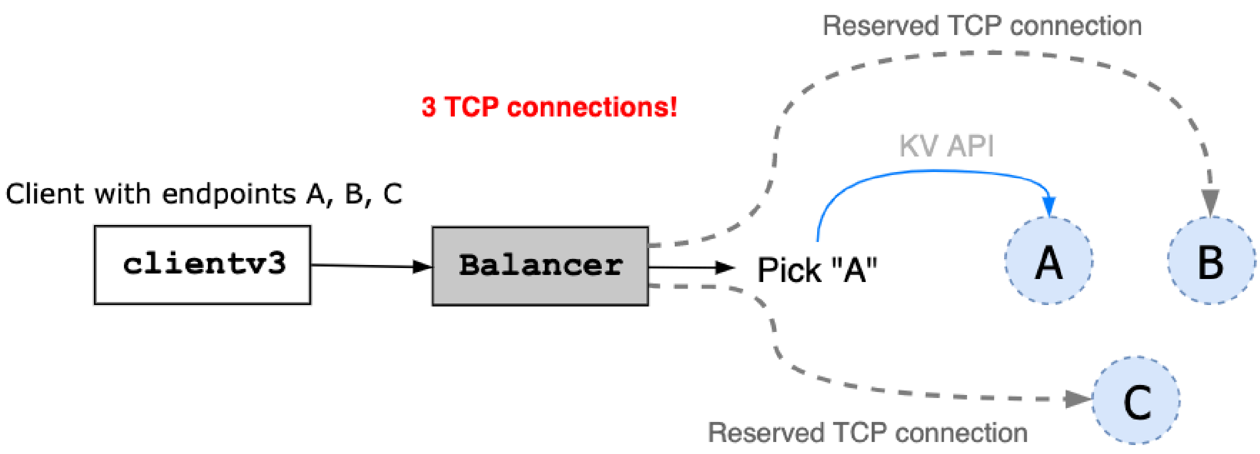


Figure 1. `clientv3-grpc1.0` passes multiple endpoints to balancer. The balancer connects to them all, picks the first established endpoint. This picked endpoint is called "pinned address". Client maintains multiple TCP connections for each endpoint, using more resources.

clientv3-grpc1.0: Balancer Limitation

`clientv3-grpc1.0` opening multiple TCP connections may provide faster balancer failover but requires more resources. The balancer does not understand node's health status or cluster membership. So, it is possible that balancer gets stuck with one failed or partitioned node.

clientv3-grpc1.7: Balancer Overview

`clientv3-grpc1.7` maintains only one TCP connection to a chosen etcd server. When given multiple cluster endpoints, a client first tries to connect to them all. As soon as one connection is up, balancer pins the address, closing others (see *Figure 2*). The pinned address is to be maintained until the client object is closed. An error, from server or client network fault, is sent to client error handler (see *Figure 3*).

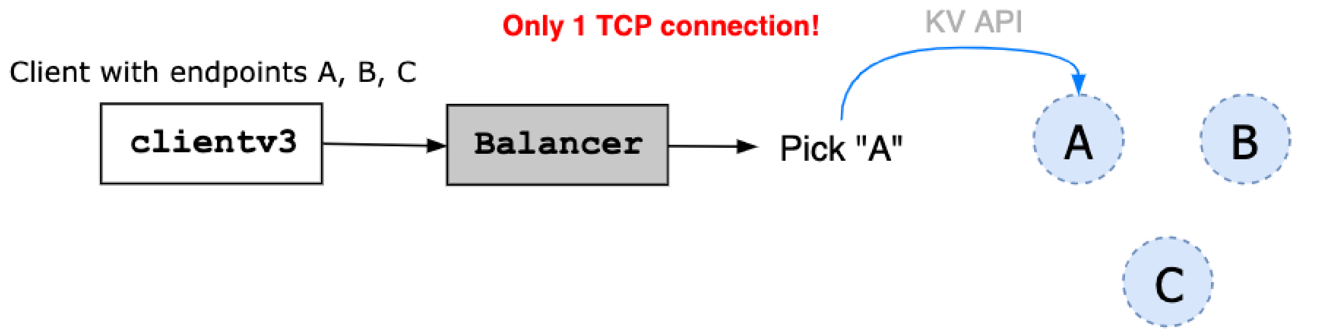


Figure 2. `clientv3-grpc1.7` passes multiple endpoints to balancer. The balancer first connects to all, picks the first established endpoint. This picked endpoint is called "pinned address". Client maintains only one TCP connection to one etcd server.

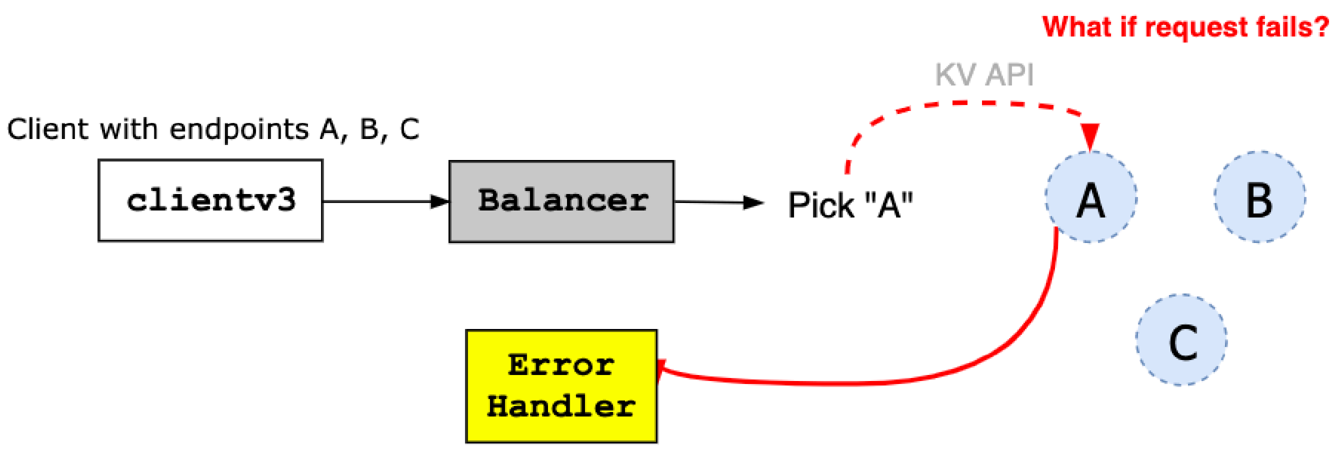


Figure 3. When request fails, the error is passed to an error handler to decide whether to retry on same endpoint, or to switch to other endpoints.

The client error handler takes an error from gRPC server, and decides whether to retry on the same endpoint, or to switch to other addresses, based on the error code and message (see *Figure 4* and *Figure 5*).

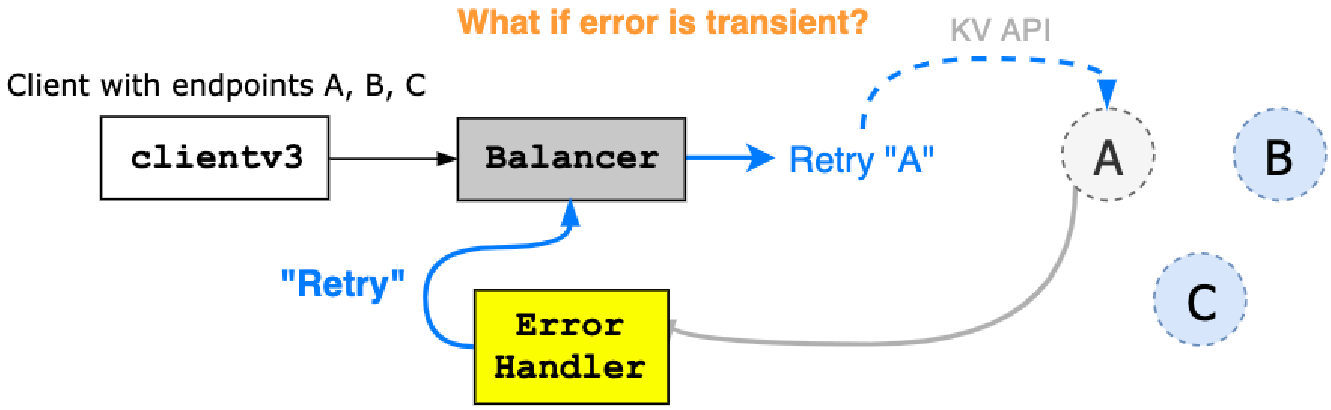


Figure 4. If error is transient which indicates that server may come back soon, retry with same endpoint (retry policy varies depending on API).

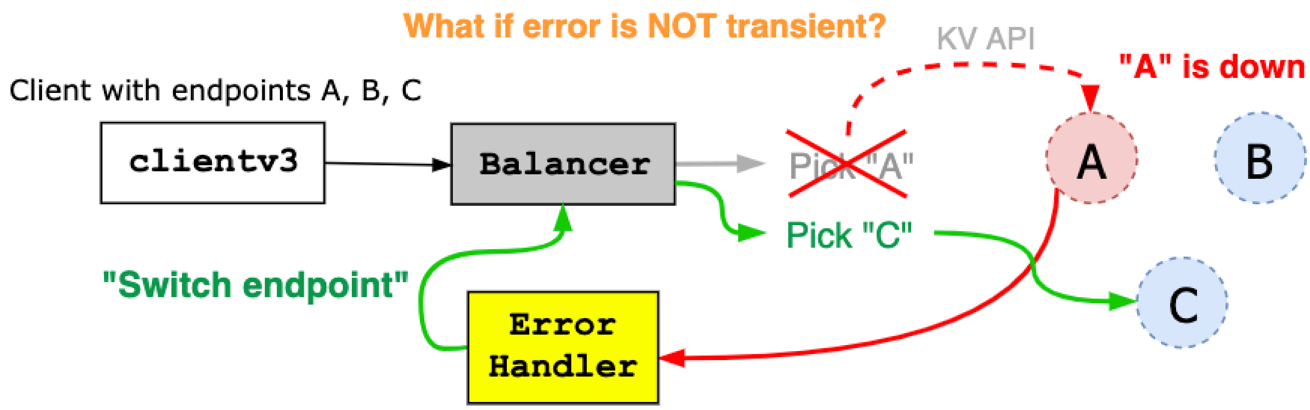


Figure 5. If the error is not transient, switch to other endpoints.

Stream RPCs, such as Watch and KeepAlive, are often requested with no timeouts. Instead, client can send periodic HTTP/2 pings to check the status of a pinned endpoint; if the server does not respond to the ping, balancer switches to other endpoints (see *Figure 6*).

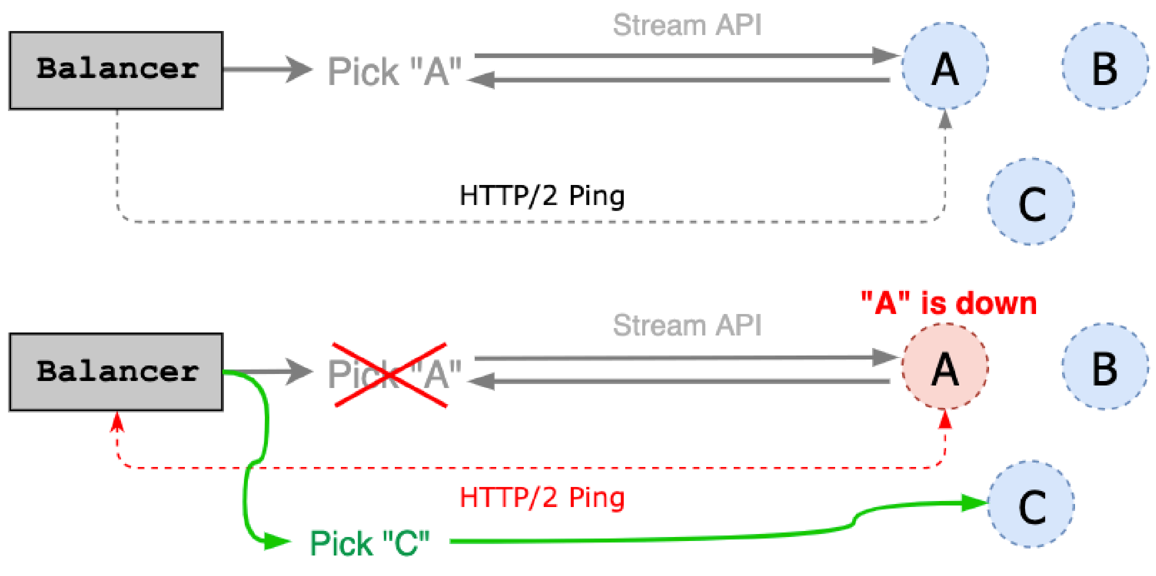


Figure 6. Stream RPCs are typically sent with no timeouts (e.g. pass `context.Background` to Go client). Instead, it sends HTTP/2 Pings to detect network disconnects.

clientv3-grpc1.7: Balancer Limitation

`clientv3-grpc1.7` balancer sends HTTP/2 keepalives to detect disconnects from streaming requests. It is a simple gRPC server ping mechanism and does not reason about cluster membership, thus unable to detect network partitions. Since partitioned gRPC server can still respond to client pings, balancer may get stuck with a partitioned node. Ideally, keepalive ping detects partition and triggers endpoint switch, before request time-out (see [etcd#8673](#) and *Figure 7*).

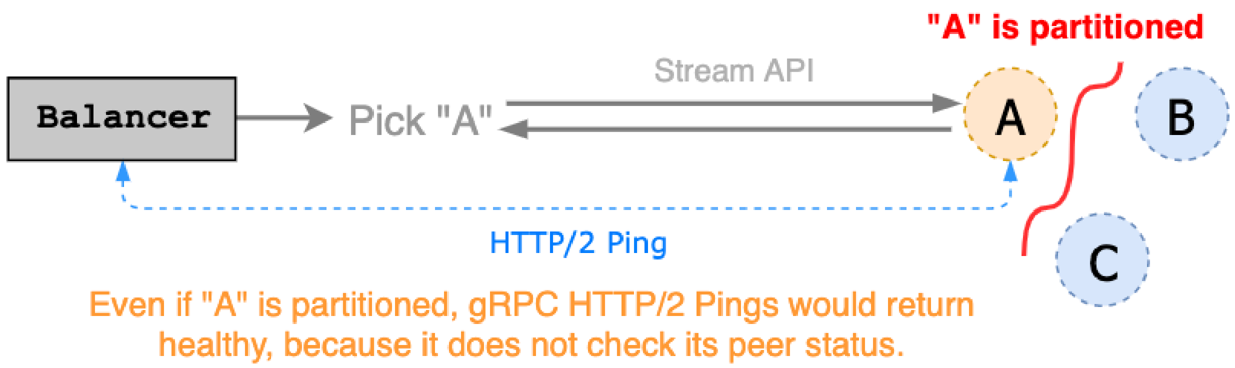


Figure 7. Neither `clientv3-grpc1.7` nor `clientv3-grpc1.23` HTTP/2 Ping cannot detect network partitions. Thus, balancer may get stuck with a partitioned node.

`clientv3-grpc1.7` balancer maintains a list of unhealthy endpoints. Disconnected addresses are added to "unhealthy" list, and considered unavailable until after wait duration, which is hard coded as dial timeout with default value 5-second. Balancer can have false positives on which endpoints are unhealthy. For instance, endpoint A may come back right after being blacklisted, but still unusable for next 5 seconds (see *Figure 8*).

`clientv3-grpc1.0` suffered the same problems above.

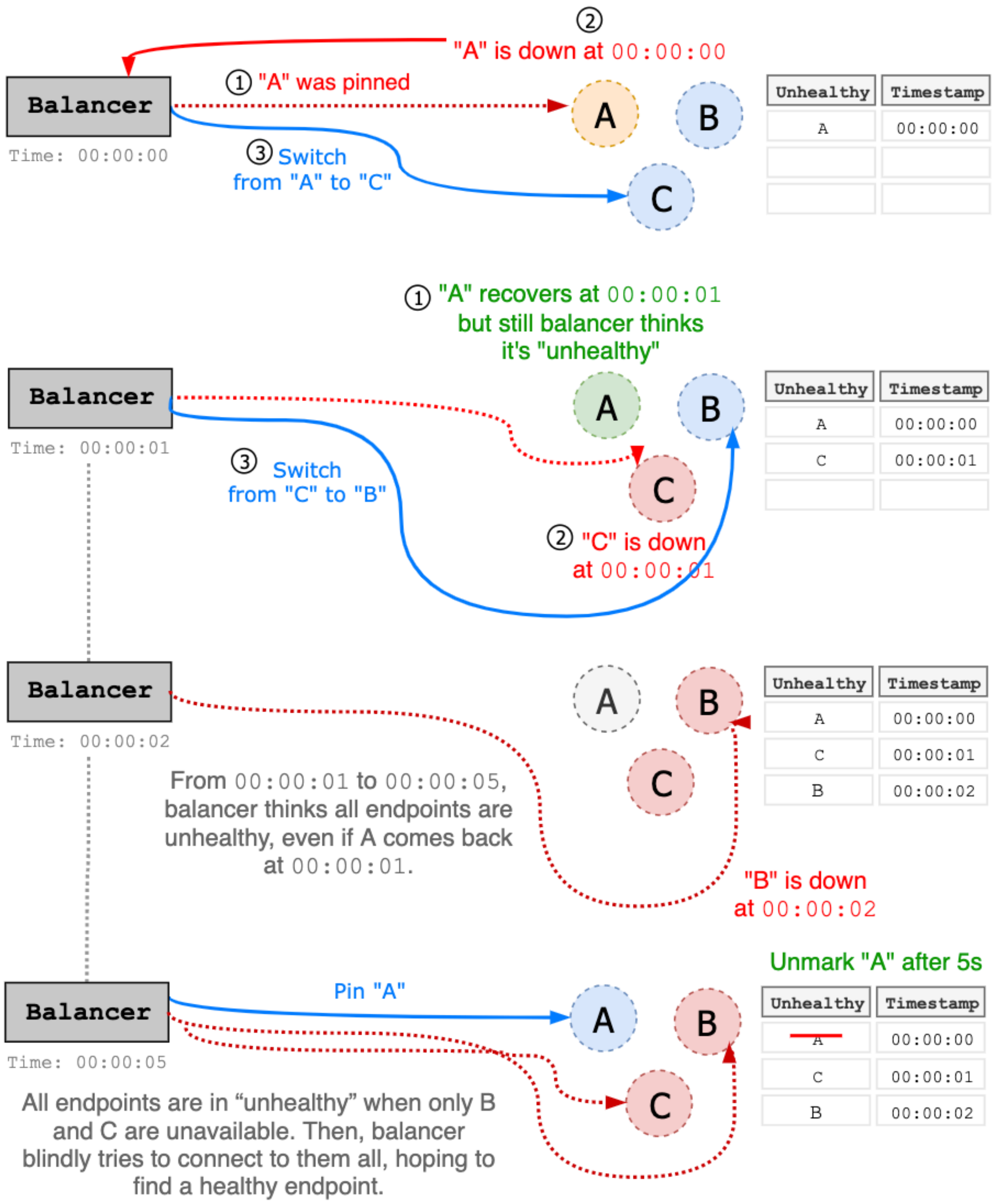


Figure 8. `clientv3-grpc1.7` maintains a list of unhealthy endpoints, which gets removed after dial timeout (`clientv3.Config.DialTimeout`). For example, if `DialTimeout` is 5-second and "A" failed at `00:00:00`, "A" will be in unhealthy endpoints until `00:00:05`, even if "A" comes back before then. "A" won't be retried unless all endpoints are in "unhealthy".

Upstream gRPC Go had already migrated to new balancer interface. For example, `clientv3-grpc1.7` underlying balancer implementation uses new gRPC balancer and tries to be consistent with old balancer behaviors. While its compatibility has been maintained reasonably well, etcd client still suffered from subtle breaking changes. Furthermore, gRPC maintainer recommends to [not rely on the old balancer interface](). In general, to get better support from upstream, it is best to be in sync with latest gRPC releases. And new features, such as retry policy, may not be backported to gRPC 1.7 branch. Thus, both etcd server and client must migrate to latest gRPC versions.

clientv3-grpc1.23: Balancer Overview

`clientv3-grpc1.7` is so tightly coupled with old gRPC interface, that every single gRPC dependency upgrade broke client behavior. Majority of development and debugging efforts were devoted to fixing those client behavior changes. As a result, its implementation has become overly complicated with bad assumptions on server connectivities.

The primary goal of `clientv3-grpc1.23` is to simplify balancer failover logic; rather than maintaining a list of unhealthy endpoints, which may be stale, simply roundrobin to the next endpoint whenever client gets disconnected from the current endpoint. It does not assume endpoint status. Thus, no more complicated status tracking is needed (see *Figure 8* and above). Upgrading to `clientv3-grpc1.23` should be no issue; all changes were internal while keeping all the backward compatibilities.

Internally, when given multiple endpoints, `clientv3-grpc1.23` creates multiple sub-connections (one sub-connection per each endpoint), while `clientv3-grpc1.7` creates only one connection to a pinned endpoint (see *Figure 9*). For instance, in 5-node cluster, `clientv3-grpc1.23` balancer would require 5 TCP connections, while `clientv3-grpc1.7` only requires one. By preserving the pool of TCP connections, `clientv3-grpc1.23` may consume more resources but provide more flexible load balancer with better failover performance. The default balancing policy is round robin but can be easily extended to support other types of balancers (e.g. power of two, pick leader, etc.). `clientv3-grpc1.23` uses gRPC resolver group and implements balancer picker policy, in order to delegate complex balancing work to upstream gRPC. On the other hand, `clientv3-grpc1.7` manually handles each gRPC connection and balancer failover, which complicates the implementation. `clientv3-grpc1.23` implements retry in the gRPC interceptor chain that automatically handles gRPC internal errors and enables more advanced retry policies like backoff, while `clientv3-grpc1.7` manually interprets gRPC errors for retries.

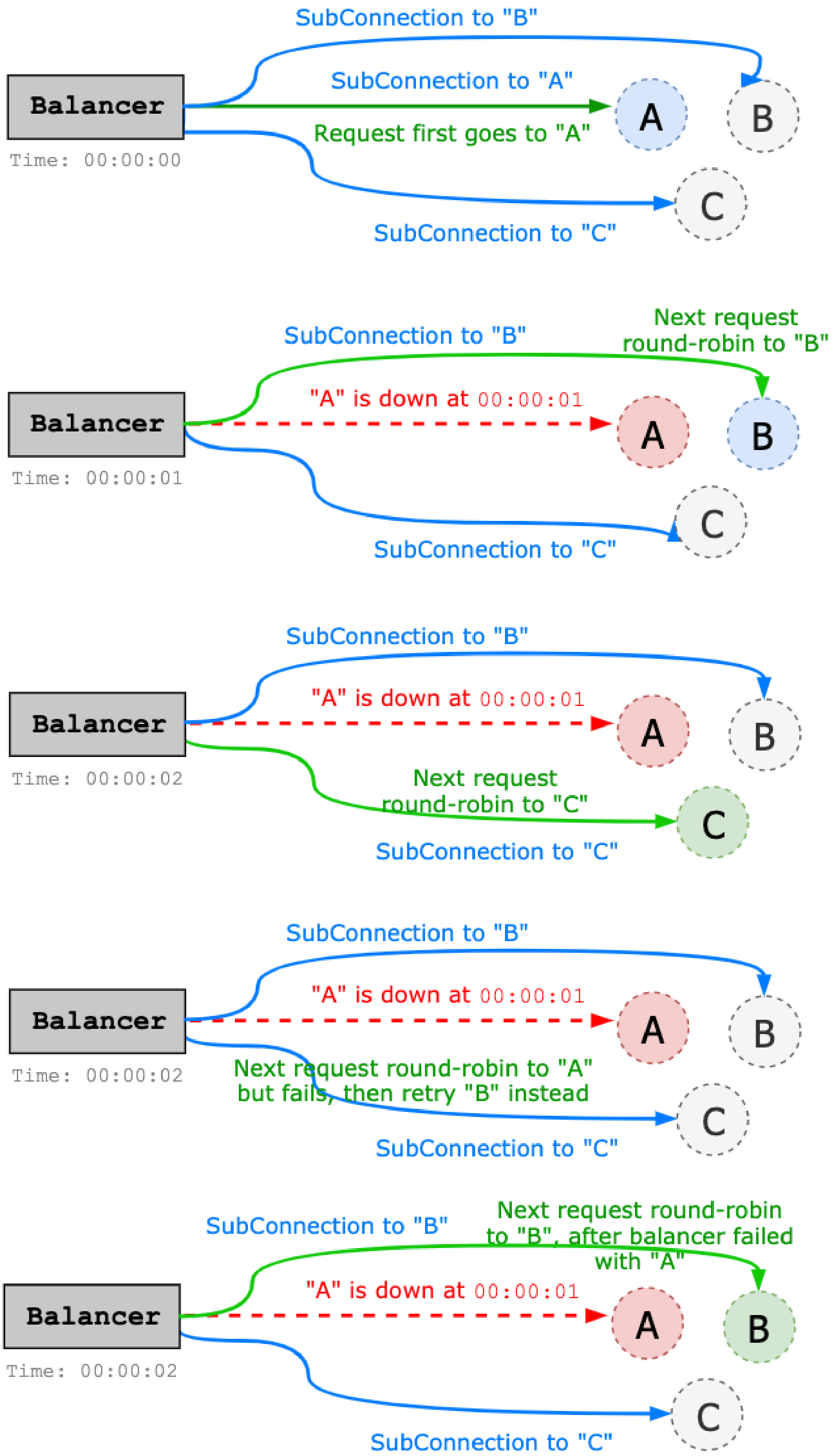


Figure 9. `clientv3-grpc1.14 picker.Policy.RoundRobinBalanced` does not maintain "unhealthy" endpoints. Instead, roundrobin to next one by one for every single request. It does not assume the status of an endpoint, thus no more false positives of endpoint status.

clientv3-grpc1.23: Balancer Limitation

Improvements can be made by caching the status of each endpoint. For instance, balancer can ping each server in advance to maintain a list of healthy candidates, and use this information when doing round-robin. Or when disconnected, balancer can prioritize healthy endpoints. This may complicate the balancer implementation, thus can be addressed in later versions.

Client-side keepalive ping still does not reason about network partitions. Streaming request may get stuck with a partitioned node. Advanced health checking service need to be implemented to understand the cluster membership (see etcd#8673 for more detail).

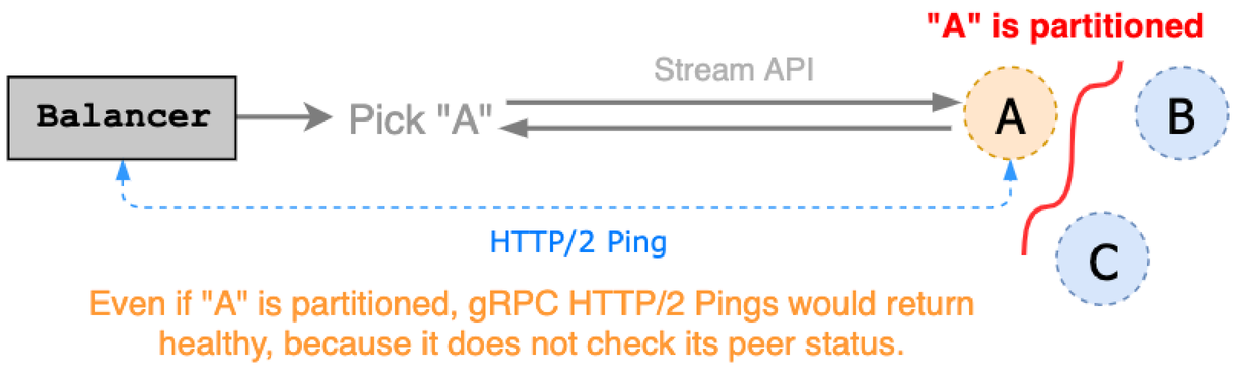


Figure 7. Neither `clientv3-grpc1.7` nor `clientv3-grpc1.23` HTTP/2 Ping cannot detect network partitions. Thus, balancer may get stuck with a partitioned node.

Currently, retry logic is handled manually as an interceptor. This may be simplified via official gRPC retries.

---

etcd learner design

Mitigating common challenges with membership reconfiguration

etcd Learner

Gyuho Lee (github.com/gyuho, Amazon Web Services, Inc.), Joe Betz (github.com/jpbetz, Google Inc.)

Background

Membership reconfiguration has been one of the biggest operational challenges. Let's review common challenges.

1. New Cluster member overloads Leader

A newly joined etcd member starts with no data, thus demanding more updates from leader until it catches up with leader's logs. Then leader's network is more likely to be overloaded, blocking or dropping leader heartbeats to followers. In such case, a follower may election-timeout to start a new leader election. That is, a cluster with a new member is more vulnerable to leader election. Both leader election and the subsequent update propagation to the new member are prone to causing periods of cluster unavailability (see *Figure 1*).

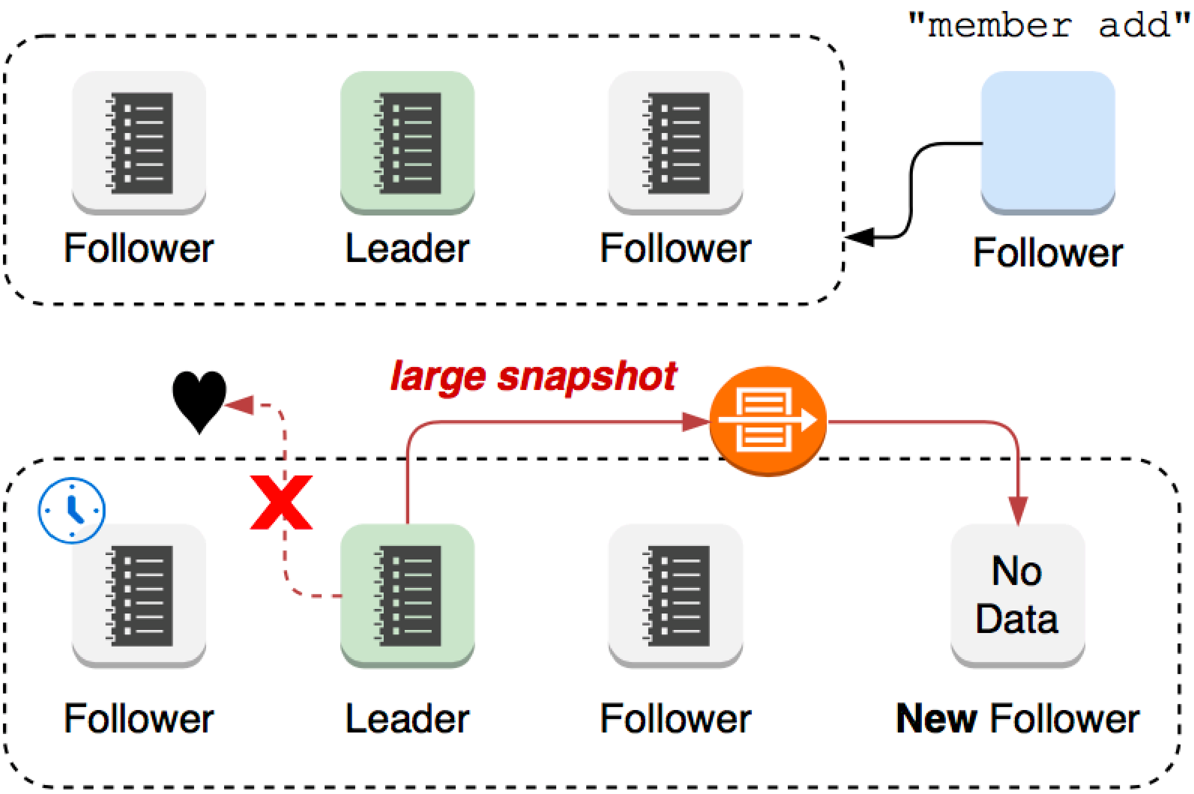


Figure 1. A new member joins with empty data, requesting more data from leader. Then leader becomes overloaded sending large snapshots. Which may block heartbeat sends. Then follower may election-timeout and start a new election.

2. Network Partitions scenarios

What if network partition happens? It depends on leader partition. If the leader still maintains the active quorum, the cluster would continue to operate (see *Figure 2*).

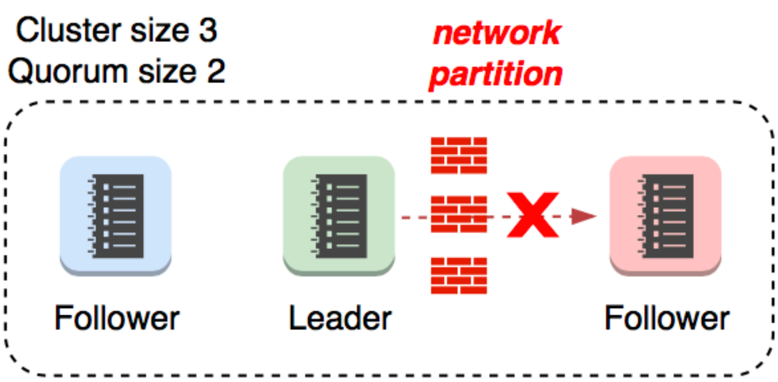


Figure 2. In 3-node cluster, a **follower gets isolated**. In this case, leader only requires 1 active follower (total 2 active nodes including leader). **No leadership election happens even with the network partition**, since the leader **still has 2 active nodes** and the size of quorum 2 is the minimum number of nodes required for cluster operation.

2.1 Leader isolation

What if the leader becomes isolated from the rest of the cluster? Leader monitors progress of each follower. When leader loses connectivity from the quorum, it reverts back to follower which will affect the cluster availability (see *Figure 3*).

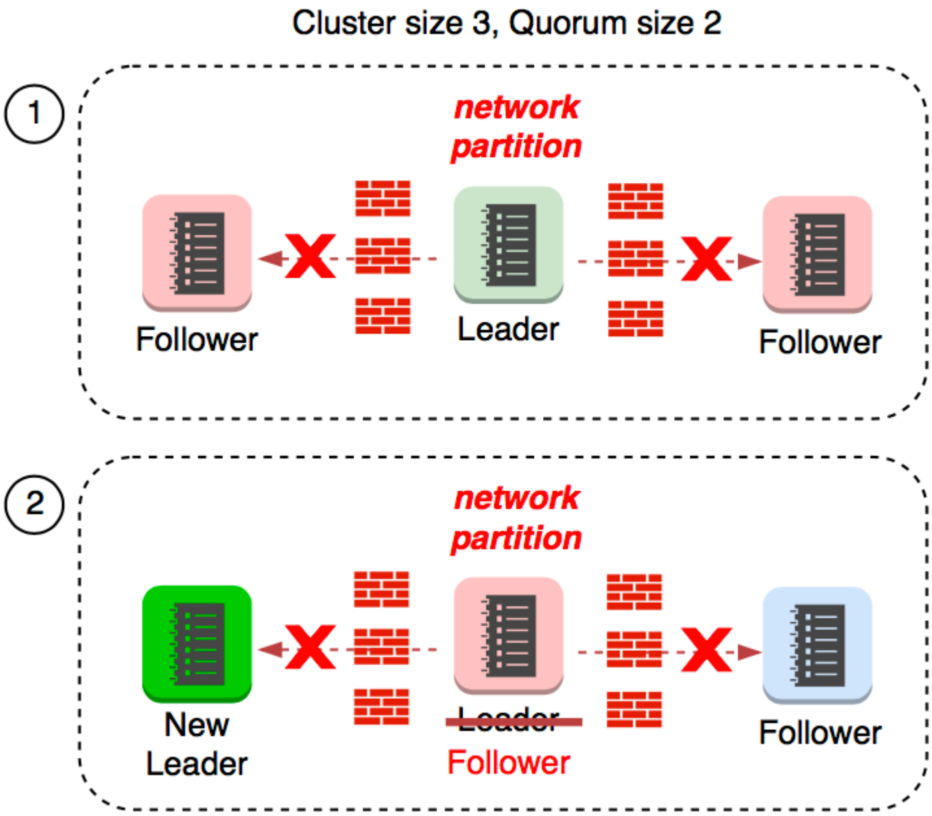


Figure 3. In 3-node cluster, the **leader node gets isolated**. In this case, leader requires at least 1 active follower (total 2 active nodes including leader). Leader had no active follower within its partition (lost quorum). Then, leader election will happen to elect a new leader.

When a new node is added to 3 node cluster, the cluster size becomes 4 and the quorum size becomes 3. What if a new node had joined the cluster, and then network partition happens? It depends on which partition the new member gets located after partition.

2.2 Cluster Split 3+1

If the new node happens to be located in the same partition as leader's, the leader still maintains the active quorum of 3. No leadership election happens, and no cluster availability gets affected (see *Figure 4*).

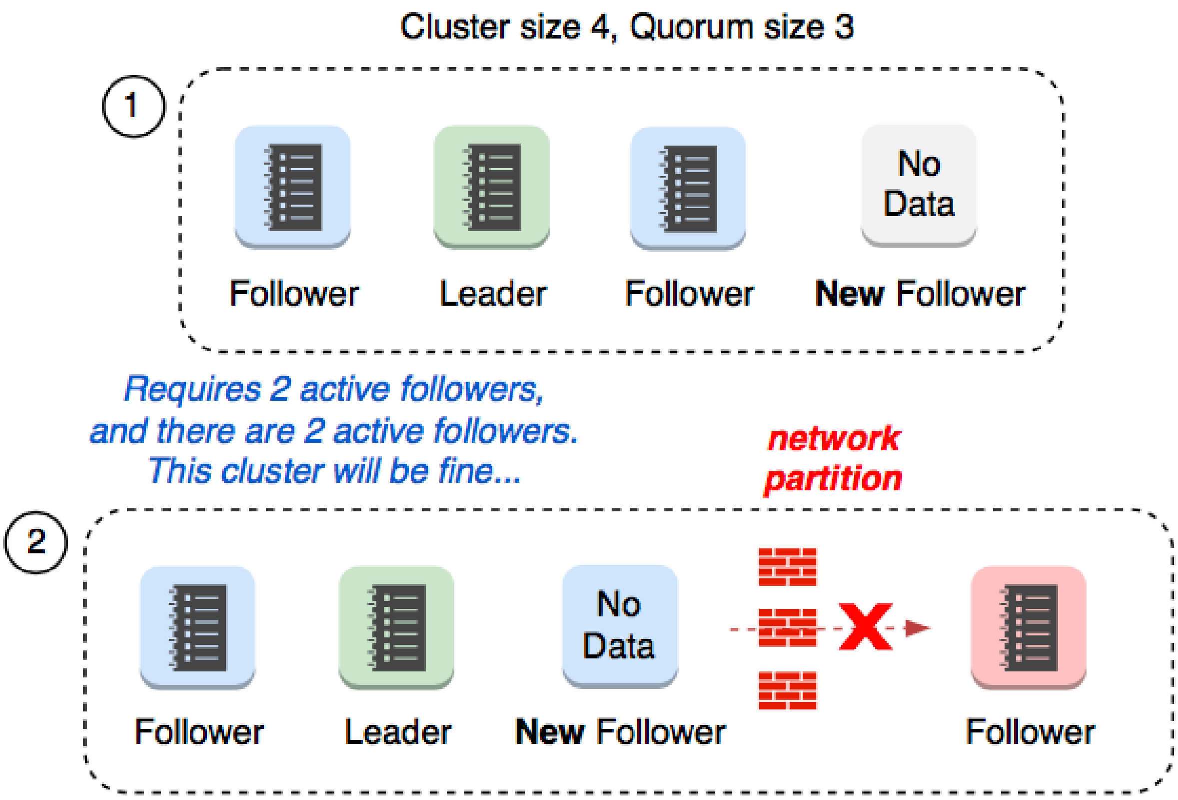


Figure 4. What if a new node has been added, and then network partition happens? When a new node joins 3-node cluster, quorum size increases to 3. And if the **new node happens to be in the same partition as leader's**, leader **still maintains the active quorum**, so cluster will continue to work under the network partition.

2.3 Cluster Split 2+2

If the cluster is 2-and-2 partitioned, then neither of partition maintains the quorum of 3. In this case, leadership election happens (see *Figure 5*).

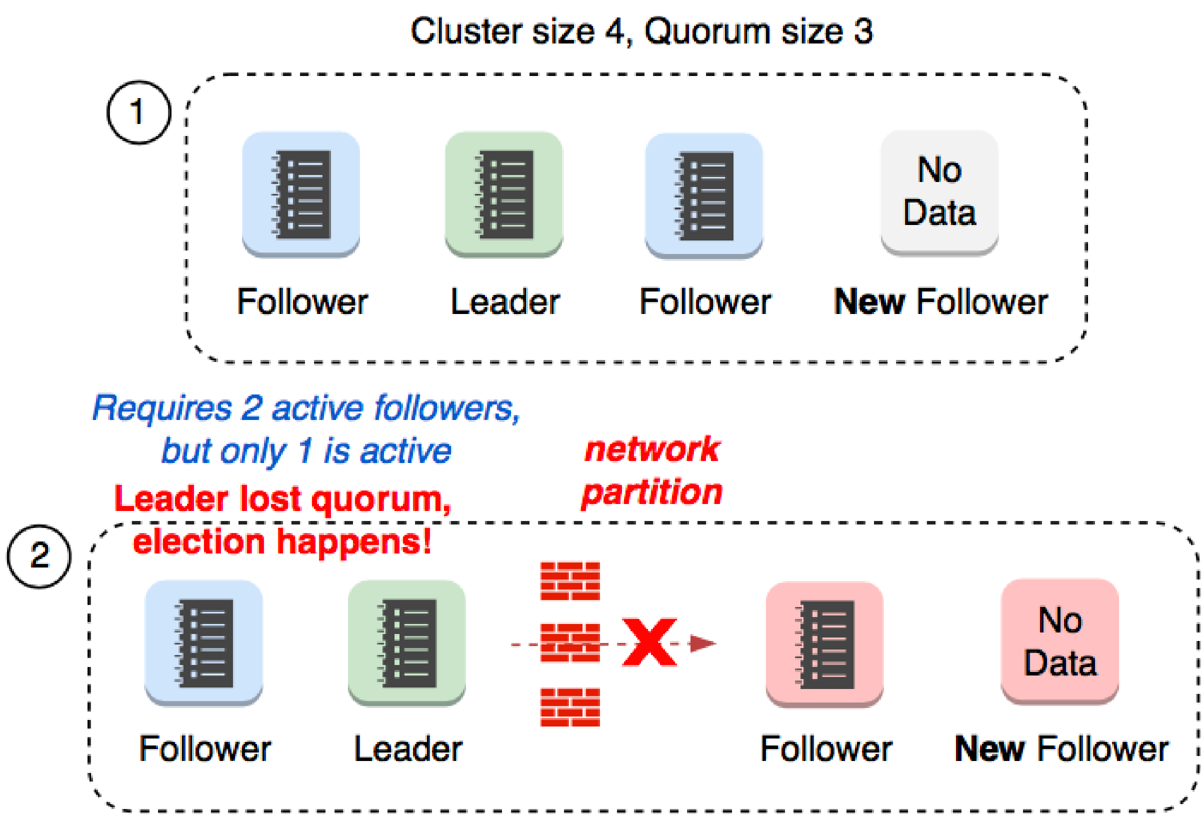


Figure 5. What if a new node has been added, and then network partition happens? When a new node joins 3-node cluster, quorum size increases to 3. In this case, **leader requires at least 2 active followers**, but due to network partition, there are **only 1 active follower**. Then, leader reverts back to follower.

2.4 Quorum Lost

What if network partition happens first, and then a new member gets added? A partitioned 3-node cluster already has one disconnected follower. When a new member is added, the quorum changes from 2 to 3. Now, this cluster has only 2 active nodes out 4, thus losing quorum and starting a new leadership election (see *Figure 6*).

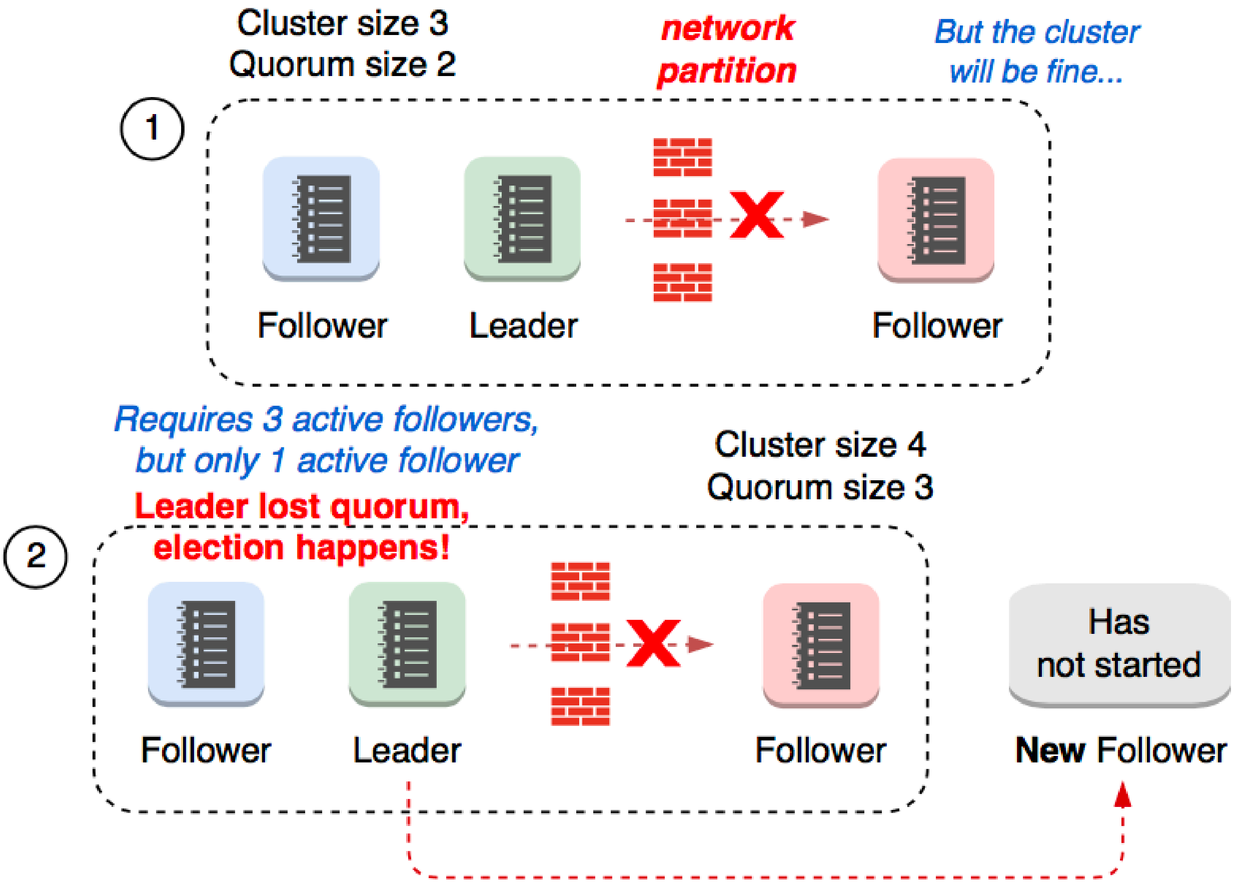


Figure 6. What if network partition has happened, and then a new node is added? When a new node is added to 3-node cluster (now 4-node cluster), quorum size changes from 2 to 3. **Since the new node is not started yet, the cluster now has only 2 active nodes and loses quorum, triggering a leadership election.**

Since member add operation can change the size of quorum, it is always recommended to “member remove” first to replace an unhealthy node.

Adding a new member to a 1-node cluster changes the quorum size to 2, immediately causing a leader election when the previous leader finds out quorum is not active. This is because “member add” operation is a 2-step process where user needs to apply “member add” command first, and then starts the new node process (see *Figure 7*).

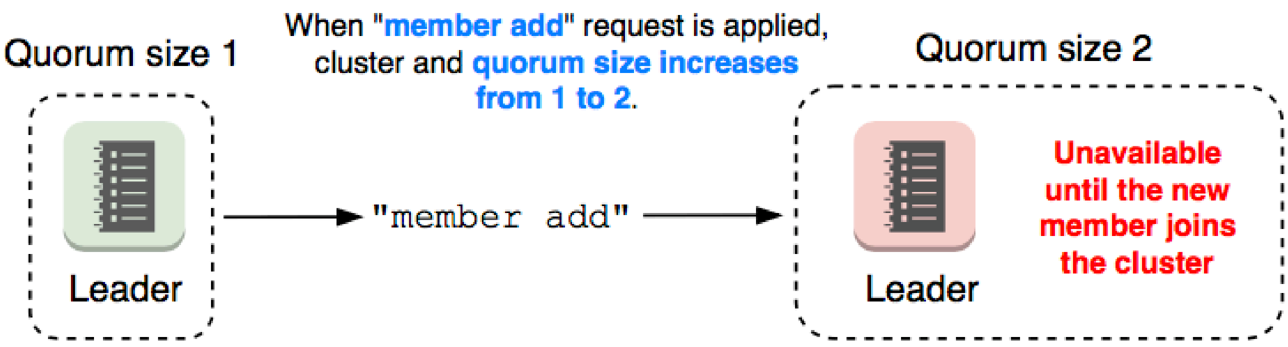


Figure 7. "member add" command to a single node cluster **increases the quorum size to 2,** causing an **immediate leader election**, because from previous leader's viewpoint, quorum is not active.

3. Cluster Misconfigurations

An even worse case is when an added member is misconfigured. Membership reconfiguration is a two-step process: "etcdctl member add" and starting an etcd server process with the given peer URL. That is, "member add" command is applied regardless of URL, even when the URL value is invalid. If the first step is applied with invalid URLs, the second step cannot even start the new etcd. Once the cluster loses quorum, there is no way to revert the membership change (see *Figure 8*).

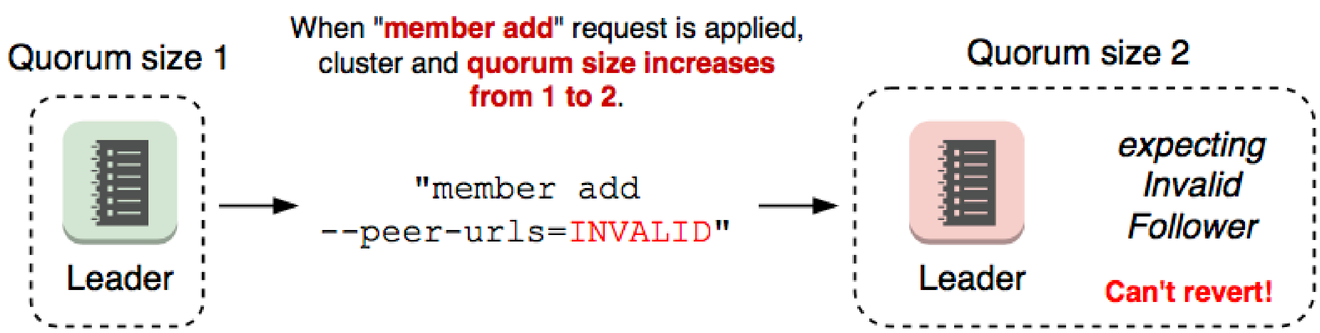


Figure 8. Imagine "member add" command was **misconfigured with a wrong URL**. The request is still applied to the 1-node cluster. And quorum size becomes 2. Then, **leader loses quorum**. Now, the whole cluster is inoperable.

Same applies to a multi-node cluster. For example, the cluster has two members down (one is failed, the other is misconfigured) and two members up, but now it requires at least 3 votes to change the cluster membership (see *Figure 9*).

