etcd

A distributed, reliable key-value store for the most critical data of a distributed system.

This is the documentation for etcd2 releases. Read etcd3 doc for etcd3 releases.

etcd API

Running a Single Machine Cluster

These examples will use a single member cluster to show you the basics of the etcd REST API. Let's start etcd:

```
./bin/etcd
```

This will bring up etcd listening on the IANA assigned ports and listening on localhost. The IANA assigned ports for etcd are 2379 for client communication and 2380 for server-to-server communication.

Getting the etcd version

The etcd version of a specific instance can be obtained from the `/version` endpoint.

```
curl -L http://127.0.0.1:2379/version
```

Key Space Operations

The primary API of etcd is a hierarchical key space. The key space consists of directories and keys which are generically referred to as "nodes".

Setting the value of a key

Let's set the first key-value pair in the datastore. In this case the key is `/message` and the value is `Hello world`.

```
curl http://127.0.0.1:2379/v2/keys/message -XPUT -d value="Hello world"
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 2,
        "key": "/message",
        "modifiedIndex": 2,
        "value": "Hello world"
    }
}
```

The response object contains several attributes:

1. `action`: the action of the request that was just made. The request attempted to modify `node.value` via a `PUT` HTTP request, thus the value of action is `set`.
2. `node.key`: the HTTP path to which the request was made. We set `/message` to `Hello world`, so the key field is `/message`. etcd uses a file-system-like structure to represent the key-value pairs, therefore all keys start with `/`.
3. `node.value`: the value of the key after resolving the request. In this case, a successful request was made that attempted to change the node's value to `Hello world`.
4. `node.createdIndex`: an index is a unique, monotonically-incrementing integer created for each change to etcd. This specific index reflects the point in the etcd state member at which a given key was created. You may notice that in this example the index is `2` even though it is the first request you sent to the server. This is because there are internal commands that also change the state behind the scenes, like adding and syncing servers.
5. `node.modifiedIndex`: like `node.createdIndex`, this attribute is also an etcd index. Actions that cause the value to change include `set`, `delete`, `update`, `create`, `compareAndSwap` and `compareAndDelete`. Since the `get` and `watch` commands do not change state in the store, they do not change the value of `node.modifiedIndex`.

Response Headers

etcd includes a few HTTP headers in responses that provide global information about the etcd cluster that serviced a request:

```
X-Etcd-Index: 35
X-Raft-Index: 5398
X-Raft-Term: 1
```

- `X-Etcd-Index` is the current etcd index as explained above. When request is a watch on key space, `X-Etcd-Index` is the current etcd index when the watch starts, which means that the watched event may happen after `X-Etcd-Index`.
- `X-Raft-Index` is similar to the etcd index but is for the underlying raft protocol.
- `X-Raft-Term` is an integer that will increase whenever an etcd master election happens in the cluster. If this number is increasing rapidly, you may need to tune the election timeout. See the tuning section for details.

Get the value of a key

We can get the value that we just set in `/message` by issuing a `GET` request:

```
curl http://127.0.0.1:2379/v2/keys/message
```

```
{
    "action": "get",
    "node": {
        "createdIndex": 2,
        "key": "/message",
        "modifiedIndex": 2,
        "value": "Hello world"
    }
}
```

Changing the value of a key

You can change the value of `/message` from `Hello world` to `Hello etcd` with another `PUT` request to the key:

```
curl http://127.0.0.1:2379/v2/keys/message -XPUT -d value="Hello etcd"
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 3,
        "key": "/message",
        "modifiedIndex": 3,
        "value": "Hello etcd"
    },
    "prevNode": {
     "createdIndex": 2,
     "key": "/message",
     "value": "Hello world",
     "modifiedIndex": 2
    }
}
```

Here we introduce a new field: `prevNode`. The `prevNode` field represents what the state of a given node was before resolving the request at hand. The `prevNode` field follows the same format as the `node`, and is omitted in the event that there was no previous state for a given node.

