Handling multi-layer database replication with Docker Swarm to increase availability, data consistency and load-balancing performance

Andrea Cardiota
andrea.cardiota@studio.unibo.it

December 2019

Databases replication and sharding operations are not widely spread as the use of databases themselvers. This is amenable to its complexity that might discourage from embarking in the development steps and action needed to deploy it. This project aims to discover a potential solution on how to integrate such technologies with Docker Swarm's orchestration capabilities to improve load-balancing perfomance and data consistency, while simplifying the process and providing a basic "*know-how*", from a technological standpoint, on how all the services used in the operation work.

Goals The end result of this work will be a detailed report on the core aspects of the technologies used, with the addition of some testbed scenario to put it to the test and to evaluate technical limits. To achieve this, I'll take advantage of a database technology that needs to have robust and scalable performance, combined with the possibility of choosing from a variety of storage engines. This is particulaty helpful when planning to develop a distributed storage system, giving many options to best adapt to the desired class of services in mind.
Most of today's database replication techniques involve a master-slave approach, which means that actually only reads operations can be off-loaded to the distributed cluster of database replicas, leaving the heavy weight workload of writes to the main master node.
Of course, this only partially brings us to the goal of true replication, full scalablity and load-balancing, resulting in the bigger and main goal of this project being finding an approach that sets up a truly distributed storage system and integrates it into Docker Swarm to reach maximum traffic, stability, consistency and scalability performance.

Work plan

1. Learning the *modus operandi* of Docker and Docker Swarm, including the notions of stack and services;
2. Analysis of how a three-tier (client - application - storage) application, with replicated application layer and non-replicated storage layer, work;
3. Analysis and building of a distributed database cluster into a Docker Swarm multiple containers system;
4. Evaluation, basing on the output of the previous phase, of a plausible solution which adapts well to the project requirements and, in case there's not such a solution, study of a clever way to bypass the aforesaid issues.

Index

Chapter 1

State of the art

Virtual Machines VMs are software that, like a physical computer, run an operating system and applications. A VM is comprised of a set of specification and configuration files and is backed by the physical resources of a host. Every VM has virtual devices that provide the same functionality as physical hardware and have additional benefits in terms of portability, manageability, security and the possibility to run multiple operating systems concurrently on a single host.

Containers on the other hand, are a solution to the problem of *how* to get software to run reliably when moved from one computing environment to another. This could be from a developer's laptop to a test environment, from a staging environment into production and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another without needing any further configuration procedure. A *Docker* container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

1.1 Docker

But *what* is Docker? Docker is a tool that was engineered to ship, deploy and run software applications using, in fact, containers. A docker container, unlike a VM, doesn't require or include a separate operating system. Instead, it relies on the Linux kernel's functionalities and uses resource isolation. The primary Docker focus is to automate the deployment of applications inside software containers and the automation of operating system level virtualization on Linux.
newline So why do we favour containers over VMs? Firstly, they are more lightweight than VMs, use a lot more resources and boot up in seconds, making them more resource-efficient both technically and economically. Also, they need a reduced size for snapshots, are quicker spinning up apps, reduce and simplify security updates and need less code to transfer, migrate and upload workloads.

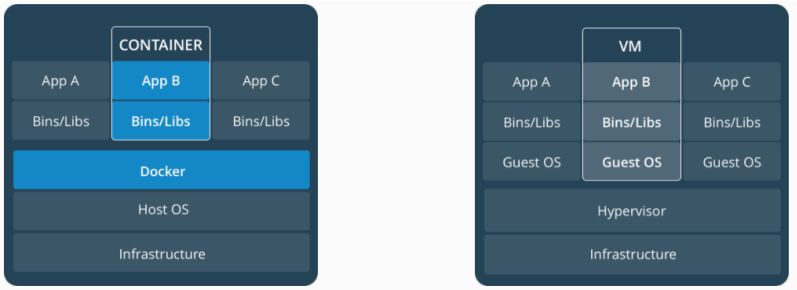

Figure 1.1: Containers stack vs VMs stack

"So, do we always prefer containers over VMs?"

There's no easy answer to this, because it really depends on what we are trying to build. As a rule of thumb, VMs are a better choice for running apps that require all of the operating system's resources and functionality when there's the need to run multiple applications on servers, or have a wide variety of operating systems to manage. Containers instead are a better choice when the biggest priority is maximizing the number of applications running on a minimal number of servers.
In the general case, the ideal setup is likely to include both. With the current state of virtualization technology, the flexibility of VMs and the minimal resource requirements of containers work together to provide environments with maximum functionality, as we'll see in the development phase later on.