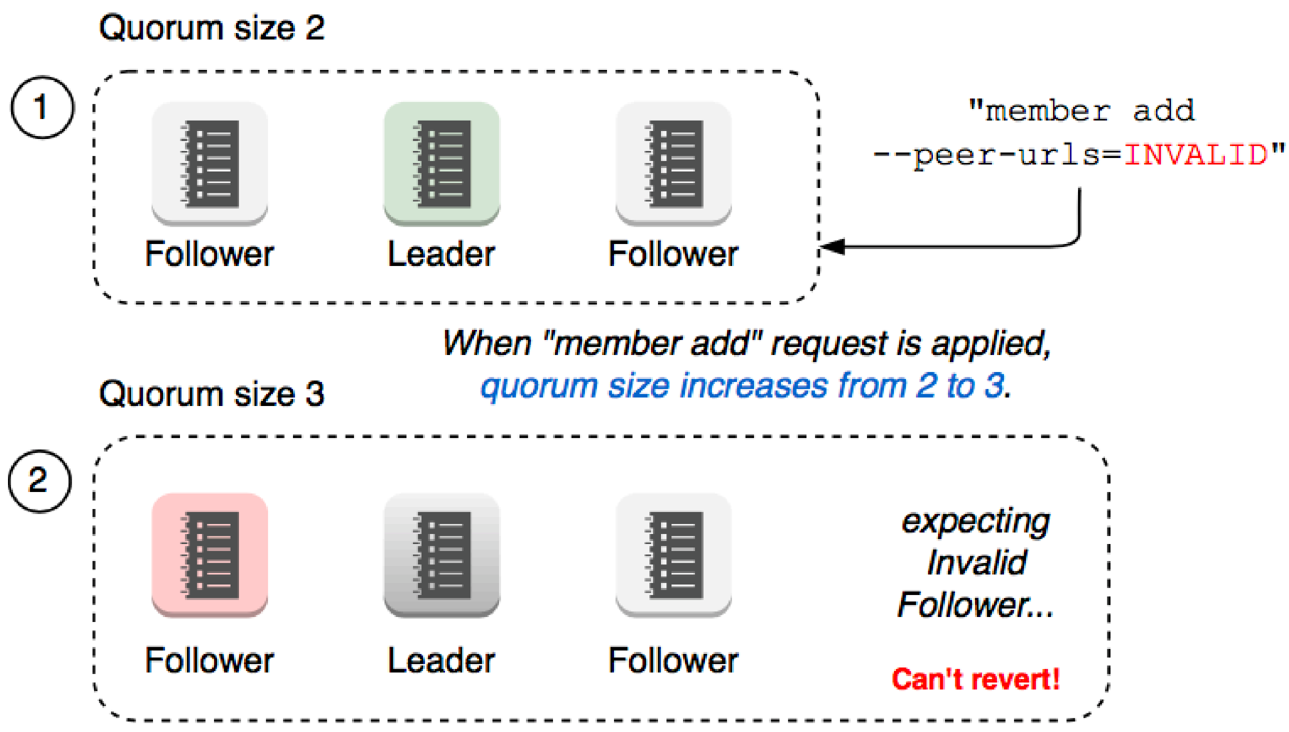


Figure 9. When a wrong member entry is applied to 3-node cluster, quorum size becomes 3. Which requires 3 votes (or 3 active nodes) to commit new entries. With one node being misconfigured but still counted in quorum, even one node failure makes the whole cluster unavailable.

As seen above, a simple misconfiguration can fail the whole cluster into an inoperative state. In such case, an operator need manually recreate the cluster with `etcd --force-new-cluster` flag. As etcd has become a mission-critical service for Kubernetes, even the slightest outage may have significant impact on users. What can we better to make etcd such operations easier? Among other things, leader election is most critical to cluster availability: Can we make membership reconfiguration less disruptive by not changing the size of quorum? Can a new node be idle, only requesting the minimum updates from leader, until it catches up? Can membership misconfiguration be always reversible and handled in a more secure way (wrong member add command run should never fail the cluster)? Should an user worry about network topology when adding a new member? Can member add API work regardless of the location of nodes and ongoing network partitions?

Raft Learner

In order to mitigate such availability gaps in the previous section, Raft §4.2.1 introduces a new node state “Learner”, which joins the cluster as a **non-voting member** until it catches up to leader’s logs.

Features in v3.4

An operator should do the minimum amount of work possible to add a new learner node. `member add --learner` command to add a new learner, which joins cluster as a non-voting member but still receives all data from leader (see *Figure 10*).

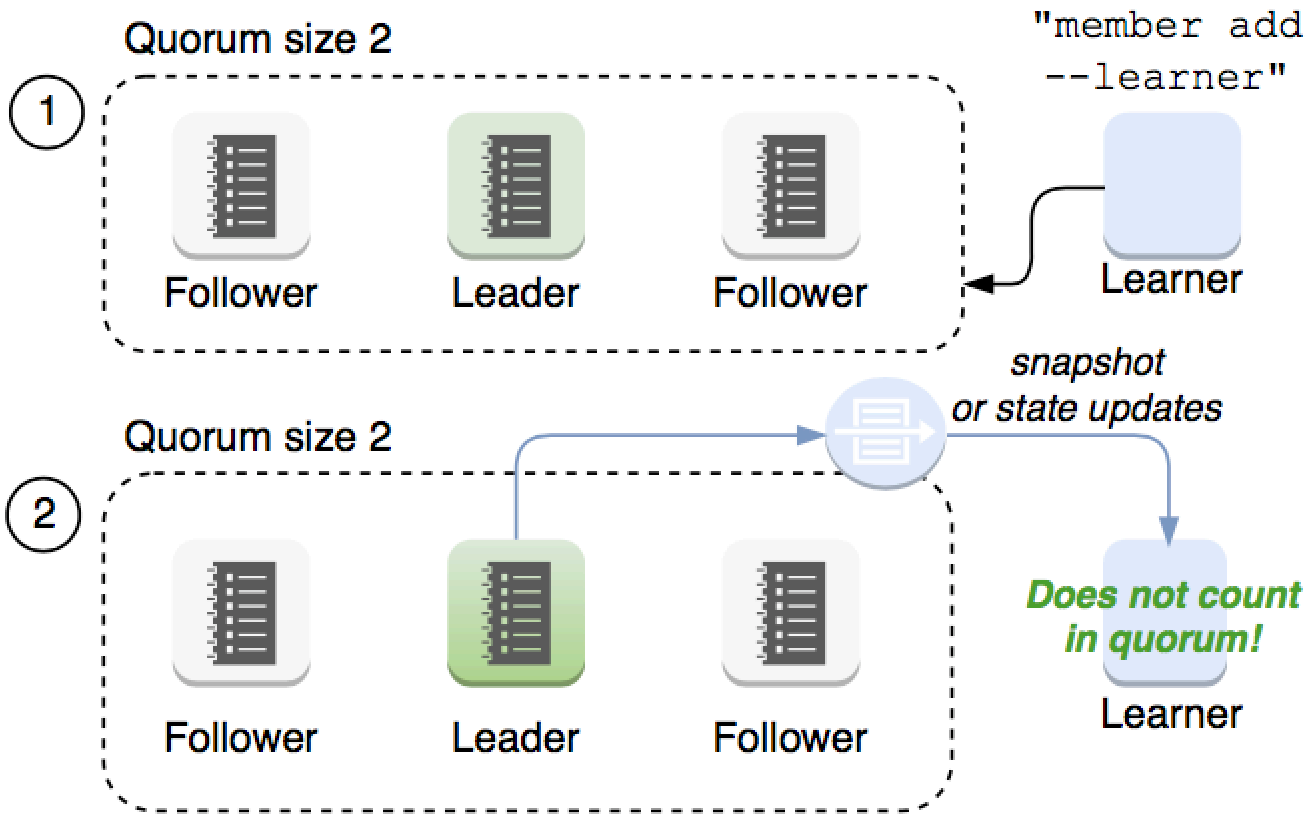


Figure 10. Add a learner node as a non-voting member. Wait until learner node catches up to leader's logs. Until then, learner node neither votes nor counts towards quorum.

When a learner has caught up with leader's progress, the learner can be promoted to a voting member using `member promote` API, which then counts towards the quorum (see *Figure 11*).

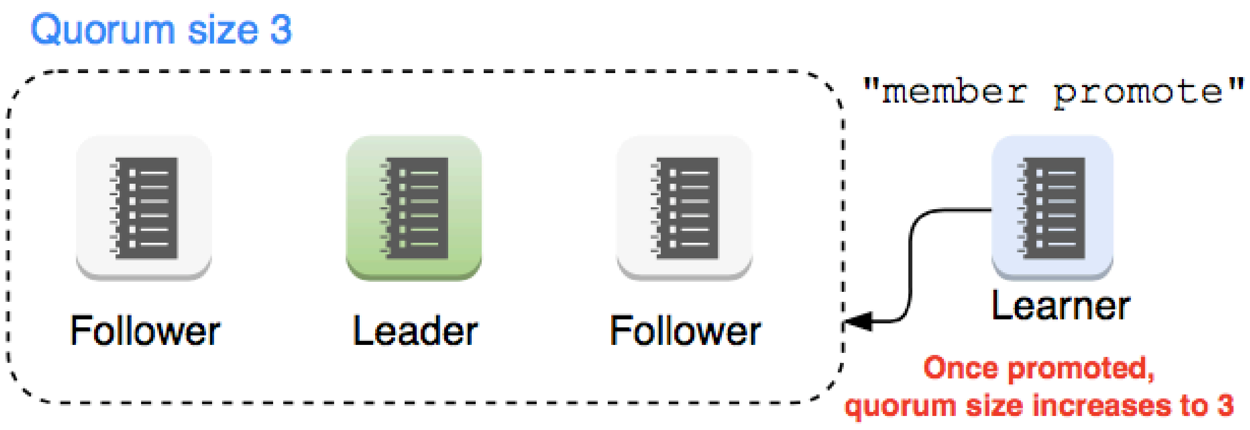


Figure 11. Once learner node has caught up to leader's log, "member promote" API can promote it to a normal voting node that counts towards quorum. In this case, it will increase the size of quorum to 3.

etcd server validates promote request to ensure its operational safety. Only after its log has caught up to leader's can learner be promoted to a voting member (see *Figure 12*).

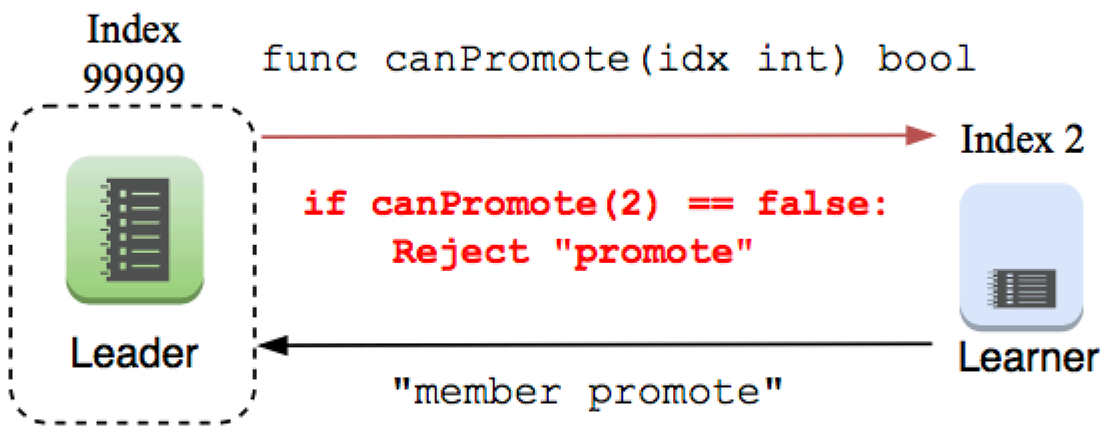


Figure 12. etcd v3.4 learner can be promoted only when it satisfies the safety requirement. Otherwise, promote request will be rejected.

Learner only serves as a standby node until promoted: Leadership cannot be transferred to learner. Learner rejects client reads and writes (client balancer should not route requests to learner). Which means learner does not need issue Read Index requests to leader. Such limitation simplifies the initial learner implementation in v3.4 release (see *Figure 13*).

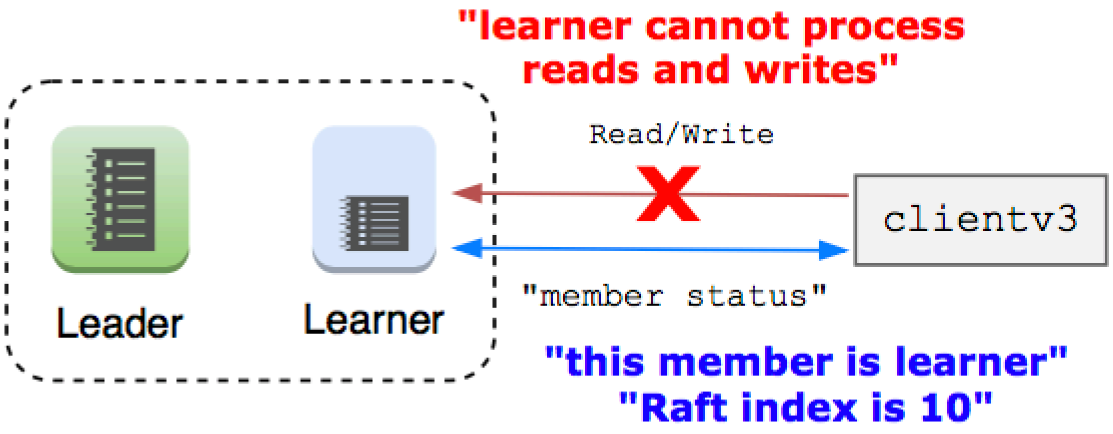


Figure 13. etcd v3.4 learner serves as a standby node only. Learner node rejects client reads and writes but allows status checks.

In addition, etcd limits the total number of learners that a cluster can have, and avoids overloading the leader with log replication. Learner never promotes itself. While etcd provides learner status information and safety checks, cluster operator must make the final decision whether to promote learner or not.

Proposed features for future releases

Make learner state only and default: Defaulting a new member state to learner will greatly improve membership reconfiguration safety, because learner does not change the size of quorum. Misconfiguration will always be reversible without losing the quorum.

Make voting-member promotion fully automatic: Once a learner catches up to leader's logs, a cluster can automatically promote the learner. etcd requires certain thresholds to be defined by the user, and once the requirements are satisfied, learner promotes itself to a voting member. From a user's perspective, "member add" command would work the same way as today but with greater safety provided by learner feature.

Make learner standby failover node: A learner joins as a standby node, and gets automatically promoted when the cluster availability is affected.

Make learner read-only: A learner can serve as a read-only node that never gets promoted. In a weak consistency mode, learner only receives data from leader and never process writes. Serving reads locally without consensus overhead would greatly decrease the workloads to leader but may serve stale data. In a strong consistency mode, learner requests read index from leader to serve latest data, but still rejects writes.

Learner vs. Mirror Maker

etcd implements “mirror maker” using watch API to continuously relay key creates and updates to a separate cluster. Mirroring usually has low latency overhead once it completes initial synchronization. Learner and mirroring overlap in that both can be used to replicate existing data for read-only. However, mirroring does not guarantee linearizability. During network disconnects, previous key-values might have been discarded, and clients are expected to verify watch responses for correct ordering. Thus, there is no ordering guarantee in mirror. Use mirror for minimum latency (e.g. cross data center) at the costs of consistency. Use learner to retain all historical data and its ordering.

Appendix: Learner Implementation in v3.4

Expose “Learner” node type to “MemberAdd” API.

etcd client adds a flag to “MemberAdd” API for learner node. And etcd server handler applies membership change entry with `pb.ConfChangeAddLearnerNode` type. Once the command has been applied, a server joins the cluster with `etcd --initial-cluster-state=existing` flag. This learner node can neither vote nor count as quorum.

etcd server must not transfer leadership to learner, since it may still lag behind and does not count as quorum. etcd server limits the number of learners that cluster can have to one: the more learners we have, the more data the leader has to propagate. Clients may talk to learner node, but learner rejects all requests other than serializable read and member status API. This is for simplicity of initial implementation. In the future, learner can be extended as a read-only server that continuously mirrors cluster data. Client balancer must provide helper function to exclude learner node endpoint. Otherwise, request sent to learner may fail. Client sync member call should factor into learner node type. So should client endpoints update call.

`MemberList` and `MemberStatus` responses should indicate which node is learner.

Add “MemberPromote” API.

Internally in Raft, second `MemberAdd` call to learner node promotes it to a voting member. Leader maintains the progress of each follower and learner. If learner has not completed its snapshot message, reject promote request. Only accept promote request if and only if: The learner node is in a healthy state. The learner is in sync with leader or the delta is within the threshold (e.g. the number of entries to replicate to learner is less than 1/10 of snapshot count, which means it is less likely that even after promotion leader would not need send snapshot to the learner). All these logic are hard-coded in `etcdserver` package and not configurable.

Reference

- Original github issue: [etcd#9161]
- Use case: [etcd#3715]

- Use case: etcd#8888
- Use case: etcd#10114

Last modified July 23, 2021: learners: clarify features as future vs v3.5 (9dea14b)

etcd v3 authentication design

etcd v3 authentication

Why not reuse the v2 auth system?

The v3 protocol uses gRPC as its transport instead of a RESTful interface like v2. This new protocol provides an opportunity to iterate on and improve the v2 design. For example, v3 auth has connection based authentication, rather than v2's slower per-request authentication. Additionally, v2 auth's semantics tend to be unwieldy in practice with respect to reasoning about consistency, which will be described in the next sections. For v3, there is a well-defined description and implementation of the authentication mechanism which fixes the deficiencies in the v2 auth system.

Functionality requirements

- Per connection authentication, not per request
    - User ID + password based authentication implemented for the gRPC API
    - Authentication must be refreshed after auth policy changes
- Its functionality should be as simple and useful as v2
    - v3 provides a flat key space, unlike the directory structure of v2. Permission checking will be provided as interval matching.
- It should have stronger consistency guarantees than v2 auth

Main required changes

- A client must create a dedicated connection only for authentication before sending authenticated requests
- Add permission information (user ID and authorized revision) to the Raft commands (`etcdserverpb.InternalRaftRequest`)
- Every request is permission checked in the state machine layer, rather than API layer

Permission metadata consistency

The metadata for auth should also be stored and managed in the storage controlled by etcd's Raft protocol like other data stored in etcd. It is required for not sacrificing availability and consistency of the entire etcd cluster. If reading or writing the metadata (e.g. permission information) needs an agreement of every node (more than quorum), single node failure can stop the entire cluster. Requiring all nodes to agree at once means that checking ordinary read/write requests cannot be completed if any cluster member is down, even if the cluster has an available quorum. This unanimous scheme ultimately degrades cluster availability; quorum based consensus from raft should suffice since agreement follows from consistent ordering.

The authentication mechanism in the etcd v2 protocol has a tricky part because the metadata consistency should work as in the above, but does not: each permission check is processed by the etcd member that receives the client request (etcdserver/api/v2http/client.go), including follower members. Therefore, it's possible the check may be based on stale metadata.

This staleness means that auth configuration cannot be reflected as soon as operators execute etcdctl. Therefore there is no way to know how long the stale metadata is active. Practically, the configuration change is reflected immediately after the command execution. However, in some cases of heavy load, the inconsistent state can be prolonged and it might result in counter-intuitive situations for users and developers. It requires a workaround like this: https://github.com/etcd-io/etcd/pull/4317#issuecomment-179037582

Inconsistent permissions are unsafe for linearized requests

Inconsistent authentication state is most serious for writes. Even if an operator disables write on a user, if the write is only ordered with respect to the key value store but not the authentication system, it's possible the write will complete successfully. Without ordering on both the auth store and the key-value store, the system will be susceptible to stale permission attacks.

Therefore, the permission checking logic should be added to the state machine of etcd. Each state machine should check the requests based on its permission information in the apply phase (so the auth information must not be stale).

Design and implementation

Authentication

At first, a client must create a gRPC connection only to authenticate its user ID and password. An etcd server will respond with an authentication reply. The response will be an authentication token on success or an error on failure. The client can use its authentication token to present its credentials to etcd when making API requests.

The client connection used to request the authentication token is typically thrown away; it cannot carry the new token's credentials. This is because gRPC doesn't provide a way for adding per RPC credential after creation of the connection (calling `grpc.Dial()`). Therefore, a client cannot assign a token to its connection that is obtained through the connection. The client needs a new connection for using the token.

Notes on the implementation of `Authenticate()` RPC

`Authenticate()` RPC generates an authentication token based on a given user name and password. etcd saves and checks a configured password and a given password using Go's `bcrypt` package. By design, `bcrypt`'s password checking mechanism is computationally expensive, taking nearly 100ms on an ordinary x64 server. Therefore, performing this check in the state machine apply phase would cause performance trouble: the entire etcd cluster can only serve almost 10 `Authenticate()` requests per second.

For good performance, the v3 auth mechanism checks passwords in etcd's API layer, where it can be parallelized outside of raft. However, this can lead to potential time-of-check/time-of-use (TOCTOU) permission lapses:

1. client A sends a request `Authenticate()`
2. the API layer processes the password checking part of `Authenticate()`
3. another client B sends a request of `ChangePassword()` and the server completes it
4. the state machine layer processes the part of getting a revision number for the `Authenticate()` from A
5. the server returns a success to A
6. now A is authenticated on an obsolete password

For avoiding such a situation, the API layer performs *version number validation* based on the revision number of the auth store. During password checking, the API layer saves the revision number of auth store. After successful password checking, the API layer compares the saved revision number and the latest revision number. If the numbers differ, it means someone else updated the auth metadata. So it retries the checking. With this mechanism, the successful password checking based on the obsolete password can be avoided.

Resolving a token in the API layer

After authenticating with `Authenticate()`, a client can create a gRPC connection as it would without auth. In addition to the existing initialization process, the client must associate the token with the newly created connection. `grpc.WithPerRPCCredentials()` provides the functionality for this purpose.

Every authenticated request from the client has a token. The token can be obtained with `grpc.metadata.FromIncomingContext()` in the server side. The server can obtain who is issuing the request and when the user was authorized. The information will be filled by the API layer in the header (`etcdserverpb.RequestHeader.Username` and `etcdserverpb.RequestHeader.AuthRevision`) of a raft log entry (`etcdserverpb.InternalRaftRequest`).

Checking permission in the state machine

The auth info in `etcdserverpb.RequestHeader` is checked in the apply phase of the state machine. This step checks the user is granted permission to requested keys on the latest revision of auth store.

Two types of tokens: simple and JWT

There are two kinds of token types: simple and JWT. The simple token isn't designed for production use cases. Its tokens aren't cryptographically signed and servers must statefully track token-user correspondence; it is meant for development testing. JWT tokens should be used for production deployments since it is cryptographically signed and verified. From the implementation perspective, JWT is stateless. Its token can include metadata including username and revision, so servers don't need to remember correspondence between tokens and the metadata.

Note : There is a known issue #18437 with simple tokens. Within etcd servers, tokens are resolved at the API layer and simple tokens are stateful. The process is not protected by a linearizable check, meaning an etcd member may not have completed processing a previous authentication request before receiving the next one. In such cases, the member might return an "invalid auth token" error to the client. This issue is usually rare on a node with good network conditions but can occur if there is significant latency. As a workaround, applications can implement a retry mechanism to handle this error.

Notes on the difference between KVS models and file system models

etcd v3 is a KVS, not a file system. So the permissions can be granted to the users in form of an exact key name or a key range like `["start key", "end key")`. It means that granting a permission of a nonexistent key is possible. Users should care about unintended permission granting. In a case of file system like system (e.g. Chubby or ZooKeeper), an inode like data structure can include the permission information. So granting permission to a nonexist key won't be possible (except the case of sticky bits).

The etcd v3 model requires multiple lookup of the metadata unlike the file system like systems. The worst case lookup cost will be sum the user's total granted keys and intervals. The cost cannot be avoided because v3's flat key space is completely different from Unix's file system model (every inode includes permission metadata). Practically the cost won't be a serious problem because the metadata is small enough to benefit from caching.

Last modified August 19, 2024: update doc auth design with simple token known issue (65d913f)

etcd3 API

etcd3 API central design overview

This document is meant to give an overview of the etcd3 API's central design. It is by no means all encompassing, but intended to focus on the basic ideas needed to understand etcd without the distraction of less common API calls. All etcd3 API's are defined in gRPC services, which categorize remote procedure calls (RPCs) understood by the etcd server. A full listing of all etcd RPCs are documented in markdown in the gRPC API listing.

gRPC Services

Every API request sent to an etcd server is a gRPC remote procedure call. RPCs in etcd3 are categorized based on functionality into services.

Services important for dealing with etcd's key space include:

- KV - Creates, updates, fetches, and deletes key-value pairs.
- Watch - Monitors changes to keys.
- Lease - Primitives for consuming client keep-alive messages.

Services which manage the cluster itself include:

- Auth - Role based authentication mechanism for authenticating users.
- Cluster - Provides membership information and configuration facilities.
- Maintenance - Takes recovery snapshots, defragments the store, and returns per-member status information.

Requests and Responses

All RPCs in etcd3 follow the same format. Each RPC has a function `Name` which takes `NameRequest` as an argument and returns `NameResponse` as a response. For example, here is the `Range` RPC description:

```
service KV {
  Range(RangeRequest) returns (RangeResponse)
```

```
  ...
}
```

Response header

All Responses from etcd API have an attached response header which includes cluster metadata for the response:

```
message ResponseHeader {
  uint64 cluster_id = 1;
  uint64 member_id = 2;
  int64 revision = 3;
  uint64 raft_term = 4;
}
```

- Cluster_ID - the ID of the cluster generating the response.
- Member_ID - the ID of the member generating the response.
- Revision - the revision of the key-value store when generating the response.
- Raft_Term - the Raft term of the member when generating the response.

An application may read the `Cluster_ID` or `Member_ID` field to ensure it is communicating with the intended cluster (member).

Applications can use the `Revision` field to know the latest revision of the key-value store. This is especially useful when applications specify a historical revision to make a `time travel query` and wish to know the latest revision at the time of the request.

Applications can use `Raft_Term` to detect when the cluster completes a new leader election.

Key-Value API

The Key-Value API manipulates key-value pairs stored inside etcd. The majority of requests made to etcd are usually key-value requests.

System primitives

Key-Value pair

A key-value pair is the smallest unit that the key-value API can manipulate. Each key-value pair has a number of fields, defined in [protobuf format](#):

```
message KeyValue {
  bytes key = 1;
  int64 create_revision = 2;
  int64 mod_revision = 3;
  int64 version = 4;
  bytes value = 5;
  int64 lease = 6;
}
```

- Key - key in bytes. An empty key is not allowed.
- Value - value in bytes.
- Version - version is the version of the key. A deletion resets the version to zero and any modification of the key increases its version.
- Create_Revision - revision of the last creation on the key.
- Mod_Revision - revision of the last modification on the key.
- Lease - the ID of the lease attached to the key. If lease is 0, then no lease is attached to the key.

In addition to just the key and value, etcd attaches additional revision metadata as part of the key message. This revision information orders keys by time of creation and modification, which is useful for managing concurrency for distributed synchronization. The etcd client's distributed shared locks use the creation revision to wait for lock ownership. Similarly, the modification revision is used for detecting software transactional memory read set conflicts and waiting on leader election updates.

Revisions

etcd maintains a 64-bit cluster-wide counter, the store revision, that is incremented each time the key space is modified. The revision serves as a global logical clock, sequentially ordering all updates to the store. The change represented by a new revision is incremental; the data associated with a revision is the data that changed the store. Internally, a new revision means writing the changes to the backend's B+tree, keyed by the incremented revision.

Revisions become more valuable when considering etcd3's multi-version concurrency control backend. The MVCC model means that the key-value store can be viewed from past revisions since historical key revisions are retained. The retention policy for this history can be configured by cluster administrators for fine-grained storage management; usually etcd3 discards old revisions of keys on a timer. A typical etcd3 cluster retains superseded key data for hours. This also provides reliable handling for long client disconnection, not just transient network disruptions: watchers simply resume from the last observed historical revision. Similarly, to read from the store at a particular point-in-time, read requests can be tagged with a revision to return keys from a view of the key space at the point-in-time that revision was committed.

Key ranges

The etcd3 data model indexes all keys over a flat binary key space. This differs from other key-value store systems that use a hierarchical system of organizing keys into directories. Instead of listing keys by directory, keys are listed by key intervals `[a, b)`.

These intervals are often referred to as "ranges" in etcd3. Operations over ranges are more powerful than operations on directories. Like a hierarchical store, intervals support single key lookups via `[a, a+1)` (e.g., ['a', 'a\x00') looks up 'a') and directory lookups by encoding keys by directory depth. In addition to those operations, intervals can also encode prefixes; for example the interval `['a', 'b')` looks up all keys prefixed by the string 'a'.

By convention, ranges for a request are denoted by the fields `key` and `range_end`. The `key` field is the first key of the range and should be non-empty. The `range_end` is the key following the last key of the range. If `range_end` is not given or empty, the range is defined to contain only the key argument. If `range_end` is `key` plus one (e.g., "aa"+1 == "ab", "a\xff"+1 == "b"), then the range represents all keys prefixed with key. If both `key` and `range_end` are '\0', then range represents all keys. If `range_end` is '\0', the range is all keys greater than or equal to the key argument.

Range

Keys are fetched from the key-value store using the `Range` API call, which takes a `RangeRequest`:

```
message RangeRequest {
  enum SortOrder {
    NONE = 0; // default, no sorting
    ASCEND = 1; // lowest target value first
    DESCEND = 2; // highest target value first
  }
  enum SortTarget {
    KEY = 0;
    VERSION = 1;
    CREATE = 2;
    MOD = 3;
    VALUE = 4;
  }

  bytes key = 1;
  bytes range_end = 2;
  int64 limit = 3;
```

```
  int64 revision = 4;
  SortOrder sort_order = 5;
  SortTarget sort_target = 6;
  bool serializable = 7;
  bool keys_only = 8;
  bool count_only = 9;
  int64 min_mod_revision = 10;
  int64 max_mod_revision = 11;
  int64 min_create_revision = 12;
  int64 max_create_revision = 13;
}
```

- Key, Range_End - The key range to fetch.
- Limit - the maximum number of keys returned for the request. When limit is set to 0, it is treated as no limit.
- Revision - the point-in-time of the key-value store to use for the range. If revision is less or equal to zero, the range is over the latest key-value store. If the revision is compacted, ErrCompacted is returned as a response.
- Sort_Order - the ordering for sorted requests.
- Sort_Target - the key-value field to sort.
- Serializable - sets the range request to use serializable member-local reads. By default, Range is linearizable; it reflects the current consensus of the cluster. For better performance and availability, in exchange for possible stale reads, a serializable range request is served locally without needing to reach consensus with other nodes in the cluster.
- Keys_Only - return only the keys and not the values.
- Count_Only - return only the count of the keys in the range.
- Min_Mod_Revision - the lower bound for key mod revisions; filters out lesser mod revisions.
- Max_Mod_Revision - the upper bound for key mod revisions; filters out greater mod revisions.
- Min_Create_Revision - the lower bound for key create revisions; filters out lesser create revisions.
- Max_Create_Revision - the upper bound for key create revisions; filters out greater create revisions.

The client receives a `RangeResponse` message from the `Range` call:

```
message RangeResponse {
  ResponseHeader header = 1;
  repeated mvccpb.KeyValue kvs = 2;
  bool more = 3;
```

```
  int64 count = 4;
}
```

- Kvs - the list of key-value pairs matched by the range request. When `Count_Only` is set, `Kvs` is empty.
- More - indicates if there are more keys to return in the requested range if `limit` is set.
- Count - the total number of keys satisfying the range request.

Put

Keys are saved into the key-value store by issuing a `Put` call, which takes a `PutRequest`:

```
message PutRequest {
  bytes key = 1;
  bytes value = 2;
  int64 lease = 3;
  bool prev_kv = 4;
  bool ignore_value = 5;
  bool ignore_lease = 6;
}
```

- Key - the name of the key to put into the key-value store.
- Value - the value, in bytes, to associate with the key in the key-value store.
- Lease - the lease ID to associate with the key in the key-value store. A lease value of 0 indicates no lease.
- Prev_Kv - when set, responds with the key-value pair data before the update from this `Put` request.
- Ignore_Value - when set, update the key without changing its current value. Returns an error if the key does not exist.
- Ignore_Lease - when set, update the key without changing its current lease. Returns an error if the key does not exist.

The client receives a `PutResponse` message from the `Put` call:

```
message PutResponse {
  ResponseHeader header = 1;
  mvccpb.KeyValue prev_kv = 2;
}
```

- Prev_Kv - the key-value pair overwritten by the `Put`, if `Prev_Kv` was set in the `PutRequest`.

Delete Range

Ranges of keys are deleted using the `DeleteRange` call, which takes a `DeleteRangeRequest`:

```
message DeleteRangeRequest {
  bytes key = 1;
  bytes range_end = 2;
  bool prev_kv = 3;
}
```

- Key, Range_End - The key range to delete.
- Prev_Kv - when set, return the contents of the deleted key-value pairs.

The client receives a `DeleteRangeResponse` message from the `DeleteRange` call:

```
message DeleteRangeResponse {
  ResponseHeader header = 1;
  int64 deleted = 2;
  repeated mvccpb.KeyValue prev_kvs = 3;
}
```

- Deleted - number of keys deleted.
- Prev_Kv - a list of all key-value pairs deleted by the `DeleteRange` operation.

Transaction

A transaction is an atomic If/Then/Else construct over the key-value store. It provides a primitive for grouping requests together in atomic blocks (i.e., then/else) whose execution is guarded (i.e., if) based on the contents of the key-value store. Transactions can be used for protecting keys from unintended concurrent updates, building compare-and-swap operations, and developing higher-level concurrency control.

A transaction can atomically process multiple requests in a single request. For modifications to the key-value store, this means the store's revision is incremented only once for the transaction and all events generated by the transaction will have the same revision. However, modifications to the same key multiple times within a single transaction are forbidden.

All transactions are guarded by a conjunction of comparisons, similar to an `If` statement. Each comparison checks a single key in the store. It may check for the absence or presence of a value, compare with a given value, or check a key's revision or version. Two different comparisons may apply to the same or different keys. All comparisons are applied atomically; if all comparisons are true, the transaction is said to succeed and etcd applies the transaction's then / `success` request block, otherwise it is said to fail and applies the else / `failure` request block.

Each comparison is encoded as a `Compare` message:

```
message Compare {
  enum CompareResult {
    EQUAL = 0;
    GREATER = 1;
    LESS = 2;
    NOT_EQUAL = 3;
  }
  enum CompareTarget {
    VERSION = 0;
    CREATE = 1;
    MOD = 2;
    VALUE= 3;
  }
  CompareResult result = 1;
  // target is the key-value field to inspect for the comparison.
  CompareTarget target = 2;
  // key is the subject key for the comparison operation.
  bytes key = 3;
  oneof target_union {
    int64 version = 4;
    int64 create_revision = 5;
    int64 mod_revision = 6;
    bytes value = 7;
  }
}
```

- Result - the kind of logical comparison operation (e.g., equal, less than, etc).
- Target - the key-value field to be compared. Either the key's version, create revision, modification revision, or value.
- Key - the key for the comparison.
- Target_Union - the user-specified data for the comparison.

After processing the comparison block, the transaction applies a block of requests. A block is a list of `RequestOp` messages:

```
message RequestOp {
  // request is a union of request types accepted by a transaction.
  oneof request {
    RangeRequest request_range = 1;
    PutRequest request_put = 2;
    DeleteRangeRequest request_delete_range = 3;
  }
}
```

- Request_Range - a `RangeRequest`.
- Request_Put - a `PutRequest`. The keys must be unique. It may not share keys with any other Puts or Deletes.
- Request_Delete_Range - a `DeleteRangeRequest`. It may not share keys with any Puts or Deletes requests.

All together, a transaction is issued with a `Txn` API call, which takes a `TxnRequest`:

```
message TxnRequest {
  repeated Compare compare = 1;
  repeated RequestOp success = 2;
  repeated RequestOp failure = 3;
}
```

- Compare - A list of predicates representing a conjunction of terms for guarding the transaction.
- Success - A list of requests to process if all compare tests evaluate to true.
- Failure - A list of requests to process if any compare test evaluates to false.

The client receives a `TxnResponse` message from the `Txn` call:

```
message TxnResponse {
  ResponseHeader header = 1;
  bool succeeded = 2;
  repeated ResponseOp responses = 3;
}
```

- Succeeded - Whether `Compare` evaluated to true or false.
- Responses - A list of responses corresponding to the results from applying the `Success` block if succeeded is true or the `Failure` if succeeded is false.

The `Responses` list corresponds to the results from the applied `RequestOp` list, with each response encoded as a `ResponseOp`:

```
message ResponseOp {
  oneof response {
    RangeResponse response_range = 1;
    PutResponse response_put = 2;
    DeleteRangeResponse response_delete_range = 3;
  }
}
```

The `ResponseHeader` included in each inner response shouldn't be interpreted in any way. If clients need to get the latest revision, then they should always check the top level `ResponseHeader` in `TxnResponse`.

Watch API

The `Watch` API provides an event-based interface for asynchronously monitoring changes to keys. An etcd3 watch waits for changes to keys by continuously watching from a given revision, either current or historical, and streams key updates back to the client.

Events

Every change to every key is represented with `Event` messages. An `Event` message provides both the update's data and the type of update:

```
message Event {
  enum EventType {
    PUT = 0;
    DELETE = 1;
  }
  EventType type = 1;
  KeyValue kv = 2;
  KeyValue prev_kv = 3;
}
```

- Type - The kind of event. A PUT type indicates new data has been stored to the key. A DELETE indicates the key was deleted.
- KV - The KeyValue associated with the event. A PUT event contains current kv pair. A PUT event with kv.Version=1 indicates the creation of a key. A DELETE event contains the

deleted key with its modification revision set to the revision of deletion.
- Prev_KV - The key-value pair for the key from the revision immediately before the event. To save bandwidth, it is only filled out if the watch has explicitly enabled it.

Watch streams

Watches are long-running requests and use gRPC streams to stream event data. A watch stream is bi-directional; the client writes to the stream to establish watches and reads to receive watch events. A single watch stream can multiplex many distinct watches by tagging events with per-watch identifiers. This multiplexing helps reducing the memory footprint and connection overhead on the core etcd cluster.

To read about guarantees made about watch events, please read [etcd api guarantees].

A client creates a watch by sending a `WatchCreateRequest` over a stream returned by `Watch`:

```
message WatchCreateRequest {
  bytes key = 1;
  bytes range_end = 2;
  int64 start_revision = 3;
  bool progress_notify = 4;

  enum FilterType {
    NOPUT = 0;
    NODELETE = 1;
  }
  repeated FilterType filters = 5;
  bool prev_kv = 6;
}
```

- Key, Range_End - The key range to watch.
- Start_Revision - An optional revision for where to inclusively begin watching. If not given, it will stream events following the revision of the watch creation response header revision. The entire available event history can be watched starting from the last compaction revision.
- Progress_Notify - When set, the watch will periodically receive a WatchResponse with no events, if there are no recent events. It is useful when clients wish to recover a disconnected watcher starting from a recent known revision. The etcd server decides how often to send notifications based on current server load.
- Filters - A list of event types to filter away at server side.
- Prev_Kv - When set, the watch receives the key-value data from before the event happens. This is useful for knowing what data has been overwritten.

In response to a `WatchCreateRequest` or if there is a new event for some established watch, the client receives a `WatchResponse`:

```
message WatchResponse {
  ResponseHeader header = 1;
  int64 watch_id = 2;
  bool created = 3;
  bool canceled = 4;
  int64 compact_revision = 5;

  repeated mvccpb.Event events = 11;
}
```

- Watch_ID - the ID of the watch that corresponds to the response.
- Created - set to true if the response is for a create watch request. The client should store the ID and expect to receive events for the watch on the stream. All events sent to the created watcher will have the same watch_id.
- Canceled - set to true if the response is for a cancel watch request. No further events will be sent to the canceled watcher.
- Compact_Revision - set to the minimum historical revision available to etcd if a watcher tries watching at a compacted revision. This happens when creating a watcher at a compacted revision or the watcher cannot catch up with the progress of the key-value store. The watcher will be canceled; creating new watches with the same start_revision will fail.
- Events - a list of new events in sequence corresponding to the given watch ID.

If the client wishes to stop receiving events for a watch, it issues a `WatchCancelRequest`:

```
message WatchCancelRequest {
   int64 watch_id = 1;
}
```

- Watch_ID - the ID of the watch to cancel so that no more events are transmitted.

Lease API

Leases are a mechanism for detecting client liveness. The cluster grants leases with a time-to-live. A lease expires if the etcd cluster does not receive a keepAlive within a given TTL period.

To tie leases into the key-value store, each key may be attached to at most one lease. When a lease expires or is revoked, all keys attached to that lease will be deleted. Each expired key generates a delete event in the event history.

Obtaining leases

Leases are obtained through the `LeaseGrant` API call, which takes a `LeaseGrantRequest`:

```
message LeaseGrantRequest {
  int64 TTL = 1;
  int64 ID = 2;
}
```

- TTL - the advisory time-to-live, in seconds.
- ID - the requested ID for the lease. If ID is set to 0, etcd will choose an ID.

The client receives a `LeaseGrantResponse` from the `LeaseGrant` call:

```
message LeaseGrantResponse {
  ResponseHeader header = 1;
  int64 ID = 2;
  int64 TTL = 3;
}
```

- ID - the lease ID for the granted lease.
- TTL - is the server selected time-to-live, in seconds, for the lease.

```
message LeaseRevokeRequest {
  int64 ID = 1;
}
```

- ID - the lease ID to revoke. When the lease is revoked, all attached keys are deleted.

Keep alives

Leases are refreshed using a bi-directional stream created with the `LeaseKeepAlive` API call. When the client wishes to refresh a lease, it sends a `LeaseKeepAliveRequest` over the stream:

```
message LeaseKeepAliveRequest {
  int64 ID = 1;
}
```

- ID - the lease ID for the lease to keep alive.

The keep alive stream responds with a `LeaseKeepAliveResponse`:

```
message LeaseKeepAliveResponse {
  ResponseHeader header = 1;
  int64 ID = 2;
  int64 TTL = 3;
}
```

- ID - the lease that was refreshed with a new TTL.
- TTL - the new time-to-live, in seconds, that the lease has remaining.

Last modified April 6, 2024: Fix markdown links in api docs for v3.5 v3.4 Fixes issue #830 (8711d61)

etcd API guarantees

API guarantees made by etcd

etcd is a consistent and durable key value store. The key value store is exposed through gRPC Services. etcd ensures the strongest consistency and durability guarantees for a distributed system. This specification enumerates the API guarantees made by etcd.

APIs to consider

- KV APIs
  - Range
  - Put
  - Delete
  - Transaction
- Watch APIs
  - Watch
- Lease APIs
  - Grant
  - [Revoke]
  - Keep alive

KV API allows for direct reading and manipulation of key value store. Watch API allows subscribing to key value store changes. Lease API allows assigning a time to live to a key.

Both KV and Watch APIs allow access to not only the latest versions of keys, but also previous versions are accessible within a continuous history window, limited by a compaction operation.

Calling KV API will take an immediate effect, while Watch API will return with some unbounded delay. In correctly working etcd cluster you should expect to see watch events to appear with 10ms delay after them happening. However, there is no limit and events in unhealthy clusters might never arrive.

KV APIs

etcd ensures durability and strict serializability for all KV api calls. Those are the strongest isolation guarantee of distributed transactional database systems.

Durability

Any completed operations are durable. All accessible data is also durable data. A read will never return data that has not been made durable.

Strict serializability

KV Service operations are atomic and occur in a total order, consistent with real-time order of those operations. Total order is implied through [revision](). Read more about [strict serializability]().

Strict serializability implies other weaker guarantees that might be easier to understand:

Atomicity

All API requests are atomic; an operation either completes entirely or not at all. For watch requests, all events generated by one operation will be in one watch response. Watch never observes partial events for a single operation.

Linearizability

From the perspective of client, linearizability provides useful properties which make reasoning easily. This is a clean description quoted from [the original paper](): `Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.`

For example, consider a client completing a write at time point 1 (*t1*). A client issuing a read at *t2* (for *t2* > *t1*) should receive a value at least as recent as the previous write, completed at *t1*. However, the read might actually complete only by *t3*. Linearizability guarantees the read returns the most current value. Without linearizability guarantee, the returned value, current at *t2* when the read began, might be "stale" by *t3* because a concurrent write might happen between *t2* and *t3*.

etcd ensures linearizability for all other operations by default. Linearizability comes with a cost, however, because linearized requests must go through the Raft consensus process. To obtain lower latencies and higher throughput for read requests, clients can configure a request's consistency mode to `serializable`, which may access stale data with respect to quorum, but removes the performance penalty of linearized accesses' reliance on live consensus.

Watch APIs

Watches make guarantees about events:

- Ordered - events are ordered by revision. An event will never appear on a watch if it precedes an event in time that has already been posted.
- Unique - an event will never appear on a watch twice.
- Reliable - a sequence of events will never drop any subsequence of events within the available history window. If there are events ordered in time as a < b < c, then if the watch receives events a and c, it is guaranteed to receive b as long b is in the available history window.
- Atomic - a list of events is guaranteed to encompass complete revisions. Updates in the same revision over multiple keys will not be split over several lists of events.
- Resumable - A broken watch can be resumed by establishing a new watch starting after the last revision received in a watch event before the break, so long as the revision is in the history window.
- Bookmarkable - Progress notification events guarantee that all events up to a revision have been already delivered.

etcd does not ensure linearizability for watch operations. Users are expected to verify the revision of watch events to ensure correct ordering with other operations.

Lease APIs

etcd provides [a lease mechanism](). The primary use case of a lease is implementing distributed coordination mechanisms like distributed locks. The lease mechanism itself is simple: a lease can be created with the grant API, attached to a key with the put API, revoked with the revoke API, and will be expired by the wall clock time to live (TTL). However, users need to be aware about the important properties of the APIs and usage for implementing correct distributed coordination mechanisms.

etcd specific definitions

Operation completed

An etcd operation is considered complete when it is committed through consensus, and therefore “executed” -- permanently stored -- by the etcd storage engine. The client knows an operation is completed when it receives a response from the etcd server. Note that the client may be uncertain about the status of an operation if it times out, or there is a network disruption between the client and the etcd member. etcd may also abort operations when there is a leader election. etcd does not send `abort` responses to clients' outstanding requests in this event.

Revision

An etcd operation that modifies the key value store is assigned a single increasing revision. A transaction operation might modify the key value store multiple times, but only one revision is assigned. The revision attribute of a key value pair that was modified by the operation has the same value as the revision of the operation. The revision can be used as a logical clock for key value store. A key value pair that has a larger revision is modified after a key value pair with a smaller revision. Two key value pairs that have the same revision are modified by an operation "concurrently".

Last modified March 27, 2024: Backpropagate improvements of api guarantees documentation to v3.5 and v3.4 release (d8ae837)

etcd versus other key-value stores

History and use of etcd & comparison with other tools

The name “etcd” originated from two ideas, the unix “/etc” folder and “d"istributed systems. The “/etc” folder is a place to store configuration data for a single system whereas etcd stores configuration information for large scale distributed systems. Hence, a “d"istributed “/etc” is “etcd”.

etcd is designed as a general substrate for large scale distributed systems. These are systems that will never tolerate split-brain operation and are willing to sacrifice availability to achieve this end. etcd stores metadata in a consistent and fault-tolerant way. An etcd cluster is meant to provide key-value storage with best of class stability, reliability, scalability and performance.

Distributed systems use etcd as a consistent key-value store for configuration management, service discovery, and coordinating distributed work. Many organizations use etcd to implement production systems such as container schedulers, service discovery services, and distributed data storage. Common distributed patterns using etcd include leader election, distributed locks, and monitoring machine liveness.

Use cases

- Container Linux by CoreOS: Applications running on Container Linux get automatic, zero-downtime Linux kernel updates. Container Linux uses locksmith to coordinate updates. Locksmith implements a distributed semaphore over etcd to ensure only a subset of a cluster is rebooting at any given time.
- Kubernetes stores configuration data into etcd for service discovery and cluster management; etcd’s consistency is crucial for correctly scheduling and operating services. The Kubernetes API server persists cluster state into etcd. It uses etcd’s watch API to monitor the cluster and roll out critical configuration changes.

Comparison chart

Perhaps etcd already seems like a good fit, but as with all technological decisions, proceed with caution. Please note this documentation is written by the etcd team. Although the ideal is a disinterested comparison of technology and features, the authors' expertise and biases obviously favor etcd. Use only as directed.

The table below is a handy quick reference for spotting the differences among etcd and its most popular alternatives at a glance. Further commentary and details for each column are in the sections following the table.

	etcd	ZooKeeper	Consul	NewSQL (Cloud Spanner, CockroachDB, TiDB)
Concurrency Primitives	Lock RPCs, Election RPCs, command line locks, command line elections, recipes in go	External curator recipes in Java	Native lock API	Rare, if any
Linearizable Reads	Yes	No	Yes	Sometimes
Multi-version Concurrency Control	Yes	No	No	Sometimes
Transactions	Field compares, Read, Write	Version checks, Write	Field compare, Lock, Read, Write	SQL-style
Change Notification	Historical and current key intervals	Current keys and directories	Current keys and prefixes	Triggers (sometimes)
User permissions	Role based	ACLs	ACLs	Varies (per-table GRANT, per-database roles)
HTTP/JSON API	Yes	No	Yes	Rarely
Membership Reconfiguration	Yes	>3.5.0	Yes	Yes
Maximum reliable database	Several gigabytes	Hundreds of megabytes	Hundreds of MBs	Terabytes+

	etcd	ZooKeeper	Consul	NewSQL (Cloud Spanner, CockroachDB, TiDB)
size		(sometimes several gigabytes)		
Minimum read linearization latency	Network RTT	No read linearization	RTT + fsync	Clock barriers (atomic, NTP)

ZooKeeper

ZooKeeper solves the same problem as etcd: distributed system coordination and metadata storage. However, etcd has the luxury of hindsight taken from engineering and operational experience with ZooKeeper's design and implementation. The lessons learned from Zookeeper certainly informed etcd's design, helping it support large scale systems like Kubernetes. The improvements etcd made over Zookeeper include:

- Dynamic cluster membership reconfiguration
- Stable read/write under high load
- A multi-version concurrency control data model
- Reliable key monitoring which never silently drop events
- Lease primitives decoupling connections from sessions
- APIs for safe distributed shared locks

Furthermore, etcd supports a wide range of languages and frameworks out of the box. Whereas Zookeeper has its own custom Jute RPC protocol, which is totally unique to Zookeeper and limits its supported language bindings, etcd's client protocol is built from gRPC, a popular RPC framework with language bindings for go, C++, Java, and more. Likewise, gRPC can be serialized into JSON over HTTP, so even general command line utilities like `curl` can talk to it. Since systems can select from a variety of choices, they are built on etcd with native tooling rather than around etcd with a single fixed set of technologies.

When considering features, support, and stability, new applications planning to use Zookeeper for a consistent key value store would do well to choose etcd instead.

Consul

Consul is an end-to-end service discovery framework. It provides built-in health checking, failure detection, and DNS services. In addition, Consul exposes a key value store with

RESTful HTTP APIs. As it stands in Consul 1.0, the storage system does not scale as well as other systems like etcd or Zookeeper in key-value operations; systems requiring millions of keys will suffer from high latencies and memory pressure. The key value API is missing, most notably, multi-version keys, conditional transactions, and reliable streaming watches.

etcd and Consul solve different problems. If looking for a distributed consistent key value store, etcd is a better choice over Consul. If looking for end-to-end cluster service discovery, etcd will not have enough features; choose Kubernetes, Consul, or SmartStack.

NewSQL (Cloud Spanner, CockroachDB, TiDB)

Both etcd and NewSQL databases (e.g., Cockroach, TiDB, Google Spanner) provide strong data consistency guarantees with high availability. However, the significantly different system design parameters lead to significantly different client APIs and performance characteristics.

NewSQL databases are meant to horizontally scale across data centers. These systems typically partition data across multiple consistent replication groups (shards), potentially distant, storing data sets on the order of terabytes and above. This sort of scaling makes them poor candidates for distributed coordination as they have long latencies from waiting on clocks and expect updates with mostly localized dependency graphs. The data is organized into tables, including SQL-style query facilities with richer semantics than etcd, but at the cost of additional complexity for processing, planning, and optimizing queries.

In short, choose etcd for storing metadata or coordinating distributed applications. If storing more than a few GB of data or if full SQL queries are needed, choose a NewSQL database.

Using etcd for metadata

etcd replicates all data within a single consistent replication group. For storing up to a few GB of data with consistent ordering, this is the most efficient approach. Each modification of cluster state, which may change multiple keys, is assigned a global unique ID, called a revision in etcd, from a monotonically increasing counter for reasoning over ordering. Since there's only a single replication group, the modification request only needs to go through the raft protocol to commit. By limiting consensus to one replication group, etcd gets distributed consistency with a simple protocol while achieving low latency and high throughput.