Deleting a key

You can remove the `/message` key with a `DELETE` request:

```
curl http://127.0.0.1:2379/v2/keys/message -XDELETE
```

```
{
    "action": "delete",
    "node": {
        "createdIndex": 3,
        "key": "/message",
        "modifiedIndex": 4
    },
    "prevNode": {
     "key": "/message",
     "value": "Hello etcd",
     "modifiedIndex": 3,
     "createdIndex": 3
    }
}
```

Using key TTL

Keys in etcd can be set to expire after a specified number of seconds. You can do this by setting a TTL (time to live) on the key when sending a `PUT` request:

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -d ttl=5
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 5,
        "expiration": "2013-12-04T12:01:21.874888581-08:00",
        "key": "/foo",
        "modifiedIndex": 5,
        "ttl": 5,
        "value": "bar"
    }
}
```

Note the two new fields in response:

1. The `expiration` is the time at which this key will expire and be deleted.
2. The `ttl` is the specified time to live for the key, in seconds.

NOTE: Keys can only be expired by a cluster leader, so if a member gets disconnected from the cluster, its keys will not expire until it rejoins.

Now you can try to get the key by sending a `GET` request:

```
curl http://127.0.0.1:2379/v2/keys/foo
```

If the TTL has expired, the key will have been deleted, and you will be returned a 100.

```
{
    "cause": "/foo",
    "errorCode": 100,
    "index": 6,
    "message": "Key not found"
}
```

The TTL can be unset to avoid expiration through update operation:

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -d ttl= -d prevExist=true
```


```
{
    "action": "update",
    "node": {
        "createdIndex": 5,
        "key": "/foo",
        "modifiedIndex": 6,
        "value": "bar"
    },
    "prevNode": {
        "createdIndex": 5,
        "expiration": "2013-12-04T12:01:21.874888581-08:00",
        "key": "/foo",
        "modifiedIndex": 5,
        "ttl": 3,
        "value": "bar"
    }
}
```


Refreshing key TTL

Keys in etcd can be refreshed without notifying current watchers.

This can be achieved by setting the refresh to true when updating a TTL.

You cannot update the value of a key when refreshing it.

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -d ttl=5
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d ttl=5 -d refresh=true -d prevExist=true
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 5,
        "expiration": "2013-12-04T12:01:21.874888581-08:00",
        "key": "/foo",
        "modifiedIndex": 5,
        "ttl": 5,
        "value": "bar"
    }
}
{
   "action":"update",
   "node":{
       "key":"/foo",
       "value":"bar",
       "expiration": "2013-12-04T12:01:26.874888581-08:00",
       "ttl":5,
       "modifiedIndex":6,
       "createdIndex":5
    },
   "prevNode":{
       "key":"/foo",
       "value":"bar",
       "expiration":"2013-12-04T12:01:21.874888581-08:00",
       "ttl":3,
       "modifiedIndex":5,
       "createdIndex":5
     }
}
```


Waiting for a change

We can watch for a change on a key and receive a notification by using long polling. This also works for child keys by passing `recursive=true` in curl.

In one terminal, we send a `GET` with `wait=true` :

```
curl http://127.0.0.1:2379/v2/keys/foo?
wait=true
```

Now we are waiting for any changes at path `/foo` .

In another terminal, we set a key `/foo` with value `bar` :

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar
```

The first terminal should get the notification and return with the same response as the set request:

```
{
    "action": "set",
    "node": {
        "createdIndex": 7,
        "key": "/foo",
        "modifiedIndex": 7,
        "value": "bar"
    },
    "prevNode": {
        "createdIndex": 6,
        "key": "/foo",
        "modifiedIndex": 6,
        "value": "bar"
    }
}
```


However, the watch command can do more than this. Using the index, we can watch for commands that have happened in the past. This is useful for ensuring you don't miss events between watch commands. Typically, we watch again from the `modifiedIndex` + 1 of the node we got.