1.1.1 Docker Engine

The Docker Engine is the underlying client-server technology that builds and runs containers using Docker's components and services. The Docker Engine comprises the *Docker Daemon*, a REST API and the Command Line Interface (CLI) that talks to the daemon through the API, the Swarm Mode, load balancing and distributed storage features. The Docker Engine supports the tasks and workflows involved to build, ship and run container-based applications. It creates a server-side daemon process that hosts images, containers, networks and storage volumes. It is built to be declarative, which means an administrator programs a specific set of conditions as the desired state. The Engine automatically also adjusts settings and conditions to ensure the actual state and the desired state match at all times.

1.1.2 Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does builds, runs and distributes containers. The Docker client and daemon can run on the same system. The Docker client and daemon communicate using a REST API.

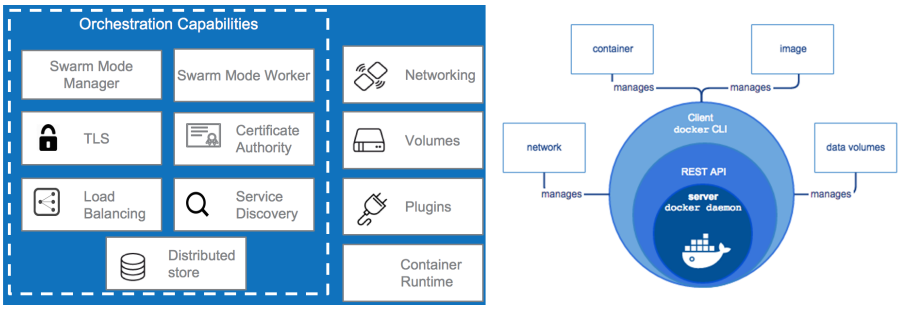

Figure 1.2: Docker Engine features

Daemon The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks and volumes. A daemon has the ability to communicate with other daemons to manage Docker services.

Client The Docker client is the primary way that many Docker users interact with Docker. When commands such as *docker run* are invoked, the client sends these commands to the daemon, which handles them. The Docker client has the ability to communicate with more than one daemon.

Images A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container. It essentially is built from the instructions for an application to run, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container. To build a custom image, it's necessary to create a Dockerfile with very simple syntax to define every step needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image.

1.1.3 Dockerfile

Say, for istance, there's the need to pull some open source software image for development. Before it is possible to develop with that container, there are a number of modifications needed to make to the image (such as upgrading software and adding the necessary development packages for the job at hand). To solve this problem, the best practise is to write a Dockerfile for each variation. Basically, the Dockerfile builds a private file system which may also contain some metadata to precisely run a container basing off the current image. Once we have our Dockerfile constructed, we can build the same image over and over, without having to do it manually everytime.

1.1.4 Stack and services

A stack is a group of interrelated services that share dependencies and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application. Docker services meanwhile allow us to scale containers across multiple Docker daemons, which all work together as a *Swarm* with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate

using the Docker API. A service allows us to define the desired state, such as the number of replicas of the service that must be available. By default, the service is load-balanced across all worker nodes.

1.1.5 Deployment

To deploy a container, the following command is needed:

```
$ docker run --name NAME -p PORTS IMAGE
```

where:

- NAME equals a name we want to give the container;
- PORTS equals to the ports we want to use (in the form of NETWORK PORT:CONTAINER PORT);
- IMAGE the image to be used for the container.

If some container was still running, we'd have to kill it before we could deploy another container on the same port, otherwise the ports would conflict, preventing the container from deploying and aborting the operation. To access a running container, the container ID is needed. With the ID in hand, run the command:

```
$ docker exec -it CONTAINER_ID bash
```

This is the most basic Docker deployment option. There's the option to use Docker Cloud to help manage the dockerized application on services like Amazon Web Services, Google Cloud Platform and Microsoft Azure. Alternately, once we initiated the Swarm Mode, we could also deploy to the swarm.

