MySQL on Docker: Deploy a Homogeneous Galera Cluster with etcd

Ashraf Sharif

December 07, 2016
Posted in:
DB Ops Devops MySQL on Docker

In the previous blog post, we have looked into the multi-host networking capabilities with Docker with native network and Calico. In this blog post, our journey to make Galera Cluster run smoothly on Docker containers continues. Deploying Galera Cluster on Docker is tricky when using orchestration tools. Due to the nature of the scheduler in container orchestration tools and the assumption of homogenous images, the scheduler will just fire the respective containers according to the run command and leave the bootstrapping process to the container's entrypoint logic when starting up. And you do not want to do that for Galera - starting all nodes at once means each node will form a "1-node cluster" and you'll end up with a disjointed system.

"Homogeneousing" Galera Cluster

Related posts
MySQL on Docker: Multi-Host Networking for MySQL Containers (Part 2 - Calico)
MySQL on Docker: Introduction to Docker Swarm Mode and Multi-Host Networking
MySQL on Docker: Building the Container Image
MySQL Docker Containers: Understanding the basics

That might be a new word, but it holds true for stateful services like MySQL Replication and Galera Cluster. As one might know, the bootstrapping process for Galera Cluster usually requires manual intervention, where you usually have to decide which node is the most advanced node to start bootstrapping from. There is nothing wrong with this step, you need to be aware of the state of each database node before deciding on the sequence of how to start them up. Galera Cluster is a distributed system, and its redundancy model works like that.

However, container orchestration tools like Docker Engine Swarm Mode and Kubernetes are not aware of the redundancy model of Galera. The orchestration tool presumes containers are independent from each other. If they are dependent, then you have to have an external service that monitors the state. The best way to achieve this is to use a key/value store as a reference point for other containers when starting up.

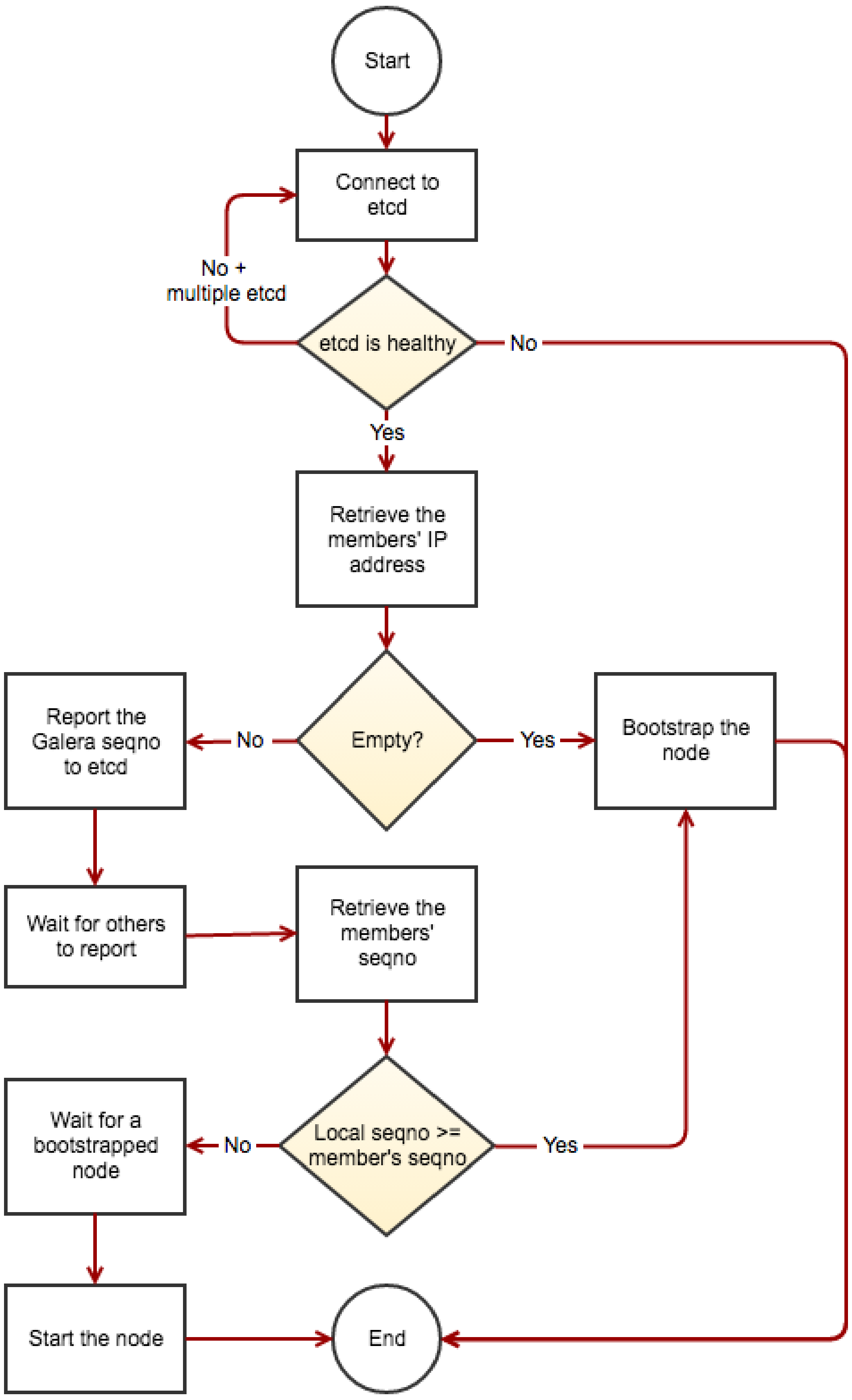
This is where service discovery like etcd comes into the picture. The basic idea is, each node should report its state periodically to the service. This simplifies the decision process when starting up. For Galera Cluster, the node that has `wsrep_local_state_comment` equal to Synced shall be used as a reference node when constructing the Galera communication address (gcomm) during joining. Otherwise, the most updated node has to get bootstrapped first.