Let's try to watch for the set command of index 7 again:

```
curl 'http://127.0.0.1:2379/v2/keys/foo?wait=true&waitIndex=7'
```

The watch command returns immediately with the same response as previously.

If we were to restart the watch from index 8 with:

```
curl 'http://127.0.0.1:2379/v2/keys/foo?wait=true&waitIndex=8'
```

Then even if etcd is on index 9 or 800, the first event to occur to the `/foo` key between 8 and the current index will be returned.

Note: etcd only keeps the responses of the most recent 1000 events across all etcd keys. It is recommended to send the response to another thread to process immediately instead of blocking the watch while processing the result.

Watch from cleared event index

If we miss all the 1000 events, we need to recover the current state of the watching key space through a get and then start to watch from the `X-Etcd-Index` + 1.

For example, we set `/other="bar"` for 2000 times and try to wait from index 8.

```
curl 'http://127.0.0.1:2379/v2/keys/foo?wait=true&waitIndex=8'
```

We get the index is outdated response, since we miss the 1000 events kept in etcd.


```
{"errorCode":401,"message":"The event in requested index is outdated and cleared","cau
se":"the requested history has been cleared [1008/8]","index":2007}
```


To start watch, first we need to fetch the current state of key `/foo`:

```
curl 'http://127.0.0.1:2379/v2/keys/foo' -vv
```

```
< HTTP/1.1 200 OK
< Content-Type: application/json
< X-Etcd-Cluster-Id: 7e27652122e8b2ae
< X-Etcd-Index: 2007
< X-Raft-Index: 2615
< X-Raft-Term: 2
< Date: Mon, 05 Jan 2015 18:54:43 GMT
< Transfer-Encoding: chunked
<
{"action":"get","node":
{"key":"/foo","value":"bar","modifiedIndex":7,"createdIndex":7}}
```

Unlike watches we use the `X-Etcd-Index` + 1 of the response as a `waitIndex` instead of the node's `modifiedIndex` + 1 for two reasons:

1. The `X-Etcd-Index` is always greater than or equal to the `modifiedIndex` when getting a key because `X-Etcd-Index` is the current etcd index, and the `modifiedIndex` is the index of an event already stored in etcd.
2. None of the events represented by indexes between `modifiedIndex` and `X-Etcd-Index` will be related to the key being fetched.

Using the `modifiedIndex` + 1 is functionally equivalent for subsequent watches, but since it is smaller than the `X-Etcd-Index` + 1, we may receive a `401 EventIndexCleared` error immediately.

So the first watch after the get should be:

```
curl 'http://127.0.0.1:2379/v2/keys/foo?wait=true&waitIndex=2008'
```

Connection being closed prematurely

The server may close a long polling connection before emitting any events. This can happen due to a timeout or the server being shutdown. Since the HTTP header is sent immediately upon accepting the connection, the response will be seen as empty: `200 OK` and empty body. The clients should be prepared to deal with this scenario and retry the watch.

Atomically Creating In-Order Keys

Using `POST` on a directory, you can create keys with key names that are created in-order. This can be used in a variety of useful patterns, like implementing queues of keys which need to be processed in strict order. An example use case would be ensuring clients get fair access to a mutex.

Creating an in-order key is easy:

```
curl http://127.0.0.1:2379/v2/keys/queue -XPOST -d value=Job1
```

```
{
    "action": "create",
    "node": {
        "createdIndex": 6,
        "key": "/queue/00000000000000000006",
        "modifiedIndex": 6,
        "value": "Job1"
    }
}
```

If you create another entry some time later, it is guaranteed to have a key name that is greater than the previous key. Also note the key names use the global etcd index, so the next key can be more than `previous + 1`.

```
curl http://127.0.0.1:2379/v2/keys/queue -XPOST -d value=Job2
```

```
{
    "action": "create",
    "node": {
        "createdIndex": 29,
        "key": "/queue/00000000000000000029",
        "modifiedIndex": 29,
        "value": "Job2"
    }
}
```