1.2 Swarm mode

A swarm is a cluster of one or more computers running Docker. It can consist of one or more machines and it runs in a Master-Salve configuration. With the introduction of swarm mode, Docker enabled built-in orchestration, clustering and scheduling capabilities. Features for handling multiple machines, like health checks, load balancing and other are now part of Docker. Unlike the legacy standalone Swarm, the functionality is now built into the Docker engine itself. It was a separate service, not part of the docker daemon and interacted with the Docker API. With Swarm Mode enabled and a docker-compose.yml file - which we'll study later on) - we can use docker stack commands which bring up stacks of services. If we are using swarm mode, we are dealing on a higher abstraction level, which means we are using services instead of containers. We can still deal with both kinds on the same machine. When a container is running in service mode, swarm mode takes over and manages the lifecycle of the service. It works to bring up a desired number of replicas, as specified in the definition. Every swarm object could - and *should* - be descripted in a *stack* file; there *YAML* files describe all the application components and configurations and they can be used to easily create and destroy it in all swarm environments. A key difference between standalone containers and swarm services is that only swarm managers

can manage a swarm, while standalone containers can be started on any daemon. Docker daemons can participate in a swarm as managers, workers, or both.
As the Docker Swarm documentation says, swarm mode comes with a bunch of different features, like:

- **Cluster management integrated with Docker Engine:** using the Docker Engine CLI it's possible to create a swarm of Docker Engines where we can deploy application services. Additional orchestration software to create or manage a swarm isn't needed;
- **Declarative service model:** the Docker Engine uses a declarative approach to let define the desired state of the various services in the application stack;
- **Scaling:** for each and every service, we can declare the number of tasks we want to run. When we scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state;
- **Multi-host networking:** we can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application;
- **Service discovery:** swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. We can query every container running in the swarm through a DNS server embedded in the swarm;
- **Load balancing algorithm:** it's possible to expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.

Nodes A node is an instance of the Docker engine participating in the swarm. To deploy an application to a swarm, submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes, who dispatch them Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. The worker nodes notify the manager node of the current state of assigned tasks so that the manager can maintain the desired state of each worker.

1.3 Docker Compose

When using Docker extensively, the management of several different containers quickly becomes problematic .Docker Compose is a tool that helps overcome this problem and easily handle multiple containers at once. Docker Compose works by applying many rules declared within a single docker-compose.yml configuration file. These YAML rules, both human-readable and machine-optimized, provide us an effective way to snapshot the whole project in a few lines. Almost every rule replaces a specific Docker command so that in the end we just need to run:

```
$ docker-compose up
```

It's possible to get a very large number of configurations applied by Compose. The main requirement is to specify the version of the Compose file format, at least one service and optionally volumes and networks, so that the general form will look like this:

```
version: '3'
    services:
        service 1:
        .
        .
        .
        service 2:
        .
        .
        .
    volumes:
        ...
    networks:
        ...
```

Using Compose is three-step process. Firstly, we need to define the application's environment with a Dockerfile so it can be reproduced anywhere. Then, we need to define the services that make up the application in the docker-compose.yml as is explained above; lastly, we need to run the trigger the Compose start command to run the entire app. Docker Compose has multiple commands to manage the whole lifecycle of the dockerized application. In particular, it can start, stop and rebuild services and view the status of running services. Luckily for us, Docker Compose creates a virtual network for all of the containers/nodes. So by default, each and everyone of them can access all of the others defined in the compose file and their host names match their services name, which is very convenient

1.3.1 Docker volumes

Docker images are stored as series of read-only layers. When we start a container, Docker takes the read-only image and adds a read-write layer on top. If the running container modifies an existing file, the file is copied out of the underlying read-only layer and into the top-most read-write layer where the changes are applied. When a Docker container is deleted, relaunching the image will start a fresh container without any of the changes made in the previously running container, meaning that those changes are lost. Docker calls this combination of read-only layers with a read-write layer on top a Union File System. In order to be able to persist data, Docker came up with the concept of volumes, which also doubles up as a way to share data between multiple containers. So, volumes are directories that are outside of the default Union File System and exist as normal directories and files on the host original filesystem. Meanwhile, bind mounts - which are another way of persisting data, are dependent on the directory structure of the host machine, while volumes are handled single-handed by Docker. We can manage volumes using the Docker APIs. Also, volumes can be *more* safely shared among multiple containers. In addition,

volumes are often a better choice than persisting data in a container's writable layer, because a volume doesn't increase the size of the containers using it.

1.4 Database replication

Database replication is the sharing of transactional data across multiple servers to ensure consistency between redundant database nodes. In other words, replication is the operation intended for the copying of data from a database in one computer or server to a database in another so that all users share the same level of information at any given time. The result is a distributed database in which users can quickly read and write data relevant to their tasks without interfering with the work of other entities. Numerous elements contribute to the overall process of creating and managing database replication. The case of database replication is found in applications that connect a primary storage location and a secondary location that is often off site, but it may also be on site. While fast transactions and queries are a major reason to employ database replication, having multiple copies of databases is also valuable for testing and operational reporting. Overall, Distributed Database Management Systems (DDBMS) work to ensure that every change on the data at any given location is automatically reflected in the data stored at all the other locations. There are multiple types of database replication, each serving a different purpose.