Etcd has a very nice feature called TTL, where you can expire a key after a certain amount of time. This is useful to determine the state of a node, where the key/value entry only exists if an alive node reports to it. As a result, the node won't have to connect to each other to determine state (which is very troublesome in a dynamic environment) when forming a cluster. For example, consider the following keys:

```
{
    "createdIndex": 10074,
    "expiration": "2016-11-29T10:55:35.218496083Z",
    "key": "/galera/my_wsrep_cluster/10.255.0.7/wsrep_last_committed",
    "modifiedIndex": 10074,
    "ttl": 10,
    "value": "2881"
},
{
    "createdIndex": 10072,
    "expiration": "2016-11-29T10:55:34.650574629Z",
    "key": "/galera/my_wsrep_cluster/10.255.0.7/wsrep_local_state_comment",
    "modifiedIndex": 10072,
    "ttl": 10,
    "value": "Synced"
}
```

After 10 seconds (ttl value), those keys will be removed from the entry. Basically, all nodes should report to etcd periodically with an expiring key. Container should report every N seconds when it's alive (wsrep_cluster_state_comment=Synced and wsrep_last_committed=#value) via a background process. If a container is down, it will no longer send the update to etcd, thus the keys are removed after expiration. This simply indicates that the node was registered but is no longer synced with the cluster. It will be skipped when constructing the Galera communication address at a later point.

The overall flow of joining procedure is illustrated in the following flow chart:


Start
Connect to etcd
No + multiple etcd
etcd is healthy
No
Yes
Retrieve the members' IP address
Report the Galera seqno to etcd
No
Empty?
Yes
Bootstrap the node
Wait for others to report
Retrieve the members' seqno
Wait for a bootstrapped node
No
Local seqno >= member's seqno
Yes
Start the node
End

We have built a Docker image that follows the above. It is specifically built for running Galera Cluster using Docker's orchestration tool. It is available at Docker Hub and our Github repository. It requires an etcd cluster as the discovery service (supports multiple etcd hosts) and based on Percona XtraDB Cluster 5.6. The image includes Percona Xtrabackup, jq (JSON processor) and also a shell script tailored for Galera health check called report_status.sh.

You are welcome to fork or contribute to the project. Any bugs can be reported via Github or via our support page.

Deploying etcd Cluster

etcd is a distributed key value store that provides a simple and efficient way to store data across a cluster of machines. It's open-source and available on GitHub. It provides shared configuration and service discovery. A simple use-case is to store database connection details or feature flags in etcd as key value pairs. It gracefully handles leader elections during network partitions and will tolerate machine failures, including the leader.