To enumerate the in-order keys as a sorted list, use the "sorted" parameter.

```
curl -s 'http://127.0.0.1:2379/v2/keys/queue?recursive=true&sorted=true'
```

```
{
    "action": "get",
    "node": {
        "createdIndex": 2,
        "dir": true,
        "key": "/queue",
        "modifiedIndex": 2,
        "nodes": [
            {
                "createdIndex": 2,
                "key": "/queue/00000000000000000002",
                "modifiedIndex": 2,
                "value": "Job1"
            },
            {
                "createdIndex": 3,
                "key": "/queue/00000000000000000003",
                "modifiedIndex": 3,
                "value": "Job2"
            }
        ]
    }
}
```

Using a directory TTL

Like keys, directories in etcd can be set to expire after a specified number of seconds. You can

do this by setting a TTL (time to live) on a directory when it is created with a `PUT`:

```
curl http://127.0.0.1:2379/v2/keys/dir -XPUT -d ttl=30 -d dir=true
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 17,
        "dir": true,
        "expiration": "2013-12-11T10:37:33.689275857-08:00",
        "key": "/dir",
        "modifiedIndex": 17,
        "ttl": 30
    }
}
```

The directory's TTL can be refreshed by making an update. You can do this by making a PUT with `prevExist=true` and a new TTL.

```
curl http://127.0.0.1:2379/v2/keys/dir -XPUT -d ttl=30 -d dir=true -d prevExist=true
```

Keys that are under this directory work as usual, but when the directory expires, a watcher on a key under the directory will get an expire event:

```
curl 'http://127.0.0.1:2379/v2/keys/dir?wait=true'
```

```
{
    "action": "expire",
    "node": {
        "createdIndex": 8,
        "key": "/dir",
        "modifiedIndex": 15
    },
    "prevNode": {
        "createdIndex": 8,
        "key": "/dir",
        "dir":true,
        "modifiedIndex": 17,
        "expiration": "2013-12-11T10:39:35.689275857-08:00"
    }
}
```

Atomic Compare-and-Swap

etcd can be used as a centralized coordination service in a cluster, and `CompareAndSwap` (CAS) is the most basic operation used to build a distributed lock service.

This command will set the value of a key only if the client-provided conditions are equal to the current conditions.

Note that `CompareAndSwap` does not work with directories. If an attempt is made to `CompareAndSwap` a directory, a 102 "Not a file" error will be returned.

The current comparable conditions are:

1. `prevValue` - checks the previous value of the key.
2. `prevIndex` - checks the previous modifiedIndex of the key.
3. `prevExist` - checks existence of the key: if `prevExist` is true, it is an `update` request; if `prevExist` is `false`, it is a `create` request.

Here is a simple example. Let's create a key-value pair first: `foo=one`.

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=one
```

```
{
    "action":"set",
    "node":{
        "key":"/foo",
        "value":"one",
        "modifiedIndex":4,
        "createdIndex":4
    }
}
```

Specifying `noValueOnSuccess` option skips returning the node as value.

```
curl http://127.0.0.1:2379/v2/keys/foo?noValueOnSuccess=true -XPUT -d
value=one
# {"action":"set"}
```

Now let's try some invalid `CompareAndSwap` commands.

Trying to set this existing key with `prevExist=false` fails as expected:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevExist=false -XPUT -d
value=three
```

The error code explains the problem:

```
{
    "cause": "/foo",
    "errorCode": 105,
    "index": 39776,
    "message": "Key already
exists"
}
```

Now let's provide a `prevValue` parameter:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevValue=two -XPUT -d
value=three
```

This will try to compare the previous value of the key and the previous value we provided. If they are equal, the value of the key will change to three.