The replication behind etcd cannot horizontally scale because it lacks data sharding. In contrast, NewSQL databases usually shard data across multiple consistent replication groups, storing data sets on the order of terabytes and above. However, to assign each modification a global unique and increasing ID, each request must go through an additional coordination protocol among replication groups. This extra coordination step may potentially conflict on the global ID, forcing ordered requests to retry. The result is a more complicated approach with typically worse performance than etcd for strict ordering.

If an application reasons primarily about metadata or metadata ordering, such as to coordinate processes, choose etcd. If the application needs a large data store spanning multiple data centers and does not heavily depend on strong global ordering properties, choose a NewSQL database.

Using etcd for distributed coordination

etcd has distributed coordination primitives such as event watches, leases, elections, and distributed shared locks out of the box. These primitives are both maintained and supported by the etcd developers; leaving these primitives to external libraries shirks the responsibility of developing foundational distributed software, essentially leaving the system incomplete. NewSQL databases usually expect these distributed coordination primitives to be authored by third parties. Likewise, ZooKeeper famously has a separate and independent library of coordination recipes. Consul, which provides a native locking API, goes so far as to apologize that it's "not a bulletproof method".

In theory, it's possible to build these primitives atop any storage systems providing strong consistency. However, the algorithms tend to be subtle; it is easy to develop a locking algorithm that appears to work, only to suddenly break due to thundering herd and timing skew. Furthermore, other primitives supported by etcd, such as transactional memory depend on etcd's MVCC data model; simple strong consistency is not enough.

For distributed coordination, choosing etcd can help prevent operational headaches and save engineering effort.

Last modified March 27, 2024: Backpropagate improvements of api guarantees documentation to v3.5 and v3.4 release (d8ae837)

Glossary

Terms used in etcd documentation, command line, and source code

This document defines the various terms used in etcd documentation, command line and source code.

Alarm

The etcd server raises an alarm whenever the cluster needs operator intervention to remain reliable.

Authentication

Authentication manages user access permissions for etcd resources.

Client

A client connects to the etcd cluster to issue service requests such as fetching key-value pairs, writing data, or watching for updates.

Cluster

Cluster consists of several members.

The node in each member follows raft consensus protocol to replicate logs. Cluster receives proposals from members, commits them and apply to local store.

Compaction

Compaction discards all etcd event history and superseded keys prior to a given revision. It is used to reclaim storage space in the etcd backend database.

Election

The etcd cluster holds elections among its members to choose a leader as part of the raft consensus protocol.

Endpoint

A URL pointing to an etcd service or resource.

Key

A user-defined identifier for storing and retrieving user-defined values in etcd.

Key range

A set of keys containing either an individual key, a lexical interval for all x such that a < x <= b, or all keys greater than a given key.

Keyspace

The set of all keys in an etcd cluster.

Lease

A short-lived renewable contract that deletes keys associated with it on its expiry.

Member

A logical etcd server that participates in serving an etcd cluster.

Modification Revision

The first revision to hold the last write to a given key.

Peer

Peer is another member of the same cluster.

Proposal

A proposal is a request (for example a write request, a configuration change request) that needs to go through raft protocol.

Quorum

The number of active members needed for consensus to modify the cluster state. etcd requires a member majority to reach quorum.

Revision

A 64-bit cluster-wide counter that starts at 1 and is incremented each time the keyspace is modified.

Role

A unit of permissions over a set of key ranges which may be granted to a set of users for access control.

Snapshot

A point-in-time backup of the etcd cluster state.

Store

The physical storage backing the cluster keyspace.

Transaction

An atomically executed set of operations. All modified keys in a transaction share the same modification revision.

Key Version

The number of writes to a key since it was created, starting at 1. The version of a nonexistent or deleted key is 0.

Watcher

A client opens a watcher to observe updates on a given key range.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Developer guide

etcd guide for developers

Last modified April 26, 2021: [Docsy theme (#244) (86b070b)]

Discovery service protocol

Discover other etcd members in a cluster bootstrap phase

Discovery service protocol helps new etcd member to discover all other members in cluster bootstrap phase using a shared discovery URL.

Discovery service protocol is *only* used in cluster bootstrap phase, and cannot be used for runtime reconfiguration or cluster monitoring.

The protocol uses a new discovery token to bootstrap one *unique* etcd cluster. Remember that one discovery token can represent only one etcd cluster. As long as discovery protocol on this token starts, even if it fails halfway, it must not be used to bootstrap another etcd cluster.

The rest of this article will walk through the discovery process with examples that correspond to a self-hosted discovery cluster. The public discovery service, discovery.etcd.io, functions the same way, but with a layer of polish to abstract away ugly URLs, generate UUIDs automatically, and provide some protections against excessive requests. At its core, the public discovery service still uses an etcd cluster as the data store as described in this document.

Protocol workflow

The idea of discovery protocol is to use an internal etcd cluster to coordinate bootstrap of a new cluster. First, all new members interact with discovery service and help to generate the expected member list. Then each new member bootstraps its server using this list, which performs the same functionality as -initial-cluster flag.

In the following example workflow, we will list each step of protocol in curl format for ease of understanding.

By convention the etcd discovery protocol uses the key prefix `_etcd/registry`. If `http://example.com` hosts an etcd cluster for discovery service, a full URL to discovery keyspace will be `http://example.com/v2/keys/_etcd/registry`. We will use this as the URL prefix in the example.

Creating a new discovery token

Generate a unique token that will identify the new cluster. This will be used as a unique prefix in discovery keyspace in the following steps. An easy way to do this is to use `uuidgen`:

```
UUID=$(uuidgen)
```

Specifying the expected cluster size

The discovery token expects a cluster size that must be specified. The size is used by the discovery service to know when it has found all members that will initially form the cluster.

```
curl -X PUT http://example.com/v2/keys/_etcd/registry/${UUID}/_config/size -d value=${clu
```

Usually the cluster size is 3, 5 or 7. Check optimal cluster size for more details.

Bringing up etcd processes

Given the discovery URL, use it as `-discovery` flag and bring up etcd processes. Every etcd process will follow this next few steps internally if given a `-discovery` flag.

Registering itself

The first thing for etcd process is to register itself into the discovery URL as a member. This is done by creating member ID as a key in the discovery URL.

```
curl -X PUT http://example.com/v2/keys/_etcd/registry/${UUID}/${member_id}?prevExist=fal
```

Checking the status

It checks the expected cluster size and registration status in discovery URL, and decides what the next action is.

```
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}/_config/size
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}
```

If registered members are still not enough, it will wait for left members to appear.

If the number of registered members is bigger than the expected size N, it treats the first N registered members as the member list for the cluster. If the member itself is in the member list, the discovery procedure succeeds and it fetches all peers through the member list. If it is not in the member list, the discovery procedure finishes with the failure that the cluster has been full.

In etcd implementation, the member may check the cluster status even before registering itself. So it could fail quickly if the cluster has been full.

Waiting for all members

The wait process is described in detail in the etcd API documentation.

```
curl -X GET http://example.com/v2/keys/_etcd/registry/${UUID}?wait=true&waitIndex=${curre
```

It keeps waiting until finding all members.

Public discovery service

CoreOS Inc. hosts a public discovery service at https://discovery.etcd.io/ , which provides some nice features for ease of use.

Mask key prefix

Public discovery service will redirect `https://discovery.etcd.io/${UUID}` to etcd cluster behind for the key at `/v2/keys/_etcd/registry`. It masks register key prefix for short and readable discovery url.

Get new token

```
GET /new

Sent query:
	size=${cluster_size}
Possible status codes:
	200 OK
	400 Bad Request
200 Body:
	generated discovery url
```

The generation process in the service follows the steps from Creating a New Discovery Token to Specifying the Expected Cluster Size.

Check discovery status

```
GET /${UUID}
```

The status for this discovery token, including the machines that have been registered, can be checked by requesting the value of the UUID.

Open-source repository

The repository is located at https://github.com/coreos/discovery.etcd.io. It could be used to build a custom discovery service.

Last modified March 16, 2022: Remove contributor documentation (04f6278)

Set up a local cluster

Configuring local clusters for testing and development

For testing and development deployments, the quickest and easiest way is to configure a local cluster. For a production deployment, refer to the clustering section.

Local standalone cluster

Starting a cluster

Run the following to deploy an etcd cluster as a standalone cluster:

```
$ ./etcd
...
```

If the `etcd` binary is not present in the current working directory, it might be located either at `$GOPATH/bin/etcd` or at `/usr/local/bin/etcd`. Run the command appropriately.

The running etcd member listens on `localhost:2379` for client requests.

Interacting with the cluster

Use `etcdctl` to interact with the running cluster:

1. Store an example key-value pair in the cluster:

   ```
   $ ./etcdctl put foo bar
   OK
   ```

   If OK is printed, storing key-value pair is successful.

2. Retrieve the value of `foo`:

   ```
   $ ./etcdctl get foo
   bar
   ```

If `bar` is returned, interaction with the etcd cluster is working as expected.

Local multi-member cluster

Starting a cluster

A `Procfile` at the base of the etcd git repository is provided to easily configure a local multi-member cluster. To start a multi-member cluster, navigate to the root of the etcd source tree and perform the following:

1. Install `goreman` to control Procfile-based applications:

   ```
   $ go install github.com/mattn/goreman@latest
   ```

2. Start a cluster with `goreman` using etcd's stock Procfile:

   ```
   $ goreman -f Procfile start
   ```

   The members start running. They listen on `localhost:2379`, `localhost:22379`, and `localhost:32379` respectively for client requests.

Interacting with the cluster

Use `etcdctl` to interact with the running cluster:

1. Print the list of members:

   ```
   $ etcdctl --write-out=table --endpoints=localhost:2379 member list
   ```

   The list of etcd members are displayed as follows:

   ```
   +------------------+---------+--------+------------------------+--------------------
   |        ID        | STATUS  |  NAME  |       PEER ADDRS       |      CLIENT ADDRS
   +------------------+---------+--------+------------------------+--------------------
   | 8211f1d0f64f3269 | started | infra1 | http://127.0.0.1:2380  | http://127.0.0.1:23
   | 91bc3c398fb3c146 | started | infra2 | http://127.0.0.1:22380 | http://127.0.0.1:22
   | fd422379fda50e48 | started | infra3 | http://127.0.0.1:32380 | http://127.0.0.1:32
   +------------------+---------+--------+------------------------+--------------------
   ```

2. Store an example key-value pair in the cluster:

```
$ etcdctl put foo bar
OK
```

If OK is printed, storing key-value pair is successful.

Testing fault tolerance

To exercise etcd's fault tolerance, kill a member and attempt to retrieve the key.

1. Identify the process name of the member to be stopped.

   The `Procfile` lists the properties of the multi-member cluster. For example, consider the member with the process name, `etcd2`.

2. Stop the member:

   ```
   # kill etcd2
   $ goreman run stop etcd2
   ```

3. Store a key:

   ```
   $ etcdctl put key hello
   OK
   ```

4. Retrieve the key that is stored in the previous step:

   ```
   $ etcdctl get key
   hello
   ```

5. Retrieve a key from the stopped member:

   ```
   $ etcdctl --endpoints=localhost:22379 get key
   ```

   The command should display an error caused by connection failure:

   ```
   2017/06/18 23:07:35 grpc: Conn.resetTransport failed to create client transport: con
   Error:  grpc: timed out trying to connect
   ```

6. Restart the stopped member:

```
$ goreman run restart etcd2
```

7. Get the key from the restarted member:

```
$ etcdctl --endpoints=localhost:22379 get key
hello
```

Restarting the member re-establish the connection. `etcdctl` will now be able to retrieve the key successfully. To learn more about interacting with etcd, read interacting with etcd section.

Last modified April 18, 2023: Fix instructions for installing goreman. (964f7ab)

Interacting with etcd

etcdctl: a command line tool for interacting with the etcd server

Users mostly interact with etcd by putting or getting the value of a key. This section describes how to do that by using etcdctl, a command line tool for interacting with etcd server. The concepts described here should apply to the gRPC APIs or client library APIs.

The API version used by etcdctl to speak to etcd may be set to version `2` or `3` via the `ETCDCTL_API` environment variable. By default, etcdctl on master (3.4) uses the v3 API and earlier versions (3.3 and earlier) default to the v2 API.

Note that any key that was created using the v2 API will not be able to be queried via the v3 API. A v3 API `etcdctl get` of a v2 key will exit with 0 and no key data, this is the expected behaviour.

```
export ETCDCTL_API=3
```

Find versions

etcdctl version and Server API version can be useful in finding the appropriate commands to be used for performing various operations on etcd.

Here is the command to find the versions:

```
$ etcdctl version
etcdctl version: 3.1.0-alpha.0+git
API version: 3.1
```

Write a key

Applications store keys into the etcd cluster by writing to keys. Every stored key is replicated to all etcd cluster members through the Raft protocol to achieve consistency and reliability.

Here is the command to set the value of key `foo` to `bar` :

```
$ etcdctl put foo bar
OK
```

Also a key can be set for a specified interval of time by attaching lease to it.

Here is the command to set the value of key `foo1` to `bar1` for 10s.

```
$ etcdctl put foo1 bar1 --lease=1234abcd
OK
```

Note: The lease id `1234abcd` in the above command refers to id returned on creating the lease of 10s. This id can then be attached to the key.

Read keys

Applications can read values of keys from an etcd cluster. Queries may read a single key, or a range of keys.

Suppose the etcd cluster has stored the following keys:

```
foo = bar
foo1 = bar1
foo2 = bar2
foo3 = bar3
```

Here is the command to read the value of key `foo` :

```
$ etcdctl get foo
foo
bar
```

Here is the command to read the value of key `foo` in hex format:

```
$ etcdctl get foo --hex
\x66\x6f\x6f          # Key
\x62\x61\x72          # Value
```

Here is the command to read only the value of key `foo`:

```
$ etcdctl get foo --print-value-only
bar
```

Here is the command to range over the keys from `foo` to `foo3`:

```
$ etcdctl get foo foo3
foo
bar
foo1
bar1
foo2
bar2
```

Note that `foo3` is excluded since the range is over the half-open interval `[foo, foo3)`, excluding `foo3`.

Here is the command to range over all keys prefixed with `foo`:

```
$ etcdctl get --prefix foo
foo
bar
foo1
bar1
foo2
bar2
foo3
bar3
```

Here is the command to range over all keys prefixed with `foo`, limiting the number of results to 2:

```
$ etcdctl get --prefix --limit=2 foo
```

```
foo
bar
foo1
bar1
```

Read past version of keys

Applications may want to read superseded versions of a key. For example, an application may wish to roll back to an old configuration by accessing an earlier version of a key. Alternatively, an application may want a consistent view over multiple keys through multiple requests by accessing key history. Since every modification to the etcd cluster key-value store increments the global revision of an etcd cluster, an application can read superseded keys by providing an older etcd revision.

Suppose an etcd cluster already has the following keys:

```
foo = bar         # revision = 2
foo1 = bar1       # revision = 3
foo = bar_new     # revision = 4
foo1 = bar1_new   # revision = 5
```

Here are an example to access the past versions of keys:

```
$ etcdctl get --prefix foo # access the most recent versions of keys
foo
bar_new
foo1
bar1_new

$ etcdctl get --prefix --rev=4 foo # access the versions of keys at revision 4
foo
bar_new
foo1
bar1

$ etcdctl get --prefix --rev=3 foo # access the versions of keys at revision 3
foo
bar
foo1
bar1

$ etcdctl get --prefix --rev=2 foo # access the versions of keys at revision 2
```

```
foo
bar

$ etcdctl get --prefix --rev=1 foo # access the versions of keys at revision 1
```

Read keys which are greater than or equal to the byte value of the specified key

Applications may want to read keys which are greater than or equal to the byte value of the specified key.

Suppose an etcd cluster already has the following keys:

```
a = 123
b = 456
z = 789
```

Here is the command to read keys which are greater than or equal to the byte value of key `b`:

```
$ etcdctl get --from-key b
b
456
z
789
```

Delete keys

Applications can delete a key or a range of keys from an etcd cluster.

Suppose an etcd cluster already has the following keys:

```
foo = bar
foo1 = bar1
foo3 = bar3
zoo = val
zoo1 = val1
zoo2 = val2
```

```
a = 123
b = 456
z = 789
```

Here is the command to delete key `foo` :

```
$ etcdctl del foo
1 # one key is deleted
```

Here is the command to delete keys ranging from `foo` to `foo9` :

```
$ etcdctl del foo foo9
2 # two keys are deleted
```

Here is the command to delete key `zoo` with the deleted key value pair returned:

```
$ etcdctl del --prev-kv zoo
1   # one key is deleted
zoo # deleted key
val # the value of the deleted key
```

Here is the command to delete keys having prefix as `zoo` :

```
$ etcdctl del --prefix zoo
2 # two keys are deleted
```

Here is the command to delete keys which are greater than or equal to the byte value of key `b` :

```
$ etcdctl del --from-key b
2 # two keys are deleted
```

Watch key changes

Applications can watch on a key or a range of keys to monitor for any updates.

Here is the command to watch on key `foo`:

```
$ etcdctl watch foo
# in another terminal: etcdctl put foo bar
PUT
foo
bar
```

Here is the command to watch on key `foo` in hex format:

```
$ etcdctl watch foo --hex
# in another terminal: etcdctl put foo bar
PUT
\x66\x6f\x6f          # Key
\x62\x61\x72          # Value
```

Here is the command to watch on a range key from `foo` to `foo9`:

```
$ etcdctl watch foo foo9
# in another terminal: etcdctl put foo bar
PUT
foo
bar
# in another terminal: etcdctl put foo1 bar1
PUT
foo1
bar1
```

Here is the command to watch on keys having prefix `foo`:

```
$ etcdctl watch --prefix foo
# in another terminal: etcdctl put foo bar
PUT
foo
bar
# in another terminal: etcdctl put fooz1 barz1
PUT
fooz1
```

```
barz1
```

Here is the command to watch on multiple keys `foo` and `zoo` :

```
$ etcdctl watch -i
$ watch foo
$ watch zoo
# in another terminal: etcdctl put foo bar
PUT
foo
bar
# in another terminal: etcdctl put zoo val
PUT
zoo
val
```

Watch historical changes of keys

Applications may want to watch for historical changes of keys in etcd. For example, an application may wish to receive all the modifications of a key; if the application stays connected to etcd, then `watch` is good enough. However, if the application or etcd fails, a change may happen during the failure, and the application will not receive the update in real time. To guarantee the update is delivered, the application must be able to watch for historical changes to keys. To do this, an application can specify a historical revision on a watch, just like reading past version of keys.

Suppose we finished the following sequence of operations:

```
$ etcdctl put foo bar         # revision = 2
OK
$ etcdctl put foo1 bar1       # revision = 3
OK
$ etcdctl put foo bar_new     # revision = 4
OK
$ etcdctl put foo1 bar1_new   # revision = 5
OK
```

Here is an example to watch the historical changes:

```
```

```
# watch for changes on key `foo` since revision 2
$ etcdctl watch --rev=2 foo
PUT
foo
bar
PUT
foo
bar_new
```

```
# watch for changes on key `foo` since revision 3
$ etcdctl watch --rev=3 foo
PUT
foo
bar_new
```

Here is an example to watch only from the last historical change:

```
# watch for changes on key `foo` and return last revision value along with modified value
$ etcdctl watch --prev-kv foo
# in another terminal: etcdctl put foo bar_latest
PUT
foo         # key
bar_new     # last value of foo key before modification
foo         # key
bar_latest  # value of foo key after modification
```

Watch progress

Applications may want to check the progress of a watch to determine how up-to-date the watch stream is. For example, if a watch is used to update a cache, it can be useful to know if the cache is stale compared to the revision from a quorum read.

Progress requests can be issued using the "progress" command in interactive watch session to ask the etcd server to send a progress notify update in the watch stream:

```
$ etcdctl watch -i
$ watch a
$ progress
progress notify: 1
# in another terminal: etcdctl put x 0
```

```
# in another terminal: etcdctl put y 1
$ progress
progress notify: 3
```

Note: The revision number in the progress notify response is the revision from the local etcd server node that the watch stream is connected to. If this node is partitioned and not part of quorum, this progress notify revision might be lower than than the revision returned by a quorum read against a non-partitioned etcd server node.

Compacted revisions

As we mentioned, etcd keeps revisions so that applications can read past versions of keys. However, to avoid accumulating an unbounded amount of history, it is important to compact past revisions. After compacting, etcd removes historical revisions, releasing resources for future use. All superseded data with revisions before the compacted revision will be unavailable.

Here is the command to compact the revisions:

```
$ etcdctl compact 5
compacted revision 5

# any revisions before the compacted one are not accessible
$ etcdctl get --rev=4 foo
Error:  rpc error: code = 11 desc = etcdserver: mvcc: required revision has been compacte
```

Note: The current revision of etcd server can be found using get command on any key (existent or non-existent) in json format. Example is shown below for mykey which does not exist in etcd server:

```
$ etcdctl get mykey -w=json
{"header":{"cluster_id":14841639068965178418,"member_id":10276657743932975437,"revision"
```

Grant leases

Applications can grant leases for keys from an etcd cluster. When a key is attached to a lease, its lifetime is bound to the lease's lifetime which in turn is governed by a time-to-live (TTL). Each lease has a minimum time-to-live (TTL) value specified by the application at grant time.

The lease's actual TTL value is at least the minimum TTL and is chosen by the etcd cluster. Once a lease's TTL elapses, the lease expires and all attached keys are deleted.

Here is the command to grant a lease:

```
# grant a lease with 60 second TTL
$ etcdctl lease grant 60
lease 32695410dcc0ca06 granted with TTL(60s)

# attach key foo to lease 32695410dcc0ca06
$ etcdctl put --lease=32695410dcc0ca06 foo bar
OK
```

Revoke leases

Applications revoke leases by lease ID. Revoking a lease deletes all of its attached keys.

Suppose we finished the following sequence of operations:

```
$ etcdctl lease grant 60
lease 32695410dcc0ca06 granted with TTL(60s)
$ etcdctl put --lease=32695410dcc0ca06 foo bar
OK
```

Here is the command to revoke the same lease:

```
$ etcdctl lease revoke 32695410dcc0ca06
lease 32695410dcc0ca06 revoked

$ etcdctl get foo
# empty response since foo is deleted due to lease revocation
```

Keep leases alive

Applications can keep a lease alive by refreshing its TTL so it does not expire.

Suppose we finished the following sequence of operations:

```
$ etcdctl lease grant 60
lease 32695410dcc0ca06 granted with TTL(60s)
```

Here is the command to keep the same lease alive:

```
$ etcdctl lease keep-alive 32695410dcc0ca06
lease 32695410dcc0ca06 keepalived with TTL(60)
lease 32695410dcc0ca06 keepalived with TTL(60)
lease 32695410dcc0ca06 keepalived with TTL(60)
...
```

Get lease information

Applications may want to know about lease information, so that they can be renewed or to check if the lease still exists or it has expired. Applications may also want to know the keys to which a particular lease is attached.

Suppose we finished the following sequence of operations:

```
# grant a lease with 500 second TTL
$ etcdctl lease grant 500
lease 694d5765fc71500b granted with TTL(500s)

# attach key zoo1 to lease 694d5765fc71500b
$ etcdctl put zoo1 val1 --lease=694d5765fc71500b
OK

# attach key zoo2 to lease 694d5765fc71500b
$ etcdctl put zoo2 val2 --lease=694d5765fc71500b
OK
```

Here is the command to get information about the lease:

```
$ etcdctl lease timetolive 694d5765fc71500b
lease 694d5765fc71500b granted with TTL(500s), remaining(258s)
```

Here is the command to get information about the lease along with the keys attached with the lease:

```
$ etcdctl lease timetolive --keys 694d5765fc71500b
lease 694d5765fc71500b granted with TTL(500s), remaining(132s), attached keys([zoo2 zoo1

# if the lease has expired or does not exist it will give the below response:
Error:  etcdserver: requested lease not found
```

Last modified April 26, 2021: [Docsy theme (#244) (86b070b)]

Why gRPC gateway

Why you should consider using the gRPC gateway

etcd v3 uses gRPC for its messaging protocol. The etcd project includes a gRPC-based Go client and a command line utility, etcdctl, for communicating with an etcd cluster through gRPC. For languages with no gRPC support, etcd provides a JSON gRPC gateway. This gateway serves a RESTful proxy that translates HTTP/JSON requests into gRPC messages.

Using gRPC gateway

The gateway accepts a JSON mapping for etcd's protocol buffer message definitions. Note that `key` and `value` fields are defined as byte arrays and therefore must be base64 encoded in JSON. The following examples use `curl`, but any HTTP/JSON client should work all the same.

Notes

gRPC gateway endpoint has changed since etcd v3.3:

- etcd v3.2 or before uses only `[CLIENT-URL]/v3alpha/*`.
- etcd v3.3 uses `[CLIENT-URL]/v3beta/*` while keeping `[CLIENT-URL]/v3alpha/*`.
- etcd v3.4 uses `[CLIENT-URL]/v3/*` while keeping `[CLIENT-URL]/v3beta/*`.
  - **`[CLIENT-URL]/v3alpha/*` is deprecated**.
- etcd v3.5 or later uses only `[CLIENT-URL]/v3/*`.
  - **`[CLIENT-URL]/v3beta/*` is deprecated**.

gRPC-gateway does not support authentication using TLS Common Name.

Put and get keys

Use the `/v3/kv/range` and `/v3/kv/put` services to read and write keys:

```
<<COMMENT
https://www.base64encode.org/
foo is 'Zm9v' in Base64
```

```
bar is 'YmFy'
COMMENT

curl -L http://localhost:2379/v3/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
# {"header":{"cluster_id":"12585971608760269493","member_id":"13847567121247652255","rev

curl -L http://localhost:2379/v3/kv/range \
  -X POST -d '{"key": "Zm9v"}'
# {"header":{"cluster_id":"12585971608760269493","member_id":"13847567121247652255","rev

# get all keys prefixed with "foo"
curl -L http://localhost:2379/v3/kv/range \
  -X POST -d '{"key": "Zm9v", "range_end": "Zm9w"}'
# {"header":{"cluster_id":"12585971608760269493","member_id":"13847567121247652255","rev
```

Watch keys

Use the `/v3/watch` service to watch keys:

```
curl -N http://localhost:2379/v3/watch \
  -X POST -d '{"create_request": {"key":"Zm9v"} }' &
# {"result":{"header":{"cluster_id":"12585971608760269493","member_id":"1384756712124765

curl -L http://localhost:2379/v3/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}' >/dev/null 2>&1
# {"result":{"header":{"cluster_id":"12585971608760269493","member_id":"1384756712124765
```

Transactions

Issue a transaction with `/v3/kv/txn`:

```
# target CREATE
curl -L http://localhost:2379/v3/kv/txn \
  -X POST \
  -d '{"compare":[{"target":"CREATE","key":"Zm9v","createRevision":"2"}],"success":[{"re
# {"header":{"cluster_id":"12585971608760269493","member_id":"13847567121247652255","rev
```

```
# target VERSION
curl -L http://localhost:2379/v3/kv/txn \
  -X POST \
  -d '{"compare":[{"version":"4","result":"EQUAL","target":"VERSION","key":"Zm9v"}],"suc
```

```
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev
```

Authentication

Set up authentication with the `/v3/auth` service:

```
# create root user
curl -L http://localhost:2379/v3/auth/user/add \
  -X POST -d '{"name": "root", "password": "pass"}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev

# create root role
curl -L http://localhost:2379/v3/auth/role/add \
  -X POST -d '{"name": "root"}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev

# grant root role
curl -L http://localhost:2379/v3/auth/user/grant \
  -X POST -d '{"user": "root", "role": "root"}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev

# enable auth
curl -L http://localhost:2379/v3/auth/enable -X POST -d '{}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev
```

Authenticate with etcd for an authentication token using `/v3/auth/authenticate` :

```
# get the auth token for the root user
curl -L http://localhost:2379/v3/auth/authenticate \
  -X POST -d '{"name": "root", "password": "pass"}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev
```

Set the `Authorization` header to the authentication token to fetch a key using authentication credentials:

```
curl -L http://localhost:2379/v3/kv/put \
  -H 'Authorization: sssvIpwfnLAcWAQH.9' \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
# {"header":{"cluster_id":"14841639068965178418","member_id":"10276657743932975437","rev
```

Swagger

Generated Swagger API definitions can be found at rpc.swagger.json.

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

gRPC naming and discovery

go-grpc: for resolving gRPC endpoints with an etcd backend

etcd provides a gRPC resolver to support an alternative name system that fetches endpoints from etcd for discovering gRPC services. The underlying mechanism is based on watching updates to keys prefixed with the service name.

Using etcd discovery with go-grpc

The etcd client provides a gRPC resolver for resolving gRPC endpoints with an etcd backend. The resolver is initialized with an etcd client and given a target for resolution:

```
import (
    "go.etcd.io/etcd/clientv3"
    etcdnaming "go.etcd.io/etcd/clientv3/naming/resolver"

    "google.golang.org/grpc"
)

...

cli, err := clientv3.NewFromURL("http://localhost:2379")
if err != nil {
    // ...
}
r, err := etcdnaming.NewBuilder(cli)
if err != nil {
    // ...
}
conn, gerr := grpc.Dial("my-service", grpc.WithResolvers(r), grpc.WithBlock(), ...
```

Managing service endpoints

The etcd resolver treats all keys under the prefix of the resolution target following a “/” (e.g., “my-service/”) with JSON-encoded go-grpc `naming.Update` values as potential service endpoints. Endpoints are added to the service by creating new keys and removed from the service by deleting keys.

Adding an endpoint

New endpoints can be added to the service through `etcdctl`:

```
ETCDCTL_API=3 etcdctl put my-service/1.2.3.4 '{"Addr":"1.2.3.4","Metadata":"..."}'
```

The etcd client's `GRPCResolver.Update` method can also register new endpoints with a key matching the `Addr`:

```
r.Update(context.TODO(), "my-service", naming.Update{Op: naming.Add, Addr: "1.2.3.4", Me
```

Deleting an endpoint

Hosts can be deleted from the service through `etcdctl`:

```
ETCDCTL_API=3 etcdctl del my-service/1.2.3.4
```

The etcd client's `GRPCResolver.Update` method also supports deleting endpoints:

```
r.Update(context.TODO(), "my-service", naming.Update{Op: naming.Delete, Addr: "1.2.3.4"}
```

Registering an endpoint with a lease

Registering an endpoint with a lease ensures that if the host can't maintain a keepalive heartbeat (e.g., its machine fails), it will be removed from the service:

```
lease=`ETCDCTL_API=3 etcdctl lease grant 5 | cut -f2 -d' '`
ETCDCTL_API=3 etcdctl put --lease=$lease my-service/1.2.3.4 '{"Addr":"1.2.3.4","Metadata
ETCDCTL_API=3 etcdctl lease keep-alive $lease
```

Last modified August 1, 2024: fix: update v3.4 grpc_naming.md guide (8b4e231)

System limits

etcd limits: requests and storage

Request size limit

etcd is designed to handle small key value pairs typical for metadata. Larger requests will work, but may increase the latency of other requests. By default, the maximum size of any request is 1.5 MiB. This limit is configurable through `--max-request-bytes` flag for etcd server.

Storage size limit

The default storage size limit is 2 GiB, configurable with `--quota-backend-bytes` flag. 8 GiB is a suggested maximum size for normal environments and etcd warns at startup if the configured value exceeds it.

Last modified April 20, 2023: Corrected the data size units. (9e92641)

etcd features

using etcd features

This document provides an overview of etcd features to help users better understand the features and related deprecation process. If you are interested in knowing about how features are developed in the etcd, please see these development guidelines.

The etcd features fall into three stages, experimental, stable, and unsafe. You can get the list of features by running `etcd --help`.

Experimental

In order to get early feedback, any new feature is usually added as an experimental feature. The experimental feature can be identified by looking at the flag name, which should have `--experimental` as a prefix. Please consider the following points while using an experimental feature:

- It might be buggy due to a lack of user testing. Enabling the feature may not work as expected.
- It is disabled by default.
- Support for such a feature may be dropped at any time without notice
  - It can be removed in the next minor or major release without following the feature deprecation policy unless it graduates to a stable future.
  - The project team would appreciate users reporting any issues related to experimental features. However, such issues may be given lower priorities compared to the issues related to stable featuers.
- An experimental feature flag deprecates when it graduates to the stable stage. Users should start using a stable feature flag as soon as possible.

Stable

This is the most common stage of features in the etcd. A stable feature is characterized as below:

- Supported as part of the supported releases of etcd.

- May be enabled by default.
- Discontinuation of support must follow the feature deprecation policy.

Unsafe

Unsafe features are rare and listed under the `Unsafe feature:` section in the etcd usage documentation. By default, they are disabled. They should be used with caution following documentation. An unsafe feature can be removed in the next minor or major release without following the feature deprecation policy.

Feature Deprecation

Experimental

An experimental feature deprecates when it graduates to the stable stage.

- The experimental feature documentation will show a deprecation message with a recommendation to use a related stable feature flag. e.g. `DEPRECATED. Use <feature-name> instead.`
- A deprecated feature will be removed in the following release.

Stable

As the project evolves, a stable feature may sometimes need to be deprecated and removed. When that happens,

- The feature documentation will show a warning message before a planned release for deprecation. e.g. `To be deprecated in <release>.`. If a new feature is already planned to replace the `To be deprecated` feature, then the documentation will also provide a message saying so. e.g. `Use <feature-name> instead.`.
- The feature will be deprecated in the planned release. At that time, the feature documentation will show a deprecation message with a recommendation to use a related stable feature. e.g. `DEPRECATED. Use <feature-name> instead.`
- A deprecated feature will be removed in the following release.

Last modified September 30, 2022: Add user doc for etcd features (5413d18)

API reference

This API reference is autogenerated from the named `.proto` files.

service `Auth` (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type
AuthEnable	AuthEnableRequest	AuthEnableResponse
AuthDisable	AuthDisableRequest	AuthDisableResponse
Authenticate	AuthenticateRequest	AuthenticateResponse
UserAdd	AuthUserAddRequest	AuthUserAddResponse
UserGet	AuthUserGetRequest	AuthUserGetResponse
UserList	AuthUserListRequest	AuthUserListResponse
UserDelete	AuthUserDeleteRequest	AuthUserDeleteResponse
UserChangePassword	AuthUserChangePasswordRequest	AuthUserChangePasswordRes
UserGrantRole	AuthUserGrantRoleRequest	AuthUserGrantRoleResponse

Method	Request Type	Response Type
UserRevokeRole	AuthUserRevokeRoleRequest	AuthUserRevokeRoleRespons
RoleAdd	AuthRoleAddRequest	AuthRoleAddResponse
RoleGet	AuthRoleGetRequest	AuthRoleGetResponse
RoleList	AuthRoleListRequest	AuthRoleListResponse
RoleDelete	AuthRoleDeleteRequest	AuthRoleDeleteResponse
RoleGrantPermission	AuthRoleGrantPermissionRequest	AuthRoleGrantPermissionRes
RoleRevokePermission	AuthRoleRevokePermissionRequest	AuthRoleRevokePermissionRe

service `Cluster` (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type	Description
MemberAdd	MemberAddRequest	MemberAddResponse	MemberAdd adds a member into the cluster.
MemberRemove	MemberRemoveRequest	MemberRemoveResponse	MemberRemove removes an existing member from the cluster.
MemberUpdate	MemberUpdateRequest	MemberUpdateResponse	MemberUpdate updates the

Method	Request Type	Response Type	Description
			member configuration.
MemberList	MemberListRequest	MemberListResponse	MemberList lists all the members in the cluster.
MemberPromote	MemberPromoteRequest	MemberPromoteResponse	MemberPromote promotes a member from raft learner (non voting) to raft voting member.

service KV (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type	Description
Range	RangeRequest	RangeResponse	Range gets the keys in the range from the key-value store.
Put	PutRequest	PutResponse	Put puts the given key into the key-value store. A put request increments the revision of the key-value store and generates one event in the event history.
DeleteRange	DeleteRangeRequest	DeleteRangeResponse	DeleteRange deletes the given range from the key-value store. A delete request increments the revision of the key-value store and generates a delete event in the event history for every deleted key.
Txn	TxnRequest	TxnResponse	Txn processes multiple requests in a single transaction. A txn request increments the revision of the key-value store and

Method	Request Type	Response Type	Description
			generates events with the same revision for every completed request. It is not allowed to modify the same key several times within one txn.
Compact	CompactionRequest	CompactionResponse	Compact compacts the event history in the etcd key-value store. The key-value store should be periodically compacted or the event history will continue to grow indefinitely.

service `Lease` (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type	Description
LeaseGrant	LeaseGrantRequest	LeaseGrantResponse	LeaseGrant creates a lease which expires if the server does not receive a keepAlive within a given time to live period. All keys attached to the lease will be expired and deleted if the lease expires. Each expired key generates a delete event in the event history.
LeaseRevoke	LeaseRevokeRequest	LeaseRevokeResponse	LeaseRevoke revokes a lease. All keys attached to the lease will

Method	Request Type	Response Type	Description
			expire and be deleted.
LeaseKeepAlive	LeaseKeepAliveRequest	LeaseKeepAliveResponse	LeaseKeepAlive keeps the lease alive by streaming keep alive requests from the client to the server and streaming keep alive responses from the server to the client.
LeaseTimeToLive	LeaseTimeToLiveRequest	LeaseTimeToLiveResponse	LeaseTimeToLive retrieves lease information.
LeaseLeases	LeaseLeasesRequest	LeaseLeasesResponse	LeaseLeases lists all existing leases.

service `Maintenance` (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type	Description
Alarm	AlarmRequest	AlarmResponse	Alarm activates, deactivates, and queries alarms regarding cluster health.
Status	StatusRequest	StatusResponse	Status gets the status of the member.
Defragment	DefragmentRequest	DefragmentResponse	Defragment defragments a member's backend database to recover storage space.
Hash	HashRequest	HashResponse	Hash computes the hash of whole backend keyspace, including key, lease, and other buckets in storage. This

Method	Request Type	Response Type	Description
			is designed for testing ONLY! Do not rely on this in production with ongoing transactions, since Hash operation does not hold MVCC locks. Use “HashKV” API instead for “key” bucket consistency checks.
HashKV	HashKVRequest	HashKVResponse	HashKV computes the hash of all MVCC keys up to a given revision. It only iterates “key” bucket in backend storage.
Snapshot	SnapshotRequest	SnapshotResponse	Snapshot sends a snapshot of the entire backend from a member over a stream to a client.
MoveLeader	MoveLeaderRequest	MoveLeaderResponse	MoveLeader requests current leader node to transfer its leadership to transferee.

service `Watch` (etcdserver/etcdserverpb/rpc.proto)

Method	Request Type	Response Type	Description
Watch	WatchRequest	WatchResponse	Watch watches for events happening or that have happened. Both input and output are streams; the input stream is for creating and canceling watchers and the output stream sends events. One watch RPC can watch on multiple key ranges, streaming events for several watches at once. The entire event history can be watched starting from the last compaction revision.

message `AlarmMember` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
memberID	memberID is the ID of the member associated with the raised alarm.	uint64
alarm	alarm is the type of alarm which has been raised.	AlarmType

message `AlarmRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
action	action is the kind of alarm request to issue. The action may GET alarm statuses, ACTIVATE an alarm, or DEACTIVATE a raised alarm.	AlarmAction
memberID	memberID is the ID of the member associated with the alarm. If memberID is 0, the alarm request covers all members.	uint64
alarm	alarm is the type of alarm to consider for this request.	AlarmType

message `AlarmResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
alarms	alarms is a list of alarms associated with the alarm request.	(slice of) AlarmMember

message `AuthDisableRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `AuthDisableResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthEnableRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `AuthEnableResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthRoleAddRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name	name is the name of the role to add to the authentication system.	string

message `AuthRoleAddResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthRoleDeleteRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
role		string

message `AuthRoleDeleteResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthRoleGetRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
role		string

message `AuthRoleGetResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

Field	Description	Type
perm		(slice of) authpb.Permission

message `AuthRoleGrantPermissionRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name	name is the name of the role which will be granted the permission.	string
perm	perm is the permission to grant to the role.	authpb.Permission

message `AuthRoleGrantPermissionResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthRoleListRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `AuthRoleListResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
roles		(slice of) string

message `AuthRoleRevokePermissionRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
role		string
key		bytes
range_end		bytes

message `AuthRoleRevokePermissionResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthUserAddRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name		string
password		string
options		authpb.UserAddOptions

message `AuthUserAddResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthUserChangePasswordRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name	name is the name of the user whose password is being changed.	string
password	password is the new password for the user.	string

message `AuthUserChangePasswordResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthUserDeleteRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name	name is the name of the user to delete.	string

message `AuthUserDeleteResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthUserGetRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name		string

message `AuthUserGetResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
roles		(slice of) string

message `AuthUserGrantRoleRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
user	user is the name of the user which should be granted a given role.	string
role	role is the name of the role to grant to the user.	string

message `AuthUserGrantRoleResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthUserListRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `AuthUserListResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

Field	Description	Type
users		(slice of) string

message `AuthUserRevokeRoleRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name		string
role		string

message `AuthUserRevokeRoleResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `AuthenticateRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
name		string
password		string

message `AuthenticateResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
token	token is an authorized token that can be used in succeeding RPCs	string

message `CompactionRequest` (etcdserver/etcdserverpb/rpc.proto)

CompactionRequest compacts the key-value store up to a given revision. All superseded keys with a revision less than the compaction revision will be removed.

Field	Description	Type
revision	revision is the key-value store revision for the compaction operation.	int64
physical	physical is set so the RPC will wait until the compaction is physically applied to the local database such that compacted entries are totally removed from the backend database.	bool

message `CompactionResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `Compare` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
result	result is logical comparison operation for this comparison.	CompareResult
target	target is the key-value field to inspect for the comparison.	CompareTarget
key	key is the subject key for the comparison operation.	bytes
target_union		oneof
version	version is the version of the given key	int64
create_revision	create_revision is the creation revision of the given key	int64
mod_revision	mod_revision is the last modified revision of the given key.	int64
value	value is the value of the given key, in bytes.	bytes
lease	lease is the lease id of the given key.	int64
range_end	range_end compares the given target to all keys in the range [key, range_end). See RangeRequest for more details on key ranges.	bytes

message `DefragmentRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `DefragmentResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `DeleteRangeRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
key	key is the first key to delete in the range.	bytes
range_end	range_end is the key following the last key to delete for the range [key, range_end). If range_end is not given, the range is defined to contain only the key argument. If range_end is one bit larger than the given key, then the range is all the keys with the prefix (the given key). If range_end is '\0', the range is all keys greater than or equal to the key argument.	bytes
prev_kv	If prev_kv is set, etcd gets the previous key-value pairs before deleting it. The previous key-value pairs will be returned in the delete response.	bool

message `DeleteRangeResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
deleted	deleted is the number of keys deleted by the delete range request.	int64
prev_kvs	if prev_kv is set in the request, the previous key-value pairs will be returned.	(slice of) mvccpb.KeyValue

message `HashKVRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
revision	revision is the key-value store revision for the hash operation.	int64

message `HashKVResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
hash	hash is the hash value computed from the responding member's MVCC keys up to a given revision.	uint32
compact_revision	compact_revision is the compacted revision of key-value store when hash begins.	int64

message `HashRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `HashResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
hash	hash is the hash value computed from the responding member's KV's backend.	uint32

message `LeaseCheckpoint` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the lease ID to checkpoint.	int64
remaining_TTL	Remaining_TTL is the remaining time until expiry of the lease.	int64

message `LeaseCheckpointRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
checkpoints		(slice of) LeaseCheckpoint

message `LeaseCheckpointResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `LeaseGrantRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
TTL	TTL is the advisory time-to-live in seconds. Expired lease will return -1.	int64
ID	ID is the requested ID for the lease. If ID is set to 0, the lessor chooses an ID.	int64

message `LeaseGrantResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
ID	ID is the lease ID for the granted lease.	int64
TTL	TTL is the server chosen lease time-to-live in seconds.	int64
error		string

message `LeaseKeepAliveRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the lease ID for the lease to keep alive.	int64

message `LeaseKeepAliveResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
ID	ID is the lease ID from the keep alive request.	int64
TTL	TTL is the new time-to-live for the lease.	int64

message `LeaseLeasesRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `LeaseLeasesResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
leases		(slice of) LeaseStatus

message `LeaseRevokeRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the lease ID to revoke. When the ID is revoked, all associated keys will be deleted.	int64

message `LeaseRevokeResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `LeaseStatus` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID		int64

message `LeaseTimeToLiveRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the lease ID for the lease.	int64
keys	keys is true to query all the keys attached to this lease.	bool

message `LeaseTimeToLiveResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
ID	ID is the lease ID from the keep alive request.	int64

Field	Description	Type
TTL	TTL is the remaining TTL in seconds for the lease; the lease will expire in under TTL+1 seconds.	int64
grantedTTL	GrantedTTL is the initial granted time in seconds upon lease creation/renewal.	int64
keys	Keys is the list of keys attached to this lease.	(slice of) bytes

message `Member` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the member ID for this member.	uint64
name	name is the human-readable name of the member. If the member is not started, the name will be an empty string.	string
peerURLs	peerURLs is the list of URLs the member exposes to the cluster for communication.	(slice of) string
clientURLs	clientURLs is the list of URLs the member exposes to clients for communication. If the member is not started, clientURLs will be empty.	(slice of) string
isLearner	isLearner indicates if the member is raft learner.	bool

message `MemberAddRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
peerURLs	peerURLs is the list of URLs the added member will use to communicate with the cluster.	(slice of) string
isLearner	isLearner indicates if the added member is raft learner.	bool

message `MemberAddResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
member	member is the member information for the added member.	Member

Field	Description	Type
members	members is a list of all members after adding the new member.	(slice of) Member

message `MemberListRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `MemberListResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
members	members is a list of all members associated with the cluster.	(slice of) Member

message `MemberPromoteRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the member ID of the member to promote.	uint64

message `MemberPromoteResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
members	members is a list of all members after promoting the member.	(slice of) Member

message `MemberRemoveRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the member ID of the member to remove.	uint64

message `MemberRemoveResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
members	members is a list of all members after removing the member.	(slice of) Member

message `MemberUpdateRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
ID	ID is the member ID of the member to update.	uint64
peerURLs	peerURLs is the new list of URLs the member will use to communicate with the cluster.	(slice of) string

message `MemberUpdateResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
members	members is a list of all members after updating the member.	(slice of) Member

message `MoveLeaderRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
targetID	targetID is the node ID for the new leader.	uint64

message `MoveLeaderResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader

message `PutRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
key	key is the key, in bytes, to put into the key-value store.	bytes

Field	Description	Type
value	value is the value, in bytes, to associate with the key in the key-value store.	bytes
lease	lease is the lease ID to associate with the key in the key-value store. A lease value of 0 indicates no lease.	int64
prev_kv	If prev_kv is set, etcd gets the previous key-value pair before changing it. The previous key-value pair will be returned in the put response.	bool
ignore_value	If ignore_value is set, etcd updates the key using its current value. Returns an error if the key does not exist.	bool
ignore_lease	If ignore_lease is set, etcd updates the key using its current lease. Returns an error if the key does not exist.	bool

message `PutResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
prev_kv	if prev_kv is set in the request, the previous key-value pair will be returned.	mvccpb.KeyValue

message `RangeRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
key	key is the first key for the range. If range_end is not given, the request only looks up key.	bytes
range_end	range_end is the upper bound on the requested range [key, range_end). If range_end is '\0', the range is all keys >= key. If range_end is key plus one (e.g., "aa"+1 == "ab", "a\xff"+1 == "b"), then the range request gets all keys prefixed with key. If both key and range_end are '\0', then the range request returns all keys.	bytes
limit	limit is a limit on the number of keys returned for the request. When limit is set to 0, it is treated as no limit.	int64

Field	Description	Type
revision	revision is the point-in-time of the key-value store to use for the range. If revision is less or equal to zero, the range is over the newest key-value store. If the revision has been compacted, ErrCompacted is returned as a response.	int64
sort_order	sort_order is the order for returned sorted results.	SortOrder
sort_target	sort_target is the key-value field to use for sorting.	SortTarget
serializable	serializable sets the range request to use serializable member-local reads. Range requests are linearizable by default; linearizable requests have higher latency and lower throughput than serializable requests but reflect the current consensus of the cluster. For better performance, in exchange for possible stale reads, a serializable range request is served locally without needing to reach consensus with other nodes in the cluster.	bool
keys_only	keys_only when set returns only the keys and not the values.	bool
count_only	count_only when set returns only the count of the keys in the range.	bool
min_mod_revision	min_mod_revision is the lower bound for returned key mod revisions; all keys with lesser mod revisions will be filtered away.	int64
max_mod_revision	max_mod_revision is the upper bound for returned key mod revisions; all keys with greater mod revisions will be filtered away.	int64
min_create_revision	min_create_revision is the lower bound for returned key create revisions; all keys with lesser create revisions will be filtered away.	int64
max_create_revision	max_create_revision is the upper bound for returned key create revisions; all keys with greater create revisions will be filtered away.	int64

message `RangeResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
kvs	kvs is the list of key-value pairs matched by the range request. kvs is empty when count is requested.	(slice of) mvccpb.KeyValue
more	more indicates if there are more keys to return in the requested range.	bool
count	count is set to the number of keys within the range when requested.	int64

message `RequestOp` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
request	request is a union of request types accepted by a transaction.	oneof
request_range		RangeRequest
request_put		PutRequest
request_delete_range		DeleteRangeRequest
request_txn		TxnRequest

message `ResponseHeader` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
cluster_id	cluster_id is the ID of the cluster which sent the response.	uint64
member_id	member_id is the ID of the member which sent the response.	uint64
revision	revision is the key-value store revision when the request was applied. For watch progress responses, the header.revision indicates progress. All future events received in this stream are guaranteed to have a higher revision number than the header.revision number.	int64
raft_term	raft_term is the raft term when the request was applied.	uint64

message `ResponseOp` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
response	response is a union of response types returned by a transaction.	oneof
response_range		RangeResponse
response_put		PutResponse
response_delete_range		DeleteRangeResponse
response_txn		TxnResponse

message `SnapshotRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `SnapshotResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header	header has the current key-value store information. The first header in the snapshot stream indicates the point in time of the snapshot.	ResponseHeader
remaining_bytes	remaining_bytes is the number of blob bytes to be sent after this message	uint64
blob	blob contains the next chunk of the snapshot in the snapshot stream.	bytes

message `StatusRequest` (etcdserver/etcdserverpb/rpc.proto)

Empty field.

message `StatusResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
version	version is the cluster protocol version used by the responding member.	string

Field	Description	Type
dbSize	dbSize is the size of the backend database physically allocated, in bytes, of the responding member.	int64
leader	leader is the member ID which the responding member believes is the current leader.	uint64
raftIndex	raftIndex is the current raft committed index of the responding member.	uint64
raftTerm	raftTerm is the current raft term of the responding member.	uint64
raftAppliedIndex	raftAppliedIndex is the current raft applied index of the responding member.	uint64
errors	errors contains alarm/health information and status.	(slice of) string
dbSizeInUse	dbSizeInUse is the size of the backend database logically in use, in bytes, of the responding member.	int64
isLearner	isLearner indicates if the member is raft learner.	bool

message `TxnRequest` (etcdserver/etcdserverpb/rpc.proto)

From google paxosdb paper: Our implementation hinges around a powerful primitive which we call MultiOp. All other database operations except for iteration are implemented as a single call to MultiOp. A MultiOp is applied atomically and consists of three components: 1. A list of tests called guard. Each test in guard checks a single entry in the database. It may check for the absence or presence of a value, or compare with a given value. Two different tests in the guard may apply to the same or different entries in the database. All tests in the guard are applied and MultiOp returns the results. If all tests are true, MultiOp executes t op (see item 2 below), otherwise it executes f op (see item 3 below). 2. A list of database operations called t op. Each operation in the list is either an insert, delete, or lookup operation, and applies to a single database entry. Two different operations in the list may apply to the same or different entries in the database. These operations are executed if guard evaluates to true. 3. A list of database operations called f op. Like t op, but executed if guard evaluates to false.