Since etcd is the brain of the setup, we are going to deploy it as a cluster daemon, on three nodes, instead of using containers. In this example, we are going to install etcd on each of the Docker hosts and form a three-node etcd cluster for better availability.

We used CentOS 7 as the operating system, with Docker v1.12.3, build 6b644ec. The deployment steps in this blog post are basically similar to the one used in our previous blog post.

1. Install etcd packages:

```
$ yum install etcd
```

2. Modify the configuration file accordingly depending on the Docker hosts:

```
$ vim /etc/etcd/etcd.conf
```

For docker1 with IP address 192.168.55.111:

```
ETCD_NAME=etcd1
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://0.0.0.0:2380"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.55.111:2380"
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.55.111:2380,etcd2=http://192.168.55.112:2380,etcd3=http://192.168.55.113:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-1"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

For docker2 with IP address 192.168.55.112:

```
ETCD_NAME=etcd2
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://0.0.0.0:2380"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.55.112:2380"
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.55.111:2380,etcd2=http://192.168.55.112:2380,etcd3=http://192.168.55.113:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-1"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

For docker3 with IP address 192.168.55.113:

```
ETCD_NAME=etcd3
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_PEER_URLS="http://0.0.0.0:2380"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.55.113:2380"
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.55.111:2380,etcd2=http://192.168.55.112:2380,etcd3=http://192.168.55.113:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster-1"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

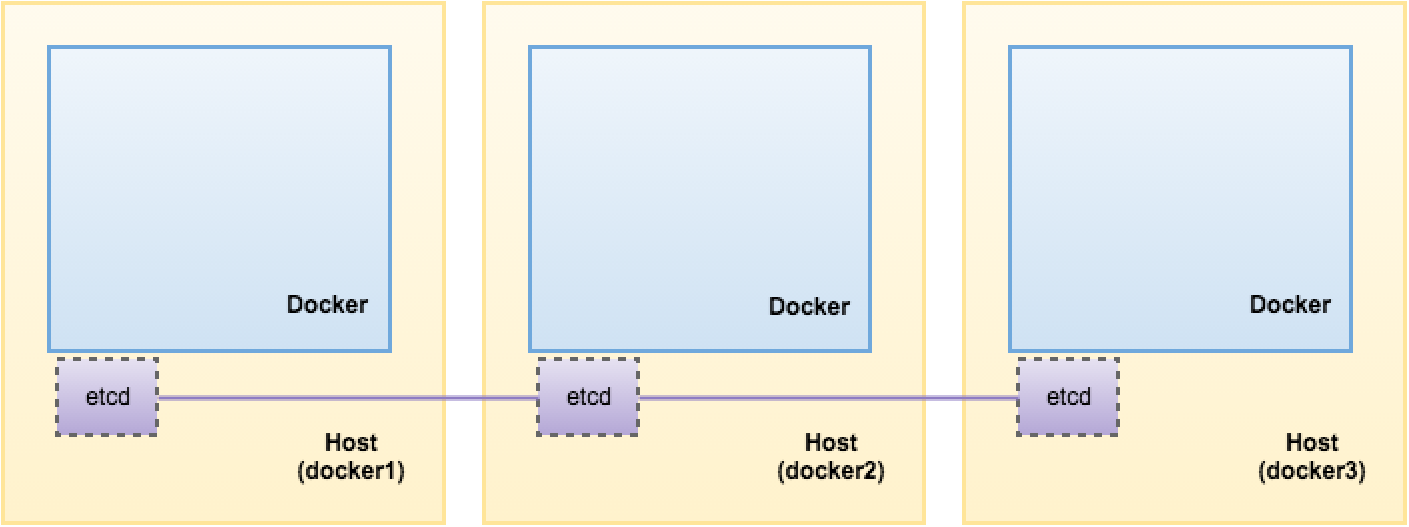
3. Start the service on docker1, followed by docker2 and docker3:

```
$ systemctl enable etcd
$ systemctl start etcd
```

4. Verify our cluster status using etcdctl:

```
[docker3 ]$ etcdctl cluster-health
member 2f8ec0a21c11c189 is healthy: got healthy result from http://0.0.0.0:2379
member 589a7883a7ee56ec is healthy: got healthy result from http://0.0.0.0:2379
member fcacfa3f23575abe is healthy: got healthy result from http://0.0.0.0:2379
cluster is healthy
```

That's it. Our etcd is now running as a cluster on three nodes. The below illustrates our architecture:



Deploying Galera Cluster

Minimum of 3 containers is recommended for high availability setup. Thus, we are going to create 3 replicas to start with, it can be scaled up and down afterwards. Running standalone is also possible with standard "docker run" command as shown further down.

Before we start, it's a good idea to remove any sort of keys related to our cluster name in etcd:

```
$ etcdctl rm /galera/my_wsrep_cluster --recursive
```

Ephemeral Storage

This is a recommended way if you plan on scaling the cluster out on more nodes (or scale back by removing nodes). To create a three-node Galera Cluster with ephemeral storage (MySQL datadir will be lost if the container is removed), you can use the following command:

```
$ docker service create \
--name mysql-galera \
--replicas 3 \
-p 3306:3306 \
--network galera-net \
--env MYSQL_ROOT_PASSWORD=mypassword \
--env DISCOVERY_SERVICE=192.168.55.111:2379,192.168.55.112:2379,192.168.55.113:2379 \
--env XTRABACKUP_PASSWORD=mypassword \
--env CLUSTER_NAME=my_wsrep_cluster \
severalnines/pxc56
```

Persistent Storage

To create a three-node Galera Cluster with persistent storage (MySQL datadir persists if the container is removed), add the mount option with *type=volume*:

```
$ docker service create \
--name mysql-galera \
--replicas 3 \
-p 3306:3306 \
--network galera-net \
--mount type=volume,source=galera-vol,destination=/var/lib/mysql \
--env MYSQL_ROOT_PASSWORD=mypassword \
--env DISCOVERY_SERVICE=192.168.55.111:2379,192.168.55.112:2379,192.168.55.113:2379 \
--env XTRABACKUP_PASSWORD=mypassword \
--env CLUSTER_NAME=my_wsrep_cluster \
severalnines/pxc56
```

Custom my.cnf

If you would like to include a customized MySQL configuration file, create a directory on the physical host beforehand:

```
$ mkdir /mnt/docker/mysql-config # repeat on all Docker hosts
```

Then, use the mount option with "type=bind" to map the path into the container. In the following example, the custom my.cnf is located at `/mnt/docker/mysql-config/my-custom.cnf` on each Docker host:

```
$ docker service create \
--name mysql-galera \
--replicas 3 \
-p 3306:3306 \
--network galera-net \
--mount type=volume,source=galera-vol,destination=/var/lib/mysql \
--mount type=bind,src=/mnt/docker/mysql-config,dst=/etc/my.cnf.d \
--env MYSQL_ROOT_PASSWORD=mypassword \
--env DISCOVERY_SERVICE=192.168.55.111:2379,192.168.55.112:2379,192.168.55.113:2379 \
--env XTRABACKUP_PASSWORD=mypassword \
--env CLUSTER_NAME=my_wsrep_cluster \
severalnines/pxc56
```

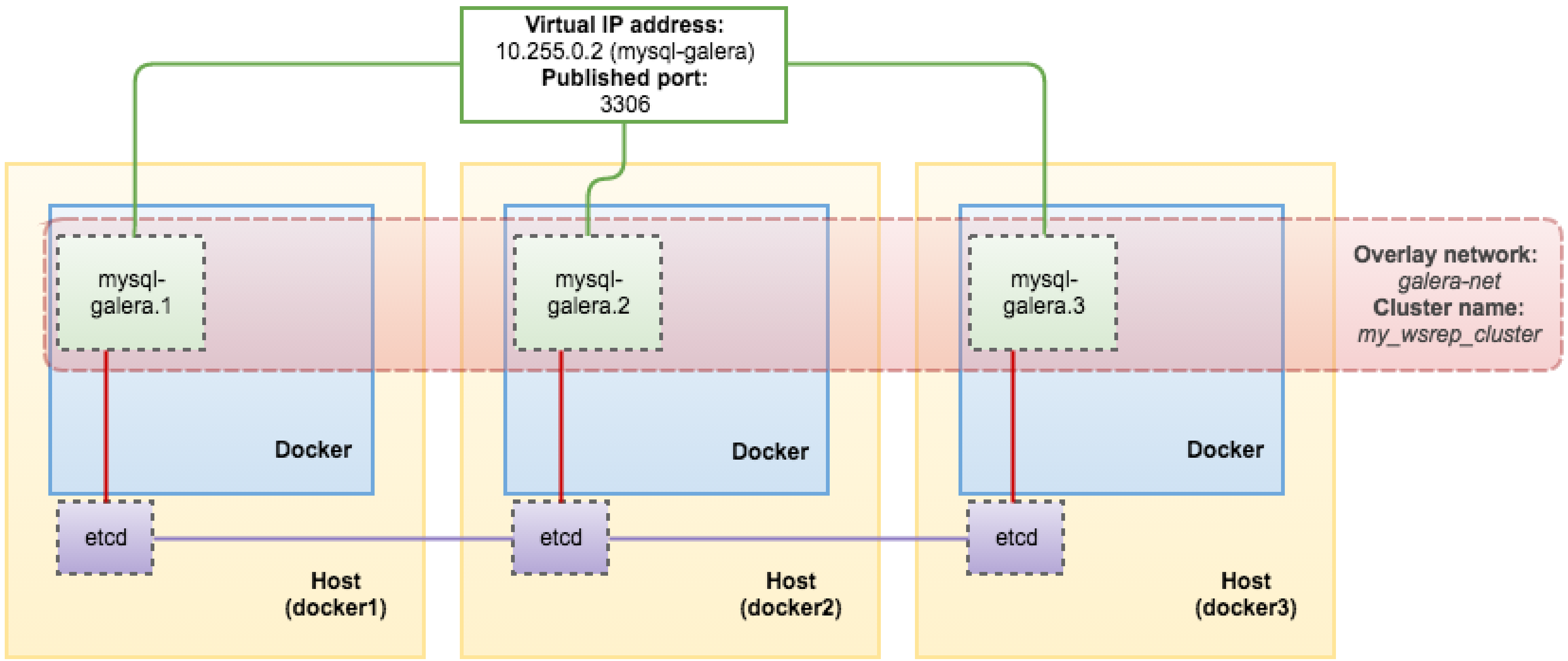
Wait for a couple of minutes and verify the service is running (CURRENT STATE = Running):

```
$ docker service ls mysql-galera
ID                          NAME            IMAGE               NODE            DESIRED STATE   CURRENT STATE           ERROR
2vw40cavru9w4crr4d2fg83j4   mysql-galera.1  severalnines/pxc56  docker1.local   Running         Running 5 minutes ago
1cw6jeyb966326xu68lsjqoe1   mysql-galera.2  severalnines/pxc56  docker3.local   Running         Running 12 seconds ago
753x1edjlspqxmte96f7pzxs1   mysql-galera.3  severalnines/pxc56  docker2.local   Running         Running 5 seconds ago
```

External applications/clients can connect to any Docker host IP address or hostname on port 3306, requests will be load balanced between the Galera containers. The connection gets NATed to a Virtual IP address for each service "task" (container, in this case) using the Linux kernel's built-in load balancing functionality, IPVS. If the application containers reside in the same overlay network (galera-net), then **use the assigned virtual IP address instead.** You can retrieve it using the inspect option:

```
$ docker service inspect mysql-galera -f "{{ .Endpoint.VirtualIPs }}"
[{89n5idmdcswqqha7wcswbn6pw 10.255.0.2/16} {1ufbr56pyhhbkbgtgsfy9xkww 10.0.0.2/24}]
```

Our architecture is now looking like this:



As a side note, you can also run Galera in standalone mode. This is probably useful for testing purposes like backup and restore, testing the impact of queries and so on. To run it just like a standalone MySQL container, use the standard docker run command:

```
$ docker run -d \
-p 3306 \
--name=galera-single \
-e MYSQL_ROOT_PASSWORD=mypassword \
-e DISCOVERY_SERVICE=192.168.55.111:2379,192.168.55.112:2379,192.168.55.113:2379 \
-e CLUSTER_NAME=my_wsrep_cluster \
-e XTRABACKUP_PASSWORD=mypassword \
severalnines/pxc56
```