```
{
    "cause": "[two != one]",
    "errorCode": 101,
    "index": 8,
    "message": "Compare failed"
}
```

which means `CompareAndSwap` failed. `cause` explains why the test failed. Note: the condition prevIndex=0 always passes.

Let's try a valid condition:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevValue=one -XPUT -d
value=two
```

The response should be:

```
{
    "action": "compareAndSwap",
    "node": {
        "createdIndex": 8,
        "key": "/foo",
        "modifiedIndex": 9,
        "value": "two"
    },
    "prevNode": {
     "createdIndex": 8,
     "key": "/foo",
     "modifiedIndex": 8,
     "value": "one"
    }
}
```

We successfully changed the value from "one" to "two" since we gave the correct previous value.

Atomic Compare-and-Delete

This command will delete a key only if the client-provided conditions are equal to the current conditions.

Note that `CompareAndDelete` does not work with directories. If an attempt is made to `CompareAndDelete` a directory, a 102 "Not a file" error will be returned.

The current comparable conditions are:

1. `prevValue` - checks the previous value of the key.
2. `prevIndex` - checks the previous modifiedIndex of the key.

Here is a simple example. Let's first create a key: `foo=one`.

```
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=one
```

Now let's try some `CompareAndDelete` commands.

Trying to delete the key with `prevValue=two` fails as expected:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevValue=two -
XDELETE
```

The error code explains the problem:

```
{
 "errorCode": 101,
 "message": "Compare failed"
,
 "cause": "[two != one]",
 "index": 8
}
```

As does a `CompareAndDelete` with a mismatched `prevIndex`:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevIndex=1 -XDELETE
```

```
{
 "errorCode": 101,
 "message": "Compare failed"
,
 "cause": "[1 != 8]",
 "index": 8
}
```

And now a valid `prevValue` condition:

```
curl http://127.0.0.1:2379/v2/keys/foo?prevValue=one -
XDELETE
```

The successful response will look something like:

```
{
 "action": "compareAndDelete",
 "node": {
  "key": "/foo",
  "modifiedIndex": 9,
  "createdIndex": 8
 },
 "prevNode": {
  "key": "/foo",
  "value": "one",
  "modifiedIndex": 8,
  "createdIndex": 8
 }
}
```

Creating Directories

In most cases, directories for a key are automatically created. But there are cases where you will want to create a directory or remove one.

Creating a directory is just like a key except you cannot provide a value and must add the `dir=true` parameter.

```
curl http://127.0.0.1:2379/v2/keys/dir -XPUT -d dir=true
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 30,
        "dir": true,
        "key": "/dir",
        "modifiedIndex": 30
    }
}
```

Listing a directory

In etcd we can store two types of things: keys and directories. Keys store a single string value. Directories store a set of keys and/or other directories.

In this example, let's first create some keys:

We already have `/foo=two` so now we'll create another one called `/foo_dir/foo` with the value of `bar`:

```
curl http://127.0.0.1:2379/v2/keys/foo_dir/foo -XPUT -d value=bar
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 2,
        "key": "/foo_dir/foo",
        "modifiedIndex": 2,
        "value": "bar"
    }
}
```

Now we can list the keys under root `/`:

```
curl http://127.0.0.1:2379/v2/keys/
```

We should see the response as an array of items:

```
{
    "action": "get",
    "node": {
        "key": "/",
        "dir": true,
        "nodes": [
            {
                "key": "/foo_dir",
                "dir": true,
                "modifiedIndex": 2,
                "createdIndex": 2
            },
            {
                "key": "/foo",
                "value": "two",
                "modifiedIndex": 1,
                "createdIndex": 1
            }
        ]
    }
}
```


Here we can see `/foo` is a key-value pair under `/` and `/foo_dir` is a directory. We can also recursively get all the contents under a directory by adding `recursive=true`.

```
curl http://127.0.0.1:2379/v2/keys/?recursive=true
```


```
{
    "action": "get",
    "node": {
        "key": "/",
        "dir": true,
        "nodes": [
            {
                "key": "/foo_dir",
                "dir": true,
                "nodes": [
                    {
                        "key": "/foo_dir/foo",
                        "value": "bar",
                        "modifiedIndex": 2,
                        "createdIndex": 2
                    }
                ],
                "modifiedIndex": 2,
                "createdIndex": 2
            },
            {
                "key": "/foo",
                "value": "two",
                "modifiedIndex": 1,
                "createdIndex": 1
            }
        ]
    }
}
```

Deleting a Directory

Now let's try to delete the directory `/foo_dir`.