Field	Description	Type
compare	compare is a list of predicates representing a conjunction of terms. If the comparisons succeed, then the success requests will be processed in order, and the response will contain their respective responses in order. If the comparisons fail, then the	(slice of) Compare

Field	Description	Type
	failure requests will be processed in order, and the response will contain their respective responses in order.	
success	success is a list of requests which will be applied when compare evaluates to true.	(slice of) RequestOp
failure	failure is a list of requests which will be applied when compare evaluates to false.	(slice of) RequestOp

message `TxnResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
succeeded	succeeded is set to true if the compare evaluated to true or false otherwise.	bool
responses	responses is a list of responses corresponding to the results from applying success if succeeded is true or failure if succeeded is false.	(slice of) ResponseOp

message `WatchCancelRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
watch_id	watch_id is the watcher id to cancel so that no more events are transmitted.	int64

message `WatchCreateRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
key	key is the key to register for watching.	bytes
range_end	range_end is the end of the range [key, range_end) to watch. If range_end is not given, only the key argument is watched. If range_end is equal to '\0', all keys greater than or equal to the key argument are watched. If the range_end is one bit larger than the given key, then all keys with the prefix (the given key) will be watched.	bytes

Field	Description	Type
start_revision	start_revision is an optional revision to watch from (inclusive). No start_revision is “now”.	int64
progress_notify	progress_notify is set so that the etcd server will periodically send a WatchResponse with no events to the new watcher if there are no recent events. It is useful when clients wish to recover a disconnected watcher starting from a recent known revision. The etcd server may decide how often it will send notifications based on current load.	bool
filters	filters filter the events at server side before it sends back to the watcher.	(slice of) FilterType
prev_kv	If prev_kv is set, created watcher gets the previous KV before the event happens. If the previous KV is already compacted, nothing will be returned.	bool
watch_id	If watch_id is provided and non-zero, it will be assigned to this watcher. Since creating a watcher in etcd is not a synchronous operation, this can be used ensure that ordering is correct when creating multiple watchers on the same stream. Creating a watcher with an ID already in use on the stream will cause an error to be returned.	int64
fragment	fragment enables splitting large revisions into multiple watch responses.	bool

message `WatchProgressRequest` (etcdserver/etcdserverpb/rpc.proto)

Requests the a watch stream progress status be sent in the watch response stream as soon as possible.

Empty field.

message `WatchRequest` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
request_union	request_union is a request to either create a new watcher or cancel an existing watcher.	oneof
create_request		WatchCreateRequest

Field	Description	Type
cancel_request		WatchCancelRequest
progress_request		WatchProgressRequest

message `WatchResponse` (etcdserver/etcdserverpb/rpc.proto)

Field	Description	Type
header		ResponseHeader
watch_id	watch_id is the ID of the watcher that corresponds to the response.	int64
created	created is set to true if the response is for a create watch request. The client should record the watch_id and expect to receive events for the created watcher from the same stream. All events sent to the created watcher will attach with the same watch_id.	bool
canceled	canceled is set to true if the response is for a cancel watch request. No further events will be sent to the canceled watcher.	bool
compact_revision	compact_revision is set to the minimum index if a watcher tries to watch at a compacted index. This happens when creating a watcher at a compacted revision or the watcher cannot catch up with the progress of the key-value store. The client should treat the watcher as canceled and should not try to create any watcher with the same start_revision again.	int64
cancel_reason	cancel_reason indicates the reason for canceling the watcher.	string
fragment	framgment is true if large watch response was split over multiple responses.	bool
events		(slice of) mvccpb.Event

message `Event` (mvcc/mvccpb/kv.proto)

Field	Description	Type
type	type is the kind of event. If type is a PUT, it indicates new data has been stored to the key. If type is a DELETE, it indicates the key was deleted.	EventType
kv	kv holds the KeyValue for the event. A PUT event contains current kv pair. A PUT event with kv.Version=1 indicates the creation of a key. A DELETE/EXPIRE event contains the deleted key with its modification revision set to the revision of deletion.	KeyValue
prev_kv	prev_kv holds the key-value pair before the event happens.	KeyValue

message `KeyValue` (mvcc/mvccpb/kv.proto)

Field	Description	Type
key	key is the key in bytes. An empty key is not allowed.	bytes
create_revision	create_revision is the revision of last creation on this key.	int64
mod_revision	mod_revision is the revision of last modification on this key.	int64
version	version is the version of the key. A deletion resets the version to zero and any modification of the key increases its version.	int64
value	value is the value held by the key, in bytes.	bytes
lease	lease is the ID of the lease that attached to key. When the attached lease expires, the key will be deleted. If lease is 0, then no lease is attached to the key.	int64

message `Lease` (lease/leasepb/lease.proto)

Field	Description	Type
ID		int64
TTL		int64
RemainingTTL		int64

message `LeaseInternalRequest` (lease/leasepb/lease.proto)

Field	Description	Type
LeaseTimeToLiveRequest		etcdserverpb.LeaseTimeToLiveRequest

message `LeaseInternalResponse` (lease/leasepb/lease.proto)

Field	Description	Type
LeaseTimeToLiveResponse		etcdserverpb.LeaseTimeToLiveResponse

message `Permission` (auth/authpb/auth.proto)

Permission is a single entity

Field	Description	Type
permType		Type
key		bytes
range_end		bytes

message `Role` (auth/authpb/auth.proto)

Role is a single entry in the bucket authRoles

Field	Description	Type
name		bytes
keyPermission		(slice of) Permission

message `User` (auth/authpb/auth.proto)

User is a single entry in the bucket authUsers

Field	Description	Type
name		bytes
password		bytes
roles		(slice of) string

Field	Description	Type
options		UserAddOptions

message `UserAddOptions` (auth/authpb/auth.proto)

Field	Description	Type
no_password		bool

Last modified April 9, 2022: Fix typos (a2da31e)

API reference: concurrency

This API reference is autogenerated from the named `.proto` files.

service `Lock` (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

The lock service exposes client-side locking facilities as a gRPC interface.

Method	Request Type	Response Type	Description
Lock	LockRequest	LockResponse	Lock acquires a distributed shared lock on a given named lock. On success, it will return a unique key that exists so long as the lock is held by the caller. This key can be used in conjunction with transactions to safely ensure updates to etcd only occur while holding lock ownership. The lock is held until Unlock is called on the key or the lease associate with the owner expires.
Unlock	UnlockRequest	UnlockResponse	Unlock takes a key returned by Lock and releases the hold on lock. The next Lock caller waiting for the lock will then be woken up and given ownership of the lock.

message `LockRequest` (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
name	name is the identifier for the distributed shared lock to be acquired.	bytes
lease	lease is the ID of the lease that will be attached to ownership of the lock. If the lease expires or is revoked and currently holds the lock, the lock is automatically released. Calls to Lock with the same lease will be treated as a single acquisition; locking twice with the same lease is a no-op.	int64

message `LockResponse` (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader
key	key is a key that will exist on etcd for the duration that the Lock caller owns the lock. Users should not modify this key or the lock may exhibit undefined behavior.	bytes

message `UnlockRequest` (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
key	key is the lock ownership key granted by Lock.	bytes

message `UnlockResponse` (etcdserver/api/v3lock/v3lockpb/v3lock.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader

service `Election` (etcdserver/api/v3election/v3electionpb/v3election.proto)

The election service exposes client-side election facilities as a gRPC interface.

Method	Request Type	Response Type	Description
Campaign	CampaignRequest	CampaignResponse	Campaign waits to acquire leadership in an election, returning a LeaderKey representing the leadership if successful. The LeaderKey can then be used to issue new values on the election, transactionally guard API requests on leadership still being held, and resign from the election.
Proclaim	ProclaimRequest	ProclaimResponse	Proclaim updates the leader's posted value with a new value.
Leader	LeaderRequest	LeaderResponse	Leader returns the current election proclamation, if any.

Method	Request Type	Response Type	Description
Observe	LeaderRequest	LeaderResponse	Observe streams election proclamations in-order as made by the election's elected leaders.
Resign	ResignRequest	ResignResponse	Resign releases election leadership so other campaigners may acquire leadership on the election.

message `CampaignRequest` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
name	name is the election's identifier for the campaign.	bytes
lease	lease is the ID of the lease attached to leadership of the election. If the lease expires or is revoked before resigning leadership, then the leadership is transferred to the next campaigner, if any.	int64
value	value is the initial proclaimed value set when the campaigner wins the election.	bytes

message `CampaignResponse` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader
leader	leader describes the resources used for holding leadereship of the election.	LeaderKey

message `LeaderKey` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
name	name is the election identifier that corresponds to the leadership key.	bytes
key	key is an opaque key representing the ownership of the election. If the key is deleted, then leadership is lost.	bytes

Field	Description	Type
rev	rev is the creation revision of the key. It can be used to test for ownership of an election during transactions by testing the key's creation revision matches rev.	int64
lease	lease is the lease ID of the election leader.	int64

message `LeaderRequest` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
name	name is the election identifier for the leadership information.	bytes

message `LeaderResponse` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader
kv	kv is the key-value pair representing the latest leader update.	mvccpb.KeyValue

message `ProclaimRequest` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
leader	leader is the leadership hold on the election.	LeaderKey
value	value is an update meant to overwrite the leader's current value.	bytes

message `ProclaimResponse` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader

message `ResignRequest` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
leader	leader is the leadership to relinquish by resignation.	LeaderKey

message `ResignResponse` (etcdserver/api/v3election/v3electionpb/v3election.proto)

Field	Description	Type
header		etcdserverpb.ResponseHeader

message `Event` (mvcc/mvccpb/kv.proto)

Field	Description	Type
type	type is the kind of event. If type is a PUT, it indicates new data has been stored to the key. If type is a DELETE, it indicates the key was deleted.	EventType
kv	kv holds the KeyValue for the event. A PUT event contains current kv pair. A PUT event with kv.Version=1 indicates the creation of a key. A DELETE/EXPIRE event contains the deleted key with its modification revision set to the revision of deletion.	KeyValue
prev_kv	prev_kv holds the key-value pair before the event happens.	KeyValue

message `KeyValue` (mvcc/mvccpb/kv.proto)

Field	Description	Type
key	key is the key in bytes. An empty key is not allowed.	bytes
create_revision	create_revision is the revision of last creation on this key.	int64
mod_revision	mod_revision is the revision of last modification on this key.	int64
version	version is the version of the key. A deletion resets the version to zero and any modification of the key increases its version.	int64
value	value is the value held by the key, in bytes.	bytes
lease	lease is the ID of the lease that attached to key. When the attached lease expires, the key will be deleted. If lease is 0, then no lease is attached to the key.	int64

Last modified April 9, 2022: Fix typos (a2da31e)

Operations guide

etcd installation, maintenance, and troubleshooting guides

Hardware recommendations
Hardware guidelines for administering etcd clusters

Maintenance
Periodic etcd cluster maintenance guide

Performance
Understanding performance: latency & throughput

Design of runtime reconfiguration
The design of etcd's runtime reconfiguration commands

Runtime reconfiguration
etcd incremental runtime reconfiguration support

Supported platforms
etcd support status for common architectures & operating systems

Migrate applications from using API v2 to API v3
A guide for migrating from API v2 to API v3

Versioning
Versioning support by etcd

Data Corruption
etcd data corruption and recovery

Monitoring etcd

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Configuration options

etcd configuration: files, flags, and environment variables

etcd is configurable through a configuration file, various command-line flags, and environment variables.

A reusable configuration file is a YAML file made with name and value of one or more command-line flags described below. In order to use this file, specify the file path as a value to the `--config-file` flag or `ETCD_CONFIG_FILE` environment variable. The sample configuration file can be used as a starting point to create a new configuration file as needed.

Options set on the command line take precedence over those from the environment. If a configuration file is provided, other command line flags and environment variables will be ignored. For example, `etcd --config-file etcd.conf.yml.sample --data-dir /tmp` will ignore the `--data-dir` flag.

The format of environment variable for flag `--my-flag` is `ETCD_MY_FLAG`. It applies to all flags.

The official etcd ports are 2379 for client requests and 2380 for peer communication. The etcd ports can be set to accept TLS traffic, non-TLS traffic, or both TLS and non-TLS traffic.

To start etcd automatically using custom settings at startup in Linux, using a systemd unit is highly recommended.

The list of flags provided below may not be up-to-date due to ongoing development changes. For the latest available flags, run `etcd --help` or refer to the etcd help.

Member flags

--name

- Human-readable name for this member.
- default: "default"
- env variable: ETCD_NAME
- This value is referenced as this node's own entries listed in the `--initial-cluster` flag (e.g., `default=http://localhost:2380`). This needs to match the key used in the flag if using

static bootstrapping. When using discovery, each member must have a unique name. `Hostname` or `machine-id` can be a good choice.

--data-dir

- Path to the data directory.
- default: "${name}.etcd"
- env variable: ETCD_DATA_DIR

--wal-dir

- Path to the dedicated wal directory. If this flag is set, etcd will write the WAL files to the walDir rather than the dataDir. This allows a dedicated disk to be used, and helps avoid io competition between logging and other IO operations.
- default: ""
- env variable: ETCD_WAL_DIR

--snapshot-count

- Number of committed transactions to trigger a snapshot to disk.
- default: "100000"
- env variable: ETCD_SNAPSHOT_COUNT

--heartbeat-interval

- Time (in milliseconds) of a heartbeat interval.
- default: "100"
- env variable: ETCD_HEARTBEAT_INTERVAL

--election-timeout

- Time (in milliseconds) for an election to timeout. See Documentation/tuning.md for details.
- default: "1000"
- env variable: ETCD_ELECTION_TIMEOUT

--initial-election-tick-advance

- Whether to fast-forward initial election ticks on boot for faster election. When it is true, then local member fast-forwards election ticks to speed up "initial" leader election trigger. This benefits the case of larger election ticks. Disabling this would slow down initial bootstrap process for cross datacenter deployments. Make your own tradeoffs by configuring this flag at the cost of slow initial bootstrap.

- default: true
- env variable: ETCD_INITIAL_ELECTION_TICK_ADVANCE

--listen-peer-urls

- List of URLs to listen on for peer traffic. This flag tells the etcd to accept incoming requests from its peers on the specified `scheme://IP:port` combinations. Scheme can be `http` or `https`. Alternatively, use `unix://<file-path>` or `unixs://<file-path>` for unix sockets. If `0.0.0.0` is specified as the IP, etcd listens to the given port on all interfaces. If an IP address is given as well as a port, etcd will listen on the given port and interface. Multiple URLs may be used to specify a number of addresses and ports to listen on. The etcd will respond to requests from any of the listed addresses and ports.
- default: "http://localhost:2380"
- env variable: ETCD_LISTEN_PEER_URLS
- example: "http://10.0.0.1:2380"
- invalid example: "http://example.com:2380" (domain name is invalid for binding)

--listen-client-urls

- List of URLs to listen on for client traffic. This flag tells the etcd to accept incoming requests from the clients on the specified `scheme://IP:port` combinations as long as `--listen-client-http-urls` is not specified. Scheme can be either `http` or `https`. Alternatively, use `unix://<file-path>` or `unixs://<file-path>` for unix sockets. If `0.0.0.0` is specified as the IP, etcd listens to the given port on all interfaces. If an IP address is given as well as a port, etcd will listen on the given port and interface. Multiple URLs may be used to specify a number of addresses and ports to listen on. The etcd will respond to requests from any of the listed addresses and ports.
- default: "http://localhost:2379"
- env variable: ETCD_LISTEN_CLIENT_URLS
- example: "http://10.0.0.1:2379"
- invalid example: "http://example.com:2379" (domain name is invalid for binding)

--listen-client-http-urls

- List of URLs to listen on for HTTP-only client traffic. This flag tells the etcd to accept incoming requests from clients on the specified `http://IP:port` combinations. Enabling this flag removes HTTP services from `--listen-client-urls`. Use this flag when you want to segregate HTTP traffic from other protocols. If `0.0.0.0` is specified as the IP, etcd listens to the given port on all interfaces. If an IP address is given as well as a port, etcd will listen on the given port and interface. Multiple URLs may be used to specify a number of addresses and ports to listen on. The etcd will respond to requests from any of the listed addresses and ports.

- default: ""
- env variable: ETCD_LISTEN_CLIENT_HTTP_URLS
- example: “http://10.0.0.1:2383”
- invalid example: “http://example.com:2383” (domain name is invalid for binding)

--max-snapshots

- Maximum number of snapshot files to retain (0 is unlimited)
- default: 5
- env variable: ETCD_MAX_SNAPSHOTS
- The default for users on Windows is unlimited, and manual purging down to 5 (or some preference for safety) is recommended.

--max-wals

- Maximum number of wal files to retain (0 is unlimited)
- default: 5
- env variable: ETCD_MAX_WALS
- The default for users on Windows is unlimited, and manual purging down to 5 (or some preference for safety) is recommended.

--cors

- Comma-separated white list of origins for CORS (cross-origin resource sharing).
- default: ""
- env variable: ETCD_CORS

--quota-backend-bytes

- Raise alarms when backend size exceeds the given quota (0 defaults to low space quota).
- default: 0
- env variable: ETCD_QUOTA_BACKEND_BYTES

--backend-batch-limit

- BackendBatchLimit is the maximum operations before commit the backend transaction.
- default: 0

- env variable: ETCD_BACKEND_BATCH_LIMIT

--backend-batch-interval

- BackendBatchInterval is the maximum time before commit the backend transaction.
- default: 0
- env variable: ETCD_BACKEND_BATCH_INTERVAL

--max-txn-ops

- Maximum number of operations permitted in a transaction.
- default: 128
- env variable: ETCD_MAX_TXN_OPS

--max-request-bytes

- Maximum client request size in bytes the server will accept.
- default: 1572864
- env variable: ETCD_MAX_REQUEST_BYTES

--grpc-keepalive-min-time

- Minimum duration interval that a client should wait before pinging server.
- default: 5s
- env variable: ETCD_GRPC_KEEPALIVE_MIN_TIME

--grpc-keepalive-interval

- Frequency duration of server-to-client ping to check if a connection is alive (0 to disable).
- default: 2h
- env variable: ETCD_GRPC_KEEPALIVE_INTERVAL

--grpc-keepalive-timeout

- Additional duration of wait before closing a non-responsive connection (0 to disable).
- default: 20s
- env variable: ETCD_GRPC_KEEPALIVE_TIMEOUT

Clustering flags

`--initial-advertise-peer-urls`, `--initial-cluster`, `--initial-cluster-state`, and `--initial-cluster-token` flags are used in bootstrapping (static bootstrap, discovery-service bootstrap or runtime reconfiguration) a new member, and ignored when restarting an existing member.

`--discovery` prefix flags need to be set when using discovery service.

--initial-advertise-peer-urls

- List of this member's peer URLs to advertise to the rest of the cluster. These addresses are used for communicating etcd data around the cluster. At least one must be routable to all cluster members. These URLs can contain domain names.
- default: "http://localhost:2380"
- env variable: ETCD_INITIAL_ADVERTISE_PEER_URLS
- example: "http://example.com:2380, http://10.0.0.1:2380"

--initial-cluster

- Initial cluster configuration for bootstrapping.
- default: "default=http://localhost:2380"
- env variable: ETCD_INITIAL_CLUSTER
- The key is the value of the `--name` flag for each node provided. The default uses `default` for the key because this is the default for the `--name` flag.

--initial-cluster-state

- Initial cluster state ("new" or "existing"). Set to `new` for all members present during initial static or DNS bootstrapping. If this option is set to `existing`, etcd will attempt to join the existing cluster. If the wrong value is set, etcd will attempt to start but fail safely.
- default: "new"
- env variable: ETCD_INITIAL_CLUSTER_STATE

--initial-cluster-token

- Initial cluster token for the etcd cluster during bootstrap.
- default: "etcd-cluster"
- env variable: ETCD_INITIAL_CLUSTER_TOKEN

--advertise-client-urls

- List of this member's client URLs to advertise to the rest of the cluster. These URLs can contain domain names.
- default: "http://localhost:2379"
- env variable: ETCD_ADVERTISE_CLIENT_URLS
- example: "http://example.com:2379, http://10.0.0.1:2379"
- Be careful if advertising URLs such as http://localhost:2379 from a cluster member and are using the proxy feature of etcd. This will cause loops, because the proxy will be forwarding requests to itself until its resources (memory, file descriptors) are eventually depleted.

--discovery

- Discovery URL used to bootstrap the cluster.
- default: ""
- env variable: ETCD_DISCOVERY

--discovery-srv

- DNS srv domain used to bootstrap the cluster.
- default: ""
- env variable: ETCD_DISCOVERY_SRV

--discovery-srv-name

- Suffix to the DNS srv name queried when bootstrapping using DNS.
- default: ""
- env variable: ETCD_DISCOVERY_SRV_NAME

--discovery-fallback

- Expected behavior ("exit" or "proxy") when discovery services fails. "proxy" supports v2 API only.
- default: "proxy"
- env variable: ETCD_DISCOVERY_FALLBACK

--discovery-proxy

- HTTP proxy to use for traffic to discovery service.
- default: ""
- env variable: ETCD_DISCOVERY_PROXY

--strict-reconfig-check

- Reject reconfiguration requests that would cause quorum loss.
- default: true
- env variable: ETCD_STRICT_RECONFIG_CHECK

--auto-compaction-retention

- Auto compaction retention for mvcc key value store in hour. 0 means disable auto compaction.
- default: 0
- env variable: ETCD_AUTO_COMPACTION_RETENTION

--auto-compaction-mode

- Interpret 'auto-compaction-retention' one of: 'periodic', 'revision'. 'periodic' for duration based retention, defaulting to hours if no time unit is provided (e.g. '5m'). 'revision' for revision number based retention.
- default: periodic
- env variable: ETCD_AUTO_COMPACTION_MODE

--enable-v2

- Accept etcd V2 client requests
- default: false
- env variable: ETCD_ENABLE_V2

--next-cluster-version-compatible

- Enable a 3.4 server to be compatible with version 3.5, allowing the 3.4 server to join a 3.5 cluster and start on 3.5 schema for cluster downgrade purposes.
- default: false
- env variable: ETCD_NEXT_CLUSTER_VERSION_COMPATIBLE

Proxy flags

`--proxy` prefix flags configures etcd to run in [proxy mode](). "proxy" supports v2 API only.

--proxy

- Proxy mode setting (“off”, “readonly” or “on”).
- default: “off”
- env variable: ETCD_PROXY

--proxy-failure-wait

- Time (in milliseconds) an endpoint will be held in a failed state before being reconsidered for proxied requests.
- default: 5000
- env variable: ETCD_PROXY_FAILURE_WAIT

--proxy-refresh-interval

- Time (in milliseconds) of the endpoints refresh interval.
- default: 30000
- env variable: ETCD_PROXY_REFRESH_INTERVAL

--proxy-dial-timeout

- Time (in milliseconds) for a dial to timeout or 0 to disable the timeout
- default: 1000
- env variable: ETCD_PROXY_DIAL_TIMEOUT

--proxy-write-timeout

- Time (in milliseconds) for a write to timeout or 0 to disable the timeout.
- default: 5000
- env variable: ETCD_PROXY_WRITE_TIMEOUT

--proxy-read-timeout

- Time (in milliseconds) for a read to timeout or 0 to disable the timeout.
- Don’t change this value if using watches because use long polling requests.
- default: 0
- env variable: ETCD_PROXY_READ_TIMEOUT

Security flags

The security flags help to build a secure etcd cluster.

--ca-file

DEPRECATED

- Path to the client server TLS CA file. `--ca-file ca.crt` could be replaced by `--trusted-ca-file ca.crt --client-cert-auth` and etcd will perform the same.
- default: ""
- env variable: ETCD_CA_FILE

--cert-file

- Path to the client server TLS cert file.
- default: ""
- env variable: ETCD_CERT_FILE

--key-file

- Path to the client server TLS key file.
- default: ""
- env variable: ETCD_KEY_FILE

--client-cert-auth

- Enable client cert authentication.
- default: false
- env variable: ETCD_CLIENT_CERT_AUTH
- CN authentication is not supported by gRPC-gateway. It’s recommended to enable client cert authentication to prevent attacks from unauthenticated clients (e.g. CVE-2023-44487), especially when running etcd as a public service.

--client-crl-file

- Path to the client certificate revocation list file.
- default: ""
- env variable: ETCD_CLIENT_CRL_FILE

--client-cert-allowed-hostname

- Allowed Allowed TLS name for client cert authentication.
- default: ""
- env variable: ETCD_CLIENT_CERT_ALLOWED_HOSTNAME

--trusted-ca-file

- Path to the client server TLS trusted CA cert file.
- default: ""
- env variable: ETCD_TRUSTED_CA_FILE
- Note setting this parameter will also automatically enable client cert authentication no matter what value is set for `--client-cert-auth`.

--auto-tls

- Client TLS using generated certificates
- default: false
- env variable: ETCD_AUTO_TLS

--peer-ca-file

DEPRECATED

- Path to the peer server TLS CA file. `--peer-ca-file ca.crt` could be replaced by `--peer-trusted-ca-file ca.crt --peer-client-cert-auth` and etcd will perform the same.
- default: ""
- env variable: ETCD_PEER_CA_FILE

--peer-cert-file

- Path to the peer server TLS cert file. This is the cert for peer-to-peer traffic, used both for server and client.
- default: ""
- env variable: ETCD_PEER_CERT_FILE

--peer-key-file

- Path to the peer server TLS key file. This is the key for peer-to-peer traffic, used both for server and client.
- default: ""
- env variable: ETCD_PEER_KEY_FILE

--peer-client-cert-auth

- Enable peer client cert authentication.
- default: false

- env variable: ETCD_PEER_CLIENT_CERT_AUTH
- It's recommended to enable peer client cert authentication to prevent attacks from unauthenticated forged peers (e.g. CVE-2023-44487).

--peer-crl-file

- Path to the peer certificate revocation list file.
- default: ""
- env variable: ETCD_PEER_CRL_FILE

--peer-trusted-ca-file

- Path to the peer server TLS trusted CA file.
- default: ""
- env variable: ETCD_PEER_TRUSTED_CA_FILE

--peer-auto-tls

- Peer TLS using generated certificates
- default: false
- env variable: ETCD_PEER_AUTO_TLS

--peer-cert-allowed-cn

- Allowed CommonName for inter peer authentication.
- default: ""
- env variable: ETCD_PEER_CERT_ALLOWED_CN

--peer-cert-allowed-hostname

- Allowed TLS certificate name for inter peer authentication.
- default: ""
- env variable: ETCD_PEER_CERT_ALLOWED_HOSTNAME

--cipher-suites

- Comma-separated list of supported TLS cipher suites between server/client and peers.
- default: ""
- env variable: ETCD_CIPHER_SUITES

--tls-min-version

- Minimum TLS version supported by etcd.
- default: “TLS1.2”

--tls-max-version

- Maximum TLS version supported by etcd.
- detault: ""

Logging flags

--logger

Available from v3.4. WARNING: `--logger=capnslog` to be deprecated in v3.5.

- Specify ‘zap’ for structured logging or ‘capnslog’.
- default: capnslog
- env variable: ETCD_LOGGER

--log-outputs

- Specify ‘stdout’ or ‘stderr’ to skip journald logging even when running under systemd, or list of comma separated output targets.
- default: default
- env variable: ETCD_LOG_OUTPUTS
- ‘default’ use ‘stderr’ config for v3.4 during zap logger migraion

--log-level

Available from v3.4.

- Configures log level. Only supports debug, info, warn, error, panic, or fatal.
- default: info
- env variable: ETCD_LOG_LEVEL
- ‘default’ use ‘info’.

--debug

WARNING: to be deprecated in v3.5.

- Drop the default log level to DEBUG for all subpackages.
- default: false (INFO for all packages)
- env variable: ETCD_DEBUG

--log-package-levels

WARNING: to be deprecated in v3.5.

- Set individual etcd subpackages to specific log levels. An example being `etcdserver=WARNING,security=DEBUG`
- default: "" (INFO for all packages)
- env variable: ETCD_LOG_PACKAGE_LEVELS

Unsafe flags

Please be CAUTIOUS when using unsafe flags because it will break the guarantees given by the consensus protocol. For example, it may panic if other members in the cluster are still alive. Follow the instructions when using these flags.

--force-new-cluster

- Force to create a new one-member cluster. It commits configuration changes forcing to remove all existing members in the cluster and add itself, but is strongly discouraged. Please review the [disaster recovery]() documentation for preferred v3 recovery procedures.
- default: false
- env variable: ETCD_FORCE_NEW_CLUSTER

Miscellaneous flags

--version

- Print the version and exit.
- default: false

--config-file

- Load server configuration from a file. Note that if a configuration file is provided, other command line flags and environment variables will be ignored.
- default: ""

- example: sample configuration file
- env variable: ETCD_CONFIG_FILE

Profiling flags

--enable-pprof

- Enable runtime profiling data via HTTP server. Address is at client URL + "/debug/pprof/"
- default: false
- env variable: ETCD_ENABLE_PPROF

--metrics

- Set level of detail for exported metrics, specify 'extensive' to include server side grpc histogram metrics.
- default: basic
- env variable: ETCD_METRICS

--listen-metrics-urls

- List of additional URLs to listen on that will respond to both the `/metrics` and `/health` endpoints
- default: ""
- env variable: ETCD_LISTEN_METRICS_URLS

Auth flags

--auth-token

- Specify a token type and token specific options, especially for JWT. Its format is "type,var1=val1,var2=val2,...". Possible type is 'simple' or 'jwt'. Possible variables are 'sign-method' for specifying a sign method of jwt (its possible values are 'ES256', 'ES384', 'ES512', 'HS256', 'HS384', 'HS512', 'RS256', 'RS384', 'RS512', 'PS256', 'PS384', or 'PS512'), 'pub-key' for specifying a path to a public key for verifying jwt, 'priv-key' for specifying a path to a private key for signing jwt, and 'ttl' for specifying TTL of jwt tokens.
- For asymmetric algorithms ('RS', 'PS', 'ES'), the public key is optional, as the private key contains enough information to both sign and verify tokens.
- Example option of JWT: '--auth-token jwt,pub-key=app.rsa.pub,priv-key=app.rsa,sign-method=RS512,ttl=10m'
- default: "simple"

- env variable: ETCD_AUTH_TOKEN

--bcrypt-cost

- Specify the cost / strength of the bcrypt algorithm for hashing auth passwords. Valid values are between 4 and 31.
- default: 10
- env variable: (not supported)

Experimental flags

--experimental-backend-bbolt-freelist-type

- The freelist type that etcd backend(bboltdb) uses (array and map are supported types).
- default: array
- env variable: ETCD_EXPERIMENTAL_BACKEND_BBOLT_FREELIST_TYPE

--experimental-corrupt-check-time

- Duration of time between cluster corruption check passes
- default: 0s
- env variable: ETCD_EXPERIMENTAL_CORRUPT_CHECK_TIME

--experimental-compaction-batch-limit

- Sets the maximum revisions deleted in each compaction batch.
- default: 1000
- env variable: ETCD_EXPERIMENTAL_COMPACTION_BATCH_LIMIT

--experimental-peer-skip-client-san-verification

- Skip verification of SAN field in client certificate for peer connections. This can be helpful e.g. if cluster members run in different networks behind a NAT.

  In this case make sure to use peer certificates based on a private certificate authority using `--peer-cert-file`, `--peer-key-file`, `--peer-trusted-ca-file`

- default: false

- env variable: ETCD_EXPERIMENTAL_PEER_SKIP_CLIENT_SAN_VERIFICATION

Last modified June 24, 2024: Update doc of listen-client-urls and listen-client-http-urls (0d6e903)

Role-based access control

A basic authentication and role-based access control guide

Overview

Authentication was added in etcd 2.1. The etcd v3 API slightly modified the authentication feature's API and user interface to better fit the new data model. This guide is intended to help users set up basic authentication and role-based access control in etcd v3.

Special users and roles

There is one special user, `root`, and one special role, `root`.

User `root`

The `root` user, which has full access to etcd, must be created before activating authentication. The idea behind the `root` user is for administrative purposes: managing roles and ordinary users. The `root` user must have the `root` role and is allowed to change anything inside etcd.

Role `root`

The role `root` may be granted to any user, in addition to the root user. A user with the `root` role has both global read-write access and permission to update the cluster's authentication configuration. Furthermore, the `root` role grants privileges for general cluster maintenance, including modifying cluster membership, defragmenting the store, and taking snapshots.

Working with users

The `user` subcommand for `etcdctl` handles all things having to do with user accounts.

A listing of users can be found with:

```
$ etcdctl user list
```

Creating a user is as easy as

```
$ etcdctl user add myusername
```

Creating a new user will prompt for a new password. The password can be supplied from standard input when an option `--interactive=false` is given. `--new-user-password` can also be used for supplying the password.

Roles can be granted and revoked for a user with:

```
$ etcdctl user grant-role myusername foo
$ etcdctl user revoke-role myusername bar
```

The user's settings can be inspected with:

```
$ etcdctl user get myusername
```

And the password for a user can be changed with

```
$ etcdctl user passwd myusername
```

Changing the password will prompt again for a new password. The password can be supplied from standard input when an option `--interactive=false` is given.

Delete an account with:

```
$ etcdctl user delete myusername
```

Working with roles

The `role` subcommand for `etcdctl` handles all things having to do with access controls for particular roles, as were granted to individual users.

List roles with:

```
$ etcdctl role list
```

Create a new role with:

```
$ etcdctl role add myrolename
```

A role has no password; it merely defines a new set of access rights.

Roles are granted access to a single key or a range of keys.

The range can be specified as an interval [start-key, end-key) where start-key should be lexically less than end-key in an alphabetical manner.

Access can be granted as either read, write, or both, as in the following examples:

```
# Give read access to a key /foo
$ etcdctl role grant-permission myrolename read /foo

# Give read access to keys with a prefix /foo/. The prefix is equal to the range [/foo/,
$ etcdctl role grant-permission myrolename --prefix=true read /foo/

# Give write-only access to the key at /foo/bar
$ etcdctl role grant-permission myrolename write /foo/bar

# Give full access to keys in a range of [key1, key5)
$ etcdctl role grant-permission myrolename readwrite key1 key5

# Give full access to keys with a prefix /pub/
$ etcdctl role grant-permission myrolename --prefix=true readwrite /pub/
```

To see what's granted, we can look at the role at any time:

```
$ etcdctl role get myrolename
```

Revocation of permissions is done the same logical way:

```
$ etcdctl role revoke-permission myrolename /foo/bar
```

As is removing a role entirely:

```
$ etcdctl role delete myrolename
```

Enabling authentication

The minimal steps to enabling auth are as follows. The administrator can set up users and roles before or after enabling authentication, as a matter of preference.

Make sure the root user is created:

```
$ etcdctl user add root
Password of root:
```

Enable authentication:

```
$ etcdctl auth enable
```

After this, etcd is running with authentication enabled. To disable it for any reason, use the reciprocal command:

```
$ etcdctl --user root:rootpw auth disable
```

Using `etcdctl` to authenticate

`etcdctl` supports a similar flag as `curl` for authentication.

```
$ etcdctl --user user:password get foo
```

The password can be taken from a prompt:

```
$ etcdctl --user user get foo
```

The password can also be taken from a command line flag `--password` :

```
$ etcdctl --user user --password password get foo
```

Creating a user which cannot be authenticated with password is also possible like below:

```
$ etcdctl user add myusername --no-password
```

Such a user can only be authenticated with TLS Common Name.

Otherwise, all `etcdctl` commands remain the same. Users and roles can still be created and modified, but require authentication by a user with the root role.

Using TLS Common Name

As of version v3.2 if an etcd server is launched with the option `--client-cert-auth=true`, the field of Common Name (CN) in the client's TLS cert will be used as an etcd user. In this case, the common name authenticates the user and the client does not need a password. Note that if both of 1. `--client-cert-auth=true` is passed and CN is provided by the client, and 2. username and password are provided by the client, the username and password based authentication is prioritized. Note that this feature cannot be used with gRPC-proxy and gRPC-gateway. This is because gRPC-proxy terminates TLS from its client so all the clients share a cert of the proxy. gRPC-gateway uses a TLS connection internally for transforming HTTP request to gRPC request so it shares the same limitation. Therefore the clients cannot provide their CN to the server correctly. gRPC-proxy will cause an error and stop if a given cert has non empty CN. gRPC-proxy returns an error which indicates that the client has an non empty CN in its cert.

Notes on password strength

The `etcdctl` and etcd API do not enforce a specific password length during user creation or user password update operations. It is the responsibility of the administrator to enforce these requirements. For avoiding security risks related to password strength, TLS Common Name based authentication and users created with `--no-password` option can be utilized.

Last modified April 17, 2023: Update the RBAC page (26fb46d)

Transport security model

Securing data in transit

etcd supports automatic TLS as well as authentication through client certificates for both clients to server as well as peer (server to server / cluster) communication.

To get up and running, first have a CA certificate and a signed key pair for one member. It is recommended to create and sign a new key pair for every member in a cluster.

For convenience, the cfssl tool provides an easy interface to certificate generation, and we provide an example using the tool here. Alternatively, try this guide to generating self-signed key pairs.

Basic setup

etcd takes several certificate related configuration options, either through command-line flags or environment variables:

Client-to-server communication:

`--cert-file=<path>` : Certificate used for SSL/TLS connections **to** etcd. When this option is set, advertise-client-urls can use the HTTPS schema.

`--key-file=<path>` : Key for the certificate. Must be unencrypted.

`--client-cert-auth` : When this is set etcd will check all incoming HTTPS requests for a client certificate signed by the trusted CA, requests that don't supply a valid client certificate will fail. If authentication is enabled, the certificate provides credentials for the user name given by the Common Name field.

`--trusted-ca-file=<path>` : Trusted certificate authority.

`--auto-tls` : Use automatically generated self-signed certificates for TLS connections with clients.

Peer (server-to-server / cluster) communication:

The peer options work the same way as the client-to-server options:

`--peer-cert-file=<path>` : Certificate used for SSL/TLS connections between peers. This will be used both for listening on the peer address as well as sending requests to other peers.

`--peer-key-file=<path>` : Key for the certificate. Must be unencrypted.

`--peer-client-cert-auth` : When set, etcd will check all incoming peer requests from the cluster for valid client certificates signed by the supplied CA.

`--peer-trusted-ca-file=<path>` : Trusted certificate authority.

`--peer-auto-tls` : Use automatically generated self-signed certificates for TLS connections between peers.

If either a client-to-server or peer certificate is supplied the key must also be set. All of these configuration options are also available through the environment variables, `ETCD_CA_FILE` , `ETCD_PEER_CA_FILE` and so on.

Common options:

`--cipher-suites` : Comma-separated list of supported TLS cipher suites between server/client and peers (empty will be auto-populated by Go).

`--tls-min-version=<version>` Sets the minimum TLS version supported by etcd.

`--tls-max-version=<version>` Sets the maximum TLS version supported by etcd. If not set the maximum version supported by Go will be used.

Example 1: Client-to-server transport security with HTTPS

For this, have a CA certificate (`ca.crt`) and signed key pair (`server.crt` , `server.key`) ready.

Let us configure etcd to provide simple HTTPS transport security step by step:

```
$ etcd --name infra0 --data-dir infra0 \
  --cert-file=/path/to/server.crt --key-file=/path/to/server.key \
  --advertise-client-urls=https://127.0.0.1:2379 --listen-client-urls=https://127.0.0.1:
```

This should start up fine and it will be possible to test the configuration by speaking HTTPS to etcd:

```
$ curl --cacert /path/to/ca.crt https://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -v
```

The command should show that the handshake succeed. Since we use self-signed certificates with our own certificate authority, the CA must be passed to curl using the `--cacert` option. Another possibility would be to add the CA certificate to the system's trusted certificates directory (usually in `/etc/pki/tls/certs` or `/etc/ssl/certs`).

OSX 10.9+ Users: curl 7.30.0 on OSX 10.9+ doesn't understand certificates passed in on the command line. Instead, import the dummy ca.crt directly into the keychain or add the `-k` flag to curl to ignore errors. To test without the `-k` flag, run `open ./fixtures/ca/ca.crt` and follow the prompts. Please remove this certificate after testing! If there is a workaround, let us know.

Example 2: Client-to-server authentication with HTTPS client certificates

For now we've given the etcd client the ability to verify the server identity and provide transport security. We can however also use client certificates to prevent unauthorized access to etcd.

The clients will provide their certificates to the server and the server will check whether the cert is signed by the supplied CA and decide whether to serve the request.

The same files mentioned in the first example are needed for this, as well as a key pair for the client (`client.crt`, `client.key`) signed by the same certificate authority.

```
$ etcd --name infra0 --data-dir infra0 \
  --client-cert-auth --trusted-ca-file=/path/to/ca.crt --cert-file=/path/to/server.crt -
  --advertise-client-urls https://127.0.0.1:2379 --listen-client-urls https://127.0.0.1:
```

Now try the same request as above to this server:

```
$ curl --cacert /path/to/ca.crt https://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -v
```

The request should be rejected by the server:

```
...
routines:SSL3_READ_BYTES:sslv3 alert bad certificate
...
```

To make it succeed, we need to give the CA signed client certificate to the server:

```
$ curl --cacert /path/to/ca.crt --cert /path/to/client.crt --key /path/to/client.key \
  -L https://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -v
```

The output should include:

```
...
SSLv3, TLS handshake, CERT verify (15):
...
TLS handshake, Finished (20)
```

And also the response from the server:

```
{
    "action": "set",
    "node": {
        "createdIndex": 12,
        "key": "/foo",
        "modifiedIndex": 12,
        "value": "bar"
    }
}
```

Specify cipher suites to block weak TLS cipher suites.

TLS handshake would fail when client hello is requested with invalid cipher suites.

For instance:

```
$ etcd \
  --cert-file ./server.crt \
  --key-file ./server.key \
  --trusted-ca-file ./ca.crt \
  --cipher-suites TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SH
```

Then, client requests must specify one of the cipher suites specified in the server:

```
# valid cipher suite
$ curl \
  --cacert /path/to/ca.crt \
```

```
  --cert /path/to/client.crt \
  --key /path/to/client.key \
  -L [CLIENT-URL]/metrics \
  --ciphers ECDHE-RSA-AES128-GCM-SHA256

# request succeeds
etcd_server_version{server_version="3.2.22"} 1
...
```

```
# invalid cipher suite
$ curl \
  --cacert /path/to/ca.crt \
  --cert /path/to/client.crt \
  --key /path/to/client.key \
  -L [CLIENT-URL]/metrics \
  --ciphers ECDHE-RSA-DES-CBC3-SHA

# request fails with
(35) error:14094410:SSL routines:ssl3_read_bytes:sslv3 alert handshake failure
```

Example 3: Transport security & client certificates in a cluster

etcd supports the same model as above for **peer communication**, that means the communication between etcd members in a cluster.

Assuming we have our `ca.crt` and two members with their own key pairs (`member1.crt` & `member1.key` , `member2.crt` & `member2.key`) signed by this CA, we launch etcd as follows:

```
DISCOVERY_URL=... # from https://discovery.etcd.io/new

# member1
$ etcd --name infra1 --data-dir infra1 \
  --peer-client-cert-auth --peer-trusted-ca-file=/path/to/ca.crt --peer-cert-file=/path/
  --initial-advertise-peer-urls=https://10.0.1.10:2380 --listen-peer-urls=https://10.0.1
  --discovery ${DISCOVERY_URL}

# member2
$ etcd --name infra2 --data-dir infra2 \
  --peer-client-cert-auth --peer-trusted-ca-file=/path/to/ca.crt --peer-cert-file=/path/
  --initial-advertise-peer-urls=https://10.0.1.11:2380 --listen-peer-urls=https://10.0.1
  --discovery ${DISCOVERY_URL}
```

The etcd members will form a cluster and all communication between members in the cluster will be encrypted and authenticated using the client certificates. The output of etcd will show that the addresses it connects to use HTTPS.

Example 4: Automatic self-signed transport security

For cases where communication encryption, but not authentication, is needed, etcd supports encrypting its messages with automatically generated self-signed certificates. This simplifies deployment because there is no need for managing certificates and keys outside of etcd.

Configure etcd to use self-signed certificates for client and peer connections with the flags `--auto-tls` and `--peer-auto-tls`:

```
DISCOVERY_URL=... # from https://discovery.etcd.io/new

# member1
$ etcd --name infra1 --data-dir infra1 \
  --auto-tls --peer-auto-tls \
  --initial-advertise-peer-urls=https://10.0.1.10:2380 --listen-peer-urls=https://10.0.1
  --discovery ${DISCOVERY_URL}

# member2
$ etcd --name infra2 --data-dir infra2 \
  --auto-tls --peer-auto-tls \
  --initial-advertise-peer-urls=https://10.0.1.11:2380 --listen-peer-urls=https://10.0.1
  --discovery ${DISCOVERY_URL}
```

Self-signed certificates do not authenticate identity so curl will return an error:

```
curl: (60) SSL certificate problem: Invalid certificate chain
```

To disable certificate chain checking, invoke curl with the `-k` flag:

```
$ curl -k https://127.0.0.1:2379/v2/keys/foo -Xput -d value=bar -v
```

Notes for DNS SRV

Since v3.1.0 (except v3.2.9), discovery SRV bootstrapping authenticates `ServerName` with a root domain name from `--discovery-srv` flag. This is to avoid man-in-the-middle cert attacks, by requiring a certificate to have matching root domain name in its Subject Alternative Name (SAN) field. For instance, `etcd --discovery-srv=etcd.local` will only authenticate peers/clients when the provided certs have root domain `etcd.local` as an entry in Subject Alternative Name (SAN) field

Notes for etcd proxy

etcd proxy terminates the TLS from its client if the connection is secure, and uses proxy's own key/cert specified in `--peer-key-file` and `--peer-cert-file` to communicate with etcd members.

The proxy communicates with etcd members through both the `--advertise-client-urls` and `--advertise-peer-urls` of a given member. It forwards client requests to etcd members' advertised client urls, and it syncs the initial cluster configuration through etcd members' advertised peer urls.

When client authentication is enabled for an etcd member, the administrator must ensure that the peer certificate specified in the proxy's `--peer-cert-file` option is valid for that authentication. The proxy's peer certificate must also be valid for peer authentication if peer authentication is enabled.

Notes for TLS authentication

Since v3.2.0, TLS certificates get reloaded on every client connection. This is useful when replacing expiry certs without stopping etcd servers; it can be done by overwriting old certs with new ones. Refreshing certs for every connection should not have too much overhead, but can be improved in the future, with caching layer. Example tests can be found here.

Since v3.2.0, server denies incoming peer certs with wrong IP `SAN`. For instance, if peer cert contains any IP addresses in Subject Alternative Name (SAN) field, server authenticates a peer only when the remote IP address matches one of those IP addresses. This is to prevent unauthorized endpoints from joining the cluster. For example, peer B's CSR (with `cfssl`) is:

```
{
  "CN": "etcd peer",
  "hosts": [
    "*.example.default.svc",
    "*.example.default.svc.cluster.local",
    "10.138.0.27"
  ],
```

```
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "CA",
      "ST": "San Francisco"
    }
  ]
}
```

when peer B's actual IP address is `10.138.0.2`, not `10.138.0.27`. When peer B tries to join the cluster, peer A will reject B with the error `x509: certificate is valid for 10.138.0.27, not 10.138.0.2`, because B's remote IP address does not match the one in Subject Alternative Name (SAN) field.

Since [v3.2.0](), [server resolves TLS `DNSNames` when checking `SAN`](). For instance, if peer cert contains only DNS names (no IP addresses) in Subject Alternative Name (SAN) field, server authenticates a peer only when forward-lookups (`dig b.com`) on those DNS names have matching IP with the remote IP address. For example, peer B's CSR (with `cfssl`) is:

```
{
  "CN": "etcd peer",
  "hosts": [
    "b.com"
  ],
```

when peer B's remote IP address is `10.138.0.2`. When peer B tries to join the cluster, peer A looks up the incoming host `b.com` to get the list of IP addresses (e.g. `dig b.com`). And rejects B if the list does not contain the IP `10.138.0.2`, with the error `tls: 10.138.0.2 does not match any of DNSNames ["b.com"]`.

Since [v3.2.2](), [server accepts connections if IP matches, without checking DNS entries](). For instance, if peer cert contains IP addresses and DNS names in Subject Alternative Name (SAN) field, and the remote IP address matches one of those IP addresses, server just accepts connection without further checking the DNS names. For example, peer B's CSR (with `cfssl`) is:

```
{
  "CN": "etcd peer",
```

```
  "hosts": [
    "invalid.domain",
    "10.138.0.2"
  ],
```

when peer B's remote IP address is `10.138.0.2` and `invalid.domain` is a invalid host. When peer B tries to join the cluster, peer A successfully authenticates B, since Subject Alternative Name (SAN) field has a valid matching IP address. See issue#8206 for more detail.

Since v3.2.5, server supports reverse-lookup on wildcard DNS `SAN`. For instance, if peer cert contains only DNS names (no IP addresses) in Subject Alternative Name (SAN) field, server first reverse-lookups the remote IP address to get a list of names mapping to that address (e.g. `nslookup IPADDR`). Then accepts the connection if those names have a matching name with peer cert's DNS names (either by exact or wildcard match). If none is matched, server forward-lookups each DNS entry in peer cert (e.g. look up `example.default.svc` when the entry is `*.example.default.svc`), and accepts connection only when the host's resolved addresses have the matching IP address with the peer's remote IP address. For example, peer B's CSR (with `cfssl`) is:

```
{
  "CN": "etcd peer",
  "hosts": [
    "*.example.default.svc",
    "*.example.default.svc.cluster.local"
  ],
```

when peer B's remote IP address is `10.138.0.2`. When peer B tries to join the cluster, peer A reverse-lookup the IP `10.138.0.2` to get the list of host names. And either exact or wildcard match the host names with peer B's cert DNS names in Subject Alternative Name (SAN) field. If none of reverse/forward lookups worked, it returns an error `"tls: "10.138.0.2" does not match any of DNSNames ["*.example.default.svc","*.example.default.svc.cluster.local"]`. See issue#8268 for more detail.

v3.3.0 adds `etcd --peer-cert-allowed-cn` flag to support CN(Common Name)-based auth for inter-peer connections. Kubernetes TLS bootstrapping involves generating dynamic certificates for etcd members and other system components (e.g. API server, kubelet, etc.). Maintaining different CAs for each component provides tighter access control to etcd cluster but often tedious. When `--peer-cert-allowed-cn` flag is specified, node can only join with matching common name even with shared CAs. For example, each member in 3-node cluster is set up with CSRs (with `cfssl`) as below:

```
{
  "CN": "etcd.local",
  "hosts": [
    "m1.etcd.local",
    "127.0.0.1",
    "localhost"
  ],
```

```
{
  "CN": "etcd.local",
  "hosts": [
    "m2.etcd.local",
    "127.0.0.1",
    "localhost"
  ],
```

```
{
  "CN": "etcd.local",
  "hosts": [
    "m3.etcd.local",
    "127.0.0.1",
    "localhost"
  ],
```

Then only peers with matching common names will be authenticated if `--peer-cert-allowed-cn etcd.local` is given. And nodes with different CNs in CSRs or different `--peer-cert-allowed-cn` will be rejected:

```
$ etcd --peer-cert-allowed-cn m1.etcd.local

I | embed: rejected connection from "127.0.0.1:48044" (error "CommonName authentication
I | embed: rejected connection from "127.0.0.1:55702" (error "remote error: tls: bad cer
```

Each process should be started with:

```
etcd --peer-cert-allowed-cn etcd.local

I | pkg/netutil: resolving m3.etcd.local:32380 to 127.0.0.1:32380
I | pkg/netutil: resolving m2.etcd.local:22380 to 127.0.0.1:22380
```

```
I | pkg/netutil: resolving m1.etcd.local:2380 to 127.0.0.1:2380
I | etcdserver: published {Name:m3 ClientURLs:[https://m3.etcd.local:32379]} to cluster 9
I | embed: ready to serve client requests
I | etcdserver: published {Name:m1 ClientURLs:[https://m1.etcd.local:2379]} to cluster 9
I | embed: ready to serve client requests
I | etcdserver: published {Name:m2 ClientURLs:[https://m2.etcd.local:22379]} to cluster 9
I | embed: ready to serve client requests
I | embed: serving client requests on 127.0.0.1:32379
I | embed: serving client requests on 127.0.0.1:22379
I | embed: serving client requests on 127.0.0.1:2379
```

v3.2.19 and v3.3.4 fixes TLS reload when certificate SAN field only includes IP addresses but no domain names. For example, a member is set up with CSRs (with `cfssl`) as below:

```
{
  "CN": "etcd.local",
  "hosts": [
    "127.0.0.1"
  ],
```

In Go, server calls `(*tls.Config).GetCertificate` for TLS reload if and only if server's `(*tls.Config).Certificates` field is not empty, or `(*tls.ClientHelloInfo).ServerName` is not empty with a valid SNI from the client. Previously, etcd always populates `(*tls.Config).Certificates` on the initial client TLS handshake, as non-empty. Thus, client was always expected to supply a matching SNI in order to pass the TLS verification and to trigger `(*tls.Config).GetCertificate` to reload TLS assets.

However, a certificate whose SAN field does not include any domain names but only IP addresses would request `*tls.ClientHelloInfo` with an empty `ServerName` field, thus failing to trigger the TLS reload on initial TLS handshake; this becomes a problem when expired certificates need to be replaced online.

Now, `(*tls.Config).Certificates` is created empty on initial TLS client handshake, first to trigger `(*tls.Config).GetCertificate`, and then to populate rest of the certificates on every new TLS connection, even when client SNI is empty (e.g. cert only includes IPs).

Notes for Host Whitelist

`etcd --host-whitelist` flag specifies acceptable hostnames from HTTP client requests. Client origin policy protects against "DNS Rebinding" attacks to insecure etcd servers. That is, any website can simply create an authorized DNS name, and direct DNS to `"localhost"` (or any other address). Then, all HTTP endpoints of etcd server listening on `"localhost"` becomes accessible, thus vulnerable to DNS rebinding attacks. See CVE-2018-5702 for more detail.

Client origin policy works as follows:

1. If client connection is secure via HTTPS, allow any hostnames.
2. If client connection is not secure and `"HostWhitelist"` is not empty, only allow HTTP requests whose Host field is listed in whitelist.

Note that the client origin policy is enforced whether authentication is enabled or not, for tighter controls.

By default, `etcd --host-whitelist` and `embed.Config.HostWhitelist` are set *empty* to allow all hostnames. Note that when specifying hostnames, loopback addresses are not added automatically. To allow loopback interfaces, add them to whitelist manually (e.g. `"localhost"`, `"127.0.0.1"`, etc.).

Frequently asked questions

I'm seeing a SSLv3 alert handshake failure when using TLS client authentication?

The `crypto/tls` package of `golang` checks the key usage of the certificate public key before using it. To use the certificate public key to do client auth, we need to add `clientAuth` to `Extended Key Usage` when creating the certificate public key.

Here is how to do it:

Add the following section to openssl.cnf:

```
[ ssl_client ]
...
  extendedKeyUsage = clientAuth
...
```

When creating the cert be sure to reference it in the `-extensions` flag:

```
$ openssl ca -config openssl.cnf -policy policy_anything -extensions ssl_client -out cert
```

With peer certificate authentication I receive "certificate is valid for 127.0.0.1, not $MY_IP"

Make sure to sign the certificates with a Subject Name the member's public IP address. The `etcd-ca` tool for example provides an `--ip=` option for its `new-cert` command.

The certificate needs to be signed for the member's FQDN in its Subject Name, use Subject Alternative Names (short IP SANs) to add the IP address. The `etcd-ca` tool provides `--domain=` option for its `new-cert` command, and openssl can make it too.

Last modified September 12, 2023: Fix paths to client certificates in curl examples (596412f)

Clustering Guide

Bootstrapping an etcd cluster: Static, etcd Discovery, and DNS Discovery

Overview

Starting an etcd cluster statically requires that each member knows another in the cluster. In a number of cases, the IPs of the cluster members may be unknown ahead of time. In these cases, the etcd cluster can be bootstrapped with the help of a discovery service.

Once an etcd cluster is up and running, adding or removing members is done via runtime reconfiguration. To better understand the design behind runtime reconfiguration, we suggest reading the runtime configuration design document.

This guide will cover the following mechanisms for bootstrapping an etcd cluster:

- Static
- etcd Discovery
- DNS Discovery

Each of the bootstrapping mechanisms will be used to create a three machine etcd cluster with the following details:

Name	Address	Hostname
infra0	10.0.1.10	infra0.example.com
infra1	10.0.1.11	infra1.example.com
infra2	10.0.1.12	infra2.example.com

Static

As we know the cluster members, their addresses and the size of the cluster before starting, we can use an offline bootstrap configuration by setting the `initial-cluster` flag. Each machine will get either the following environment variables or command line:

```
ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=ht
ETCD_INITIAL_CLUSTER_STATE=new
```

```
--initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http:/
--initial-cluster-state new
```

Note that the URLs specified in `initial-cluster` are the *advertised peer URLs*, i.e. they should match the value of `initial-advertise-peer-urls` on the respective nodes.

If spinning up multiple clusters (or creating and destroying a single cluster) with same configuration for testing purpose, it is highly recommended that each cluster is given a unique `initial-cluster-token`. By doing this, etcd can generate unique cluster IDs and member IDs for the clusters even if they otherwise have the exact same configuration. This can protect etcd from cross-cluster-interaction, which might corrupt the clusters.

etcd listens on [`listen-client-urls`] to accept client traffic. etcd member advertises the URLs specified in [`advertise-client-urls`] to other members, proxies, clients. Please make sure the `advertise-client-urls` are reachable from intended clients. A common mistake is setting `advertise-client-urls` to localhost or leave it as default if the remote clients should reach etcd.

On each machine, start etcd with these flags:

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http
  --initial-cluster-state new
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
  --listen-peer-urls http://10.0.1.11:2380 \
  --listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.11:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http
  --initial-cluster-state new
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \
  --listen-peer-urls http://10.0.1.12:2380 \
  --listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \
```

```
  --advertise-client-urls http://10.0.1.12:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http
  --initial-cluster-state new
```

The command line parameters starting with `--initial-cluster` will be ignored on subsequent runs of etcd. Feel free to remove the environment variables or command line flags after the initial bootstrap process. If the configuration needs changes later (for example, adding or removing members to/from the cluster), see the runtime configuration guide.

TLS

etcd supports encrypted communication through the TLS protocol. TLS channels can be used for encrypted internal cluster communication between peers as well as encrypted client traffic. This section provides examples for setting up a cluster with peer and client TLS. Additional information detailing etcd's TLS support can be found in the security guide.

Self-signed certificates

A cluster using self-signed certificates both encrypts traffic and authenticates its connections. To start a cluster with self-signed certificates, each cluster member should have a unique key pair (`member.crt`, `member.key`) signed by a shared cluster CA certificate (`ca.crt`) for both peer connections and client connections. Certificates may be generated by following the etcd TLS setup example.

On each machine, etcd would be started with these flags:

```
$ etcd --name infra0 --initial-advertise-peer-urls https://10.0.1.10:2380 \
  --listen-peer-urls https://10.0.1.10:2380 \
  --listen-client-urls https://10.0.1.10:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.10:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
  --client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
  --cert-file=/path/to/infra0-client.crt --key-file=/path/to/infra0-client.key \
  --peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
  --peer-cert-file=/path/to/infra0-peer.crt --peer-key-file=/path/to/infra0-peer.key
```

```
$ etcd --name infra1 --initial-advertise-peer-urls https://10.0.1.11:2380 \
  --listen-peer-urls https://10.0.1.11:2380 \
  --listen-client-urls https://10.0.1.11:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.11:2379 \
  --initial-cluster-token etcd-cluster-1 \
```

```
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
  --client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
  --cert-file=/path/to/infra1-client.crt --key-file=/path/to/infra1-client.key \
  --peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
  --peer-cert-file=/path/to/infra1-peer.crt --peer-key-file=/path/to/infra1-peer.key
```

```
$ etcd --name infra2 --initial-advertise-peer-urls https://10.0.1.12:2380 \
  --listen-peer-urls https://10.0.1.12:2380 \
  --listen-client-urls https://10.0.1.12:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.12:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
  --client-cert-auth --trusted-ca-file=/path/to/ca-client.crt \
  --cert-file=/path/to/infra2-client.crt --key-file=/path/to/infra2-client.key \
  --peer-client-cert-auth --peer-trusted-ca-file=ca-peer.crt \
  --peer-cert-file=/path/to/infra2-peer.crt --peer-key-file=/path/to/infra2-peer.key
```

Automatic certificates

If the cluster needs encrypted communication but does not require authenticated connections, etcd can be configured to automatically generate its keys. On initialization, each member creates its own set of keys based on its advertised IP addresses and hosts.

On each machine, etcd would be started with these flags:

```
$ etcd --name infra0 --initial-advertise-peer-urls https://10.0.1.10:2380 \
  --listen-peer-urls https://10.0.1.10:2380 \
  --listen-client-urls https://10.0.1.10:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.10:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
  --auto-tls \
  --peer-auto-tls
```

```
$ etcd --name infra1 --initial-advertise-peer-urls https://10.0.1.11:2380 \
  --listen-peer-urls https://10.0.1.11:2380 \
  --listen-client-urls https://10.0.1.11:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.11:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
```

```
  --auto-tls \
  --peer-auto-tls
```

```
$ etcd --name infra2 --initial-advertise-peer-urls https://10.0.1.12:2380 \
  --listen-peer-urls https://10.0.1.12:2380 \
  --listen-client-urls https://10.0.1.12:2379,https://127.0.0.1:2379 \
  --advertise-client-urls https://10.0.1.12:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=https://10.0.1.10:2380,infra1=https://10.0.1.11:2380,infra2=ht
  --initial-cluster-state new \
  --auto-tls \
  --peer-auto-tls
```

Error cases

In the following example, we have not included our new host in the list of enumerated nodes. If this is a new cluster, the node *must* be added to the list of initial cluster members.

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
  --listen-peer-urls https://10.0.1.11:2380 \
  --listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.11:2379 \
  --initial-cluster infra0=http://10.0.1.10:2380 \
  --initial-cluster-state new
etcd: infra1 not listed in the initial cluster config
exit 1
```

In this example, we are attempting to map a node (infra0) on a different address (127.0.0.1:2380) than its enumerated address in the cluster list (10.0.1.10:2380). If this node is to listen on multiple addresses, all addresses *must* be reflected in the "initial-cluster" configuration directive.

```
$ etcd --name infra0 --initial-advertise-peer-urls http://127.0.0.1:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=http
  --initial-cluster-state=new
etcd: error setting up initial cluster: infra0 has different advertised URLs in the clus
exit 1
```

If a peer is configured with a different set of configuration arguments and attempts to join this cluster, etcd will report a cluster ID mismatch will exit.

```
$ etcd --name infra3 --initial-advertise-peer-urls http://10.0.1.13:2380 \
  --listen-peer-urls http://10.0.1.13:2380 \
  --listen-client-urls http://10.0.1.13:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.13:2379 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra3=http
  --initial-cluster-state=new
etcd: conflicting cluster ID to the target cluster (c6ab534d07e8fcc4 != bc25ea2a74fb18b0
exit 1
```

Discovery

In a number of cases, the IPs of the cluster peers may not be known ahead of time. This is common when utilizing cloud providers or when the network uses DHCP. In these cases, rather than specifying a static configuration, use an existing etcd cluster to bootstrap a new one. This process is called "discovery".

There two methods that can be used for discovery:

- etcd discovery service
- DNS SRV records

etcd discovery

To better understand the design of the discovery service protocol, we suggest reading the discovery service protocol documentation.

Lifetime of a discovery URL

A discovery URL identifies a unique etcd cluster. Instead of reusing an existing discovery URL, each etcd instance shares a new discovery URL to bootstrap the new cluster.

Moreover, discovery URLs should ONLY be used for the initial bootstrapping of a cluster. To change cluster membership after the cluster is already running, see the runtime reconfiguration guide.

Custom etcd discovery service

Discovery uses an existing cluster to bootstrap itself. If using a private etcd cluster, create a URL like so:

```
$ curl -X PUT https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f
```

By setting the size key to the URL, a discovery URL is created with an expected cluster size of 3.

The URL to use in this case will be `https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83` and the etcd members will use the `https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83` directory for registration as they start.

Each member must have a different name flag specified. `Hostname` **or** `machine-id` **can be a good choice. Or discovery will fail due to duplicated name.**

Now we start etcd with those relevant flags for each member:

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
  --listen-peer-urls http://10.0.1.11:2380 \
  --listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.11:2379 \
  --discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \
  --listen-peer-urls http://10.0.1.12:2380 \
  --listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.12:2379 \
  --discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f
```

This will cause each member to register itself with the custom etcd discovery service and begin the cluster once all machines have been registered.

Public etcd discovery service

If no existing cluster is available, use the public discovery service hosted at `discovery.etcd.io`. To create a private discovery URL using the "new" endpoint, use the command:

```
$ curl https://discovery.etcd.io/new?size=3
```

```
https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

This will create the cluster with an initial size of 3 members. If no size is specified, a default of 3 is used.

```
ETCD_DISCOVERY=https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
--discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

Each member must have a different name flag specified or else discovery will fail due to duplicated names. `Hostname` or `machine-id` can be a good choice.

Now we start etcd with those relevant flags for each member:

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
$ etcd --name infra1 --initial-advertise-peer-urls http://10.0.1.11:2380 \
  --listen-peer-urls http://10.0.1.11:2380 \
  --listen-client-urls http://10.0.1.11:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.11:2379 \
  --discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

```
$ etcd --name infra2 --initial-advertise-peer-urls http://10.0.1.12:2380 \
  --listen-peer-urls http://10.0.1.12:2380 \
  --listen-client-urls http://10.0.1.12:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.12:2379 \
  --discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
```

This will cause each member to register itself with the discovery service and begin the cluster once all members have been registered.

Use the environment variable `ETCD_DISCOVERY_PROXY` to cause etcd to use an HTTP proxy to connect to the discovery service.

Error and warning cases

Discovery server errors

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
etcd: error: the cluster doesn’t have a size configuration value in https://discovery.etc
exit 1
```

Warnings

This is a harmless warning indicating the discovery URL will be ignored on this machine.

```
$ etcd --name infra0 --initial-advertise-peer-urls http://10.0.1.10:2380 \
  --listen-peer-urls http://10.0.1.10:2380 \
  --listen-client-urls http://10.0.1.10:2379,http://127.0.0.1:2379 \
  --advertise-client-urls http://10.0.1.10:2379 \
  --discovery https://discovery.etcd.io/3e86b59982e49066c5d813af1c2e2579cbf573de
etcdserver: discovery token ignored since a cluster has already been initialized. Valid l
```

DNS discovery

DNS SRV records can be used as a discovery mechanism. The `--discovery-srv` flag can be used to set the DNS domain name where the discovery SRV records can be found. Setting `--discovery-srv example.com` causes DNS SRV records to be looked up in the listed order:

- _etcd-server-ssl._tcp.example.com
- _etcd-server._tcp.example.com

If `_etcd-server-ssl._tcp.example.com` is found then etcd will attempt the bootstrapping process over TLS.

To help clients discover the etcd cluster, the following DNS SRV records are looked up in the listed order:

- _etcd-client._tcp.example.com
- _etcd-client-ssl._tcp.example.com

If `_etcd-client-ssl._tcp.example.com` is found, clients will attempt to communicate with the etcd cluster over SSL/TLS.

If etcd is using TLS, the discovery SRV record (e.g. `example.com`) must be included in the SSL certificate DNS SAN along with the hostname, or clustering will fail with log messages like the following:

```
[...] rejected connection from "10.0.1.11:53162" (error "remote error: tls: bad certifica
```

If etcd is using TLS without a custom certificate authority, the discovery domain (e.g., example.com) must match the SRV record domain (e.g., infra1.example.com). This is to mitigate attacks that forge SRV records to point to a different domain; the domain would have a valid certificate under PKI but be controlled by an unknown third party.

The `-discovery-srv-name` flag additionally configures a suffix to the SRV name that is queried during discovery. Use this flag to differentiate between multiple etcd clusters under the same domain. For example, if `discovery-srv=example.com` and `-discovery-srv-name=foo` are set, the following DNS SRV queries are made:

- _etcd-server-ssl-foo._tcp.example.com
- _etcd-server-foo._tcp.example.com

Create DNS SRV records

```
$ dig +noall +answer SRV _etcd-server._tcp.example.com
_etcd-server._tcp.example.com. 300 IN  SRV  0 0 2380 infra0.example.com.
_etcd-server._tcp.example.com. 300 IN  SRV  0 0 2380 infra1.example.com.
_etcd-server._tcp.example.com. 300 IN  SRV  0 0 2380 infra2.example.com.
```

```
$ dig +noall +answer SRV _etcd-client._tcp.example.com
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra0.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra1.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra2.example.com.
```

```
$ dig +noall +answer infra0.example.com infra1.example.com infra2.example.com
infra0.example.com.  300  IN  A  10.0.1.10
infra1.example.com.  300  IN  A  10.0.1.11
infra2.example.com.  300  IN  A  10.0.1.12
```

Bootstrap the etcd cluster using DNS

etcd cluster members can advertise domain names or IP address, the bootstrap process will resolve DNS A records. Since 3.2 (3.1 prints warnings) `--listen-peer-urls` and `--listen-client-urls` will reject domain name for the network interface binding.

The resolved address in `--initial-advertise-peer-urls` *must match* one of the resolved addresses in the SRV targets. The etcd member reads the resolved address to find out if it belongs to the cluster defined in the SRV records.

```
$ etcd --name infra0 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra0.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra0.example.com:2379 \
--listen-client-urls http://0.0.0.0:2379 \
--listen-peer-urls http://0.0.0.0:2380
```

```
$ etcd --name infra1 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra1.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra1.example.com:2379 \
--listen-client-urls http://0.0.0.0:2379 \
--listen-peer-urls http://0.0.0.0:2380
```

```
$ etcd --name infra2 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://infra2.example.com:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://infra2.example.com:2379 \
--listen-client-urls http://0.0.0.0:2379 \
--listen-peer-urls http://0.0.0.0:2380
```

The cluster can also bootstrap using IP addresses instead of domain names:

```
$ etcd --name infra0 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.10:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.10:2379 \
--listen-client-urls http://10.0.1.10:2379 \
--listen-peer-urls http://10.0.1.10:2380
```

```
$ etcd --name infra1 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.11:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.11:2379 \
--listen-client-urls http://10.0.1.11:2379 \
--listen-peer-urls http://10.0.1.11:2380
```

```
$ etcd --name infra2 \
--discovery-srv example.com \
--initial-advertise-peer-urls http://10.0.1.12:2380 \
--initial-cluster-token etcd-cluster-1 \
--initial-cluster-state new \
--advertise-client-urls http://10.0.1.12:2379 \
--listen-client-urls http://10.0.1.12:2379 \
--listen-peer-urls http://10.0.1.12:2380
```

Since v3.1.0 (except v3.2.9), when `etcd --discovery-srv=example.com` is configured with TLS, server will only authenticate peers/clients when the provided certs have root domain `example.com` as an entry in Subject Alternative Name (SAN) field. See Notes for DNS SRV.

Gateway

etcd gateway is a simple TCP proxy that forwards network data to the etcd cluster. Please read gateway guide for more information.

Proxy

When the `--proxy` flag is set, etcd runs in proxy mode. This proxy mode only supports the etcd v2 API; there are no plans to support the v3 API. Instead, for v3 API support, there will be a new proxy with enhanced features following the etcd 3.0 release.

To setup an etcd cluster with proxies of v2 API, please read the the clustering doc in etcd 2.3 release.

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

Run etcd clusters inside containers

Running etcd with rkt and Docker using static bootstrapping

The following guide shows how to run etcd with rkt and Docker using the static bootstrap process.

rkt

Running a single node etcd

The following rkt run command will expose the etcd client API on port 2379 and expose the peer API on port 2380.

Use the host IP address when configuring etcd.

```
export NODE1=192.168.1.21
```

Trust the CoreOS App Signing Key.

```
sudo rkt trust --prefix quay.io/coreos/etcd
# gpg key fingerprint is: 18AD 5014 C99E F7E3 BA5F  6CE9 50BD D3E0 FC8A 365E
```

Run the `v3.2` version of etcd or specify another release version.

```
sudo rkt run --net=default:IP=${NODE1} quay.io/coreos/etcd:v3.2 -- -name=node1 -advertise
```

List the cluster member.

```
etcdctl --endpoints=http://192.168.1.21:2379 member list
```

Running a 3 node etcd cluster

Setup a 3 node cluster with rkt locally, using the `-initial-cluster` flag.

```
export NODE1=172.16.28.21
export NODE2=172.16.28.22
export NODE3=172.16.28.23
```

```
# node 1
sudo rkt run --net=default:IP=${NODE1} quay.io/coreos/etcd:v3.2 -- -name=node1 -advertise

# node 2
sudo rkt run --net=default:IP=${NODE2} quay.io/coreos/etcd:v3.2 -- -name=node2 -advertise

# node 3
sudo rkt run --net=default:IP=${NODE3} quay.io/coreos/etcd:v3.2 -- -name=node3 -advertise
```

Verify the cluster is healthy and can be reached.

```
ETCDCTL_API=3 etcdctl --endpoints=http://172.16.28.21:2379,http://172.16.28.22:2379,http
```

DNS

Production clusters which refer to peers by DNS name known to the local resolver must mount the host's DNS configuration.

Docker

In order to expose the etcd API to clients outside of Docker host, use the host IP address of the container. Please see `docker inspect` for more detail on how to get the IP address. Alternatively, specify `--net=host` flag to `docker run` command to skip placing the container inside of a separate network stack.

Running a single node etcd

Use the host IP address when configuring etcd:

```
export NODE1=192.168.1.21
```

Configure a Docker volume to store etcd data:

```
docker volume create --name etcd-data
```

```
export DATA_DIR="etcd-data"
```

Run the latest version of etcd:

```
REGISTRY=quay.io/coreos/etcd
# available from v3.2.5
REGISTRY=gcr.io/etcd-development/etcd

docker run \
  -p 2379:2379 \
  -p 2380:2380 \
  --volume=${DATA_DIR}:/etcd-data \
  --name etcd ${REGISTRY}:latest \
  /usr/local/bin/etcd \
  --data-dir=/etcd-data --name node1 \
  --initial-advertise-peer-urls http://${NODE1}:2380 --listen-peer-urls http://0.0.0.0:23
  --advertise-client-urls http://${NODE1}:2379 --listen-client-urls http://0.0.0.0:2379 \
  --initial-cluster node1=http://${NODE1}:2380
```

List the cluster member:

```
etcdctl --endpoints=http://${NODE1}:2379 member list
```

Running a 3 node etcd cluster

```
REGISTRY=quay.io/coreos/etcd
# available from v3.2.5
REGISTRY=gcr.io/etcd-development/etcd

# For each machine
ETCD_VERSION=latest
TOKEN=my-etcd-token
CLUSTER_STATE=new
NAME_1=etcd-node-0
NAME_2=etcd-node-1
NAME_3=etcd-node-2
HOST_1=10.20.30.1
HOST_2=10.20.30.2
HOST_3=10.20.30.3
CLUSTER=${NAME_1}=http://${HOST_1}:2380,${NAME_2}=http://${HOST_2}:2380,${NAME_3}=http:/
DATA_DIR=/var/lib/etcd

# For node 1
THIS_NAME=${NAME_1}
THIS_IP=${HOST_1}
```

```
docker run \
  -p 2379:2379 \
  -p 2380:2380 \
  --volume=${DATA_DIR}:/etcd-data \
  --name etcd ${REGISTRY}:${ETCD_VERSION} \
  /usr/local/bin/etcd \
  --data-dir=/etcd-data --name ${THIS_NAME} \
  --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://0.0.0.0
  --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://0.0.0.0:2379
  --initial-cluster ${CLUSTER} \
  --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For node 2
THIS_NAME=${NAME_2}
THIS_IP=${HOST_2}
docker run \
  -p 2379:2379 \
  -p 2380:2380 \
  --volume=${DATA_DIR}:/etcd-data \
  --name etcd ${REGISTRY}:${ETCD_VERSION} \
  /usr/local/bin/etcd \
  --data-dir=/etcd-data --name ${THIS_NAME} \
  --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://0.0.0.0
  --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://0.0.0.0:2379
  --initial-cluster ${CLUSTER} \
  --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}

# For node 3
THIS_NAME=${NAME_3}
THIS_IP=${HOST_3}
docker run \
  -p 2379:2379 \
  -p 2380:2380 \
  --volume=${DATA_DIR}:/etcd-data \
  --name etcd ${REGISTRY}:${ETCD_VERSION} \
  /usr/local/bin/etcd \
  --data-dir=/etcd-data --name ${THIS_NAME} \
  --initial-advertise-peer-urls http://${THIS_IP}:2380 --listen-peer-urls http://0.0.0.0
  --advertise-client-urls http://${THIS_IP}:2379 --listen-client-urls http://0.0.0.0:2379
  --initial-cluster ${CLUSTER} \
  --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

To run `etcdctl` using API version 3:

```
docker exec etcd /usr/local/bin/etcdctl put foo bar
```

Bare Metal

To provision a 3 node etcd cluster on bare-metal, the examples in the baremetal repo may be useful.

Mounting a certificate volume

The etcd release container does not include default root certificates. To use HTTPS with certificates trusted by a root authority (e.g., for discovery), mount a certificate directory into the etcd container:

```
REGISTRY=quay.io/coreos/etcd
# available from v3.2.5
REGISTRY=docker://gcr.io/etcd-development/etcd

rkt run \
  --insecure-options=image \
  --volume etcd-ssl-certs-bundle,kind=host,source=/etc/ssl/certs/ca-certificates.crt \
  --mount volume=etcd-ssl-certs-bundle,target=/etc/ssl/certs/ca-certificates.crt \
  ${REGISTRY}:latest -- --name my-name \
  --initial-advertise-peer-urls http://localhost:2380 --listen-peer-urls http://localhos
  --advertise-client-urls http://localhost:2379 --listen-client-urls http://localhost:23
  --discovery https://discovery.etcd.io/c11fbcdc16972e45253491a24fcf45e1
```

```
REGISTRY=quay.io/coreos/etcd
# available from v3.2.5
REGISTRY=gcr.io/etcd-development/etcd

docker run \
  -p 2379:2379 \
  -p 2380:2380 \
  --volume=/etc/ssl/certs/ca-certificates.crt:/etc/ssl/certs/ca-certificates.crt \
  ${REGISTRY}:latest \
  /usr/local/bin/etcd --name my-name \
  --initial-advertise-peer-urls http://localhost:2380 --listen-peer-urls http://localhos
  --advertise-client-urls http://localhost:2379 --listen-client-urls http://localhost:23
  --discovery https://discovery.etcd.io/86a9ff6c8cb8b4c4544c1a2f88f8b801
```

Last modified January 24, 2023: removed /bin/sh (1be83b2)

Failure modes

Kinds of failures and etcd's tolerance for them

Failures are common in a large deployment of machines. A machine fails when its hardware or software malfunctions. Multiple machines fail together when there are power failures or network issues. Multiple kinds of failures can also happen at once; it is almost impossible to enumerate all possible failure cases.

In this section, we catalog kinds of failures and discuss how etcd is designed to tolerate these failures. Most users, if not all, can map a particular failure into one kind of failure. To prepare for rare or unrecoverable failures, always back up the etcd cluster.

Minor followers failure

When fewer than half of the followers fail, the etcd cluster can still accept requests and make progress without any major disruption. For example, two follower failures will not affect a five member etcd cluster's operation. However, clients will lose connectivity to the failed members. Client libraries should hide these interruptions from users for read requests by automatically reconnecting to other members. Operators should expect the system load on the other members to increase due to the reconnections.

Leader failure

When a leader fails, the etcd cluster automatically elects a new leader. The election does not happen instantly once the leader fails. It takes about an election timeout to elect a new leader since the failure detection model is timeout based.

During the leader election the cluster cannot process any writes. Write requests sent during the election are queued for processing until a new leader is elected.

Writes already sent to the old leader but not yet committed may be lost. The new leader has the power to rewrite any uncommitted entries from the previous leader. From the user perspective, some write requests might time out after a new leader election. However, no committed writes are ever lost.

The new leader extends timeouts automatically for all leases. This mechanism ensures a lease will not expire before the granted TTL even if it was granted by the old leader.

Majority failure

When the majority members of the cluster fail, the etcd cluster fails and cannot accept more writes.

The etcd cluster can only recover from a majority failure once the majority of members become available. If a majority of members cannot come back online, then the operator must start disaster recovery to recover the cluster.

Once a majority of members works, the etcd cluster elects a new leader automatically and returns to a healthy state. The new leader extends timeouts automatically for all leases. This mechanism ensures no lease expires due to server side unavailability.

Network partition

A network partition is similar to a minor followers failure or a leader failure. A network partition divides the etcd cluster into two parts; one with a member majority and the other with a member minority. The majority side becomes the available cluster and the minority side is unavailable; there is no “split-brain” in etcd.

If the leader is on the majority side, then from the majority point of view the failure is a minority follower failure. If the leader is on the minority side, then it is a leader failure. The leader on the minority side steps down and the majority side elects a new leader.

Once the network partition clears, the minority side automatically recognizes the leader from the majority side and recovers its state.

Failure during bootstrapping

A cluster bootstrap is only successful if all required members successfully start. If any failure happens during bootstrapping, remove the data directories on all members and re-bootstrap the cluster with a new cluster-token or new discovery token.

Of course, it is possible to recover a failed bootstrapped cluster like recovering a running cluster. However, it almost always takes more time and resources to recover that cluster than bootstrapping a new one, since there is no data to recover.

Last modified February 20, 2023: [Fix minor spelling error in upgrade docs. (6aa3ed0)]

Disaster recovery

etcd v3 snapshot & restore facilities

etcd is designed to withstand machine failures. An etcd cluster automatically recovers from temporary failures (e.g., machine reboots) and tolerates up to *(N-1)/2* permanent failures for a cluster of N members. When a member permanently fails, whether due to hardware failure or disk corruption, it loses access to the cluster. If the cluster permanently loses more than *(N-1)/2* members then it disastrously fails, irrevocably losing quorum. Once quorum is lost, the cluster cannot reach consensus and therefore cannot continue accepting updates.

To recover from disastrous failure, etcd v3 provides snapshot and restore facilities to recreate the cluster without v3 key data loss. To recover v2 keys, refer to the v2 admin guide.

Snapshotting the keyspace

Recovering a cluster first needs a snapshot of the keyspace from an etcd member. A snapshot may either be taken from a live member with the `etcdctl snapshot save` command or by copying the `member/snap/db` file from an etcd data directory. For example, the following command snapshots the keyspace served by `$ENDPOINT` to the file `snapshot.db`:

```
$ ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshot.db
```

Restoring a cluster

To restore a cluster, all that is needed is a single snapshot "db" file. A cluster restore with `etcdctl snapshot restore` creates new etcd data directories; all members should restore using the same snapshot. Restoring overwrites some snapshot metadata (specifically, the member ID and cluster ID); the member loses its former identity. This metadata overwrite prevents the new member from inadvertently joining an existing cluster. Therefore in order to start a cluster from a snapshot, the restore must start a new logical cluster.

Snapshot integrity may be optionally verified at restore time. If the snapshot is taken with `etcdctl snapshot save`, it will have an integrity hash that is checked by `etcdctl snapshot` `restore`. If the snapshot is copied from the data directory, there is no integrity hash and it will only restore by using `--skip-hash-check`.

A restore initializes a new member of a new cluster, with a fresh cluster configuration using `etcd`'s cluster configuration flags, but preserves the contents of the etcd keyspace. Continuing from the previous example, the following creates new etcd data directories (`m1.etcd`, `m2.etcd`, `m3.etcd`) for a three member cluster:

```
$ ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
  --name m1 \
  --initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://host1:2380
$ ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
  --name m2 \
  --initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://host2:2380
$ ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
  --name m3 \
  --initial-cluster m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://host3:2380
```

Next, start `etcd` with the new data directories:

```
$ etcd \
  --name m1 \
  --listen-client-urls http://host1:2379 \
  --advertise-client-urls http://host1:2379 \
  --listen-peer-urls http://host1:2380 &
$ etcd \
  --name m2 \
  --listen-client-urls http://host2:2379 \
  --advertise-client-urls http://host2:2379 \
  --listen-peer-urls http://host2:2380 &
$ etcd \
  --name m3 \
  --listen-client-urls http://host3:2379 \
  --advertise-client-urls http://host3:2379 \
  --listen-peer-urls http://host3:2380 &
```

Now the restored etcd cluster should be available and serving the keyspace given by the snapshot.

Restoring a cluster from membership mis-reconfiguration with wrong URLs

Previously, etcd panics on membership mis-reconfiguration with wrong URLs (v3.2.15 or later returns error early in client-side before etcd server panic).

Recommended way is restore from snapshot. `--force-new-cluster` can be used to overwrite cluster membership while keeping existing application data, but is strongly discouraged because it will panic if other members from previous cluster are still alive. Make sure to save snapshot periodically.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

etcd gateway

etcd gateway, when to use it, and how to set it up

What is etcd gateway

etcd gateway is a simple TCP proxy that forwards network data to the etcd cluster. The gateway is stateless and transparent; it neither inspects client requests nor interferes with cluster responses.

The gateway supports multiple etcd server endpoints and works on a simple round-robin policy. It only routes to available endpoints and hides failures from its clients. Other retry policies, such as weighted round-robin, may be supported in the future.

When to use etcd gateway

Every application that accesses etcd must first have the address of an etcd cluster client endpoint. If multiple applications on the same server access the same etcd cluster, every application still needs to know the advertised client endpoints of the etcd cluster. If the etcd cluster is reconfigured to have different endpoints, every application may also need to update its endpoint list. This wide-scale reconfiguration is both tedious and error prone.

etcd gateway solves this problem by serving as a stable local endpoint. A typical etcd gateway configuration has each machine running a gateway listening on a local address and every etcd application connecting to its local gateway. The upshot is only the gateway needs to update its endpoints instead of updating each and every application.

In summary, to automatically propagate cluster endpoint changes, the etcd gateway runs on every machine serving multiple applications accessing the same etcd cluster.

When not to use etcd gateway

- Improving performance

The gateway is not designed for improving etcd cluster performance. It does not provide caching, watch coalescing or batching. The etcd team is developing a caching proxy designed for improving cluster scalability.

- Running on a cluster management system

Advanced cluster management systems like Kubernetes natively support service discovery. Applications can access an etcd cluster with a DNS name or a virtual IP address managed by the system. For example, kube-proxy is equivalent to etcd gateway.

Start etcd gateway

Consider an etcd cluster with the following static endpoints:

Name	Address	Hostname	Port
infra0	10.0.1.10	infra0.example.com	2379
infra1	10.0.1.11	infra1.example.com	2379
infra2	10.0.1.12	infra2.example.com	2379

Start the etcd gateway to use these static endpoints with the command:

```
$ etcd gateway start --endpoints=infra0.example.com:2379,infra1.example.com:2379,infra2.e
2016-08-16 11:21:18.867350 I | tcpproxy: ready to proxy client requests to [...]
```

Alternatively, if using DNS for service discovery, consider the DNS SRV entries:

```
$ dig +noall +answer SRV _etcd-client._tcp.example.com
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra0.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra1.example.com.
_etcd-client._tcp.example.com. 300 IN SRV 0 0 2379 infra2.example.com.
```

```
$ dig +noall +answer infra0.example.com infra1.example.com infra2.example.com
infra0.example.com.  300  IN  A  10.0.1.10
infra1.example.com.  300  IN  A  10.0.1.11
infra2.example.com.  300  IN  A  10.0.1.12
```

Start the etcd gateway to fetch the endpoints from the DNS SRV entries with the command:

```
$ etcd gateway start --discovery-srv=example.com
2016-08-16 11:21:18.867350 I | tcpproxy: ready to proxy client requests to [...]
```

Configuration flags

etcd cluster

--endpoints

- Comma-separated list of etcd server targets for forwarding client connections.
- Default: `127.0.0.1:2379`
- Port must be included.
- Invalid example: `https://127.0.0.1:2379` (gateway does not terminate TLS)

--discovery-srv

- DNS domain used to bootstrap cluster endpoints through SRV records.
- Default: (not set)

Network

--listen-addr

- Interface and port to bind for accepting client requests.
- Default: `127.0.0.1:23790`

--retry-delay

- Duration of delay before retrying to connect to failed endpoints.
- Default: 1m0s
- Invalid example: “123” (expects time unit in format)

Security

--insecure-discovery

- Accept SRV records that are insecure or susceptible to man-in-the-middle attacks.
- Default: `false`

--trusted-ca-file

- Path to the client TLS CA file for the etcd cluster. Used to authenticate endpoints.
- Default: (not set)

Last modified January 21, 2023: Clarify port requirement for gateway endpoints (c4ecad8)

gRPC proxy

A stateless etcd reverse proxy operating at the gRPC layer

The gRPC proxy is a stateless etcd reverse proxy operating at the gRPC layer (L7). The proxy is designed to reduce the total processing load on the core etcd cluster. For horizontal scalability, it coalesces watch and lease API requests. To protect the cluster against abusive clients, it caches key range requests.

The gRPC proxy supports multiple etcd server endpoints. When the proxy starts, it randomly picks one etcd server endpoint to use. This endpoint serves all requests until the proxy detects an endpoint failure. If the gRPC proxy detects an endpoint failure, it switches to a different endpoint, if available, to hide failures from its clients. Other retry policies, such as weighted round-robin, may be supported in the future.

Scalable watch API

The gRPC proxy coalesces multiple client watchers (`c-watchers`) on the same key or range into a single watcher (`s-watcher`) connected to an etcd server. The proxy broadcasts all events from the `s-watcher` to its `c-watchers` .

Assuming N clients watch the same key, one gRPC proxy can reduce the watch load on the etcd server from N to 1. Users can deploy multiple gRPC proxies to further distribute server load.

In the following example, three clients watch on key A. The gRPC proxy coalesces the three watchers, creating a single watcher attached to the etcd server.

```
            +-------------+
            | etcd server |
            +------+------+
                   ^ watch key A (s-watcher)
                   |
           +-------+-----+
           | gRPC proxy  | <-------+
           |             |         |
           ++-----+------+         |watch key A (c-watcher)
 watch key A ^     ^ watch key A    |
 (c-watcher) |     | (c-watcher)    |
```

```
      +-------+-+  ++---------+  +----+----+
      |  client |  |  client |  |  client |
      |         |  |         |  |         |
      +---------+  +---------+  +---------+
```

Limitations

To effectively coalesce multiple client watchers into a single watcher, the gRPC proxy coalesces new `c-watchers` into an existing `s-watcher` when possible. This coalesced `s-watcher` may be out of sync with the etcd server due to network delays or buffered undelivered events. When the watch revision is unspecified, the gRPC proxy will not guarantee the `c-watcher` will start watching from the most recent store revision. For example, if a client watches from an etcd server with revision 1000, that watcher will begin at revision 1000. If a client watches from the gRPC proxy, may begin watching from revision 990.

Similar limitations apply to cancellation. When the watcher is cancelled, the etcd server's revision may be greater than the cancellation response revision.

These two limitations should not cause problems for most use cases. In the future, there may be additional options to force the watcher to bypass the gRPC proxy for more accurate revision responses.

Scalable lease API

To keep its leases alive, a client must establish at least one gRPC stream to an etcd server for sending periodic heartbeats. If an etcd workload involves heavy lease activity spread over many clients, these streams may contribute to excessive CPU utilization. To reduce the total number of streams on the core cluster, the proxy supports lease stream coalescing.

Assuming N clients are updating leases, a single gRPC proxy reduces the stream load on the etcd server from N to 1. Deployments may have additional gRPC proxies to further distribute streams across multiple proxies.

In the following example, three clients update three independent leases (`L1` , `L2` , and `L3`). The gRPC proxy coalesces the three client lease streams (`c-streams`) into a single lease keep alive stream (`s-stream`) attached to an etcd server. The proxy forwards client-side lease heartbeats from the c-streams to the s-stream, then returns the responses to the corresponding c-streams.

```
            +-------------+
            | etcd server |
            +------+------+
                   ^
```

```
                 | heartbeat L1, L2, L3
                 | (s-stream)
                 v
         +-------+-----+
         | gRPC proxy  +<-----------+
         +---+-----+---+            | heartbeat L3
             ^     ^                | (c-stream)
heartbeat L1 |     | heartbeat L2   |
(c-stream)   v     v (c-stream)     v
      +------+-+  +-+------+  +-----+---+
      | client |  | client |  | client |
      +--------+  +--------+  +--------+
```

Abusive clients protection

The gRPC proxy caches responses for requests when it does not break consistency requirements. This can protect the etcd server from abusive clients in tight for loops.

Start etcd gRPC proxy

Consider an etcd cluster with the following static endpoints:

Name	Address	Hostname
infra0	10.0.1.10	infra0.example.com
infra1	10.0.1.11	infra1.example.com
infra2	10.0.1.12	infra2.example.com

Start the etcd gRPC proxy to use these static endpoints with the command:

```
$ etcd grpc-proxy start --endpoints=infra0.example.com,infra1.example.com,infra2.example
```

The etcd gRPC proxy starts and listens on port 2379. It forwards client requests to one of the three endpoints provided above.

Sending requests through the proxy:

```
$ ETCDCTL_API=3 etcdctl --endpoints=127.0.0.1:2379 put foo bar
```

```
OK
$ ETCDCTL_API=3 etcdctl --endpoints=127.0.0.1:2379 get foo
foo
bar
```

Client endpoint synchronization and name resolution

The proxy supports registering its endpoints for discovery by writing to a user-defined endpoint. This serves two purposes. First, it allows clients to synchronize their endpoints against a set of proxy endpoints for high availability. Second, it is an endpoint provider for etcd gRPC naming.

Register proxy(s) by providing a user-defined prefix:

```
$ etcd grpc-proxy start --endpoints=localhost:2379 \
  --listen-addr=127.0.0.1:23790 \
  --advertise-client-url=127.0.0.1:23790 \
  --resolver-prefix="___grpc_proxy_endpoint" \
  --resolver-ttl=60

$ etcd grpc-proxy start --endpoints=localhost:2379 \
  --listen-addr=127.0.0.1:23791 \
  --advertise-client-url=127.0.0.1:23791 \
  --resolver-prefix="___grpc_proxy_endpoint" \
  --resolver-ttl=60
```

The proxy will list all its members for member list:

```
ETCDCTL_API=3 etcdctl --endpoints=http://localhost:23790 member list --write-out table

+----+---------+--------------------------------+------------+-----------------+
| ID | STATUS  |              NAME              | PEER ADDRS |  CLIENT ADDRS   |
+----+---------+--------------------------------+------------+-----------------+
|  0 | started | Gyu-Hos-MBP.sfo.coreos.systems |            | 127.0.0.1:23791 |
|  0 | started | Gyu-Hos-MBP.sfo.coreos.systems |            | 127.0.0.1:23790 |
+----+---------+--------------------------------+------------+-----------------+
```

This lets clients automatically discover proxy endpoints through Sync:

```
cli, err := clientv3.New(clientv3.Config{
```

```
    Endpoints: []string{"http://localhost:23790"},
})
if err != nil {
    log.Fatal(err)
}
defer cli.Close()

// fetch registered grpc-proxy endpoints
if err := cli.Sync(context.Background()); err != nil {
    log.Fatal(err)
}
```

Note that if a proxy is configured without a resolver prefix,

```
$ etcd grpc-proxy start --endpoints=localhost:2379 \
  --listen-addr=127.0.0.1:23792 \
  --advertise-client-url=127.0.0.1:23792
```

The member list API to the grpc-proxy returns its own `advertise-client-url`:

```
ETCDCTL_API=3 etcdctl --endpoints=http://localhost:23792 member list --write-out table

+----+---------+--------------------------------+------------+-----------------+
| ID | STATUS  |              NAME              | PEER ADDRS |  CLIENT ADDRS   |
+----+---------+--------------------------------+------------+-----------------+
|  0 | started | Gyu-Hos-MBP.sfo.coreos.systems |            | 127.0.0.1:23792 |
+----+---------+--------------------------------+------------+-----------------+
```

Namespacing

Suppose an application expects full control over the entire key space, but the etcd cluster is shared with other applications. To let all applications run without interfering with each other, the proxy can partition the etcd keyspace so clients appear to have access to the complete keyspace. When the proxy is given the flag `--namespace`, all client requests going into the proxy are translated to have a user-defined prefix on the keys. Accesses to the etcd cluster will be under the prefix and responses from the proxy will strip away the prefix; to the client, it appears as if there is no prefix at all.

To namespace a proxy, start it with `--namespace`:

```
$ etcd grpc-proxy start --endpoints=localhost:2379 \
  --listen-addr=127.0.0.1:23790 \
  --namespace=my-prefix/
```

Accesses to the proxy are now transparently prefixed on the etcd cluster:

```
$ ETCDCTL_API=3 etcdctl --endpoints=localhost:23790 put my-key abc
# OK
$ ETCDCTL_API=3 etcdctl --endpoints=localhost:23790 get my-key
# my-key
# abc
$ ETCDCTL_API=3 etcdctl --endpoints=localhost:2379 get my-prefix/my-key
# my-prefix/my-key
# abc
```

TLS termination

Terminate TLS from a secure etcd cluster with the gRPC proxy by serving an unencrypted local endpoint.

To try it out, start a single member etcd cluster with client https:

```
$ etcd --listen-client-urls https://localhost:2379 --advertise-client-urls https://localh
```

Confirm the client port is serving https:

```
# fails
$ ETCDCTL_API=3 etcdctl --endpoints=http://localhost:2379 endpoint status
# works
$ ETCDCTL_API=3 etcdctl --endpoints=https://localhost:2379 --cert=client.crt --key=clien
```

Next, start a gRPC proxy on `localhost:12379` by connecting to the etcd endpoint `https://localhost:2379` using the client certificates:

```
$ etcd grpc-proxy start --endpoints=https://localhost:2379 --listen-addr localhost:12379
```

Finally, test the TLS termination by putting a key into the proxy over http:

```
$ ETCDCTL_API=3 etcdctl --endpoints=http://localhost:12379 put abc def
# OK
```

Metrics and Health

The gRPC proxy exposes `/health` and Prometheus `/metrics` endpoints for the etcd members defined by `--endpoints`. An alternative define an additional URL that will respond to both the `/metrics` and `/health` endpoints with the `--metrics-addr` flag.

```
$ etcd grpc-proxy start \
  --endpoints https://localhost:2379 \
  --metrics-addr https://0.0.0.0:4443 \
  --listen-addr 127.0.0.1:23790 \
  --key client.key \
  --key-file proxy-server.key \
  --cert client.crt \
  --cert-file proxy-server.crt \
  --cacert ca.pem \
  --trusted-ca-file proxy-ca.pem
```

Known issue

The main interface of the proxy serves both HTTP2 and HTTP/1.1. If proxy is setup with TLS as show in the above example, when using a client such as cURL against the listening interface will require explicitly setting the protocol to HTTP/1.1 on the request to return `/metrics` or `/health`. By using the `--metrics-addr` flag the secondary interface will not have this requirement.