You can remove an empty directory using the `DELETE` verb and the `dir=true` parameter.

```
curl 'http://127.0.0.1:2379/v2/keys/foo_dir?dir=true' -XDELETE
```

```
{
    "action": "delete",
    "node": {
        "createdIndex": 30,
        "dir": true,
        "key": "/foo_dir",
        "modifiedIndex": 31
    },
    "prevNode": {
     "createdIndex": 30,
     "key": "/foo_dir",
     "dir": true,
     "modifiedIndex": 30
    }
}
```

To delete a directory that holds keys, you must add `recursive=true`.

```
curl http://127.0.0.1:2379/v2/keys/dir?recursive=true -XDELETE
```

```
{
    "action": "delete",
    "node": {
        "createdIndex": 10,
        "dir": true,
        "key": "/dir",
        "modifiedIndex": 11
    },
    "prevNode": {
     "createdIndex": 10,
     "dir": true,
     "key": "/dir",
     "modifiedIndex": 10
    }
}
```

Creating a hidden node

We can create a hidden key-value pair or directory by add a `_` prefix. The hidden item will not be listed when sending a `GET` request for a directory.

First we'll add a hidden key named `/_message`:

```
curl http://127.0.0.1:2379/v2/keys/_message -XPUT -d value="Hello hidden world"
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 3,
        "key": "/_message",
        "modifiedIndex": 3,
        "value": "Hello hidden world"
    }
}
```

Next we'll add a regular key named `/message`:

```
curl http://127.0.0.1:2379/v2/keys/message -XPUT -d value="Hello world"
```

```
{
    "action": "set",
    "node": {
        "createdIndex": 4,
        "key": "/message",
        "modifiedIndex": 4,
        "value": "Hello world"
    }
}
```

Now let's try to get a listing of keys under the root directory, `/`:

```
curl http://127.0.0.1:2379/v2/keys/
```

```
{
    "action": "get",
    "node": {
        "dir": true,
        "key": "/",
        "nodes": [
            {
                "createdIndex": 2,
                "dir": true,
                "key": "/foo_dir",
                "modifiedIndex": 2
            },
            {
                "createdIndex": 4,
                "key": "/message",
                "modifiedIndex": 4,
                "value": "Hello world"
            }
        ]
    }
}
```

Here we see the `/message` key but our hidden `/_message` key is not returned.

Setting a key from a file

You can also use etcd to store small configuration files, JSON documents, XML documents, etc directly. For example you can use curl to upload a simple text file and encode it:

```
echo "Hello\nWorld" > afile.txt
curl http://127.0.0.1:2379/v2/keys/afile -XPUT --data-urlencode value@afile.txt
```


```
{
    "action": "get",
    "node": {
        "createdIndex": 2,
        "key": "/afile",
        "modifiedIndex": 2,
        "value": "Hello\nWorld\n"
    }
}
```


Read Linearization

If you want a read that is fully linearized you can use a `quorum=true` GET. The read will take a very similar path to a write and will have a similar speed. If you are unsure if you need this feature feel free to email etcd-dev for advice.

Statistics

An etcd cluster keeps track of a number of statistics including latency, bandwidth and uptime. These are exposed via the statistics endpoint to understand the internal health of a cluster.