```
 $ curl --cacert proxy-ca.pem --key proxy-client.key --cert proxy-client.crt https://127
```

Last modified April 9, 2022: [Fix typos (a2da31e)]

Hardware recommendations

Hardware guidelines for administering etcd clusters

etcd usually runs well with limited resources for development or testing purposes; it's common to develop with etcd on a laptop or a cheap cloud machine. However, when running etcd clusters in production, some hardware guidelines are useful for proper administration. These suggestions are not hard rules; they serve as a good starting point for a robust production deployment. As always, deployments should be tested with simulated workloads before running in production.

CPUs

Few etcd deployments require a lot of CPU capacity. Typical clusters need two to four cores to run smoothly. Heavily loaded etcd deployments, serving thousands of clients or tens of thousands of requests per second, tend to be CPU bound since etcd can serve requests from memory. Such heavy deployments usually need eight to sixteen dedicated cores.

Memory

etcd has a relatively small memory footprint but its performance still depends on having enough memory. An etcd server will aggressively cache key-value data and spends most of the rest of its memory tracking watchers. Typically 8GB is enough. For heavy deployments with thousands of watchers and millions of keys, allocate 16GB to 64GB memory accordingly.

Disks

Fast disks are the most critical factor for etcd deployment performance and stability.

A slow disk will increase etcd request latency and potentially hurt cluster stability. Since etcd's consensus protocol depends on persistently storing metadata to a log, a majority of etcd cluster members must write every request down to disk. Additionally, etcd will also incrementally checkpoint its state to disk so it can truncate this log. If these writes take too long, heartbeats may time out and trigger an election, undermining the stability of the cluster. In general, to tell whether a disk is fast enough for etcd, a benchmarking tool such as fio can be used. Read here for an example.

etcd is very sensitive to disk write latency. Typically 50 sequential IOPS (e.g., a 7200 RPM disk) is required. For heavily loaded clusters, 500 sequential IOPS (e.g., a typical local SSD or a high performance virtualized block device) is recommended. Note that most cloud providers publish concurrent IOPS rather than sequential IOPS; the published concurrent IOPS can be 10x greater than the sequential IOPS. To measure actual sequential IOPS, we suggest using a disk benchmarking tool such as diskbench or fio.

etcd requires only modest disk bandwidth but more disk bandwidth buys faster recovery times when a failed member has to catch up with the cluster. Typically 10MB/s will recover 100MB data within 15 seconds. For large clusters, 100MB/s or higher is suggested for recovering 1GB data within 15 seconds.

When possible, back etcd's storage with a SSD. A SSD usually provides lower write latencies and with less variance than a spinning disk, thus improving the stability and reliability of etcd. If using spinning disk, get the fastest disks possible (15,000 RPM). Using RAID 0 is also an effective way to increase disk speed, for both spinning disks and SSD. With at least three cluster members, mirroring and/or parity variants of RAID are unnecessary; etcd's consistent replication already gets high availability.

Network

Multi-member etcd deployments benefit from a fast and reliable network. In order for etcd to be both consistent and partition tolerant, an unreliable network with partitioning outages will lead to poor availability. Low latency ensures etcd members can communicate fast. High bandwidth can reduce the time to recover a failed etcd member. 1GbE is sufficient for common etcd deployments. For large etcd clusters, a 10GbE network will reduce mean time to recovery.

Deploy etcd members within a single data center when possible to avoid latency overheads and lessen the possibility of partitioning events. If a failure domain in another data center is required, choose a data center closer to the existing one. Please also read the tuning documentation for more information on cross data center deployment.

Example hardware configurations

Here are a few example hardware setups on AWS and GCE environments. As mentioned before, but must be stressed regardless, administrators should test an etcd deployment with a simulated workload before putting it into production.

Note that these configurations assume these machines are totally dedicated to etcd. Running other applications along with etcd on these machines may cause resource contentions and lead to cluster instability.

Small cluster

A small cluster serves fewer than 100 clients, fewer than 200 of requests per second, and stores no more than 100MB of data.

Example application workload: A 50-node Kubernetes cluster

Provider	Type	vCPUs	Memory (GB)	Max concurrent IOPS	Disk bandwidth (MB/s)
AWS	m4.large	2	8	3600	56.25
GCE	n1-standard-2 + 50GB PD SSD	2	7.5	1500	25

Medium cluster

A medium cluster serves fewer than 500 clients, fewer than 1,000 of requests per second, and stores no more than 500MB of data.

Example application workload: A 250-node Kubernetes cluster

Provider	Type	vCPUs	Memory (GB)	Max concurrent IOPS	Disk bandwidth (MB/s)
AWS	m4.xlarge	4	16	6000	93.75
GCE	n1-standard-4 + 150GB PD SSD	4	15	4500	75

Large cluster

A large cluster serves fewer than 1,500 clients, fewer than 10,000 of requests per second, and stores no more than 1GB of data.

Example application workload: A 1,000-node Kubernetes cluster

Provider	Type	vCPUs	Memory (GB)	Max concurrent IOPS	Disk bandwidth (MB/s)
AWS	m4.2xlarge	8	32	8000	125
GCE	n1-standard-8 + 250GB PD SSD	8	30	7500	125

xLarge cluster

An xLarge cluster serves more than 1,500 clients, more than 10,000 of requests per second, and stores more than 1GB data.

Example application workload: A 3,000 node Kubernetes cluster

Provider	Type	vCPUs	Memory (GB)	Max concurrent IOPS	Disk bandwidth (MB/s)
AWS	m4.4xlarge	16	64	16,000	250
GCE	n1-standard-16 + 500GB PD SSD	16	60	15,000	250

Last modified August 9, 2023: fix broken link again (35877ad)

Maintenance

Periodic etcd cluster maintenance guide

Overview

An etcd cluster needs periodic maintenance to remain reliable. Depending on an etcd application's needs, this maintenance can usually be automated and performed without downtime or significantly degraded performance.

All etcd maintenance manages storage resources consumed by the etcd keyspace. Failure to adequately control the keyspace size is guarded by storage space quotas; if an etcd member runs low on space, a quota will trigger cluster-wide alarms which will put the system into a limited-operation maintenance mode. To avoid running out of space for writes to the keyspace, the etcd keyspace history must be compacted. Storage space itself may be reclaimed by defragmenting etcd members. Finally, periodic snapshot backups of etcd member state makes it possible to recover any unintended logical data loss or corruption caused by operational error.

Raft log retention

`etcd --snapshot-count` configures the number of applied Raft entries to hold in-memory before compaction. When `--snapshot-count` reaches, server first persists snapshot data onto disk, and then truncates old entries. When a slow follower requests logs before a compacted index, leader sends the snapshot forcing the follower to overwrite its state.

Higher `--snapshot-count` holds more Raft entries in memory until snapshot, thus causing recurrent higher memory usage. Since leader retains latest Raft entries for longer, a slow follower has more time to catch up before leader snapshot. `--snapshot-count` is a tradeoff between higher memory usage and better availabilities of slow followers.

Since v3.2, the default value of `--snapshot-count` has changed from from 10,000 to 100,000.

In performance-wise, `--snapshot-count` greater than 100,000 may impact the write throughput. Higher number of in-memory objects can slow down Go GC mark phase `runtime.scanobject`, and infrequent memory reclamation makes allocation slow. Performance varies depending on the workloads and system environments. However, in general, too frequent compaction affects cluster availabilities and write throughputs. Too infrequent compaction is also harmful placing too much pressure on Go garbage collector. See https://www.slideshare.net/mitakeh/understanding-performance-aspects-of-etcd-and-raft for more research results.

History compaction: v3 API Key-Value Database

Since etcd keeps an exact history of its keyspace, this history should be periodically compacted to avoid performance degradation and eventual storage space exhaustion. Compacting the keyspace history drops all information about keys superseded prior to a given keyspace revision. The space used by these keys then becomes available for additional writes to the keyspace.

The keyspace can be compacted automatically with `etcd`'s time windowed history retention policy, or manually with `etcdctl`. The `etcdctl` method provides fine-grained control over the compacting process whereas automatic compacting fits applications that only need key history for some length of time.

An `etcdctl` initiated compaction works as follows:

```
# compact up to revision 3
$ etcdctl compact 3
```

Revisions prior to the compaction revision become inaccessible:

```
$ etcdctl get --rev=2 somekey
Error:  rpc error: code = 11 desc = etcdserver: mvcc: required revision has been compact
```

Auto Compaction

`etcd` can be set to automatically compact the keyspace with the `--auto-compaction-*` option with a period of hours:

```
# keep one hour of history
$ etcd --auto-compaction-retention=1
```

v3.0.0 and v3.1.0 with `--auto-compaction-retention=10` run periodic compaction on v3 key-value store for every 10-hour. Compactor only supports periodic compaction. Compactor records latest revisions every 5-minute, until it reaches the first compaction period (e.g. 10-hour). In order to retain key-value history of last compaction period, it uses the last revision that was fetched before compaction period, from the revision records that were collected every 5-minute. When `--auto-compaction-retention=10`, compactor uses revision 100 for compact revision where revision 100 is the latest revision fetched from 10 hours ago. If compaction succeeds or requested revision has already been compacted, it resets period timer and starts over with new historical revision records (e.g. restart revision collect and compact for the next 10-hour period). If compaction fails, it retries in 5 minutes.

[v3.2.0] compactor runs [every hour]. Compactor only supports periodic compaction. Compactor continues to record latest revisions every 5-minute. For every hour, it uses the last revision that was fetched before compaction period, from the revision records that were collected every 5-minute. That is, for every hour, compactor discards historical data created before compaction period. The retention window of compaction period moves to next hour. For instance, when hourly writes are 100 and `--auto-compaction-retention=10`, v3.1 compacts revision 1000, 2000, and 3000 for every 10-hour, while v3.2.x, v3.3.0, v3.3.1, and v3.3.2 compact revision 1000, 1100, and 1200 for every 1-hour. If compaction succeeds or requested revision has already been compacted, it resets period timer and removes used compacted revision from historical revision records (e.g. start next revision collect and compaction from previously collected revisions). If compaction fails, it retries in 5 minutes.

In [v3.3.0], [v3.3.1], and [v3.3.2], `--auto-compaction-mode=revision --auto-compaction-retention=1000` automatically `Compact` on `"latest revision" - 1000` every 5-minute (when latest revision is 30000, compact on revision 29000). For instance, `--auto-compaction-mode=periodic --auto-compaction-retention=72h` automatically `Compact` with 72-hour retention window, for every 7.2-hour. For instance, `--auto-compaction-mode=periodic --auto-compaction-retention=30m` automatically `Compact` with 30-minute retention window, for every 3-minute. Periodic compactor continues to record latest revisions for every 1/10 of given compaction period (e.g. 1-hour when `--auto-compaction-mode=periodic --auto-compaction-retention=10h`). For every 1/10 of given compaction period, compactor uses the last revision that was fetched before compaction period, to discard historical data. The retention window of compaction period moves for every 1/10 of given compaction period. For instance, when hourly writes are 100 and `--auto-compaction-retention=10`, v3.1 compacts revision 1000, 2000, and 3000 for every 10-hour, while v3.2.x, v3.3.0, v3.3.1, and v3.3.2 compact revision 1000, 1100, and 1200 for every 1-hour. Furthermore, when writes per minute are 1000, v3.3.0, v3.3.1, and v3.3.2 with `--auto-compaction-mode=periodic --auto-compaction-retention=30m` compact revision 30000, 33000, and 36000, for every 3-minute with more finer granularity.

When `--auto-compaction-retention=10h`, etcd first waits 10-hour for the first compaction, and then does compaction every hour (1/10 of 10-hour) afterwards like this:

```
0Hr  (rev = 1)
1hr  (rev = 10)
...
```

```
8hr  (rev = 80)
9hr  (rev = 90)
10hr (rev = 100, Compact(1))
11hr (rev = 110, Compact(10))
...
```

Whether compaction succeeds or not, this process repeats for every 1/10 of given compaction period. If compaction succeeds, it just removes compacted revision from historical revision records.

In v3.3.3, `--auto-compaction-mode=revision --auto-compaction-retention=1000` automatically `Compact` on `"latest revision" - 1000` every 5-minute (when latest revision is 30000, compact on revision 29000). Previously, `--auto-compaction-mode=periodic --auto-compaction-retention=72h` automatically `Compact` with 72-hour retention window for every 7.2-hour. **Now, `Compact` happens, for every 1-hour but still with 72-hour retention window.** Previously, `--auto-compaction-mode=periodic --auto-compaction-retention=30m` automatically `Compact` with 30-minute retention window for every 3-minute. **Now, `Compact` happens, for every 30-minute but still with 30-minute retention window.** Periodic compactor keeps recording latest revisions for every compaction period when given period is less than 1-hour, or for every 1-hour when given compaction period is greater than 1-hour (e.g. 1-hour when `--auto-compaction-mode=periodic --auto-compaction-retention=24h`). For every compaction period or 1-hour, compactor uses the last revision that was fetched before compaction period, to discard historical data. The retention window of compaction period moves for every given compaction period or hour. For instance, when hourly writes are 100 and `--auto-compaction-mode=periodic --auto-compaction-retention=24h`, `v3.2.x`, `v3.3.0`, `v3.3.1`, and `v3.3.2` compact revision 2400, 2640, and 2880 for every 2.4-hour, while `v3.3.3` *or later* compacts revision 2400, 2500, 2600 for every 1-hour. Furthermore, when `--auto-compaction-mode=periodic --auto-compaction-retention=30m` and writes per minute are about 1000, `v3.3.0`, `v3.3.1`, and `v3.3.2` compact revision 30000, 33000, and 36000, for every 3-minute, while `v3.3.3` *or later* compacts revision 30000, 60000, and 90000, for every 30-minute.

Defragmentation

After compacting the keyspace, the backend database may exhibit internal fragmentation. Any internal fragmentation is space that is free to use by the backend but still consumes storage space. Compacting old revisions internally fragments `etcd` by leaving gaps in backend database. Fragmented space is available for use by `etcd` but unavailable to the host filesystem. In other words, deleting application data does not reclaim the space on disk.

The process of defragmentation releases this storage space back to the file system. Defragmentation is issued on a per-member basis so that cluster-wide latency spikes may be avoided.

To defragment an etcd member, use the `etcdctl defrag` command:

```
$ etcdctl defrag
Finished defragmenting etcd member[127.0.0.1:2379]
```

Note that defragmentation to a live member blocks the system from reading and writing data while rebuilding its states.

Note that defragmentation request does not get replicated over cluster. That is, the request is only applied to the local node. Specify all members in `--endpoints` flag or `--cluster` flag to automatically find all cluster members.

Run defragment operations for all endpoints in the cluster associated with the default endpoint:

```
$ etcdctl defrag --cluster
Finished defragmenting etcd member[http://127.0.0.1:2379]
Finished defragmenting etcd member[http://127.0.0.1:22379]
Finished defragmenting etcd member[http://127.0.0.1:32379]
```

To defragment an etcd data directory directly, while etcd is not running, use the command:

```
$ etcdctl defrag --data-dir <path-to-etcd-data-dir>
```

Space quota

The space quota in `etcd` ensures the cluster operates in a reliable fashion. Without a space quota, `etcd` may suffer from poor performance if the keyspace grows excessively large, or it may simply run out of storage space, leading to unpredictable cluster behavior. If the keyspace's backend database for any member exceeds the space quota, `etcd` raises a cluster-wide alarm that puts the cluster into a maintenance mode which only accepts key reads and deletes. Only after freeing enough space in the keyspace and defragmenting the backend database, along with clearing the space quota alarm can the cluster resume normal operation.

By default, `etcd` sets a conservative space quota suitable for most applications, but it may be configured on the command line, in bytes:

```
# set a very small 16 MiB quota
$ etcd --quota-backend-bytes=$((16*1024*1024))
```

The space quota can be triggered with a loop:

```
# fill keyspace
$ while [ 1 ]; do dd if=/dev/urandom bs=1024 count=1024  | ETCDCTL_API=3 etcdctl put key
...
Error:  rpc error: code = 8 desc = etcdserver: mvcc: database space exceeded
# confirm quota space is exceeded
$ ETCDCTL_API=3 etcdctl --write-out=table endpoint status
+----------------+------------------+-----------+---------+-----------+-----------+-----
|    ENDPOINT    |        ID        |  VERSION  | DB SIZE | IS LEADER | RAFT TERM | RAFT
+----------------+------------------+-----------+---------+-----------+-----------+-----
| 127.0.0.1:2379 | bf9071f4639c75cc | 2.3.0+git | 18 MB   | true      |         2 |
+----------------+------------------+-----------+---------+-----------+-----------+-----
# confirm alarm is raised
$ ETCDCTL_API=3 etcdctl alarm list
memberID:13803658152347727308 alarm:NOSPACE
```

Removing excessive keyspace data and defragmenting the backend database will put the cluster back within the quota limits:

```
# get current revision
$ rev=$(ETCDCTL_API=3 etcdctl --endpoints=:2379 endpoint status --write-out="json" | egr
# compact away all old revisions
$ ETCDCTL_API=3 etcdctl compact $rev
compacted revision 1516
# defragment away excessive space
$ ETCDCTL_API=3 etcdctl defrag
Finished defragmenting etcd member[127.0.0.1:2379]
# disarm alarm
$ ETCDCTL_API=3 etcdctl alarm disarm
memberID:13803658152347727308 alarm:NOSPACE
# test puts are allowed again
$ ETCDCTL_API=3 etcdctl put newkey 123
OK
```

The metric `etcd_mvcc_db_total_size_in_use_in_bytes` indicates the actual database usage after a history compaction, while `etcd_debugging_mvcc_db_total_size_in_bytes` shows the database size including free space waiting for defragmentation. The latter increases only when the former is close to it, meaning when both of these metrics are close to the quota, a history compaction is required to avoid triggering the space quota.

`etcd_debugging_mvcc_db_total_size_in_bytes` is renamed to `etcd_mvcc_db_total_size_in_bytes` from v3.4.

Snapshot backup

Snapshotting the `etcd` cluster on a regular basis serves as a durable backup for an etcd keyspace. By taking periodic snapshots of an etcd member's backend database, an `etcd` cluster can be recovered to a point in time with a known good state.

A snapshot is taken with `etcdctl`:

```
$ etcdctl snapshot save backup.db
$ etcdctl --write-out=table snapshot status backup.db
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| fe01cf57 |       10 |          7 | 2.1 MB     |
+----------+----------+------------+------------+
```

Last modified August 17, 2024: [fix typo (264cc3e)]

Performance

Understanding performance: latency & throughput

Understanding performance

etcd provides stable, sustained high performance. Two factors define performance: latency and throughput. Latency is the time taken to complete an operation. Throughput is the total operations completed within some time period. Usually average latency increases as the overall throughput increases when etcd accepts concurrent client requests. In common cloud environments, like a standard `n-4` on Google Compute Engine (GCE) or a comparable machine type on AWS, a three member etcd cluster finishes a request in less than one millisecond under light load, and can complete more than 30,000 requests per second under heavy load.

etcd uses the Raft consensus algorithm to replicate requests among members and reach agreement. Consensus performance, especially commit latency, is limited by two physical constraints: network IO latency and disk IO latency. The minimum time to finish an etcd request is the network Round Trip Time (RTT) between members, plus the time `fdatasync` requires to commit the data to permanent storage. The RTT within a datacenter may be as long as several hundred microseconds. A typical RTT within the United States is around 50ms, and can be as slow as 400ms between continents. The typical fdatasync latency for a spinning disk is about 10ms. For SSDs, the latency is often lower than 1ms. To increase throughput, etcd batches multiple requests together and submits them to Raft. This batching policy lets etcd attain high throughput despite heavy load.

There are other sub-systems which impact the overall performance of etcd. Each serialized etcd request must run through etcd's boltdb-backed MVCC storage engine, which usually takes tens of microseconds to finish. Periodically etcd incrementally snapshots its recently applied requests, merging them back with the previous on-disk snapshot. This process may lead to a latency spike. Although this is usually not a problem on SSDs, it may double the observed latency on HDD. Likewise, inflight compactions can impact etcd's performance. Fortunately, the impact is often insignificant since the compaction is staggered so it does not compete for resources with regular requests. The RPC system, gRPC, gives etcd a well-defined, extensible API, but it also introduces additional latency, especially for local reads.

Benchmarks

Benchmarking etcd performance can be done with the benchmark CLI tool included with etcd.

For some baseline performance numbers, we consider a three member etcd cluster with the following hardware configuration:

- Google Cloud Compute Engine
- 3 machines of 8 vCPUs + 16GB Memory + 50GB SSD
- 1 machine(client) of 16 vCPUs + 30GB Memory + 50GB SSD
- Ubuntu 17.04
- etcd 3.2.0, go 1.8.3

With this configuration, etcd can approximately write:

Number of keys	Key size in bytes	Value size in bytes	Number of connections	Number of clients	Target etcd server	Average write QPS	Average latency per request	Avera ser
10,000	8	256	1	1	leader only	583	1.6ms	48
100,000	8	256	100	1000	leader only	44,341	22ms	124
100,000	8	256	100	1000	all members	50,104	20ms	126

Sample commands are:

```
# write to leader
benchmark --endpoints=${HOST_1} --target-leader --conns=1 --clients=1 \
    put --key-size=8 --sequential-keys --total=10000 --val-size=256
benchmark --endpoints=${HOST_1} --target-leader  --conns=100 --clients=1000 \
    put --key-size=8 --sequential-keys --total=100000 --val-size=256

# write to all members
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=100 --clients=1000 \
    put --key-size=8 --sequential-keys --total=100000 --val-size=256
```

Linearizable read requests go through a quorum of cluster members for consensus to fetch the most recent data. Serializable read requests are cheaper than linearizable reads since they are served by any single etcd member, instead of a quorum of members, in exchange for possibly serving stale data. etcd can read:

Number of requests	Key size in bytes	Value size in bytes	Number of connections	Number of clients	Consistency	Average read QPS	Average latency per request
10,000	8	256	1	1	Linearizable	1,353	0.7ms
10,000	8	256	1	1	Serializable	2,909	0.3ms
100,000	8	256	100	1000	Linearizable	141,578	5.5ms
100,000	8	256	100	1000	Serializable	185,758	2.2ms

Sample commands are:

```
# Single connection read requests
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=1 --clients=1 \
    range YOUR_KEY --consistency=l --total=10000
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=1 --clients=1 \
    range YOUR_KEY --consistency=s --total=10000

# Many concurrent read requests
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=100 --clients=1000 \
    range YOUR_KEY --consistency=l --total=100000
benchmark --endpoints=${HOST_1},${HOST_2},${HOST_3} --conns=100 --clients=1000 \
    range YOUR_KEY --consistency=s --total=100000
```

We encourage running the benchmark test when setting up an etcd cluster for the first time in a new environment to ensure the cluster achieves adequate performance; cluster latency and throughput can be sensitive to minor environment differences.

Last modified April 26, 2021: Fixing broken links (#203) (ae1b7f6)

Design of runtime reconfiguration

The design of etcd's runtime reconfiguration commands

Runtime reconfiguration is one of the hardest and most error prone features in a distributed system, especially in a consensus based system like etcd.

Read on to learn about the design of etcd's runtime reconfiguration commands and how we tackled these problems.

Two phase config changes keep the cluster safe

In etcd, every runtime reconfiguration has to go through two phases for safety reasons. For example, to add a member, first inform the cluster of the new configuration and then start the new member.

Phase 1 - Inform cluster of new configuration

To add a member into an etcd cluster, make an API call to request a new member to be added to the cluster. This is the only way to add a new member into an existing cluster. The API call returns when the cluster agrees on the configuration change.

Phase 2 - Start new member

To join the new etcd member into the existing cluster, specify the correct `initial-cluster` and set `initial-cluster-state` to `existing`. When the member starts, it will contact the existing cluster first and verify the current cluster configuration matches the expected one specified in `initial-cluster`. When the new member successfully starts, the cluster has reached the expected configuration.

By splitting the process into two discrete phases users are forced to be explicit regarding cluster membership changes. This actually gives users more flexibility and makes things easier to reason about. For example, if there is an attempt to add a new member with the same ID as an existing member in an etcd cluster, the action will fail immediately during phase one without impacting the running cluster. Similar protection is provided to prevent adding new members by mistake. If a new etcd member attempts to join the cluster before the cluster has accepted the configuration change, it will not be accepted by the cluster.

Without the explicit workflow around cluster membership etcd would be vulnerable to unexpected cluster membership changes. For example, if etcd is running under an init system such as systemd, etcd would be restarted after being removed via the membership API, and attempt to rejoin the cluster on startup. This cycle would continue every time a member is removed via the API and systemd is set to restart etcd after failing, which is unexpected.

We expect runtime reconfiguration to be an infrequent operation. We decided to keep it explicit and user-driven to ensure configuration safety and keep the cluster always running smoothly under explicit control.

Permanent loss of quorum requires new cluster

If a cluster permanently loses a majority of its members, a new cluster will need to be started from an old data directory to recover the previous state.

It is entirely possible to force removing the failed members from the existing cluster to recover. However, we decided not to support this method since it bypasses the normal consensus committing phase, which is unsafe. If the member to remove is not actually dead or force removed through different members in the same cluster, etcd will end up with a diverged cluster with same clusterID. This is very dangerous and hard to debug/fix afterwards.

With a correct deployment, the possibility of permanent majority loss is very low. But it is a severe enough problem that is worth special care. We strongly suggest reading the disaster recovery documentation and preparing for permanent majority loss before putting etcd into production.

Do not use public discovery service for runtime reconfiguration

The public discovery service should only be used for bootstrapping a cluster. To join member into an existing cluster, use the runtime reconfiguration API.

The discovery service is designed for bootstrapping an etcd cluster in a cloud environment, when the IP addresses of all the members are not known beforehand. After successfully bootstrapping a cluster, the IP addresses of all the members are known. Technically, the discovery service should no longer be needed.

It seems that using public discovery service is a convenient way to do runtime reconfiguration, after all discovery service already has all the cluster configuration information. However relying on public discovery service brings troubles:

1. it introduces external dependencies for the entire life-cycle of the cluster, not just bootstrap time. If there is a network issue between the cluster and public discovery service, the cluster will suffer from it.

2. public discovery service must reflect correct runtime configuration of the cluster during its life-cycle. It has to provide security mechanisms to avoid bad actions, and it is hard.

3. public discovery service has to keep tens of thousands of cluster configurations. Our public discovery service backend is not ready for that workload.

To have a discovery service that supports runtime reconfiguration, the best choice is to build a private one.

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

Runtime reconfiguration

etcd incremental runtime reconfiguration support

etcd comes with support for incremental runtime reconfiguration, which allows users to update the membership of the cluster at run time.

Reconfiguration requests can only be processed when a majority of cluster members are functioning. It is **highly recommended** to always have a cluster size greater than two in production. It is unsafe to remove a member from a two member cluster. The majority of a two member cluster is also two. If there is a failure during the removal process, the cluster might not be able to make progress and need to restart from majority failure.

To better understand the design behind runtime reconfiguration, please read the runtime reconfiguration document.

Reconfiguration use cases

This section will walk through some common reasons for reconfiguring a cluster. Most of these reasons just involve combinations of adding or removing a member, which are explained below under Cluster Reconfiguration Operations.

Cycle or upgrade multiple machines

If multiple cluster members need to move due to planned maintenance (hardware upgrades, network downtime, etc.), it is recommended to modify members one at a time.

It is safe to remove the leader, however there is a brief period of downtime while the election process takes place. If the cluster holds more than 50MB of v2 data, it is recommended to migrate the member's data directory.

Change the cluster size

Increasing the cluster size can enhance failure tolerance and provide better read performance. Since clients can read from any member, increasing the number of members increases the overall serialized read throughput.

Decreasing the cluster size can improve the write performance of a cluster, with a trade-off of decreased resilience. Writes into the cluster are replicated to a majority of members of the cluster before considered committed. Decreasing the cluster size lowers the majority, and each write is committed more quickly.

Replace a failed machine

If a machine fails due to hardware failure, data directory corruption, or some other fatal situation, it should be replaced as soon as possible. Machines that have failed but haven't been removed adversely affect the quorum and reduce the tolerance for an additional failure.

To replace the machine, follow the instructions for removing the member from the cluster, and then add a new member in its place. If the cluster holds more than 50MB, it is recommended to migrate the failed member's data directory if it is still accessible.

Restart cluster from majority failure

If the majority of the cluster is lost or all of the nodes have changed IP addresses, then manual action is necessary to recover safely. The basic steps in the recovery process include creating a new cluster using the old data, forcing a single member to act as the leader, and finally using runtime configuration to add new members to this new cluster one at a time.

Cluster reconfiguration operations

With these use cases in mind, the involved operations can be described for each.

Before making any change, a simple majority (quorum) of etcd members must be available. This is essentially the same requirement for any kind of write to etcd.

All changes to the cluster must be done sequentially:

- To update a single member peerURLs, issue an update operation
- To replace a healthy single member, remove the old member then add a new member
- To increase from 3 to 5 members, issue two add operations
- To decrease from 5 to 3, issue two remove operations

All of these examples use the `etcdctl` command line tool that ships with etcd. To change membership without `etcdctl`, use the v2 HTTP members API or the v3 gRPC members API.

Update a member

Update advertise client URLs

To update the advertise client URLs of a member, simply restart that member with updated client urls flag (`--advertise-client-urls`) or environment variable (`ETCD_ADVERTISE_CLIENT_URLS`). The restarted member will self publish the updated URLs. A wrongly updated client URL will not affect the health of the etcd cluster.

Update advertise peer URLs

To update the advertise peer URLs of a member, first update it explicitly via member command and then restart the member. The additional action is required since updating peer URLs changes the cluster wide configuration and can affect the health of the etcd cluster.

To update the advertise peer URLs, first find the target member's ID. To list all members with `etcdctl` :

```
$ etcdctl member list
6e3bd23ae5f1eae0: name=node2 peerURLs=http://localhost:23802 clientURLs=http://127.0.0.1
924e2e83e93f2560: name=node3 peerURLs=http://localhost:23803 clientURLs=http://127.0.0.1
a8266ecf031671f3: name=node1 peerURLs=http://localhost:23801 clientURLs=http://127.0.0.1
```

This example will `update` a8266ecf031671f3 member ID and change its peerURLs value to `http://10.0.1.10:2380` :

```
$ etcdctl member update a8266ecf031671f3 --peer-urls=http://10.0.1.10:2380
Updated member with ID a8266ecf031671f3 in cluster
```

Remove a member

Suppose the member ID to remove is a8266ecf031671f3. Use the `remove` command to perform the removal:

```
$ etcdctl member remove a8266ecf031671f3
Removed member a8266ecf031671f3 from cluster
```

The target member will stop itself at this point and print out the removal in the log:

```
etcd: this member has been permanently removed from the cluster. Exiting.
```

It is safe to remove the leader, however the cluster will be inactive while a new leader is elected. This duration is normally the period of election timeout plus the voting process.

Add a new member

Adding a member is a two step process:

- Add the new member to the cluster via the [HTTP members API], the [gRPC members API], or the `etcdctl member add` command.
- Start the new member with the new cluster configuration, including a list of the updated members (existing members + the new member).

`etcdctl` adds a new member to the cluster by specifying the member's [name] and [advertised peer URLs]:

```
$ etcdctl member add infra3 --peer-urls=http://10.0.1.13:2380
added member 9bf1b35fc7761a23 to cluster

ETCD_NAME="infra3"
ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=ht
ETCD_INITIAL_CLUSTER_STATE=existing
```

`etcdctl` has informed the cluster about the new member and printed out the environment variables needed to successfully start it. Now start the new etcd process with the relevant flags for the new member:

```
$ export ETCD_NAME="infra3"
$ export ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380
$ export ETCD_INITIAL_CLUSTER_STATE=existing
$ etcd --listen-client-urls http://10.0.1.13:2379 --advertise-client-urls http://10.0.1.1
```

The new member will run as a part of the cluster and immediately begin catching up with the rest of the cluster.

If adding multiple members the best practice is to configure a single member at a time and verify it starts correctly before adding more new members. If adding a new member to a 1-node cluster, the cluster cannot make progress before the new member starts because it needs two members as majority to agree on the consensus. This behavior only happens between the time `etcdctl member add` informs the cluster about the new member and the new member successfully establishing a connection to the existing one.

Add a new member as learner

Starting from v3.4, etcd supports adding a new member as learner / non-voting member. The motivation and design can be found in design doc. In order to make the process of adding a new member safer, and to reduce cluster downtime when the new member is added, it is recommended that the new member is added to cluster as a learner until it catches up. This can be described as a three step process:

- Add the new member as learner via gRPC members API or the `etcdctl member add --learner` command.
- Start the new member with the new cluster configuration, including a list of the updated members (existing members + the new member). This step is exactly the same as before.
- Promote the newly added learner to voting member via gRPC members API or the `etcdctl member promote` command. etcd server validates promote request to ensure its operational safety. Only after its raft log has caught up to leader's can learner be promoted to a voting member. If a learner member has not caught up to leader's raft log, member promote request will fail (see error cases when promoting a member section for more details). In this case, user should wait and retry later.

In v3.4, etcd server limits the number of learners that cluster can have to one. The main consideration is to limit the extra workload on leader due to propagating data from leader to learner.

Use `etcdctl member add` with flag `--learner` to add new member to cluster as learner.

```
$ etcdctl member add infra3 --peer-urls=http://10.0.1.13:2380 --learner
Member 9bf1b35fc7761a23 added to cluster a7ef944b95711739

ETCD_NAME="infra3"
ETCD_INITIAL_CLUSTER="infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=ht
ETCD_INITIAL_CLUSTER_STATE=existing
```

After new etcd process is started for the newly added learner member, use `etcdctl member promote` to promote learner to voting member.

```
$ etcdctl member promote 9bf1b35fc7761a23
Member 9e29bbaa45d74461 promoted in cluster a7ef944b95711739
```

Error cases when adding members

In the following case a new host is not included in the list of enumerated nodes. If this is a new cluster, the node must be added to the list of initial cluster members.

```
$ etcd --name infra3 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=htt
  --initial-cluster-state existing
etcdserver: assign ids error: the member count is unequal
exit 1
```

In this case, give a different address (10.0.1.14:2380) from the one used to join the cluster (10.0.1.13:2380):

```
$ etcd --name infra4 \
  --initial-cluster infra0=http://10.0.1.10:2380,infra1=http://10.0.1.11:2380,infra2=htt
  --initial-cluster-state existing
etcdserver: assign ids error: unmatched member while checking PeerURLs
exit 1
```

If etcd starts using the data directory of a removed member, etcd automatically exits if it connects to any active member in the cluster:

```
$ etcd
etcd: this member has been permanently removed from the cluster. Exiting.
exit 1
```

Error cases when adding a learner member

Cannot add learner to cluster if the cluster already has 1 learner (v3.4).

```
$ etcdctl member add infra4 --peer-urls=http://10.0.1.14:2380 --learner
Error: etcdserver: too many learner members in cluster
```

Error cases when promoting a learner member

Learner can only be promoted to voting member if it is in sync with leader.

```
$ etcdctl member promote 9bf1b35fc7761a23
Error: etcdserver: can only promote a learner member which is in sync with leader
```

Promoting a member that is not a learner will fail.

```
$ etcdctl member promote 9bf1b35fc7761a23
Error: etcdserver: can only promote a learner member
```

Promoting a member that does not exist in cluster will fail.

```
$ etcdctl member promote 12345abcde
Error: etcdserver: member not found
```

Strict reconfiguration check mode (`-strict-reconfig-check`)

As described in the above, the best practice of adding new members is to configure a single member at a time and verify it starts correctly before adding more new members. This step by step approach is very important because if newly added members is not configured correctly (for example the peer URLs are incorrect), the cluster can lose quorum. The quorum loss happens since the newly added member are counted in the quorum even if that member is not reachable from other existing members. Also quorum loss might happen if there is a connectivity issue or there are operational issues.

For avoiding this problem, etcd provides an option `-strict-reconfig-check`. If this option is passed to etcd, etcd rejects reconfiguration requests if the number of started members will be less than a quorum of the reconfigured cluster.

It is enabled by default.

Last modified August 21, 2021: [fix 3.4 links (#458) (f75a5c9)]()

Supported platforms

etcd support status for common architectures & operating systems

Current support

The following table lists etcd support status for common architectures and operating systems:

Architecture	Operating System	Status	Maintainers
amd64	Darwin	Experimental	etcd maintainers
amd64	Linux	Stable	etcd maintainers
amd64	Windows	Experimental	
arm64	Linux	Experimental	
arm	Linux	Unstable	
386	Linux	Unstable	
ppc64le	Linux	Stable	etcd maintainers

- etcd-maintainers are listed in https://github.com/etcd-io/etcd/blob/main/OWNERS.

Experimental platforms appear to work in practice and have some platform specific code in etcd, but do not fully conform to the stable support policy. Unstable platforms have been lightly tested, but less than experimental. Unlisted architecture and operating system pairs are currently unsupported; caveat emptor.

Supporting a new system platform

For etcd to officially support a new platform as stable, a few requirements are necessary to ensure acceptable quality:

1. An “official” maintainer for the platform with clear motivation; someone must be responsible for taking care of the platform.
2. Set up CI for build; etcd must compile.
3. Set up CI for running unit tests; etcd must pass simple tests.
4. Set up CI (TravisCI, SemaphoreCI or Jenkins) for running integration tests; etcd must pass intensive tests.
5. (Optional) Set up a functional testing cluster; an etcd cluster should survive stress testing.

32-bit and other unsupported systems

etcd has known issues on 32-bit systems due to a bug in the Go runtime. See the Go issue and atomic package for more information.

To avoid inadvertently running a possibly unstable etcd server, `etcd` on unstable or unsupported architectures will print a warning message and immediately exit if the environment variable `ETCD_UNSUPPORTED_ARCH` is not set to the target architecture.

Currently amd64 and ppc64le architectures are officially supported by `etcd`.

Last modified October 18, 2023: Complete migration to owners file. (bc148e9)

Migrate applications from using API v2 to API v3

A guide for migrating from API v2 to API v3

The data store v2 is still accessible from the API v2 after upgrading to etcd3. Thus, it will work as before and require no application changes. With etcd 3, applications use the new grpc API v3 to access the mvcc store, which provides more features and improved performance. The mvcc store and the old store v2 are separate and isolated; writes to the store v2 will not affect the mvcc store and, similarly, writes to the mvcc store will not affect the store v2.

Migrating an application from the API v2 to the API v3 involves two steps: 1) migrate the client library and, 2) migrate the data. If the application can rebuild the data, then migrating the data is unnecessary.

Migrate client library

API v3 is different from API v2, thus application developers need to use a new client library to send requests to etcd API v3. The documentation of the client v3 is available at https://pkg.go.dev/github.com/etcd-io/etcd/clientv3.

There are some notable differences between API v2 and API v3:

- Transaction: In v3, etcd provides multi-key conditional transactions. Applications should use transactions in place of `Compare-And-Swap` operations.
- Flat key space: There are no directories in API v3, only keys. For example, "/a/b/c/" is a key. Range queries support getting all keys matching a given prefix.
- Compacted responses: Operations like `Delete` no longer return previous values. To get the deleted value, a transaction can be used to atomically get the key and then delete its value.
- Leases: A replacement for v2 TTLs; the TTL is bound to a lease and keys attach to the lease. When the TTL expires, the lease is revoked and all attached keys are removed.

Migrate data

Application data can be migrated either offline or online. Offline migration is much simpler than online migration and is recommended.

Sometimes an etcd cluster will possibly have v3 data which should not be overwritten. In this case, the migration process may want to confirm no v3 data is committed before proceeding. One way to check the cluster has no v3 keys is to issue the following `etcdctl` command, which scans the entire v3 keyspace for any key, expecting `0` as output:

```
ETCDCTL_API=3 etcdctl get "" --from-key --keys-only --limit 1 | wc -l
```

Offline migration

Offline migration is very simple but requires etcd downtime. If an etcd downtime window spanning from seconds to minutes is acceptable, offline migration is a good choice and is easy to automate.

First, all members in the etcd cluster must converge to the same state. This can be achieved by stopping all applications that write keys to etcd. Alternatively, if the applications must remain running, configure etcd to listen on a different client URL and restart all etcd members. To check if the states converged, within a few seconds, use the `ETCDCTL_API=3 etcdctl endpoint status` command to confirm that the `raft index` of all members match (or differ by at most 1 due to an internal sync raft command).

Second, migrate the v2 keys into v3 with the migrate (`ETCDCTL_API=3 etcdctl migrate`) command. The migrate command writes keys in the v2 store to a user-provided transformer program and reads back transformed keys. It then writes transformed keys into the mvcc store. This usually takes at most tens of seconds.

> **known issue**
>
> The migrate command has a known issue that new members added after the migration do not receive existing data if no .snap file exists. The workaround is to perform a snapshot restore after the migration. For more information see GitHub issue.

Restart the etcd members and everything should just work.

For etcd v3.3+, run `ETCDCTL_API=3 etcdctl endpoint hashkv --cluster` to ensure key-value stores are consistent post migration.

Warn: When v2 store has expiring TTL keys and migrate command intends to preserve TTLs, migration may be inconsistent with the last committed v2 state when run on any member with a raft index less than the last leader's raft index.

Online migration

If the application cannot tolerate any downtime, then it must migrate online. The implementation of online migration will vary from application to application but the overall idea is the same.

First, write application code using the v3 API. The application must support two modes: a migration mode and a normal mode. The application starts in migration mode. When running in migration mode, the application reads keys using the v3 API first, and, if it cannot find the key, it retries with the API v2. In normal mode, the application only reads keys using the v3 API. The application writes keys over the API v3 in both modes. To acknowledge a switch from migration mode to normal mode, the application watches on a switch mode key. When switch key's value turns to `true`, the application switches over from migration mode to normal mode.

Second, start a background job to migrate data from the store v2 to the mvcc store by reading keys from the API v2 and writing keys to the API v3.

After finishing data migration, the background job writes `true` into the switch mode key to notify the application that it may switch modes.

Online migration can be difficult when the application logic depends on store v2 indexes. Applications will need additional logic to convert mvcc store revisions to store v2 indexes.

Last modified August 27, 2022: [Document a known issue about the v2 to v3 migration (f550fc0)](#)

Versioning

Versioning support by etcd

This document describes the versions supported by the etcd project.

Service versioning and supported versions

etcd versions are expressed as **x.y.z**, where **x** is the major version, **y** is the minor version, and **z** is the patch version, following Semantic Versioning terminology. New minor versions may add additional features to the API.

The etcd project maintains release branches for the current version and previous release. For example, when v3.5 is the current version, v3.4 is supported. When v3.6 is released, v3.4 goes out of support.

Applicable fixes, including security fixes, may be backported to those two release branches, depending on severity and feasibility. Patch releases are cut from those branches when required.

The project Maintainers own this decision.

You can check the running etcd cluster version with `etcdctl`:

```
etcdctl --endpoints=127.0.0.1:2379 endpoint status
```

API versioning

The `v3` API responses should not change after the 3.0.0 release but new features will be added over time.

Last modified October 18, 2023: Complete migration to owners file. (bc148e9)

Data Corruption

etcd data corruption and recovery

etcd has built in automated data corruption detection to prevent member state from diverging.

Enabling data corruption detection

Data corruption detection can be done in two ways:

- Initial check, enabled with `--experimental-initial-corrupt-check` flag.
- Periodic check, enabled with `--experimental-corrupt-check-time` flag.

Initial check will be executed during bootstrap of etcd member. Member will compare its persistent state vs other members and exit if there is a mismatch.

Periodic check will be executed by the cluster leader in a cluster that is already running. Leader will compare its persistent state vs other members and raise a CORRUPT ALARM if there is a mismatch. Period of checks is configured using format: `1m` - every minute, `1h` - evey hour. Recommended period is a couple of hours as there is a high performance cost. Running a check requires computing a checksum by scanning entire etcd content at given revision.

Restoring a corrupted member

There are three ways to restore a corrupted member:

- Purge member persistent state
- Replace member
- Restore whole cluster

After the corrupted member is restored, CORRUPT ALARM can be removed.

Purge member persistent state

Members state can be purged by:

1. Stopping the etcd instance.
2. Backing up etcd data directory.
3. Moving out the `snap` subdirectory from the etcd data directory.
4. Starting `etcd` with `--initial-cluster-state=existing` and cluster members listed in `--initial-cluster`.

Etcd member is expected to download up-to-date snapshot from the leader.

Replace member

Member can be replaced by:

1. Stopping the etcd instance.
2. Backing up the etcd data directory.
3. Removing the data directory.
4. Removing the member from cluster by running `etcdctl member remove`.
5. Adding it back by running `etcdctl member add`
6. Starting `etcd` with `--initial-cluster-state=existing` and cluster members listed in `--initial-cluster`.

Restore whole cluster

Cluster can be restored by saving a snapshot from current leader and restoring it to all members. Run `etcdctl snapshot save` against the leader and follow restoring a cluster procedure.

Last modified September 17, 2023: Change from it's to its (070ed10)

Monitoring etcd

Each etcd server provides local monitoring information on its client port through http endpoints. The monitoring data is useful for both system health checking and cluster debugging.

Debug endpoint

If `--debug` is set, the etcd server exports debugging information on its client port under the `/debug` path. Take care when setting `--debug`, since there will be degraded performance and verbose logging.

The `/debug/pprof` endpoint is the standard go runtime profiling endpoint. This can be used to profile CPU, heap, mutex, and goroutine utilization. For example, here `go tool pprof` gets the top 10 functions where etcd spends its time:

```
$ go tool pprof http://localhost:2379/debug/pprof/profile
Fetching profile from http://localhost:2379/debug/pprof/profile
Please wait... (30s)
Saved profile in /home/etcd/pprof/pprof.etcd.localhost:2379.samples.cpu.001.pb.gz
Entering interactive mode (type "help" for commands)
(pprof) top10
310ms of 480ms total (64.58%)
Showing top 10 nodes out of 157 (cum >= 10ms)
    flat  flat%   sum%        cum   cum%
   130ms 27.08% 27.08%      130ms 27.08%  runtime.futex
    70ms 14.58% 41.67%       70ms 14.58%  syscall.Syscall
    20ms  4.17% 45.83%       20ms  4.17%  github.com/coreos/etcd/vendor/golang.org/x/net/
    20ms  4.17% 50.00%       30ms  6.25%  runtime.pcvalue
    20ms  4.17% 54.17%       50ms 10.42%  runtime.schedule
    10ms  2.08% 56.25%       10ms  2.08%  github.com/coreos/etcd/vendor/github.com/coreos
    10ms  2.08% 58.33%       10ms  2.08%  github.com/coreos/etcd/vendor/github.com/coreos
    10ms  2.08% 60.42%       10ms  2.08%  github.com/coreos/etcd/vendor/github.com/coreos
    10ms  2.08% 62.50%       10ms  2.08%  github.com/coreos/etcd/vendor/github.com/promet
    10ms  2.08% 64.58%       10ms  2.08%  github.com/coreos/etcd/vendor/golang.org/x/net/
```

The `/debug/requests` endpoint gives gRPC traces and performance statistics through a web browser. For example, here is a `Range` request for the key `abc`:

```
When    Elapsed (s)
2017/08/18 17:34:51.999317      0.000244        /etcdserverpb.KV/Range
17:34:51.999382          .    65        ... RPC: from 127.0.0.1:47204 deadline:4.9993777
17:34:51.999395          .    13        ... recv: key:"abc"
17:34:51.999499          .   104        ... OK
17:34:51.999535          .    36        ... sent: header:<cluster_id:1484163906896517841
```

Metrics endpoint

Each etcd server exports metrics under the `/metrics` path on its client port and optionally on locations given by `--listen-metrics-urls`.

The metrics can be fetched with `curl`:

```
$ curl -L http://localhost:2379/metrics | grep -v debugging # ignore unstable debugging

# HELP etcd_disk_backend_commit_duration_seconds The latency distributions of commit cal
# TYPE etcd_disk_backend_commit_duration_seconds histogram
etcd_disk_backend_commit_duration_seconds_bucket{le="0.002"} 72756
etcd_disk_backend_commit_duration_seconds_bucket{le="0.004"} 401587
etcd_disk_backend_commit_duration_seconds_bucket{le="0.008"} 405979
etcd_disk_backend_commit_duration_seconds_bucket{le="0.016"} 406464
...
```

Health Check

Since v3.3.0, in addition to responding to the `/metrics` endpoint, any locations specified by `--listen-metrics-urls` will also respond to the `/health` endpoint. This can be useful if the standard endpoint is configured with mutual (client) TLS authentication, but a load balancer or monitoring service still needs access to the health check.

Since v3.4.29, two new endpoints `/livez` and `/readyz` are added.

- the `/livez` endpoint reflects whether the process is alive or if it needs a restart.
- the `/readyz` endpoint reflects whether the process is ready to serve traffic.

Design details of the endpoints are documented in the KEP.

Each endpoint includes several individual health checks, and you can use the `verbose` parameter to print out the details of the checks and their status, for example

```
curl -k http://localhost:2379/readyz?verbose
```

and you would see the response similar to

```
[+]data_corruption ok
[+]serializable_read ok
[+]linearizable_read ok
ok
```

The http API also supports to exclude specific checks, for example

```
curl -k http://localhost:2379/readyz?exclude=data_corruption
```

Prometheus

Running a [Prometheus](#) monitoring service is the easiest way to ingest and record etcd's metrics.

First, install Prometheus:

```
PROMETHEUS_VERSION="2.0.0"
wget https://github.com/prometheus/prometheus/releases/download/v$PROMETHEUS_VERSION/prom
tar -xvzf /tmp/prometheus-$PROMETHEUS_VERSION.linux-amd64.tar.gz --directory /tmp/ --stri
/tmp/prometheus -version
```

Set Prometheus's scraper to target the etcd cluster endpoints:

```
cat > /tmp/test-etcd.yaml <<EOF
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: test-etcd
```

```
    static_configs:
    - targets: ['10.240.0.32:2379','10.240.0.33:2379','10.240.0.34:2379']
EOF
cat /tmp/test-etcd.yaml
```

Set up the Prometheus handler:

```
nohup /tmp/prometheus \
    -config.file /tmp/test-etcd.yaml \
    -web.listen-address ":9090" \
    -storage.local.path "test-etcd.data" >> /tmp/test-etcd.log  2>&1 &
```

Now Prometheus will scrape etcd metrics every 10 seconds.

Alerting

There is a set of default alerts for etcd v3 clusters for Prometheus 1.x as well as Prometheus 2.x.

> Note: `job` labels may need to be adjusted to fit a particular need. The rules were written to apply to a single cluster so it is recommended to choose labels unique to a cluster.

Grafana

Grafana has built-in Prometheus support; just add a Prometheus data source:

```
Name:   test-etcd
Type:   Prometheus
Url:    http://localhost:9090
Access: proxy
```

Then import the default etcd dashboard template and customize. For instance, if Prometheus data source name is `my-etcd`, the `datasource` field values in JSON also need to be `my-etcd`.

Sample dashboard:

Last modified January 2, 2024: [Document the new livez/readyz endpoints. (e55accd)](#)

Benchmarks

Performance measures for etcd

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Storage Memory Usage Benchmark

Performance measures for etcd storage (in-memory index & page cache)

Two components of etcd storage consume physical memory. The etcd process allocates an *in-memory index* to speed key lookup. The process's *page cache*, managed by the operating system, stores recently-accessed data from disk for quick re-use.

The in-memory index holds all the keys in a B-tree data structure, along with pointers to the on-disk data (the values). Each key in the B-tree may contain multiple pointers, pointing to different versions of its values. The theoretical memory consumption of the in-memory index can hence be approximated with the formula:

```
N * (c1 + avg_key_size) + N * (avg_versions_of_key) * (c2 + size_of_pointer)
```

where `c1` is the key metadata overhead and `c2` is the version metadata overhead.

The graph shows the detailed structure of the in-memory index B-tree.

```
                              In mem index

                             +------------+
                             | key || ... |
   +--------------+          |     ||     |
   |              |          +------------+
   |              |          | v1  || ... |
   |   disk    <-------------|     ||     | Tree Node
   |              |          +------------+
   |              |          | v2  || ... |
   |           <-------------+     ||     |
   |              |          +------------+
   +--------------+    +-----+    |   |   |
                       |     |    |   |   |
                       |     +------------+
                       |
                       |
                       ^
                     ------+
                     | ... |
```

```
               |     |
               +-----+
               | ... | Tree Node
               |     |
               +-----+
               | ... |
               |     |
               ------+
```

[Page cache memory](#) is managed by the operating system and is not covered in detail in this document.

Testing Environment

etcd version

- git head https://github.com/etcd-io/etcd/commit/776e9fb7be7eee5e6b58ab977c8887b4fe4d48db

GCE n1-standard-2 machine type

- 7.5 GB memory
- 2x CPUs

In-memory index memory usage

In this test, we only benchmark the memory usage of the in-memory index. The goal is to find `c1` and `c2` mentioned above and to understand the hard limit of memory consumption of the storage.

We calculate the memory usage consumption via the Go runtime.ReadMemStats. We calculate the total allocated bytes difference before creating the index and after creating the index. It cannot perfectly reflect the memory usage of the in-memory index itself but can show the rough consumption pattern.

N	versions	key size	memory usage
100K	1	64bytes	22MB
100K	5	64bytes	39MB
1M	1	64bytes	218MB
1M	5	64bytes	432MB

N	versions	key size	memory usage
100K	1	256bytes	41MB
100K	5	256bytes	65MB
1M	1	256bytes	409MB
1M	5	256bytes	506MB

Based on the result, we can calculate `c1=120bytes`, `c2=30bytes`. We only need two sets of data to calculate `c1` and `c2`, since they are the only unknown variable in the formula. The `c1=120bytes` and `c2=30bytes` are the average value of the 4 sets of `c1` and `c2` we calculated. The key metadata overhead is still relatively nontrivial (50%) for small key-value pairs. However, this is a significant improvement over the old store, which had at least 1000% overhead.

Overall memory usage

The overall memory usage captures how much RSS etcd consumes with the storage. The value size should have very little impact on the overall memory usage of etcd, since we keep values on disk and only retain hot values in memory, managed by the OS page cache.

N	versions	key size	value size	memory usage
100K	1	64bytes	256bytes	40MB
100K	5	64bytes	256bytes	89MB
1M	1	64bytes	256bytes	470MB
1M	5	64bytes	256bytes	880MB
100K	1	64bytes	1KB	102MB
100K	5	64bytes	1KB	164MB
1M	1	64bytes	1KB	587MB
1M	5	64bytes	1KB	836MB

Based on the result, we know the value size does not significantly impact the memory consumption. There is some minor increase due to more data held in the OS page cache.

Last modified April 26, 2021: [Fixing broken links (#203) (ae1b7f6)]

Watch Memory Usage Benchmark

Performance measures for etcd watchers

NOTE: The watch features are under active development, and their memory usage may change as that development progresses. We do not expect it to significantly increase beyond the figures stated below.

A primary goal of etcd is supporting a very large number of watchers doing a massively large amount of watching. etcd aims to support O(10k) clients, O(100K) watch streams (O(10) streams per client) and O(10M) total watchings (O(100) watching per stream). The memory consumed by each individual watching accounts for the largest portion of etcd's overall usage, and is therefore the focus of current and future optimizations.

Three related components of etcd watch consume physical memory: each `grpc.Conn`, each watch stream, and each instance of the watching activity. `grpc.Conn` maintains the actual TCP connection and other gRPC connection state. Each `grpc.Conn` consumes O(10kb) of memory, and might have multiple watch streams attached.

Each watch stream is an independent HTTP2 connection which consumes another O(10kb) of memory. Multiple watchings might share one watch stream.

Watching is the actual struct that tracks the changes on the key-value store. Each watching should only consume < O(1kb).

```
                                          +-------+
                                          | watch |
                              +---------> | foo   |
                              |           +-------+
                       +------+-----+
                       |   stream   |
      +--------------> |            |
      |                +------+-----+     +-------+
      |                       |           | watch |
      |                       +---------> | bar   |
+-----+------+                            +-------+
|            |         +------------+
|   conn     +-------> |   stream   |
|            |         |            |
+-----+------+         +------------+
      |
```

```
      |
      |
      |                   +------------+
      +--------------> |   stream   |
                          |            |
                          +------------+
```

The theoretical memory consumption of watch can be approximated with the formula: `memory = c1 * number_of_conn + c2 * avg_number_of_stream_per_conn + c3 * avg_number_of_watch_stream`

Testing Environment

etcd version

- git head https://github.com/etcd-io/etcd/commit/185097ffaa627b909007e772c175e8fefac17af3

GCE n1-standard-2 machine type

- 7.5 GB memory
- 2x CPUs

Overall memory usage

The overall memory usage captures how much RSS etcd consumes with the client watchers. While the result may vary by as much as 10%, it is still meaningful, since the goal is to learn about the rough memory usage and the pattern of allocations.

With the benchmark result, we can calculate roughly that `c1 = 17kb`, `c2 = 18kb` and `c3 = 350bytes`. So each additional client connection consumes 17kb of memory and each additional stream consumes 18kb of memory, and each additional watching only cause 350bytes. A single etcd server can maintain millions of watchings with a few GB of memory in normal case.

clients	streams per client	watchings per stream	total watching	memory usage
1k	1	1	1k	50MB
2k	1	1	2k	90MB
5k	1	1	5k	200MB

clients	streams per client	watchings per stream	total watching	memory usage
1k	10	1	10k	217MB
2k	10	1	20k	417MB
5k	10	1	50k	980MB
1k	50	1	50k	1001MB
2k	50	1	100k	1960MB
5k	50	1	250k	4700MB
1k	50	10	500k	1171MB
2k	50	10	1M	2371MB
5k	50	10	2.5M	5710MB
1k	50	100	5M	2380MB
2k	50	100	10M	4672MB
5k	50	100	25M	*OOM*

Last modified April 26, 2021: Fixing broken links (#203) (ae1b7f6)

Benchmarking etcd v3

Performance measures for etcd v3

See etcd v3 performance benchmarking.

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

Benchmarking etcd v2.2.0-rc-memory

Performance measures for etcd v2.2.0-rc-memory

Physical machine

GCE n1-standard-2 machine type

- 1x dedicated local SSD mounted under /var/lib/etcd
- 1x dedicated slow disk for the OS
- 7.5 GB memory
- 2x CPUs

etcd

```
etcd Version: 2.2.0-rc.0+git
Git SHA: 103cb5c
Go Version: go1.5
Go OS/Arch: linux/amd64
```

Testing

Start 3-member etcd cluster, each of which uses 2 cores.

The length of key name is always 64 bytes, which is a reasonable length of average key bytes.

Memory Maximal Usage

- etcd may use maximal memory if one follower is dead and the leader keeps sending snapshots.
- `max RSS` is the maximal memory usage recorded in 3 runs.

value bytes	key number	data size(MB)	max RSS(MB)	max RSS/data rate on leader
128	50000	6	433	72x
128	100000	12	659	54x
128	200000	24	1466	61x
1024	50000	48	1253	26x
1024	100000	96	2344	24x
1024	200000	192	4361	22x

Data Size Threshold

- When etcd reaches data size threshold, it may trigger leader election easily and drop part of proposals.
- For most cases, the etcd cluster should work smoothly if it doesn't hit the threshold. If it doesn't work well due to insufficient resources, decrease its data size.

value bytes	key number limitation	suggested data size threshold(MB)	consumed RSS(MB)
128	400K	48	2400
1024	300K	292	6500

Last modified April 26, 2021: [Docsy theme (#244) (86b070b)]

Benchmarking etcd v2.2.0-rc

Performance measures for etcd v2.2.0-rc

Physical machine

GCE n1-highcpu-2 machine type

- 1x dedicated local SSD mounted under /var/lib/etcd
- 1x dedicated slow disk for the OS
- 1.8 GB memory
- 2x CPUs

etcd Cluster

3 etcd 2.2.0-rc members, each runs on a single machine.

Detailed versions:

```
etcd Version: 2.2.0-alpha.1+git
Git SHA: 59a5a7e
Go Version: go1.4.2
Go OS/Arch: linux/amd64
```

Also, we use 3 etcd 2.1.0 alpha-stage members to form cluster to get base performance. etcd's commit head is at c7146bd5, which is the same as the one that we use in etcd 2.1 benchmark.

Testing

Bootstrap another machine and use the hey HTTP benchmark tool to send requests to each etcd member. Check the benchmark hacking guide for detailed instructions.

Performance

reading one single key

key size in bytes	number of clients	target etcd server	read QPS	90th Percentile Latency (ms)
64	1	leader only	2804 (-5%)	0.4 (+0%)
64	64	leader only	17816 (+0%)	5.7 (-6%)
64	256	leader only	18667 (-6%)	20.4 (+2%)
256	1	leader only	2181 (-15%)	0.5 (+25%)
256	64	leader only	17435 (-7%)	6.0 (+9%)
256	256	leader only	18180 (-8%)	21.3 (+3%)
64	64	all servers	46965 (-4%)	2.1 (+0%)
64	256	all servers	55286 (-6%)	7.4 (+6%)
256	64	all servers	46603 (-6%)	2.1 (+5%)
256	256	all servers	55291 (-6%)	7.3 (+4%)

writing one single key

key size in bytes	number of clients	target etcd server	write QPS	90th Percentile Latency (ms)
64	1	leader only	76 (+22%)	19.4 (-15%)
64	64	leader only	2461 (+45%)	31.8 (-32%)

key size in bytes	number of clients	target etcd server	write QPS	90th Percentile Latency (ms)
64	256	leader only	4275 (+1%)	69.6 (-10%)
256	1	leader only	64 (+20%)	16.7 (-30%)
256	64	leader only	2385 (+30%)	31.5 (-19%)
256	256	leader only	4353 (-3%)	74.0 (+9%)
64	64	all servers	2005 (+81%)	49.8 (-55%)
64	256	all servers	4868 (+35%)	81.5 (-40%)
256	64	all servers	1925 (+72%)	47.7 (-59%)
256	256	all servers	4975 (+36%)	70.3 (-36%)

performance changes explanation

- read QPS in most scenarios is decreased by 5~8%. The reason is that etcd records store metrics for each store operation. The metrics is important for monitoring and debugging, so this is acceptable.
- write QPS to leader is increased by 20~30%. This is because we decouple raft main loop and entry apply loop, which avoids them blocking each other.
- write QPS to all servers is increased by 30~80% because follower could receive latest commit index earlier and commit proposals faster.

Last modified August 21, 2021: [fix 3.4 links (#458) (f75a5c9)]

Benchmarking etcd v2.2.0

Performance measures for etcd v2.2.0

Physical Machines

GCE n1-highcpu-2 machine type

- 1x dedicated local SSD mounted as etcd data directory
- 1x dedicated slow disk for the OS
- 1.8 GB memory
- 2x CPUs

etcd Cluster

3 etcd 2.2.0 members, each runs on a single machine.

Detailed versions:

```
etcd Version: 2.2.0
Git SHA: e4561dd
Go Version: go1.5
Go OS/Arch: linux/amd64
```

Testing

Bootstrap another machine, outside of the etcd cluster, and run the `hey` HTTP benchmark tool with a connection reuse patch to send requests to each etcd cluster member. See the benchmark instructions for the patch and the steps to reproduce our procedures.

The performance is calculated through results of 100 benchmark rounds.

Performance

Single Key Read Performance

key size in bytes	number of clients	target etcd server	average read QPS	read QPS stddev	average 90th Percentile Latency (ms)	latency stddev
64	1	leader only	2303	200	0.49	0.06
64	64	leader only	15048	685	7.60	0.46
64	256	leader only	14508	434	29.76	1.05
256	1	leader only	2162	214	0.52	0.06
256	64	leader only	14789	792	7.69	0.48
256	256	leader only	14424	512	29.92	1.42
64	64	all servers	45752	2048	2.47	0.14
64	256	all servers	46592	1273	10.14	0.59
256	64	all servers	45332	1847	2.48	0.12
256	256	all servers	46485	1340	10.18	0.74

Single Key Write Performance

key size in bytes	number of clients	target etcd server	average write QPS	write QPS stddev	average 90th Percentile Latency (ms)	latency stddev
64	1	leader only	55	4	24.51	13.26
64	64	leader only	2139	125	35.23	3.40
64	256	leader only	4581	581	70.53	10.22

key size in bytes	number of clients	target etcd server	average write QPS	write QPS stddev	average 90th Percentile Latency (ms)	latency stddev
256	1	leader only	56	4	22.37	4.33
256	64	leader only	2052	151	36.83	4.20
256	256	leader only	4442	560	71.59	10.03
64	64	all servers	1625	85	58.51	5.14
64	256	all servers	4461	298	89.47	36.48
256	64	all servers	1599	94	60.11	6.43
256	256	all servers	4315	193	88.98	7.01

Performance Changes

- Because etcd now records metrics for each API call, read QPS performance seems to see a minor decrease in most scenarios. This minimal performance impact was judged a reasonable investment for the breadth of monitoring and debugging information returned.
- Write QPS to cluster leaders seems to be increased by a small margin. This is because the main loop and entry apply loops were decoupled in the etcd raft logic, eliminating several blocks between them.
- Write QPS to all members seems to be increased by a significant margin, because followers now receive the latest commit index sooner, and commit proposals more quickly.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Benchmarking etcd v2.1.0

Performance measures for etcd v2.1.0

Physical machines

GCE n1-highcpu-2 machine type

- 1x dedicated local SSD mounted under /var/lib/etcd
- 1x dedicated slow disk for the OS
- 1.8 GB memory
- 2x CPUs
- etcd version 2.1.0 alpha

etcd Cluster

3 etcd members, each runs on a single machine

Testing

Bootstrap another machine and use the hey HTTP benchmark tool to send requests to each etcd member. Check the benchmark hacking guide for detailed instructions.

Performance

reading one single key

key size in bytes	number of clients	target etcd server	read QPS	90th Percentile Latency (ms)
64	1	leader only	1534	0.7
64	64	leader only	10125	9.1

key size in bytes	number of clients	target etcd server	read QPS	90th Percentile Latency (ms)
64	256	leader only	13892	27.1
256	1	leader only	1530	0.8
256	64	leader only	10106	10.1
256	256	leader only	14667	27.0
64	64	all servers	24200	3.9
64	256	all servers	33300	11.8
256	64	all servers	24800	3.9
256	256	all servers	33000	11.5

writing one single key

key size in bytes	number of clients	target etcd server	write QPS	90th Percentile Latency (ms)
64	1	leader only	60	21.4
64	64	leader only	1742	46.8
64	256	leader only	3982	90.5
256	1	leader only	58	20.3
256	64	leader only	1770	47.8
256	256	leader only	4157	105.3
64	64	all servers	1028	123.4
64	256	all servers	3260	123.8
256	64	all servers	1033	121.5
256	256	all servers	3061	119.3

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Upgrading

Upgrading etcd clusters and applications

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Upgrading etcd clusters and applications

Documentation list for upgrading etcd clusters and applications

This section contains documents specific to upgrading etcd clusters and applications.

Moving from etcd API v2 to API v3

- Migrate applications from using API v2 to API v3

Upgrading an etcd v3.x cluster

- Upgrade etcd from 3.0 to 3.1
- Upgrade etcd from 3.1 to 3.2
- Upgrade etcd from 3.2 to 3.3
- Upgrade etcd from 3.3 to 3.4

Upgrading from etcd v2.3

- Upgrade a v2.3 cluster to v3.0

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

Upgrade etcd from 3.4 to 3.5

Processes, checklists, and notes on upgrading etcd from 3.4 to 3.5

In the general case, upgrading from etcd 3.4 to 3.5 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v3.4 processes and replace them with etcd v3.5 processes
- after running all v3.5 processes, new features in v3.5 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Highlighted breaking changes in 3.5.

Deprecated `etcd_debugging_mvcc_db_total_size_in_bytes` Prometheus metrics

v3.5 promoted `etcd_debugging_mvcc_db_total_size_in_bytes` Prometheus metrics to `etcd_mvcc_db_total_size_in_bytes`, in order to encourage etcd storage monitoring. And v3.5 completely deprecates `etcd_debugging_mvcc_db_total_size_in_bytes`.

```
-etcd_debugging_mvcc_db_total_size_in_bytes
+etcd_mvcc_db_total_size_in_bytes
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecated `etcd_debugging_mvcc_put_total` Prometheus metrics

v3.5 promoted `etcd_debugging_mvcc_put_total` Prometheus metrics to `etcd_mvcc_put_total`, in order to encourage etcd storage monitoring. And v3.5 completely deprecates `etcd_debugging_mvcc_put_total`.

```
-etcd_debugging_mvcc_put_total
+etcd_mvcc_put_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecated `etcd_debugging_mvcc_delete_total` Prometheus metrics

v3.5 promoted `etcd_debugging_mvcc_delete_total` Prometheus metrics to `etcd_mvcc_delete_total`, in order to encourage etcd storage monitoring. And v3.5 completely deprecates `etcd_debugging_mvcc_delete_total`.

```
-etcd_debugging_mvcc_delete_total
+etcd_mvcc_delete_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecated `etcd_debugging_mvcc_txn_total` Prometheus metrics

v3.5 promoted `etcd_debugging_mvcc_txn_total` Prometheus metrics to `etcd_mvcc_txn_total`, in order to encourage etcd storage monitoring. And v3.5 completely deprecates `etcd_debugging_mvcc_txn_total`.

```
-etcd_debugging_mvcc_txn_total
+etcd_mvcc_txn_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecated `etcd_debugging_mvcc_range_total` Prometheus metrics

v3.5 promoted `etcd_debugging_mvcc_range_total` Prometheus metrics to `etcd_mvcc_range_total`, in order to encourage etcd storage monitoring. And v3.5 completely deprecates `etcd_debugging_mvcc_range_total`.

```
-etcd_debugging_mvcc_range_total
+etcd_mvcc_range_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecated `etcd --logger capnslog`

v3.4 defaults to `--logger=zap` in order to support multiple log outputs and structured logging.

`etcd --logger=capnslog` has been deprecated in v3.5, and now `--logger=zap` is the default.

```
-etcd --logger=capnslog
+etcd --logger=zap --log-outputs=stderr

+# to write logs to stderr and a.log file at the same time
+etcd --logger=zap --log-outputs=stderr,a.log
```

v3.4 adds `etcd --logger=zap` support for structured logging and multiple log outputs. Main motivation is to promote automated etcd monitoring, rather than looking back server logs when it starts breaking. Future development will make etcd log as few as possible, and make etcd easier to monitor with metrics and alerts. **`etcd --logger=capnslog` will be deprecated in v3.5.**

Deprecated `etcd --log-output`

v3.4 renamed [`etcd --log-output` to `--log-outputs`] to support multiple log outputs.

`etcd --log-output` has been deprecated in v3.5.

```
-etcd --log-output=stderr
+etcd --log-outputs=stderr
```

Deprecated `etcd --debug` flag (now `--log-level=debug`)

`etcd --debug` flag has been deprecated.

```
-etcd --debug
+etcd --log-level debug
```

Deprecated `etcd --log-package-levels`

`etcd --log-package-levels` flag for `capnslog` has been deprecated.

Now, `etcd --logger=zap` is the default.

```
-etcd --log-package-levels 'etcdmain=CRITICAL,etcdserver=DEBUG'
+etcd --logger=zap --log-outputs=stderr
```

Deprecated `[CLIENT-URL]/config/local/log`

`/config/local/log` endpoint is being deprecated in v3.5, as is `etcd --log-package-levels` flag.

```
-$ curl http://127.0.0.1:2379/config/local/log -XPUT -d '{"Level":"DEBUG"}'
-# debug logging enabled
```

Changed gRPC gateway HTTP endpoints (deprecated `/v3beta`)

Before

```
curl -L http://localhost:2379/v3beta/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

After

```
curl -L http://localhost:2379/v3/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

`/v3beta` has been removed in 3.5 release.

Server upgrade checklists

Upgrade requirements

To upgrade an existing etcd deployment to 3.5, the running cluster must be 3.4 or greater. If it's before 3.4, please upgrade to 3.4 before upgrading to 3.5.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl endpoint health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, download the snapshot backup. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version. Please note that the `snapshot` command only backs up the v3 data. For v2 data, see backing up v2 datastore.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.5. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

Note: If the cluster only has v3 data and no v2 data, it is not subject to this limitation.

If the cluster is serving a v2 data set larger than 50MB, each newly upgraded member may take up to two minutes to catch up with the existing cluster. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.5, the cluster will be upgraded to v3.5, and downgrade from this completed state is **not possible**. If any single member is still v3.4, however, the cluster and its operations remains “v3.4”, and it is possible from this mixed cluster state to return to using a v3.4 etcd binary on all members.

Please download the snapshot backup to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example shows how to upgrade a 3-member v3.4 etcd cluster running on a local machine.

Step 1: check upgrade requirements

Is the cluster healthy and running v3.4.x?

```
etcdctl --endpoints=localhost:2379,localhost:22379,localhost:32379 endpoint health
<<COMMENT
localhost:2379 is healthy: successfully committed proposal: took = 2.118638ms
localhost:22379 is healthy: successfully committed proposal: took = 3.631388ms
localhost:32379 is healthy: successfully committed proposal: took = 2.157051ms
COMMENT

curl http://localhost:2379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT

curl http://localhost:22379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT

curl http://localhost:32379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT
```

Step 2: download snapshot backup from leader

Download the snapshot backup to provide a downgrade path should any problems occur.

etcd leader is guaranteed to have the latest application data, thus fetch snapshot from leader:

```
curl -sL http://localhost:2379/metrics | grep etcd_server_is_leader
<<COMMENT
# HELP etcd_server_is_leader Whether or not this member is a leader. 1 if is, 0 otherwise
# TYPE etcd_server_is_leader gauge
etcd_server_is_leader 1
COMMENT

curl -sL http://localhost:22379/metrics | grep etcd_server_is_leader
<<COMMENT
etcd_server_is_leader 0
COMMENT

curl -sL http://localhost:32379/metrics | grep etcd_server_is_leader
<<COMMENT
etcd_server_is_leader 0
COMMENT

etcdctl --endpoints=localhost:2379 snapshot save backup.db
<<COMMENT
{"level":"info","ts":1526585787.148433,"caller":"snapshot/v3_snapshot.go:109","msg":"crea
{"level":"info","ts":1526585787.1485257,"caller":"snapshot/v3_snapshot.go:120","msg":"fet
{"level":"info","ts":1526585787.1519694,"caller":"snapshot/v3_snapshot.go:133","msg":"fet
{"level":"info","ts":1526585787.1520295,"caller":"snapshot/v3_snapshot.go:142","msg":"sav
Snapshot saved at backup.db
COMMENT
```

Step 3: stop one existing etcd server

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
{"level":"info","ts":1526587281.2001143,"caller":"etcdserver/server.go:2249","msg":"upda
{"level":"info","ts":1526587281.2010646,"caller":"membership/cluster.go:473","msg":"upda
{"level":"info","ts":1526587281.2012327,"caller":"api/capability.go:76","msg":"enabled c
{"level":"info","ts":1526587281.2013083,"caller":"etcdserver/server.go:2272","msg":"clus



^C{"level":"info","ts":1526587299.0717514,"caller":"osutil/interrupt_unix.go:63","msg":"
{"level":"info","ts":1526587299.0718873,"caller":"embed/etcd.go:285","msg":"closing etcd
{"level":"info","ts":1526587299.0722554,"caller":"etcdserver/server.go:1341","msg":"lead
{"level":"info","ts":1526587299.0723994,"caller":"raft/raft.go:1107","msg":"7339c4e5e833
{"level":"info","ts":1526587299.0724802,"caller":"raft/raft.go:1113","msg":"7339c4e5e833
{"level":"info","ts":1526587299.0737045,"caller":"raft/raft.go:797","msg":"7339c4e5e833c
{"level":"info","ts":1526587299.0737681,"caller":"raft/raft.go:656","msg":"7339c4e5e833c
{"level":"info","ts":1526587299.073831,"caller":"raft/raft.go:882","msg":"7339c4e5e833c0
```

```
{"level":"info","ts":1526587299.0738947,"caller":"raft/node.go:312","msg":"raft.node: 73
{"level":"info","ts":1526587299.0748374,"caller":"raft/node.go:306","msg":"raft.node: 73
{"level":"info","ts":1526587299.1726425,"caller":"etcdserver/server.go:1362","msg":"lead
{"level":"info","ts":1526587299.1728148,"caller":"rafthttp/peer.go:333","msg":"stopping
{"level":"warn","ts":1526587299.1751974,"caller":"rafthttp/stream.go:291","msg":"closed
{"level":"warn","ts":1526587299.1752589,"caller":"rafthttp/stream.go:301","msg":"stopped
{"level":"warn","ts":1526587299.177348,"caller":"rafthttp/stream.go:291","msg":"closed T
{"level":"warn","ts":1526587299.1774004,"caller":"rafthttp/stream.go:301","msg":"stopped
{"level":"info","ts":1526587299.177515,"caller":"rafthttp/pipeline.go:86","msg":"stopped
{"level":"warn","ts":1526587299.1777067,"caller":"rafthttp/stream.go:436","msg":"lost TC
{"level":"info","ts":1526587299.1778402,"caller":"rafthttp/stream.go:459","msg":"stopped
{"level":"warn","ts":1526587299.1780295,"caller":"rafthttp/stream.go:436","msg":"lost TC
{"level":"info","ts":1526587299.1780987,"caller":"rafthttp/stream.go:459","msg":"stopped
{"level":"info","ts":1526587299.1781602,"caller":"rafthttp/peer.go:340","msg":"stopped r
{"level":"info","ts":1526587299.1781986,"caller":"rafthttp/peer.go:333","msg":"stopping
{"level":"warn","ts":1526587299.1802843,"caller":"rafthttp/stream.go:291","msg":"closed
{"level":"warn","ts":1526587299.1803446,"caller":"rafthttp/stream.go:301","msg":"stopped
{"level":"warn","ts":1526587299.1824749,"caller":"rafthttp/stream.go:291","msg":"closed
{"level":"warn","ts":1526587299.18255,"caller":"rafthttp/stream.go:301","msg":"stopped T
{"level":"info","ts":1526587299.18261,"caller":"rafthttp/pipeline.go:86","msg":"stopped
{"level":"warn","ts":1526587299.1827736,"caller":"rafthttp/stream.go:436","msg":"lost TC
{"level":"info","ts":1526587299.182845,"caller":"rafthttp/stream.go:459","msg":"stopped
{"level":"warn","ts":1526587299.1830168,"caller":"rafthttp/stream.go:436","msg":"lost TC
{"level":"warn","ts":1526587299.1831107,"caller":"rafthttp/peer_status.go:65","msg":"pee
{"level":"info","ts":1526587299.1831737,"caller":"rafthttp/stream.go:459","msg":"stopped
{"level":"info","ts":1526587299.1832306,"caller":"rafthttp/peer.go:340","msg":"stopped r
{"level":"warn","ts":1526587299.1837125,"caller":"rafthttp/http.go:424","msg":"failed to
{"level":"warn","ts":1526587299.1840093,"caller":"rafthttp/http.go:424","msg":"failed to
{"level":"warn","ts":1526587299.1842315,"caller":"rafthttp/http.go:424","msg":"failed to
{"level":"warn","ts":1526587299.1844475,"caller":"rafthttp/http.go:424","msg":"failed to
{"level":"info","ts":1526587299.2056687,"caller":"embed/etcd.go:473","msg":"stopping ser
{"level":"info","ts":1526587299.205819,"caller":"embed/etcd.go:480","msg":"stopped servi
{"level":"info","ts":1526587299.2058413,"caller":"embed/etcd.go:289","msg":"closed etcd
```

Step 4: restart the etcd server with same configuration

Restart the etcd server with same configuration but with the new etcd binary.

```
-etcd-old --name s1 \
+etcd-new --name s1 \
  --data-dir /tmp/etcd/s1 \
  --listen-client-urls http://localhost:2379 \
  --advertise-client-urls http://localhost:2379 \
  --listen-peer-urls http://localhost:2380 \
  --initial-advertise-peer-urls http://localhost:2380 \
  --initial-cluster s1=http://localhost:2380,s2=http://localhost:22380,s3=http://localhos
  --initial-cluster-token tkn \
```

```
  --initial-cluster-state new
```

The new v3.5 etcd will publish its information to the cluster. At this point, cluster still operates as v3.4 protocol, which is the lowest common version.

> `{"level":"info","ts":1526586617.1647713,"caller":"membership/cluster.go:485","msg":"set initial cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"7339c4e5e833c029","cluster-version":"3.0"}`

> `{"level":"info","ts":1526586617.1648536,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.0"}`

> `{"level":"info","ts":1526586617.1649303,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"7339c4e5e833c029","from":"3.0","from":"3.4"}`

> `{"level":"info","ts":1526586617.1649797,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.4"}`

> `{"level":"info","ts":1526586617.2107732,"caller":"etcdserver/server.go:1770","msg":"published local member to cluster through raft","local-member-id":"7339c4e5e833c029","local-member-attributes":"{Name:s1 ClientURLs:[http://localhost:2379]}","request-path":"/0/members/7339c4e5e833c029/attributes","cluster-id":"7dee9ba76d59ed53","publish-timeout":7}`

Verify that each member, and then the entire cluster, becomes healthy with the new v3.5 etcd binary:

```
etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,localhost:32379
<<COMMENT
localhost:32379 is healthy: successfully committed proposal: took = 2.337471ms
localhost:22379 is healthy: successfully committed proposal: took = 1.130717ms
localhost:2379 is healthy: successfully committed proposal: took = 2.124843ms
COMMENT
```

Un-upgraded members will log warnings like the following until the entire cluster is upgraded.

This is expected and will cease after all etcd cluster members are upgraded to v3.5:

```
:41.942121 W | etcdserver: member 7339c4e5e833c029 has a higher version 3.5.0
:45.945154 W | etcdserver: the local etcd version 3.4.0 is not up-to-date
```

Step 5: repeat *step 3* and *step 4* for rest of the members

When all members are upgraded, the cluster will report upgrading to 3.5 successfully:

Member 1:

> `{"level":"info","ts":1526586949.0920913,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.5"}`
> `{"level":"info","ts":1526586949.0921566,"caller":"etcdserver/server.go:2272","msg":"cluster version is updated","cluster-version":"3.5"}`

Member 2:

> `{"level":"info","ts":1526586949.092117,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"729934363faa4a24","from":"3.4","from":"3.5"}`
> `{"level":"info","ts":1526586949.0923078,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.5"}`

Member 3:

> `{"level":"info","ts":1526586949.0921423,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"b548c2511513015","from":"3.4","from":"3.5"}`
> `{"level":"info","ts":1526586949.0922918,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.5"}`


```
endpoint health --endpoints=localhost:2379,localhost:22379,localhost:32379
<<COMMENT
localhost:2379 is healthy: successfully committed proposal: took = 492.834µs
localhost:22379 is healthy: successfully committed proposal: took = 1.015025ms
localhost:32379 is healthy: successfully committed proposal: took = 1.853077ms
COMMENT

curl http://localhost:2379/version
<<COMMENT
{"etcdserver":"3.5.0","etcdcluster":"3.5.0"}
COMMENT

curl http://localhost:22379/version
<<COMMENT
{"etcdserver":"3.5.0","etcdcluster":"3.5.0"}
COMMENT

curl http://localhost:32379/version
<<COMMENT
{"etcdserver":"3.5.0","etcdcluster":"3.5.0"}
```

```
COMMENT
```

Last modified August 19, 2023: etcd-io/website#479 Use new and better canonical link to Google Groups (cd8b01f)

Upgrade etcd from 3.3 to 3.4

Processes, checklists, and notes on upgrading etcd from 3.3 to 3.4

In the general case, upgrading from etcd 3.3 to 3.4 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v3.3 processes and replace them with etcd v3.4 processes
- after running all v3.4 processes, new features in v3.4 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Highlighted breaking changes in 3.4.

Make `ETCDCTL_API=3 etcdctl` default

`ETCDCTL_API=3` is now the default.

```
etcdctl set foo bar
Error: unknown command "set" for "etcdctl"

-etcdctl set foo bar
+ETCDCTL_API=2 etcdctl set foo bar
bar

ETCDCTL_API=3 etcdctl put foo bar
OK

-ETCDCTL_API=3 etcdctl put foo bar
+etcdctl put foo bar
```

Make `etcd --enable-v2=false` default

[`etcd --enable-v2=false`] is now the default.

This means, unless `etcd --enable-v2=true` is specified, etcd v3.4 server would not serve v2 API requests.

If v2 API were used, make sure to enable v2 API in v3.4:

```
-etcd
+etcd --enable-v2=true
```

Other HTTP APIs will still work (e.g. `[CLIENT-URL]/metrics`, `[CLIENT-URL]/health`, v3 gRPC gateway).

Deprecated `etcd --ca-file` and `etcd --peer-ca-file` flags

`--ca-file` and `--peer-ca-file` flags are deprecated; they have been deprecated since v2.1.

Note setting this parameter will also automatically enable client cert authentication no matter what value is set for `--client-cert-auth`.

```
-etcd --ca-file ca-client.crt
+etcd --trusted-ca-file ca-client.crt
```

```
-etcd --peer-ca-file ca-peer.crt
+etcd --peer-trusted-ca-file ca-peer.crt
```

Deprecated `grpc.ErrClientConnClosing` error

`grpc.ErrClientConnClosing` has been deprecated in gRPC >= 1.10.

```
import (
+    "go.etcd.io/etcd/clientv3"

     "google.golang.org/grpc"
+    "google.golang.org/grpc/codes"
+    "google.golang.org/grpc/status"
)
```

```
_, err := kvc.Get(ctx, "a")
-if err == grpc.ErrClientConnClosing {
+if clientv3.IsConnCanceled(err) {

// or
+s, ok := status.FromError(err)
+if ok {
+  if s.Code() == codes.Canceled
```

Require `grpc.WithBlock` for client dial

[The new client balancer](https://etcd.io/) uses an asynchronous resolver to pass endpoints to the gRPC dial function. As a result, v3.4 client requires `grpc.WithBlock` dial option to wait until the underlying connection is up.

```
import (
    "time"
    "go.etcd.io/etcd/clientv3"
+   "google.golang.org/grpc"
)

+// "grpc.WithBlock()" to block until the underlying connection is up
ccfg := clientv3.Config{
  Endpoints:            []string{"localhost:2379"},
  DialTimeout:          time.Second,
+ DialOptions:          []grpc.DialOption{grpc.WithBlock()},
  DialKeepAliveTime:    time.Second,
  DialKeepAliveTimeout: 500 * time.Millisecond,
}
```

Deprecating `etcd_debugging_mvcc_db_total_size_in_bytes` Prometheus metrics

v3.4 promotes `etcd_debugging_mvcc_db_total_size_in_bytes` Prometheus metrics to `etcd_mvcc_db_total_size_in_bytes`, in order to encourage etcd storage monitoring.

`etcd_debugging_mvcc_db_total_size_in_bytes` is still served in v3.4 for backward compatibilities. It will be completely deprecated in v3.5.

```
-etcd_debugging_mvcc_db_total_size_in_bytes
+etcd_mvcc_db_total_size_in_bytes
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecating `etcd_debugging_mvcc_put_total` Prometheus metrics

v3.4 promotes `etcd_debugging_mvcc_put_total` Prometheus metrics to `etcd_mvcc_put_total`, in order to encourage etcd storage monitoring.

`etcd_debugging_mvcc_put_total` is still served in v3.4 for backward compatibilities. It will be completely deprecated in v3.5.

```
-etcd_debugging_mvcc_put_total
+etcd_mvcc_put_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecating `etcd_debugging_mvcc_delete_total` Prometheus metrics

v3.4 promotes `etcd_debugging_mvcc_delete_total` Prometheus metrics to `etcd_mvcc_delete_total`, in order to encourage etcd storage monitoring.

`etcd_debugging_mvcc_delete_total` is still served in v3.4 for backward compatibilities. It will be completely deprecated in v3.5.

```
-etcd_debugging_mvcc_delete_total
+etcd_mvcc_delete_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecating `etcd_debugging_mvcc_txn_total` Prometheus metrics

v3.4 promotes `etcd_debugging_mvcc_txn_total` Prometheus metrics to `etcd_mvcc_txn_total`, in order to encourage etcd storage monitoring.

`etcd_debugging_mvcc_txn_total` is still served in v3.4 for backward compatibilities. It will be completely deprecated in v3.5.

```
-etcd_debugging_mvcc_txn_total
+etcd_mvcc_txn_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecating `etcd_debugging_mvcc_range_total` Prometheus metrics

v3.4 promotes `etcd_debugging_mvcc_range_total` Prometheus metrics to `etcd_mvcc_range_total`, in order to encourage etcd storage monitoring.

`etcd_debugging_mvcc_range_total` is still served in v3.4 for backward compatibilities. It will be completely deprecated in v3.5.

```
-etcd_debugging_mvcc_range_total
+etcd_mvcc_range_total
```

Note that `etcd_debugging_*` namespace metrics have been marked as experimental. As we improve monitoring guide, we may promote more metrics.

Deprecating `etcd --log-output` flag (now `--log-outputs`)

Rename `etcd --log-output` to `--log-outputs` to support multiple log outputs. **`etcd --logger=capnslog` does not support multiple log outputs.**

`etcd --log-output` will be deprecated in v3.5. **`etcd --logger=capnslog` will be deprecated in v3.5**.

```
-etcd --log-output=stderr
+etcd --log-outputs=stderr

+# to write logs to stderr and a.log file at the same time
+# only "--logger=zap" supports multiple writers
+etcd --logger=zap --log-outputs=stderr,a.log
```

v3.4 adds `etcd --logger=zap --log-outputs=stderr` support for structured logging and multiple log outputs. Main motivation is to promote automated etcd monitoring, rather than looking back server logs when it starts breaking. Future development will make etcd log as few as possible, and make etcd easier to monitor with metrics and alerts. **`etcd --logger=capnslog` will be deprecated in v3.5**.

Changed `log-outputs` field type in `etcd --config-file` to `[]string`

Now that `log-outputs` (old field name `log-output`) accepts multiple writers, etcd configuration YAML file `log-outputs` field must be changed to `[]string` type as below:

```
 # Specify 'stdout' or 'stderr' to skip journald logging even when running under systemd
-log-output: default
+log-outputs: [default]
```

Renamed `embed.Config.LogOutput` to `embed.Config.LogOutputs`

Renamed `embed.Config.LogOutput` to `embed.Config.LogOutputs` to support multiple log outputs. And changed `embed.Config.LogOutput` type from `string` to `[]string` to support multiple log outputs.

```
import "github.com/coreos/etcd/embed"

cfg := &embed.Config{Debug: false}
-cfg.LogOutput = "stderr"
+cfg.LogOutputs = []string{"stderr"}
```

v3.5 deprecates `capnslog`

v3.5 will deprecate `etcd --log-package-levels` flag for `capnslog`; `etcd --logger=zap --log-outputs=stderr` will the default. **v3.5 will deprecate `[CLIENT-URL]/config/local/log` endpoint.**

```
-etcd
+etcd --logger zap
```

Deprecating `etcd --debug` flag (now `--log-level=debug`)

v3.4 deprecates `etcd --debug` flag. Instead, use `etcd --log-level=debug` flag.

```
-etcd --debug
+etcd --logger zap --log-level debug
```

Deprecated `pkg/transport.TLSInfo.CAFile` field

Deprecated `pkg/transport.TLSInfo.CAFile` field.

```
import "github.com/coreos/etcd/pkg/transport"

tlsInfo := transport.TLSInfo{
    CertFile: "/tmp/test-certs/test.pem",
    KeyFile: "/tmp/test-certs/test-key.pem",
-   CAFile: "/tmp/test-certs/trusted-ca.pem",
+   TrustedCAFile: "/tmp/test-certs/trusted-ca.pem",
}
tlsConfig, err := tlsInfo.ClientConfig()
if err != nil {
    panic(err)
}
```

Changed `embed.Config.SnapCount` to `embed.Config.SnapshotCount`

To be consistent with the flag name `etcd --snapshot-count`, `embed.Config.SnapCount` field has been renamed to `embed.Config.SnapshotCount`:

```
import "github.com/coreos/etcd/embed"

cfg := embed.NewConfig()
-cfg.SnapCount = 100000
+cfg.SnapshotCount = 100000
```

Changed `etcdserver.ServerConfig.SnapCount` to `etcdserver.ServerConfig.SnapshotCount`

To be consistent with the flag name `etcd --snapshot-count`, `etcdserver.ServerConfig.SnapCount` field has been renamed to `etcdserver.ServerConfig.SnapshotCount`:

```
import "github.com/coreos/etcd/etcdserver"

srvcfg := etcdserver.ServerConfig{
-  SnapCount: 100000,
+  SnapshotCount: 100000,
```

Changed function signature in package `wal`

Changed `wal` function signatures to support structured logger.

```
import "github.com/coreos/etcd/wal"
+import "go.uber.org/zap"

+lg, _ = zap.NewProduction()

-wal.Open(dirpath, snap)
+wal.Open(lg, dirpath, snap)

-wal.OpenForRead(dirpath, snap)
+wal.OpenForRead(lg, dirpath, snap)

-wal.Repair(dirpath)
+wal.Repair(lg, dirpath)

-wal.Create(dirpath, metadata)
+wal.Create(lg, dirpath, metadata)
```

Changed `IntervalTree` type in package `pkg/adt`

`pkg/adt.IntervalTree` is now defined as an `interface`.

```
import (
    "fmt"

    "go.etcd.io/etcd/pkg/adt"
)

func main() {
-    ivt := &adt.IntervalTree{}
+    ivt := adt.NewIntervalTree()
```

Deprecated `embed.Config.SetupLogging`

`embed.Config.SetupLogging` has been removed in order to prevent wrong logging configuration, and now set up automatically.

```
```

```
import "github.com/coreos/etcd/embed"

cfg := &embed.Config{Debug: false}
-cfg.SetupLogging()
```

Changed gRPC gateway HTTP endpoints (replaced `/v3beta` with `/v3`)

Before

```
curl -L http://localhost:2379/v3beta/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

After

```
curl -L http://localhost:2379/v3/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

Requests to `/v3beta` endpoints will redirect to `/v3`, and `/v3beta` will be removed in 3.5 release.

Deprecated container image tags

`latest` and minor version images tags are deprecated:

```
-docker pull gcr.io/etcd-development/etcd:latest
+docker pull gcr.io/etcd-development/etcd:v3.4.0

-docker pull gcr.io/etcd-development/etcd:v3.4
+docker pull gcr.io/etcd-development/etcd:v3.4.0

-docker pull gcr.io/etcd-development/etcd:v3.4
+docker pull gcr.io/etcd-development/etcd:v3.4.1

-docker pull gcr.io/etcd-development/etcd:v3.4
+docker pull gcr.io/etcd-development/etcd:v3.4.2
```

Server upgrade checklists

Upgrade requirements

To upgrade an existing etcd deployment to 3.4, the running cluster must be 3.3 or greater. If it's before 3.3, please upgrade to 3.3 before upgrading to 3.4.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl endpoint health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, download the snapshot backup. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version. Please note that the `snapshot` command only backs up the v3 data. For v2 data, see backing up v2 datastore.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.4. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

Note: If the cluster only has v3 data and no v2 data, it is not subject to this limitation.

If the cluster is serving a v2 data set larger than 50MB, each newly upgraded member may take up to two minutes to catch up with the existing cluster. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.4, the cluster will be upgraded to v3.4, and downgrade from this completed state is **not possible**. If any single member is still v3.3, however, the cluster and its operations remains “v3.3”, and it is possible from this mixed cluster state to return to using a v3.3 etcd binary on all members.

Please download the snapshot backup to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example shows how to upgrade a 3-member v3.3 etcd cluster running on a local machine.

Step 1: check upgrade requirements

Is the cluster healthy and running v3.3.x?

```
etcdctl --endpoints=localhost:2379,localhost:22379,localhost:32379 endpoint health
<<COMMENT
localhost:2379 is healthy: successfully committed proposal: took = 2.118638ms
localhost:22379 is healthy: successfully committed proposal: took = 3.631388ms
localhost:32379 is healthy: successfully committed proposal: took = 2.157051ms
COMMENT

curl http://localhost:2379/version
<<COMMENT
{"etcdserver":"3.3.5","etcdcluster":"3.3.0"}
COMMENT

curl http://localhost:22379/version
<<COMMENT
{"etcdserver":"3.3.5","etcdcluster":"3.3.0"}
COMMENT

curl http://localhost:32379/version
<<COMMENT
{"etcdserver":"3.3.5","etcdcluster":"3.3.0"}
COMMENT
```

Step 2: download snapshot backup from leader

Download the snapshot backup to provide a downgrade path should any problems occur.

etcd leader is guaranteed to have the latest application data, thus fetch snapshot from leader:

```
curl -sL http://localhost:2379/metrics | grep etcd_server_is_leader
<<COMMENT
# HELP etcd_server_is_leader Whether or not this member is a leader. 1 if is, 0 otherwise
# TYPE etcd_server_is_leader gauge
etcd_server_is_leader 1
COMMENT

curl -sL http://localhost:22379/metrics | grep etcd_server_is_leader
<<COMMENT
etcd_server_is_leader 0
COMMENT

curl -sL http://localhost:32379/metrics | grep etcd_server_is_leader
<<COMMENT
etcd_server_is_leader 0
COMMENT

etcdctl --endpoints=localhost:2379 snapshot save backup.db
<<COMMENT
{"level":"info","ts":1526585787.148433,"caller":"snapshot/v3_snapshot.go:109","msg":"crea
{"level":"info","ts":1526585787.1485257,"caller":"snapshot/v3_snapshot.go:120","msg":"fet
{"level":"info","ts":1526585787.1519694,"caller":"snapshot/v3_snapshot.go:133","msg":"fet
{"level":"info","ts":1526585787.1520295,"caller":"snapshot/v3_snapshot.go:142","msg":"sav
Snapshot saved at backup.db
COMMENT
```

Step 3: stop one existing etcd server

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
10.237579 I | etcdserver: updating the cluster version from 3.0 to 3.3
10.238315 N | etcdserver/membership: updated the cluster version from 3.0 to 3.3
10.238451 I | etcdserver/api: enabled capabilities for version 3.3


^C21.192174 N | pkg/osutil: received interrupt signal, shutting down...
21.192459 I | etcdserver: 7339c4e5e833c029 starts leadership transfer from 7339c4e5e833c0
21.192569 I | raft: 7339c4e5e833c029 [term 8] starts to transfer leadership to 729934363
21.192619 I | raft: 7339c4e5e833c029 sends MsgTimeoutNow to 729934363faa4a24 immediately
WARNING: 2018/05/17 12:45:21 grpc: addrConn.resetTransport failed to create client trans
WARNING: 2018/05/17 12:45:21 grpc: addrConn.transportMonitor exits due to: grpc: the conn
21.193589 I | raft: 7339c4e5e833c029 [term: 8] received a MsgVote message with higher te
21.193626 I | raft: 7339c4e5e833c029 became follower at term 9
21.193651 I | raft: 7339c4e5e833c029 [logterm: 8, index: 9, vote: 0] cast MsgVote for 72
21.193675 I | raft: raft.node: 7339c4e5e833c029 lost leader 7339c4e5e833c029 at term 9
```

```
21.194424 I | raft: raft.node: 7339c4e5e833c029 elected leader 729934363faa4a24 at term 
21.292898 I | etcdserver: 7339c4e5e833c029 finished leadership transfer from 7339c4e5e83
21.292975 I | rafthttp: stopping peer 729934363faa4a24...
21.293206 I | rafthttp: closed the TCP streaming connection with peer 729934363faa4a24 (
21.293225 I | rafthttp: stopped streaming with peer 729934363faa4a24 (writer)
21.293437 I | rafthttp: closed the TCP streaming connection with peer 729934363faa4a24 (
21.293459 I | rafthttp: stopped streaming with peer 729934363faa4a24 (writer)
21.293514 I | rafthttp: stopped HTTP pipelining with peer 729934363faa4a24
21.293590 W | rafthttp: lost the TCP streaming connection with peer 729934363faa4a24 (st
21.293610 I | rafthttp: stopped streaming with peer 729934363faa4a24 (stream MsgApp v2 r
21.293680 W | rafthttp: lost the TCP streaming connection with peer 729934363faa4a24 (st
21.293700 I | rafthttp: stopped streaming with peer 729934363faa4a24 (stream Message rea
21.293711 I | rafthttp: stopped peer 729934363faa4a24
21.293720 I | rafthttp: stopping peer b548c2511513015...
21.293987 I | rafthttp: closed the TCP streaming connection with peer b548c2511513015 (s
21.294063 I | rafthttp: stopped streaming with peer b548c2511513015 (writer)
21.294467 I | rafthttp: closed the TCP streaming connection with peer b548c2511513015 (s
21.294561 I | rafthttp: stopped streaming with peer b548c2511513015 (writer)
21.294742 I | rafthttp: stopped HTTP pipelining with peer b548c2511513015
21.294867 W | rafthttp: lost the TCP streaming connection with peer b548c2511513015 (str
21.294892 I | rafthttp: stopped streaming with peer b548c2511513015 (stream MsgApp v2 re
21.294990 W | rafthttp: lost the TCP streaming connection with peer b548c2511513015 (str
21.295004 E | rafthttp: failed to read b548c2511513015 on stream Message (context cancel
21.295013 I | rafthttp: peer b548c2511513015 became inactive
21.295024 I | rafthttp: stopped streaming with peer b548c2511513015 (stream Message read
21.295035 I | rafthttp: stopped peer b548c2511513015
```

Step 4: restart the etcd server with same configuration

Restart the etcd server with same configuration but with the new etcd binary.

```
-etcd-old --name s1 \
+etcd-new --name s1 \
  --data-dir /tmp/etcd/s1 \
  --listen-client-urls http://localhost:2379 \
  --advertise-client-urls http://localhost:2379 \
  --listen-peer-urls http://localhost:2380 \
  --initial-advertise-peer-urls http://localhost:2380 \
  --initial-cluster s1=http://localhost:2380,s2=http://localhost:22380,s3=http://localhos
  --initial-cluster-token tkn \
+ --initial-cluster-state new \
+ --logger zap \
+ --log-outputs stderr
```

The new v3.4 etcd will publish its information to the cluster. At this point, cluster still operates as v3.3 protocol, which is the lowest common version.

> `{"level":"info","ts":1526586617.1647713,"caller":"membership/cluster.go:485","msg":"set initial cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"7339c4e5e833c029","cluster-version":"3.0"}`

> `{"level":"info","ts":1526586617.1648536,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.0"}`

> `{"level":"info","ts":1526586617.1649303,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"7339c4e5e833c029","from":"3.0","from":"3.3"}`

> `{"level":"info","ts":1526586617.1649797,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.3"}`

> `{"level":"info","ts":1526586617.2107732,"caller":"etcdserver/server.go:1770","msg":"published local member to cluster through raft","local-member-id":"7339c4e5e833c029","local-member-attributes":"{Name:s1 ClientURLs:[http://localhost:2379]}","request-path":"/0/members/7339c4e5e833c029/attributes","cluster-id":"7dee9ba76d59ed53","publish-timeout":7}`

Verify that each member, and then the entire cluster, becomes healthy with the new v3.4 etcd binary:

```
etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,localhost:32379
<<COMMENT
localhost:32379 is healthy: successfully committed proposal: took = 2.337471ms
localhost:22379 is healthy: successfully committed proposal: took = 1.130717ms
localhost:2379 is healthy: successfully committed proposal: took = 2.124843ms
COMMENT
```

Un-upgraded members will log warnings like the following until the entire cluster is upgraded.

This is expected and will cease after all etcd cluster members are upgraded to v3.4:

```
:41.942121 W | etcdserver: member 7339c4e5e833c029 has a higher version 3.4.0
:45.945154 W | etcdserver: the local etcd version 3.3.5 is not up-to-date
```

Step 5: repeat *step 3* and *step 4* for rest of the members

When all members are upgraded, the cluster will report upgrading to 3.4 successfully:

Member 1:

> `{"level":"info","ts":1526586949.0920913,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.4"}`
> `{"level":"info","ts":1526586949.0921566,"caller":"etcdserver/server.go:2272","msg":"cluster version is updated","cluster-version":"3.4"}`

Member 2:

> `{"level":"info","ts":1526586949.092117,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"729934363faa4a24","from":"3.3","from":"3.4"}`
> `{"level":"info","ts":1526586949.0923078,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.4"}`

Member 3:

> `{"level":"info","ts":1526586949.0921423,"caller":"membership/cluster.go:473","msg":"updated cluster version","cluster-id":"7dee9ba76d59ed53","local-member-id":"b548c2511513015","from":"3.3","from":"3.4"}`
> `{"level":"info","ts":1526586949.0922918,"caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.4"}`

```
endpoint health --endpoints=localhost:2379,localhost:22379,localhost:32379
<<COMMENT
localhost:2379 is healthy: successfully committed proposal: took = 492.834µs
localhost:22379 is healthy: successfully committed proposal: took = 1.015025ms
localhost:32379 is healthy: successfully committed proposal: took = 1.853077ms
COMMENT

curl http://localhost:2379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT

curl http://localhost:22379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT

curl http://localhost:32379/version
<<COMMENT
{"etcdserver":"3.4.0","etcdcluster":"3.4.0"}
COMMENT
```

Last modified September 29, 2023: Added note that --trusted-ca-file also enables client cert authentication Co-authored-by: James Blair <mail@jamesblair.net> (a412dd6)

Upgrade etcd from 3.2 to 3.3

Processes, checklists, and notes on upgrading etcd from 3.2 to 3.3

In the general case, upgrading from etcd 3.2 to 3.3 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v3.2 processes and replace them with etcd v3.3 processes
- after running all v3.3 processes, new features in v3.3 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Highlighted breaking changes in 3.3.

Changed value type of `etcd --auto-compaction-retention` flag to `string`

Changed `--auto-compaction-retention` flag to accept string values with finer granularity. Now that `--auto-compaction-retention` accepts string values, etcd configuration YAML file `auto-compaction-retention` field must be changed to `string` type. Previously, `--config-file etcd.config.yaml` can have `auto-compaction-retention: 24` field, now must be `auto-compaction-retention: "24"` or `auto-compaction-retention: "24h"`. If configured as `--auto-compaction-mode periodic --auto-compaction-retention "24h"`, the time duration value for `--auto-compaction-retention` flag must be valid for `time.ParseDuration` function in Go.

```
# etcd.config.yaml
+auto-compaction-mode: periodic
-auto-compaction-retention: 24
+auto-compaction-retention: "24"
+# Or
+auto-compaction-retention: "24h"
```

Changed `etcdserver.EtcdServer.ServerConfig` to `*etcdserver.EtcdServer.ServerConfig`

`etcdserver.EtcdServer` has changed the type of its member field `*etcdserver.ServerConfig` to `etcdserver.ServerConfig`. And `etcdserver.NewServer` now takes `etcdserver.ServerConfig`, instead of `*etcdserver.ServerConfig`.

Before and after (e.g. k8s.io/kubernetes/test/e2e_node/services/etcd.go)

```
import "github.com/coreos/etcd/etcdserver"

type EtcdServer struct {
  *etcdserver.EtcdServer
-   config *etcdserver.ServerConfig
+   config etcdserver.ServerConfig
}

func NewEtcd(dataDir string) *EtcdServer {
-   config := &etcdserver.ServerConfig{
+   config := etcdserver.ServerConfig{
    DataDir: dataDir,
        ...
  }
  return &EtcdServer{config: config}
}

func (e *EtcdServer) Start() error {
  var err error
  e.EtcdServer, err = etcdserver.NewServer(e.config)
    ...
```

Added `embed.Config.LogOutput` struct

Note that this field has been renamed to `embed.Config.LogOutputs` in `[]string` type in v3.4. Please see v3.4 upgrade guide for more details.

Field `LogOutput` is added to `embed.Config`:

```
package embed

type Config struct {
   Debug bool `json:"debug"`
   LogPkgLevels string `json:"log-package-levels"`
+   LogOutput string `json:"log-output"`
   ...
```

Before gRPC server warnings were logged in etcdserver.

```
WARNING: 2017/11/02 11:35:51 grpc: addrConn.resetTransport failed to create client transp
WARNING: 2017/11/02 11:35:51 grpc: addrConn.resetTransport failed to create client transp
```

From v3.3, gRPC server logs are disabled by default.

Note that `embed.Config.SetupLogging` method has been deprecated in v3.4. Please see [v3.4 upgrade guide](#) for more details.

```go
import "github.com/coreos/etcd/embed"

cfg := &embed.Config{Debug: false}
cfg.SetupLogging()
```

Set `embed.Config.Debug` field to `true` to enable gRPC server logs.

Changed `/health` endpoint response

Previously, `[endpoint]:[client-port]/health` returned manually marshaled JSON value. 3.3 now defines `etcdhttp.Health` struct.

Note that in v3.3.0-rc.0, v3.3.0-rc.1, and v3.3.0-rc.2, `etcdhttp.Health` has boolean type `"health"` and `"errors"` fields. For backward compatibilities, we reverted `"health"` field to `string` type and removed `"errors"` field. Further health information will be provided in separate APIs.

```
$ curl http://localhost:2379/health
{"health":"true"}
```

Changed gRPC gateway HTTP endpoints (replaced `/v3alpha` with `/v3beta`)

Before

```
curl -L http://localhost:2379/v3alpha/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

After

```
curl -L http://localhost:2379/v3beta/kv/put \
  -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
```

Requests to `/v3alpha` endpoints will redirect to `/v3beta`, and `/v3alpha` will be removed in 3.4 release.

Changed maximum request size limits

3.3 now allows custom request size limits for both server and **client side**. In previous versions(v3.2.10, v3.2.11), client response size was limited to only 4 MiB.

Server-side request limits can be configured with `--max-request-bytes` flag:

```
# limits request size to 1.5 KiB
etcd --max-request-bytes 1536

# client writes exceeding 1.5 KiB will be rejected
etcdctl put foo [LARGE VALUE...]
# etcdserver: request is too large
```

Or configure `embed.Config.MaxRequestBytes` field:

```
```

```
import "github.com/coreos/etcd/embed"
import "github.com/coreos/etcd/etcdserver/api/v3rpc/rpctypes"

// limit requests to 5 MiB
cfg := embed.NewConfig()
cfg.MaxRequestBytes = 5 * 1024 * 1024

// client writes exceeding 5 MiB will be rejected
_, err := cli.Put(ctx, "foo", [LARGE VALUE...])
err == rpctypes.ErrRequestTooLarge
```

If not specified, server-side limit defaults to 1.5 MiB.

Client-side request limits must be configured based on server-side limits.