Leader Statistics

The leader has a view of the entire cluster and keeps track of two interesting statistics: latency to each peer in the cluster, and the number of failed and successful Raft RPC requests. You can grab these statistics from the `/v2/stats/leader` endpoint:

```
curl http://127.0.0.1:2379/v2/stats/leader
```

```
{
    "followers": {
        "6e3bd23ae5f1eae0": {
            "counts": {
                "fail": 0,
                "success": 745
            },
            "latency": {
                "average": 0.017039507382550306,
                "current": 0.000138,
                "maximum": 1.007649,
                "minimum": 0,
                "standardDeviation": 0.05289178277920594
            }
        },
        "a8266ecf031671f3": {
            "counts": {
                "fail": 0,
                "success": 735
            },
            "latency": {
                "average": 0.012124141496598642,
                "current": 0.000559,
                "maximum": 0.791547,
                "minimum": 0,
                "standardDeviation": 0.04187900156583733
            }
        }
    },
    "leader": "924e2e83e93f2560"
}
```


Self Statistics

Each node keeps a number of internal statistics:

- `id` : the unique identifier for the member
- `leaderInfo.leader` : id of the current leader member
- `leaderInfo.uptime` : amount of time the leader has been leader
- `name` : this member's name
- `recvAppendRequestCnt` : number of append requests this node has processed
- `recvBandwidthRate` : number of bytes per second this node is receiving (follower only)
- `recvPkgRate` : number of requests per second this node is receiving (follower only)
- `sendAppendRequestCnt` : number of requests that this node has sent
- `sendBandwidthRate` : number of bytes per second this node is sending (leader only). This value is undefined on single member clusters.
- `sendPkgRate` : number of requests per second this node is sending (leader only). This value is undefined on single member clusters.
- `state` : either leader or follower
- `startTime` : the time when this node was started

This is an example response from a follower member:

```
curl http://127.0.0.1:2379/v2/stats/self
```


```
{
    "id": "eca0338f4ea31566",
    "leaderInfo": {
        "leader": "8a69d5f6b7814500",
        "startTime": "2014-10-24T13:15:51.186620747-07:00",
        "uptime": "10m59.322358947s"
    },
    "name": "node3",
    "recvAppendRequestCnt": 5944,
    "recvBandwidthRate": 570.6254930219969,
    "recvPkgRate": 9.00892789741075,
    "sendAppendRequestCnt": 0,
    "startTime": "2014-10-24T13:15:50.072007085-07:00",
    "state": "StateFollower"
}
```


And this is an example response from a leader member:

```
curl http://127.0.0.1:2379/v2/stats/self
```


```
{
    "id": "924e2e83e93f2560",
    "leaderInfo": {
        "leader": "924e2e83e93f2560",
        "startTime": "2015-02-09T11:38:30.177534688-08:00",
        "uptime": "9m33.891343412s"
    },
    "name": "infra3",
    "recvAppendRequestCnt": 0,
    "sendAppendRequestCnt": 6535,
    "sendBandwidthRate": 824.1758351191694,
    "sendPkgRate": 11.111234716807138,
    "startTime": "2015-02-09T11:38:28.972034204-08:00",
    "state": "StateLeader"
}
```


Store Statistics

The store statistics include information about the operations that this node has handled. Note that v2 `store Statistics` is stored in-memory. When a member stops, store statistics will reset on restart.

Operations that modify the store's state like create, delete, set and update are seen by the entire cluster and the number will increase on all nodes. Operations like get and watch are node local and will only be seen on this node.

```
curl http://127.0.0.1:2379/v2/stats/store
```

```
{
    "compareAndSwapFail": 0,
    "compareAndSwapSuccess": 0,
    "createFail": 0,
    "createSuccess": 2,
    "deleteFail": 0,
    "deleteSuccess": 0,
    "expireCount": 0,
    "getsFail": 4,
    "getsSuccess": 75,
    "setsFail": 2,
    "setsSuccess": 4,
    "updateFail": 0,
    "updateSuccess": 0,
    "watchers": 0
}
```


Cluster Config

See the members API for details on the cluster management.

Page Contents