```
# limits request size to 1 MiB
etcd --max-request-bytes 1048576
```

```
import "github.com/coreos/etcd/clientv3"

cli, _ := clientv3.New(clientv3.Config{
    Endpoints: []string{"127.0.0.1:2379"},
    MaxCallSendMsgSize: 2 * 1024 * 1024,
    MaxCallRecvMsgSize: 3 * 1024 * 1024,
})


// client writes exceeding "--max-request-bytes" will be rejected from etcd server
_, err := cli.Put(ctx, "foo", strings.Repeat("a", 1*1024*1024+5))
err == rpctypes.ErrRequestTooLarge


// client writes exceeding "MaxCallSendMsgSize" will be rejected from client-side
_, err = cli.Put(ctx, "foo", strings.Repeat("a", 5*1024*1024))
err.Error() == "rpc error: code = ResourceExhausted desc = grpc: trying to send message 


// some writes under limits
for i := range []int{0,1,2,3,4} {
    _, err = cli.Put(ctx, fmt.Sprintf("foo%d", i), strings.Repeat("a", 1*1024*1024-500))
    if err != nil {
        panic(err)
    }
}
// client reads exceeding "MaxCallRecvMsgSize" will be rejected from client-side
```

```
_, err = cli.Get(ctx, "foo", clientv3.WithPrefix())
err.Error() == "rpc error: code = ResourceExhausted desc = grpc: received message larger
```

If not specified, client-side send limit defaults to 2 MiB (1.5 MiB + gRPC overhead bytes) and receive limit to `math.MaxInt32`. Please see clientv3 godoc for more detail.

Changed raw gRPC client wrapper function signatures

3.3 changes the function signatures of `clientv3` gRPC client wrapper. This change was needed to support custom `grpc.CallOption` on message size limits.

Before and after

```
-func NewKVFromKVClient(remote pb.KVClient) KV {
+func NewKVFromKVClient(remote pb.KVClient, c *Client) KV {

-func NewClusterFromClusterClient(remote pb.ClusterClient) Cluster {
+func NewClusterFromClusterClient(remote pb.ClusterClient, c *Client) Cluster {

-func NewLeaseFromLeaseClient(remote pb.LeaseClient, keepAliveTimeout time.Duration) Lea
+func NewLeaseFromLeaseClient(remote pb.LeaseClient, c *Client, keepAliveTimeout time.Du

-func NewMaintenanceFromMaintenanceClient(remote pb.MaintenanceClient) Maintenance {
+func NewMaintenanceFromMaintenanceClient(remote pb.MaintenanceClient, c *Client) Mainte

-func NewWatchFromWatchClient(wc pb.WatchClient) Watcher {
+func NewWatchFromWatchClient(wc pb.WatchClient, c *Client) Watcher {
```

Changed clientv3 `Snapshot` API error type

Previously, clientv3 `Snapshot` API returned raw [`grpc/*status.statusError`] type error. v3.3 now translates those errors to corresponding public error types, to be consistent with other APIs.

Before

```
import "context"

// reading snapshot with canceled context should error out
ctx, cancel := context.WithCancel(context.Background())
rc, _ := cli.Snapshot(ctx)
cancel()
_, err := io.Copy(f, rc)
```

```
err.Error() == "rpc error: code = Canceled desc = context canceled"

// reading snapshot with deadline exceeded should error out
ctx, cancel = context.WithTimeout(context.Background(), time.Second)
defer cancel()
rc, _ = cli.Snapshot(ctx)
time.Sleep(2 * time.Second)
_, err = io.Copy(f, rc)
err.Error() == "rpc error: code = DeadlineExceeded desc = context deadline exceeded"
```

After

```
import "context"

// reading snapshot with canceled context should error out
ctx, cancel := context.WithCancel(context.Background())
rc, _ := cli.Snapshot(ctx)
cancel()
_, err := io.Copy(f, rc)
err == context.Canceled

// reading snapshot with deadline exceeded should error out
ctx, cancel = context.WithTimeout(context.Background(), time.Second)
defer cancel()
rc, _ = cli.Snapshot(ctx)
time.Sleep(2 * time.Second)
_, err = io.Copy(f, rc)
err == context.DeadlineExceeded
```

Changed `etcdctl lease timetolive` command output

Previously, `lease timetolive LEASE_ID` command on expired lease prints `-1s` for remaining seconds. 3.3 now outputs clearer messages.

Before

```
lease 2d8257079fa1bc0c granted with TTL(0s), remaining(-1s)
```

After

```
```

```
lease 2d8257079fa1bc0c already expired
```

Changed `golang.org/x/net/context` imports

`clientv3` has deprecated `golang.org/x/net/context`. If a project vendors `golang.org/x/net/context` in other code (e.g. etcd generated protocol buffer code) and imports `github.com/coreos/etcd/clientv3`, it requires Go 1.9+ to compile.

Before

```
import "golang.org/x/net/context"
cli.Put(context.Background(), "f", "v")
```

After

```
import "context"
cli.Put(context.Background(), "f", "v")
```

Changed gRPC dependency

3.3 now requires grpc/grpc-go `v1.7.5`.

Deprecated `grpclog.Logger`

`grpclog.Logger` has been deprecated in favor of `grpclog.LoggerV2`. `clientv3.Logger` is now `grpclog.LoggerV2`.

Before

```
import "github.com/coreos/etcd/clientv3"
clientv3.SetLogger(log.New(os.Stderr, "grpc: ", 0))
```

After

```
import "github.com/coreos/etcd/clientv3"
import "google.golang.org/grpc/grpclog"
clientv3.SetLogger(grpclog.NewLoggerV2(os.Stderr, os.Stderr, os.Stderr))
```

```
// log.New above cannot be used (not implement grpclog.LoggerV2 interface)
```

Deprecated `grpc.ErrClientConnTimeout`

Previously, `grpc.ErrClientConnTimeout` error is returned on client dial time-outs. 3.3 instead returns `context.DeadlineExceeded` (see #8504).

Before

```
// expect dial time-out on ipv4 blackhole
_, err := clientv3.New(clientv3.Config{
    Endpoints:   []string{"http://254.0.0.1:12345"},
    DialTimeout: 2 * time.Second
})
if err == grpc.ErrClientConnTimeout {
  // handle errors
}
```

After

```
_, err := clientv3.New(clientv3.Config{
    Endpoints:   []string{"http://254.0.0.1:12345"},
    DialTimeout: 2 * time.Second
})
if err == context.DeadlineExceeded {
  // handle errors
}
```

Changed official container registry

etcd now uses `gcr.io/etcd-development/etcd` as a primary container registry, and `quay.io/coreos/etcd` as secondary.

Before

```
docker pull quay.io/coreos/etcd:v3.2.5
```

After

```
docker pull gcr.io/etcd-development/etcd:v3.3.0
```

Upgrades to >= v3.3.14

v3.3.14 had to include some features from 3.4, while trying to minimize the difference between client balancer implementation. This release fixes "kube-apiserver 1.13.x refuses to work when first etcd-server is not available" (kubernetes#72102).

`grpc.ErrClientConnClosing` has been deprecated in gRPC >= 1.10.

```
import (
+   "go.etcd.io/etcd/clientv3"

  "google.golang.org/grpc"
+   "google.golang.org/grpc/codes"
+   "google.golang.org/grpc/status"
)

_, err := kvc.Get(ctx, "a")
-if err == grpc.ErrClientConnClosing {
+if clientv3.IsConnCanceled(err) {

// or
+s, ok := status.FromError(err)
+if ok {
+  if s.Code() == codes.Canceled
```

The new client balancer uses an asynchronous resolver to pass endpoints to the gRPC dial function. As a result, v3.3.14 or later requires `grpc.WithBlock` dial option to wait until the underlying connection is up.

```
import (
    "time"
    "go.etcd.io/etcd/clientv3"
+   "google.golang.org/grpc"
)

+// "grpc.WithBlock()" to block until the underlying connection is up
ccfg := clientv3.Config{
  Endpoints:            []string{"localhost:2379"},
  DialTimeout:          time.Second,
+ DialOptions:          []grpc.DialOption{grpc.WithBlock()},
```

```
  DialKeepAliveTime:    time.Second,
  DialKeepAliveTimeout: 500 * time.Millisecond,
}
```

Please see CHANGELOG for a full list of changes.

Server upgrade checklists

Upgrade requirements

To upgrade an existing etcd deployment to 3.3, the running cluster must be 3.2 or greater. If it's before 3.2, please upgrade to 3.2 before upgrading to 3.3.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl endpoint health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, backup the etcd data. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version. Please note that the `snapshot` command only backs up the v3 data. For v2 data, see backing up v2 datastore.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.3. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

Note: If the cluster only has v3 data and no v2 data, it is not subject to this limitation.

If the cluster is serving a v2 data set larger than 50MB, each newly upgraded member may take up to two minutes to catch up with the existing cluster. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.3, the cluster will be upgraded to v3.3, and downgrade from this completed state is **not possible**. If any single member is still v3.2, however, the cluster and its operations remains "v3.2", and it is possible from this mixed cluster state to return to using a v3.2 etcd binary on all members.

Please backup the data directory of all etcd members to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example shows how to upgrade a 3-member v3.2 etcd cluster running on a local machine.

1. Check upgrade requirements

Is the cluster healthy and running v3.2.x?

```
$ ETCDCTL_API=3 etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,locall
localhost:2379 is healthy: successfully committed proposal: took = 6.600684ms
localhost:22379 is healthy: successfully committed proposal: took = 8.540064ms
localhost:32379 is healthy: successfully committed proposal: took = 8.763432ms

$ curl http://localhost:2379/version
{"etcdserver":"3.2.7","etcdcluster":"3.2.0"}
```

2. Stop the existing etcd process

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
14:13:31.491746 I | raft: c89feb932daef420 [term 3] received MsgTimeoutNow from 6d4f535ba
14:13:31.491769 I | raft: c89feb932daef420 became candidate at term 4
14:13:31.491788 I | raft: c89feb932daef420 received MsgVoteResp from c89feb932daef420 at
14:13:31.491797 I | raft: c89feb932daef420 [logterm: 3, index: 9] sent MsgVote request to
14:13:31.491805 I | raft: c89feb932daef420 [logterm: 3, index: 9] sent MsgVote request to
14:13:31.491815 I | raft: raft.node: c89feb932daef420 lost leader 6d4f535bae3ab960 at ter
14:13:31.524084 I | raft: c89feb932daef420 received MsgVoteResp from 6d4f535bae3ab960 at
```

```
14:13:31.524108 I | raft: c89feb932daef420 [quorum:2] has received 2 MsgVoteResp votes ar
14:13:31.524123 I | raft: c89feb932daef420 became leader at term 4
14:13:31.524136 I | raft: raft.node: c89feb932daef420 elected leader c89feb932daef420 at
14:13:31.592650 W | rafthttp: lost the TCP streaming connection with peer 6d4f535bae3ab96
14:13:31.592825 W | rafthttp: lost the TCP streaming connection with peer 6d4f535bae3ab96
14:13:31.693275 E | rafthttp: failed to dial 6d4f535bae3ab960 on stream Message (dial tc
14:13:31.693289 I | rafthttp: peer 6d4f535bae3ab960 became inactive
14:13:31.936678 W | rafthttp: lost the TCP streaming connection with peer 6d4f535bae3ab96
```

It's a good idea at this point to backup the etcd data to provide a downgrade path should any problems occur:

```
$ etcdctl snapshot save backup.db
```

3. Drop-in etcd v3.3 binary and start the new etcd process

The new v3.3 etcd will publish its information to the cluster:

```
14:14:25.363225 I | etcdserver: published {Name:s1 ClientURLs:[http://localhost:2379]} to
```

Verify that each member, and then the entire cluster, becomes healthy with the new v3.3 etcd binary:

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,local
localhost:22379 is healthy: successfully committed proposal: took = 5.540129ms
localhost:32379 is healthy: successfully committed proposal: took = 7.321771ms
localhost:2379 is healthy: successfully committed proposal: took = 10.629901ms
```

Upgraded members will log warnings like the following until the entire cluster is upgraded. This is expected and will cease after all etcd cluster members are upgraded to v3.3:

```
14:15:17.071804 W | etcdserver: member c89feb932daef420 has a higher version 3.3.0
14:15:21.073110 W | etcdserver: the local etcd version 3.2.7 is not up-to-date
14:15:21.073142 W | etcdserver: member 6d4f535bae3ab960 has a higher version 3.3.0
14:15:21.073157 W | etcdserver: the local etcd version 3.2.7 is not up-to-date
14:15:21.073164 W | etcdserver: member c89feb932daef420 has a higher version 3.3.0
```

4. Repeat step 2 to step 3 for all other members

5. Finish

When all members are upgraded, the cluster will report upgrading to 3.3 successfully:

```
14:15:54.536901 N | etcdserver/membership: updated the cluster version from 3.2 to 3.3
14:15:54.537035 I | etcdserver/api: enabled capabilities for version 3.3
```

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,local
localhost:2379 is healthy: successfully committed proposal: took = 2.312897ms
localhost:22379 is healthy: successfully committed proposal: took = 2.553476ms
localhost:32379 is healthy: successfully committed proposal: took = 2.517902ms
```

Last modified August 19, 2023: etcd-io/website#479 Use new and better canonical link to Google Groups (cd8b01f)

Upgrade etcd from 3.1 to 3.2

Processes, checklists, and notes on upgrading etcd from 3.1 to 3.2

In the general case, upgrading from etcd 3.1 to 3.2 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v3.1 processes and replace them with etcd v3.2 processes
- after running all v3.2 processes, new features in v3.2 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Highlighted breaking changes in 3.2.

Changed default `snapshot-count` value

Higher `--snapshot-count` holds more Raft entries in memory until snapshot, thus causing recurrent higher memory usage. Since leader retains latest Raft entries for longer, a slow follower has more time to catch up before leader snapshot. `--snapshot-count` is a tradeoff between higher memory usage and better availabilities of slow followers.

Since v3.2, the default value of `--snapshot-count` has changed from from 10,000 to 100,000.

Changed gRPC dependency (>=3.2.10)

3.2.10 or later now requires grpc/grpc-go `v1.7.5` (<=3.2.9 requires `v1.2.1`).

Deprecated `grpclog.Logger`

`grpclog.Logger` has been deprecated in favor of `grpclog.LoggerV2`. `clientv3.Logger` is now `grpclog.LoggerV2`.

Before

```
import "github.com/coreos/etcd/clientv3"
clientv3.SetLogger(log.New(os.Stderr, "grpc: ", 0))
```

After

```
import "github.com/coreos/etcd/clientv3"
import "google.golang.org/grpc/grpclog"
clientv3.SetLogger(grpclog.NewLoggerV2(os.Stderr, os.Stderr, os.Stderr))

// log.New above cannot be used (not implement grpclog.LoggerV2 interface)
```

Deprecated `grpc.ErrClientConnTimeout`

Previously, `grpc.ErrClientConnTimeout` error is returned on client dial time-outs. 3.2 instead returns `context.DeadlineExceeded` (see #8504).

Before

```
// expect dial time-out on ipv4 blackhole
_, err := clientv3.New(clientv3.Config{
    Endpoints:   []string{"http://254.0.0.1:12345"},
    DialTimeout: 2 * time.Second
})
if err == grpc.ErrClientConnTimeout {
    // handle errors
}
```

After

```
_, err := clientv3.New(clientv3.Config{
    Endpoints:   []string{"http://254.0.0.1:12345"},
    DialTimeout: 2 * time.Second
})
if err == context.DeadlineExceeded {
    // handle errors
}
```

Changed maximum request size limits (>=3.2.10)

3.2.10 and 3.2.11 allow custom request size limits in server side. >=3.2.12 allows custom request size limits for both server and **client side**. In previous versions(v3.2.10, v3.2.11), client response size was limited to only 4 MiB.

Server-side request limits can be configured with `--max-request-bytes` flag:

```
# limits request size to 1.5 KiB
etcd --max-request-bytes 1536

# client writes exceeding 1.5 KiB will be rejected
etcdctl put foo [LARGE VALUE...]
# etcdserver: request is too large
```

Or configure `embed.Config.MaxRequestBytes` field:

```
import "github.com/coreos/etcd/embed"
import "github.com/coreos/etcd/etcdserver/api/v3rpc/rpctypes"

// limit requests to 5 MiB
cfg := embed.NewConfig()
cfg.MaxRequestBytes = 5 * 1024 * 1024

// client writes exceeding 5 MiB will be rejected
_, err := cli.Put(ctx, "foo", [LARGE VALUE...])
err == rpctypes.ErrRequestTooLarge
```

If not specified, server-side limit defaults to 1.5 MiB.

Client-side request limits must be configured based on server-side limits.

```
# limits request size to 1 MiB
etcd --max-request-bytes 1048576
```

```
import "github.com/coreos/etcd/clientv3"

cli, _ := clientv3.New(clientv3.Config{
    Endpoints: []string{"127.0.0.1:2379"},
    MaxCallSendMsgSize: 2 * 1024 * 1024,
```

```go
    MaxCallRecvMsgSize: 3 * 1024 * 1024,
})


// client writes exceeding "--max-request-bytes" will be rejected from etcd server
_, err := cli.Put(ctx, "foo", strings.Repeat("a", 1*1024*1024+5))
err == rpctypes.ErrRequestTooLarge


// client writes exceeding "MaxCallSendMsgSize" will be rejected from client-side
_, err = cli.Put(ctx, "foo", strings.Repeat("a", 5*1024*1024))
err.Error() == "rpc error: code = ResourceExhausted desc = grpc: trying to send message 


// some writes under limits
for i := range []int{0,1,2,3,4} {
    _, err = cli.Put(ctx, fmt.Sprintf("foo%d", i), strings.Repeat("a", 1*1024*1024-500))
    if err != nil {
        panic(err)
    }
}
// client reads exceeding "MaxCallRecvMsgSize" will be rejected from client-side
_, err = cli.Get(ctx, "foo", clientv3.WithPrefix())
err.Error() == "rpc error: code = ResourceExhausted desc = grpc: received message larger
```

If not specified, client-side send limit defaults to 2 MiB (1.5 MiB + gRPC overhead bytes) and receive limit to `math.MaxInt32`. Please see clientv3 godoc for more detail.

Changed raw gRPC client wrappers

3.2.12 or later changes the function signatures of `clientv3` gRPC client wrapper. This change was needed to support custom `grpc.CallOption` on message size limits.

Before and after

```diff
-func NewKVFromKVClient(remote pb.KVClient) KV {
+func NewKVFromKVClient(remote pb.KVClient, c *Client) KV {

-func NewClusterFromClusterClient(remote pb.ClusterClient) Cluster {
+func NewClusterFromClusterClient(remote pb.ClusterClient, c *Client) Cluster {

-func NewLeaseFromLeaseClient(remote pb.LeaseClient, keepAliveTimeout time.Duration) Leas
+func NewLeaseFromLeaseClient(remote pb.LeaseClient, c *Client, keepAliveTimeout time.Du

-func NewMaintenanceFromMaintenanceClient(remote pb.MaintenanceClient) Maintenance {
+func NewMaintenanceFromMaintenanceClient(remote pb.MaintenanceClient, c *Client) Mainten
```

```
-func NewWatchFromWatchClient(wc pb.WatchClient) Watcher {
+func NewWatchFromWatchClient(wc pb.WatchClient, c *Client) Watcher {
```

Changed `clientv3.Lease.TimeToLive` API

Previously, `clientv3.Lease.TimeToLive` API returned `lease.ErrLeaseNotFound` on non-existent lease ID. 3.2 instead returns TTL=-1 in its response and no error (see #7305).

Before

```
// when leaseID does not exist
resp, err := TimeToLive(ctx, leaseID)
resp == nil
err == lease.ErrLeaseNotFound
```

After

```
// when leaseID does not exist
resp, err := TimeToLive(ctx, leaseID)
resp.TTL == -1
err == nil
```

Moved `clientv3.NewFromConfigFile` to `clientv3.yaml.NewConfig`

`clientv3.NewFromConfigFile` is moved to `yaml.NewConfig`.

Before

```
import "github.com/coreos/etcd/clientv3"
clientv3.NewFromConfigFile
```

After

```
import clientv3yaml "github.com/coreos/etcd/clientv3/yaml"
clientv3yaml.NewConfig
```

Change in `--listen-peer-urls` and `--listen-client-urls`

3.2 now rejects domains names for `--listen-peer-urls` and `--listen-client-urls` (3.1 only prints out warnings), since domain name is invalid for network interface binding. Make sure that those URLs are properly formatted as `scheme://IP:port`.

See issue #6336 for more contexts.

Server upgrade checklists

Upgrade requirements

To upgrade an existing etcd deployment to 3.2, the running cluster must be 3.1 or greater. If it's before 3.1, please upgrade to 3.1 before upgrading to 3.2.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl endpoint health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, backup the etcd data. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version. Please note that the `snapshot` command only backs up the v3 data. For v2 data, see backing up v2 datastore.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.2. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

Note: If the cluster only has v3 data and no v2 data, it is not subject to this limitation.

If the cluster is serving a v2 data set larger than 50MB, each newly upgraded member may take up to two minutes to catch up with the existing cluster. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.2, the cluster will be upgraded to v3.2, and downgrade from this completed state is **not possible**. If any single member is still v3.1, however, the cluster and its operations remains "v3.1", and it is possible from this mixed cluster state to return to using a v3.1 etcd binary on all members.

Please backup the data directory of all etcd members to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example shows how to upgrade a 3-member v3.1 etcd cluster running on a local machine.

1. Check upgrade requirements

Is the cluster healthy and running v3.1.x?

```
$ ETCDCTL_API=3 etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,localh
localhost:2379 is healthy: successfully committed proposal: took = 6.600684ms
localhost:22379 is healthy: successfully committed proposal: took = 8.540064ms
localhost:32379 is healthy: successfully committed proposal: took = 8.763432ms

$ curl http://localhost:2379/version
{"etcdserver":"3.1.7","etcdcluster":"3.1.0"}
```

2. Stop the existing etcd process

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
2017-04-27 14:13:31.491746 I | raft: c89feb932daef420 [term 3] received MsgTimeoutNow fro
2017-04-27 14:13:31.491769 I | raft: c89feb932daef420 became candidate at term 4
2017-04-27 14:13:31.491788 I | raft: c89feb932daef420 received MsgVoteResp from c89feb93
2017-04-27 14:13:31.491797 I | raft: c89feb932daef420 [logterm: 3, index: 9] sent MsgVote
2017-04-27 14:13:31.491805 I | raft: c89feb932daef420 [logterm: 3, index: 9] sent MsgVote
2017-04-27 14:13:31.491815 I | raft: raft.node: c89feb932daef420 lost leader 6d4f535bae3
2017-04-27 14:13:31.524084 I | raft: c89feb932daef420 received MsgVoteResp from 6d4f535b
```

```
2017-04-27 14:13:31.524108 I | raft: c89feb932daef420 [quorum:2] has received 2 MsgVoteRe
2017-04-27 14:13:31.524123 I | raft: c89feb932daef420 became leader at term 4
2017-04-27 14:13:31.524136 I | raft: raft.node: c89feb932daef420 elected leader c89feb93
2017-04-27 14:13:31.592650 W | rafthttp: lost the TCP streaming connection with peer 6d4
2017-04-27 14:13:31.592825 W | rafthttp: lost the TCP streaming connection with peer 6d4
2017-04-27 14:13:31.693275 E | rafthttp: failed to dial 6d4f535bae3ab960 on stream Messa
2017-04-27 14:13:31.693289 I | rafthttp: peer 6d4f535bae3ab960 became inactive
2017-04-27 14:13:31.936678 W | rafthttp: lost the TCP streaming connection with peer 6d4
```

It's a good idea at this point to backup the etcd data to provide a downgrade path should any problems occur:

```
$ etcdctl snapshot save backup.db
```

3. Drop-in etcd v3.2 binary and start the new etcd process

The new v3.2 etcd will publish its information to the cluster:

```
2017-04-27 14:14:25.363225 I | etcdserver: published {Name:s1 ClientURLs:[http://localho
```

Verify that each member, and then the entire cluster, becomes healthy with the new v3.2 etcd binary:

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,loca
localhost:22379 is healthy: successfully committed proposal: took = 5.540129ms
localhost:32379 is healthy: successfully committed proposal: took = 7.321771ms
localhost:2379 is healthy: successfully committed proposal: took = 10.629901ms
```

Upgraded members will log warnings like the following until the entire cluster is upgraded. This is expected and will cease after all etcd cluster members are upgraded to v3.2:

```
2017-04-27 14:15:17.071804 W | etcdserver: member c89feb932daef420 has a higher version 
2017-04-27 14:15:21.073110 W | etcdserver: the local etcd version 3.1.7 is not up-to-dat
2017-04-27 14:15:21.073142 W | etcdserver: member 6d4f535bae3ab960 has a higher version 
2017-04-27 14:15:21.073157 W | etcdserver: the local etcd version 3.1.7 is not up-to-dat
2017-04-27 14:15:21.073164 W | etcdserver: member c89feb932daef420 has a higher version 
```

4. Repeat step 2 to step 3 for all other members

5. Finish

When all members are upgraded, the cluster will report upgrading to 3.2 successfully:

```
2017-04-27 14:15:54.536901 N | etcdserver/membership: updated the cluster version from 3
2017-04-27 14:15:54.537035 I | etcdserver/api: enabled capabilities for version 3.2
```

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,local
localhost:2379 is healthy: successfully committed proposal: took = 2.312897ms
localhost:22379 is healthy: successfully committed proposal: took = 2.553476ms
localhost:32379 is healthy: successfully committed proposal: took = 2.517902ms
```

Last modified August 19, 2023: etcd-io/website#479 Use new and better canonical link to Google Groups (cd8b01f)

Upgrade etcd from 3.0 to 3.1

Processes, checklists, and notes on upgrading etcd from 3.0 to 3.1

In the general case, upgrading from etcd 3.0 to 3.1 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v3.0 processes and replace them with etcd v3.1 processes
- after running all v3.1 processes, new features in v3.1 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Monitoring

Following metrics from v3.0.x have been deprecated in favor of go-grpc-prometheus:

- `etcd_grpc_requests_total`
- `etcd_grpc_requests_failed_total`
- `etcd_grpc_active_streams`
- `etcd_grpc_unary_requests_duration_seconds`

Upgrade requirements

To upgrade an existing etcd deployment to 3.1, the running cluster must be 3.0 or greater. If it's before 3.0, please upgrade to 3.0 before upgrading to 3.1.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl endpoint health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, backup the etcd data. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version. Please note that the `snapshot` command only backs up the v3 data. For v2 data, see backing up v2 datastore.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.1. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

Note: If the cluster only has v3 data and no v2 data, it is not subject to this limitation.

If the cluster is serving a v2 data set larger than 50MB, each newly upgraded member may take up to two minutes to catch up with the existing cluster. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.1, the cluster will be upgraded to v3.1, and downgrade from this completed state is **not possible**. If any single member is still v3.0, however, the cluster and its operations remains "v3.0", and it is possible from this mixed cluster state to return to using a v3.0 etcd binary on all members.

Please backup the data directory of all etcd members to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example shows how to upgrade a 3-member v3.0 etcd cluster running on a local machine.

1. Check upgrade requirements

Is the cluster healthy and running v3.0.x?

```
$ ETCDCTL_API=3 etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,localh
localhost:2379 is healthy: successfully committed proposal: took = 6.600684ms
localhost:22379 is healthy: successfully committed proposal: took = 8.540064ms
localhost:32379 is healthy: successfully committed proposal: took = 8.763432ms

$ curl http://localhost:2379/version
{"etcdserver":"3.0.16","etcdcluster":"3.0.0"}
```

2. Stop the existing etcd process

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
2017-01-17 09:34:18.352662 I | raft: raft.node: 1640829d9eea5cfb elected leader 1640829d9
2017-01-17 09:34:18.359630 W | etcdserver: failed to reach the peerURL(http://localhost:2
2017-01-17 09:34:18.359679 W | etcdserver: cannot get the version of member fd32987dcd05
2017-01-17 09:34:18.548116 W | rafthttp: lost the TCP streaming connection with peer fd32
2017-01-17 09:34:19.147816 W | rafthttp: lost the TCP streaming connection with peer fd32
2017-01-17 09:34:34.364907 W | etcdserver: failed to reach the peerURL(http://localhost:2
```

It's a good idea at this point to backup the etcd data to provide a downgrade path should any problems occur:

```
$ etcdctl snapshot save backup.db
```

3. Drop-in etcd v3.1 binary and start the new etcd process

The new v3.1 etcd will publish its information to the cluster:

```
2017-01-17 09:36:00.996590 I | etcdserver: published {Name:my-etcd-1 ClientURLs:[http://l
```

Verify that each member, and then the entire cluster, becomes healthy with the new v3.1 etcd binary:

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,local
localhost:22379 is healthy: successfully committed proposal: took = 5.540129ms
```

```
localhost:32379 is healthy: successfully committed proposal: took = 7.321671ms
localhost:2379 is healthy: successfully committed proposal: took = 10.629901ms
```

Upgraded members will log warnings like the following until the entire cluster is upgraded. This is expected and will cease after all etcd cluster members are upgraded to v3.1:

```
2017-01-17 09:36:38.406268 W | etcdserver: the local etcd version 3.0.16 is not up-to-dat
2017-01-17 09:36:38.406295 W | etcdserver: member fd32987dcd0511e0 has a higher version 3
2017-01-17 09:36:42.407695 W | etcdserver: the local etcd version 3.0.16 is not up-to-dat
2017-01-17 09:36:42.407730 W | etcdserver: member fd32987dcd0511e0 has a higher version 3
```

4. Repeat step 2 to step 3 for all other members

5. Finish

When all members are upgraded, the cluster will report upgrading to 3.1 successfully:

```
2017-01-17 09:37:03.100015 I | etcdserver: updating the cluster version from 3.0 to 3.1
2017-01-17 09:37:03.104263 N | etcdserver/membership: updated the cluster version from 3
2017-01-17 09:37:03.104374 I | etcdserver/api: enabled capabilities for version 3.1
```

```
$ ETCDCTL_API=3 /etcdctl endpoint health --endpoints=localhost:2379,localhost:22379,local
localhost:2379 is healthy: successfully committed proposal: took = 2.312897ms
localhost:22379 is healthy: successfully committed proposal: took = 2.553476ms
localhost:32379 is healthy: successfully committed proposal: took = 2.516902ms
```

Last modified August 19, 2023: etcd-io/website#479 Use new and better canonical link to Google Groups (cd8b01f)

Upgrade etcd from 2.3 to 3.0

Processes, checklists, and notes on upgrading etcd from 2.3 to 3.0

In the general case, upgrading from etcd 2.3 to 3.0 can be a zero-downtime, rolling upgrade:

- one by one, stop the etcd v2.3 processes and replace them with etcd v3.0 processes
- after running all v3.0 processes, new features in v3.0 are available to the cluster

Before starting an upgrade, read through the rest of this guide to prepare.

Upgrade checklists

NOTE: When migrating from v2 with no v3 data, etcd server v3.2+ panics when etcd restores from existing snapshots but no v3 `ETCD_DATA_DIR/member/snap/db` file. This happens when the server had migrated from v2 with no previous v3 data. This also prevents accidental v3 data loss (e.g. `db` file might have been moved). etcd requires that post v3 migration can only happen with v3 data. Do not upgrade to newer v3 versions until v3.0 server contains v3 data.

Upgrade requirements

To upgrade an existing etcd deployment to 3.0, the running cluster must be 2.3 or greater. If it's before 2.3, please upgrade to 2.3 before upgrading to 3.0.

Also, to ensure a smooth rolling upgrade, the running cluster must be healthy. Check the health of the cluster by using the `etcdctl cluster-health` command before proceeding.

Preparation

Before upgrading etcd, always test the services relying on etcd in a staging environment before deploying the upgrade to the production environment.

Before beginning, backup the etcd data directory. Should something go wrong with the upgrade, it is possible to use this backup to downgrade back to existing etcd version.

Mixed versions

While upgrading, an etcd cluster supports mixed versions of etcd members, and operates with the protocol of the lowest common version. The cluster is only considered upgraded once all of its members are upgraded to version 3.0. Internally, etcd members negotiate with each other to determine the overall cluster version, which controls the reported version and the supported features.

Limitations

It might take up to 2 minutes for the newly upgraded member to catch up with the existing cluster when the total data size is larger than 50MB. Check the size of a recent snapshot to estimate the total data size. In other words, it is safest to wait for 2 minutes between upgrading each member.

For a much larger total data size, 100MB or more , this one-time process might take even more time. Administrators of very large etcd clusters of this magnitude can feel free to contact the etcd team before upgrading, and we'll be happy to provide advice on the procedure.

Downgrade

If all members have been upgraded to v3.0, the cluster will be upgraded to v3.0, and downgrade from this completed state is **not possible**. If any single member is still v2.3, however, the cluster and its operations remains “v2.3”, and it is possible from this mixed cluster state to return to using a v2.3 etcd binary on all members.

Please backup the data directory of all etcd members to make downgrading the cluster possible even after it has been completely upgraded.

Upgrade procedure

This example details the upgrade of a three-member v2.3 etcd cluster running on a local machine.

1. Check upgrade requirements.

Is the cluster healthy and running v.2.3.x?

```
$ etcdctl cluster-health
member 6e3bd23ae5f1eae0 is healthy: got healthy result from http://localhost:22379
member 924e2e83e93f2560 is healthy: got healthy result from http://localhost:32379
member 8211f1d0f64f3269 is healthy: got healthy result from http://localhost:12379
cluster is healthy
```

```
$ curl http://localhost:2379/version
{"etcdserver":"2.3.x","etcdcluster":"2.3.8"}
```

2. Stop the existing etcd process

When each etcd process is stopped, expected errors will be logged by other cluster members. This is normal since a cluster member connection has been (temporarily) broken:

```
2016-06-27 15:21:48.624124 E | rafthttp: failed to dial 8211f1d0f64f3269 on stream Messag
2016-06-27 15:21:48.624175 I | rafthttp: the connection with 8211f1d0f64f3269 became inac
```

It's a good idea at this point to backup the etcd data directory to provide a downgrade path should any problems occur:

```
$ etcdctl backup \
      --data-dir /var/lib/etcd \
      --backup-dir /tmp/etcd_backup
```

3. Drop-in etcd v3.0 binary and start the new etcd process

The new v3.0 etcd will publish its information to the cluster:

```
09:58:25.938673 I | etcdserver: published {Name:infra1 ClientURLs:[http://localhost:1237
```

Verify that each member, and then the entire cluster, becomes healthy with the new v3.0 etcd binary:

```
$ etcdctl cluster-health
member 6e3bd23ae5f1eae0 is healthy: got healthy result from http://localhost:22379
member 924e2e83e93f2560 is healthy: got healthy result from http://localhost:32379
member 8211f1d0f64f3269 is healthy: got healthy result from http://localhost:12379
cluster is healthy
```

Upgraded members will log warnings like the following until the entire cluster is upgraded. This is expected and will cease after all etcd cluster members are upgraded to v3.0:

```
2016-06-27 15:22:05.679644 W | etcdserver: the local etcd version 2.3.7 is not up-to-date
2016-06-27 15:22:05.679660 W | etcdserver: member 8211f1d0f64f3269 has a higher version 
```

4. Repeat step 2 to step 3 for all other members

5. Finish

When all members are upgraded, the cluster will report upgrading to 3.0 successfully:

```
2016-06-27 15:22:19.873751 N | membership: updated the cluster version from 2.3 to 3.0
2016-06-27 15:22:19.914574 I | api: enabled capabilities for version 3.0.0
```

```
$ ETCDCTL_API=3 etcdctl endpoint health
127.0.0.1:12379 is healthy: successfully committed proposal: took = 18.440155ms
127.0.0.1:32379 is healthy: successfully committed proposal: took = 13.651368ms
127.0.0.1:22379 is healthy: successfully committed proposal: took = 18.513301ms
```

Further considerations

- etcdctl environment variables have been updated. If `ETCDCTL_API=2 etcdctl cluster-health` works properly but `ETCDCTL_API=3 etcdctl endpoints health` responds with `Error: grpc: timed out when dialing`, be sure to use the new variable names.

Known Issues

- etcd < v3.1 does not work properly if built with Go > v1.7. See Issue 6951 for additional information.
- If an error such as `transport: http2Client.notifyError got notified that the client transport was broken unexpected EOF.` shows up in the etcd server logs, be sure etcd is a pre-built release or built with (etcd v3.1+ & go v1.7+) or (etcd <v3.1 & go v1.6.x).
- Adding a v3 node to v2.3 cluster during upgrades is not supported and could trigger panics. See Issue 7249 for additional information. Mixed versions of etcd members are only allowed during v3 migration. Finish upgrades before making any membership changes.

Last modified August 19, 2023: etcd-io/website#479 Use new and better canonical link to Google Groups (cd8b01f)

Platforms

etcd deployments on various platform services

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Amazon Web Services

etcd deployments on AWS EC2

This guide assumes operational knowledge of Amazon Web Services (AWS), specifically Amazon Elastic Compute Cloud (EC2). This guide provides an introduction to design considerations when designing an etcd deployment on AWS EC2 and how AWS specific features may be utilized in that context.

Capacity planning

As a critical building block for distributed systems it is crucial to perform adequate capacity planning in order to support the intended cluster workload. As a highly available and strongly consistent data store increasing the number of nodes in an etcd cluster will generally affect performance adversely. This makes sense intuitively, as more nodes means more members for the leader to coordinate state across. The most direct way to increase throughput and decrease latency of an etcd cluster is allocate more disk I/O, network I/O, CPU, and memory to cluster members. In the event it is impossible to temporarily divert incoming requests to the cluster, scaling the EC2 instances which comprise the etcd cluster members one at a time may improve performance. It is, however, best to avoid bottlenecks through capacity planning.

The etcd team has produced a hardware recommendation guide which is very useful for “ballparking” how many nodes and what instance type are necessary for a cluster.

AWS provides a service for creating groups of EC2 instances which are dynamically sized to match load on the instances. Using an Auto Scaling Group (ASG) to dynamically scale an etcd cluster is not recommended for several reasons including:

- etcd performance is generally inversely proportional to the number of members in a cluster due to the synchronous replication which provides strong consistency of data stored in etcd
- the operational complexity of adding lifecycle hooks to properly add and remove members from an etcd cluster by modifying the runtime configuration

Auto Scaling Groups do provide a number of benefits besides cluster scaling which include:

- distribution of EC2 instances across Availability Zones (AZs)

- EC2 instance fail over across AZs
- consolidated monitoring and life cycle control of instances within an ASG

The use of an ASG to create a self healing etcd cluster is one of the design considerations when deploying an etcd cluster to AWS.

Cluster design

The purpose of this section is to provide foundational guidance for deploying etcd on AWS. The discussion will be framed by the following three critical design criteria about the etcd cluster itself:

- block device provider: limited to the tradeoffs between EBS or instance storage (InstanceStore)
- cluster topology: how many nodes should make up an etcd cluster; should these nodes be distributed over multiple AZs
- managing etcd members: creating a static cluster of EC2 instances or using an ASG.

The intended cluster workload should dictate the cluster design. A configuration store for microservices may require different design considerations than a distributed lock service, a secrets store, or a Kubernetes control plane. Cluster design tradeoffs include considerations such as:

- availability
- data durability after member failure
- performance/throughput
- self healing

Availability

Instance availability on AWS is ultimately determined by the Amazon EC2 Region Service Level Agreement (SLA) which is the policy by which Amazon describes their precise definition of a regional outage.

In the context of an etcd cluster this means a cluster must contain a minimum of three members where EC2 instances are spread across at least two AZs in order for an etcd cluster to be considered highly available at a Regional level.

For most use cases the additional latency associated with a cluster spanning across Availability Zones will introduce a negligible performance impact.

Availability considerations apply to all components of an application; if the application which accesses the etcd cluster will only be deployed to a single Availability Zone it may not make sense to make the etcd cluster highly available across zones.

Data durability after member failure

A highly available etcd cluster is resilient to member loss, however, it is important to consider data durability in the event of disaster when designing an etcd deployment. Deploying etcd on AWS supports multiple mechanisms for data durability.

- replication: etcd replicates all data to all members of the etcd cluster. Therefore, given more members in the cluster and more independent failure domains, the less likely that data stored in an etcd cluster will be permanently lost in the event of disaster.
- Point in time etcd snapshotting: the etcd v3 API introduced support for snapshotting clusters. The operation is cheap enough (completing in the order of minutes) to run quite frequently and the resulting archives can be archived in a storage service like Amazon Simple Storage Service (S3).
- Amazon Elastic Block Storage (EBS): an EBS volume is a replicated network attached block device which have stronger storage safety guarantees than InstanceStore which has a life cycle associated with the life cycle of the attached EC2 instance. The life cycle of an EBS volume is not necessarily tied to an EC2 instance and can be detached and snapshotted independently which means that a single node etcd cluster backed by an EBS volume can provide a fairly reasonable level of data durability.

Performance/Throughput

The performance of an etcd cluster is roughly quantifiable through latency and throughput metrics which are primarily affected by disk and network performance. Detailed performance planning information is provided in the performance section of the etcd operations guide.

Network

AWS offers EC2 Placement Groups which allow the collocation of EC2 instances within a single Availability Zone which can be utilized in order to minimize network latency between etcd members in the cluster. It is important to remember that collocation of etcd nodes within a single AZ will provide weaker fault tolerance than distributing members across multiple AZs. Enhanced networking for EC2 instances may also improve network performance of individual EC2 instances.

Disk

AWS provides two basic types of block storage: EBS volumes and EC2 Instance Store. As mentioned, an EBS volume is a network attached block device while instance storage is directly attached to the hypervisor of the EC2 host. EBS volumes will generally have higher latency, lower throughput, and greater performance variance than Instance Store volumes. If performance, rather than data safety, is the primary concern it is highly recommended that instance storage on the EC2 instances be utilized. Remember that the amount of available instance storage varies by EC2 instance types which may impose additional performance considerations.

Inconsistent EBS volume performance can introduce etcd cluster instability. Provisioned IOPS can provide more consistent performance than general purpose SSD EBS volumes. More information about EBS volume performance is available from AWS and Datadog has shared their experience with getting optimal performance with AWS EBS Provisioned IOPS in their engineering blog.

Self healing

While using an ASG to scale the size of an etcd cluster is not recommended, an ASG can be used effectively to maintain the desired number of nodes in the event of node failure. The maintenance of a stable number of etcd nodes will provide the etcd cluster with a measure of self healing.

Next steps

The operational life cycle of an etcd cluster can be greatly simplified through the use of the etcd-operator. The open source etcd operator is a Kubernetes control plane operator which deploys and manages etcd clusters atop Kubernetes. While still in its early stages the etcd-operator already offers periodic backups to S3, detection and replacement of failed nodes, and automated disaster recovery from backups in the event of permanent quorum loss.

Last modified August 21, 2021: fix 3.4 links (#458) (f75a5c9)

Container Linux with systemd

etcd deployments with systemd under Container Linux

The following guide shows how to run etcd with systemd under Container Linux.

Provisioning an etcd cluster

Cluster bootstrapping in Container Linux is simplest with Ignition; `coreos-metadata.service` dynamically fetches the machine's IP for discovery. Note that etcd's discovery service protocol is only meant for bootstrapping, and cannot be used with runtime reconfiguration or cluster monitoring.

The Container Linux Config Transpiler compiles etcd configuration files into Ignition configuration files:

```
etcd:
  version: 3.2.0
  name: s1
  data_dir: /var/lib/etcd
  advertise_client_urls:       http://{PUBLIC_IPV4}:2379
  initial_advertise_peer_urls: http://{PRIVATE_IPV4}:2380
  listen_client_urls:          http://0.0.0.0:2379
  listen_peer_urls:            http://{PRIVATE_IPV4}:2380
  discovery:                   https://discovery.etcd.io/<token>
```

`ct` would produce the following Ignition Config:

```
$ ct --platform=gce --in-file /tmp/ct-etcd.cnf
{"ignition":{"version":"2.0.0","config"...
```

```
{
  "ignition":{"version":"2.0.0","config":{}},
  "storage":{},
  "systemd":{
```

```
    "units":[{
      "name":"etcd-member.service",
      "enable":true,
      "dropins":[{
        "name":"20-clct-etcd-member.conf",
        "contents":"[Unit]\nRequires=coreos-metadata.service\nAfter=coreos-metadata.serv
      "networkd":{},
      "passwd":{}}
```

To avoid accidental misconfiguration, the transpiler helpfully verifies etcd configurations when generating Ignition files:

```
etcd:
  version: 3.2.0
  name: s1
  data_dir_x: /var/lib/etcd
  advertise_client_urls:       http://{PUBLIC_IPV4}:2379
  initial_advertise_peer_urls: http://{PRIVATE_IPV4}:2380
  listen_client_urls:          http://0.0.0.0:2379
  listen_peer_urls:            http://{PRIVATE_IPV4}:2380
  discovery:                   https://discovery.etcd.io/<token>
```

```
$ ct --platform=gce --in-file /tmp/ct-etcd.cnf
warning at line 3, column 2
Config has unrecognized key: data_dir_x
```

See Container Linux Provisioning for more details.

etcd 3.x service

Container Linux does not include etcd 3.x binaries by default. Different versions of etcd 3.x can be fetched via `etcd-member.service`.

Confirm unit file exists:

```
systemctl cat etcd-member.service
```

Check if the etcd service is running:

```
systemctl status etcd-member.service
```

Example systemd drop-in unit to override the default service settings:

```
cat > /tmp/20-cl-etcd-member.conf <<EOF
[Service]
Environment="ETCD_IMAGE_TAG=v3.2.0"
Environment="ETCD_DATA_DIR=/var/lib/etcd"
Environment="ETCD_SSL_DIR=/etc/ssl/certs"
Environment="ETCD_OPTS=--name s1 \
  --listen-client-urls https://10.240.0.1:2379 \
  --advertise-client-urls https://10.240.0.1:2379 \
  --listen-peer-urls https://10.240.0.1:2380 \
  --initial-advertise-peer-urls https://10.240.0.1:2380 \
  --initial-cluster s1=https://10.240.0.1:2380,s2=https://10.240.0.2:2380,s3=https://10.2
  --initial-cluster-token mytoken \
  --initial-cluster-state new \
  --client-cert-auth \
  --trusted-ca-file /etc/ssl/certs/etcd-root-ca.pem \
  --cert-file /etc/ssl/certs/s1.pem \
  --key-file /etc/ssl/certs/s1-key.pem \
  --peer-client-cert-auth \
  --peer-trusted-ca-file /etc/ssl/certs/etcd-root-ca.pem \
  --peer-cert-file /etc/ssl/certs/s1.pem \
  --peer-key-file /etc/ssl/certs/s1-key.pem \
  --auto-compaction-retention 1"
EOF
mv /tmp/20-cl-etcd-member.conf /etc/systemd/system/etcd-member.service.d/20-cl-etcd-membe
```

Or use a Container Linux Config:

```
systemd:
  units:
    - name: etcd-member.service
      dropins:
        - name: conf1.conf
          contents: |
            [Service]
            Environment="ETCD_SSL_DIR=/etc/ssl/certs"

etcd:
  version: 3.2.0
  name: s1
  data_dir: /var/lib/etcd
```

```
  listen_client_urls:          https://0.0.0.0:2379
  advertise_client_urls:       https://{PUBLIC_IPV4}:2379
  listen_peer_urls:            https://{PRIVATE_IPV4}:2380
  initial_advertise_peer_urls: https://{PRIVATE_IPV4}:2380
  initial_cluster:             s1=https://{PRIVATE_IPV4}:2380,s2=https://10.240.0.2:2380
  initial_cluster_token:       mytoken
  initial_cluster_state:       new
  client_cert_auth:            true
  trusted_ca_file:             /etc/ssl/certs/etcd-root-ca.pem
  cert_file:                   /etc/ssl/certs/s1.pem
  key_file:                    /etc/ssl/certs/s1-key.pem
  peer_client_cert_auth:       true
  peer_trusted_ca_file:        /etc/ssl/certs/etcd-root-ca.pem
  peer_cert_file:              /etc/ssl/certs/s1.pem
  peer_key_file:               /etc/ssl/certs/s1-key.pem
  auto_compaction_retention:   1
```

```
$ ct --platform=gce --in-file /tmp/ct-etcd.cnf
{"ignition":{"version":"2.0.0","config"...
```

To see all runtime drop-in changes for system units:

```
systemd-delta --type=extended
```

To enable and start:

```
systemctl daemon-reload
systemctl enable --now etcd-member.service
```

To see the logs:

```
journalctl --unit etcd-member.service --lines 10
```

To stop and disable the service:

```
systemctl disable --now etcd-member.service
```

etcd 2.x service

Container Linux includes a unit file `etcd2.service` for etcd 2.x, which will be removed in the near future. See Container Linux FAQ for more details.

Confirm unit file is installed:

```
systemctl cat etcd2.service
```

Check if the etcd service is running:

```
systemctl status etcd2.service
```

To stop and disable:

```
systemctl disable --now etcd2.service
```

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

FreeBSD

etcd deployments using FreeBSD

Starting with version 0.1.2 both etcd and etcdctl have been ported to FreeBSD and can be installed either via packages or ports system. Their versions have been recently updated to 0.2.0 so now etcd and etcdctl can be enjoyed on FreeBSD 10.0 (RC4 as of now) and 9.x, where they have been tested. They might also work when installed from ports on earlier versions of FreeBSD, but it is untested; caveat emptor.

Installation

Using pkgng package system

1. If pkgng is not installed, install it with command `pkg` and answering 'Y' when asked.

2. Update the repository data with `pkg update`.

3. Install etcd with `pkg install coreos-etcd coreos-etcdctl`.

4. Verify successful installation by confirming `pkg info | grep etcd` matches:

```
r@fbsd10:/ # pkg info | grep etcd
coreosetcd0.2.0             Highlyavailable key value store and service discovery
coreosetcdctl0.2.0          Simple commandline client for etcd
r@fbsd10:/ #
```

5. etcd and etcdctl are ready to use! For more information about using pkgng, please see: http://www.freebsd.org/doc/handbook/pkgngintro.html

Using ports system

1. If ports is not installed, install with `portsnap fetch extract` (it may take some time depending on hardware and network connection).

2. Build etcd with `cd /usr/ports/devel/etcd && make install clean`. There will be an option to build and install documentation and etcdctl with it.

3. If etcd wasn't installed with etcdctl, it can be built later with `cd /usr/ports/devel/etcdctl && make install clean`.

4. Verify successful installation by confirming `pkg info | grep etcd` matches:

```
r@fbsd10:/ # pkg info | grep etcd
coreosetcd0.2.0              Highlyavailable key value store and service discovery
coreosetcdctl0.2.0           Simple commandline client for etcd
r@fbsd10:/ #
```

5. etcd and etcdctl are ready to use! For more information about using ports system, please see: https://www.freebsd.org/doc/handbook/portsusing.html

Issues

If there are any issues with the build/install procedure or there's a problem that is local to FreeBSD only (for example, by not being able to reproduce it on any other platform, like OSX or Linux), please send a problem report using this page for more information: http://www.freebsd.org/sendpr.html

Triage

Managing changes in etcd

Issue Triage Guidelines
Managing incoming issues

Last modified April 26, 2021: Docsy theme (#244) (86b070b)

Issue Triage Guidelines

Managing incoming issues

Purpose

Speed up issue management.

The `etcd` issues are listed at https://github.com/etcd-io/etcd/issues and are identified with labels. For example, an issue that is identified as a bug will eventually be set to label `area/bug` . New issues will start out without any labels, but typically `etcd` maintainers and active contributors add labels based on their findings. The detailed list of labels can be found at https://github.com/kubernetes/kubernetes/labels

Following are few predetermined searches on issues for convenience:

- Bugs
- Help Wanted
- Longest untriaged issues

Scope

These guidelines serves as a primary document for triaging an incoming issues in `etcd` . Everyone is welcome to help manage issues and PRs but the work and responsibilities discussed in this document are created with `etcd` maintainers and active contributors in mind.

Validate if an issue is a bug

Validate if the issue is indeed a bug. If not, add a comment with findings and close trivial issue. For non-trivial issue, wait to hear back from issue reporter and see if there is any objection. If issue reporter does not reply in 30 days, close the issue. If the problem can not be reproduced or require more information, leave a comment for the issue reporter.

Inactive issues

Issues that lack enough information from the issue reporter should be closed if issue reporter do not provide information in 60 days.

Duplicate issues

If an issue is a duplicate, add a comment stating so along with a reference for the original issue and close it.

Issues that don't belong to etcd

Sometime issues are reported that actually belongs to other projects that `etcd` use. For example, `grpc` or `golang` issues. Such issues should be addressed by asking reporter to open issues in appropriate other project. Close the issue unless a maintainer and issue reporter see a need to keep it open for tracking purpose.

Verify important labels are in place

Make sure that issue has label on areas it belongs to, proper assignees are added and milestone is identified. If any of these labels are missing, add one. If labels can not be assigned due to limited privilege or correct label can not be decided, that's fine, contact maintainers if needed.

Poke issue owner if needed

If an issue owned by a developer has no PR created in 30 days, contact the issue owner and ask for a PR or to release ownership if needed.

Last modified April 26, 2021: Docsy theme (#244) (86b070b)