1.4.1 Streaming replication

In this approach, data is replicated from the primary node to the secondary node. This comes with a Master-Slave approach and it bring a number of setbacks which are: difficulties in introducing a new secondary, since it would require the replication of the entire state, which can be resource intensive. Secondly, lack of built-in monitoring and failover. A secondary node has to be promoted to a primary in case of the latter failure. Often this promotion may result in data inconsistency during the primary's absence.

1.4.2 Write-Ahead Log Shipping

This approach employs the streaming replication approach since the secondaries are reconstructed from a backup made by the primary which undertakes a full database backup after each day besides an incremental backup every 60 seconds. The advantage with this approach is that no additional load is subjected to the primary until the secondaries are close enough to the primary such that they start streaming the Write Ahead Log (WAL) in order to catch up with it. With this approach, we can add or remove replicas without impacting the performance of the database.

Trial and error with PostgreSQL This very technique was firstly used as a practial approach in this project, using PostgreSQL. The way that replication in PostgreSQL works is that the replicas load changes from the WAL files generated by the primary, which are guaranteed to be in order. The primary is configured to keep a certain number of WAL files and thus if it removes a log before the replica has a chance to load it, then the replica may not be able to load it and

so becomes out-of-sync. There are several reasons why a replica can become out-of-sync from your primary database:

- The replica has not communicated with the primary for a while, due to a network issue or the replica being down;
- The primary database is writing changes faster than the replica can process the changes;
- Something changed in your primary or replica configuration that is causing the WAL logs to either not be shipped or processed.

So, in the end, this was not a great choice.

1.4.3 Distributed Replicated Block Device

DRBD enables automatic failover capabilities to prevent downtime. With a Distributed Replicated Block Device, whenever new data is written to disk, the block device uses the network to replicate data to the second node. Through redundancy, it is possibile to protect from downtime and data loss and get minimal to zero interruption during software and framework-related operations. To this point, a logical choice for this project would be PostgreSQL, which - among other things - employs the following three types of replication:

1.4.4 Logical Replication

Logical replication allows a database server to send a stream of data modifications to another server. PostgreSQL logical replication constructs a stream of logical data modifications from the WAL. It allows the data changes from individual tables to be replicated and it doesn't require a particular server to be designated as a master or a replica but allows data to flow in multiple directions.

1.4.5 Trigger-Based Master-Standby Replication

A master-standby replication setup sends all data modification queries to the master server. The master server asynchronously sends data changes to the standby server. The standby can answer read-only queries while the master server is running. The standby server is ideal for data warehouse queries.

1.4.6 Statement-Based Replication Middleware

This is also a great option. With statement-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries must be sent to all servers, so that every server receives any changes. But read-only queries can be sent to just one server, allowing the read workload to be distributed among them.

1.4.7 Synchronous Multimaster Replication

In synchronous multimaster replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive

locking and commit delays, leading to poor performance. Read requests can be sent to any server. Synchronous multimaster replication is best for mostly read workloads, though its big advantage is that any server can accept write requests, there is no need to partition workloads between master and standby servers.

Chapter 2

Desing & Deployment

Before going into more detailed phases of development using Docker Swarm, it is fundamental to understand how a n-tier architecture actually works. The name comes from the fact the every such architecture is engineered to have processing, data and presentation functions physically and logically separated. This means that these different functions are hosted on several machines or clusters, ensuring that services are provided without resources being shared and these services are delivered at top capacity. This way, the software gains from being able to handle services in the best possible way, but it's also easier to manage. It's also easier to pinpoint where it originates. A 3-tier application architecture is a modular client-server architecture that consists of a presentation tier, an application tier and a data tier. The data tier stores information, the application tier handles logic and the presentation tier is a graphical user interface that communicates with the other two tiers. The three tiers are logical, not physical, and may or may not run on the same physical server. This project features, in fact, a client - application - storage architecture. The benefits of using a 3-tiered architecture include improved horizontal scalability, performance and availability. Because the programming for a tier can be changed or relocated without affecting the other tiers, the 3-tier model makes it easier to continually evolve an application as new needs arise.

2.1 Three-tiered architecture

As said before, one of the pre-requirements of this project was, in fact, to understand how a three-tiered - replicated application layer/non-replicated storage layer - work. In order to do so, I deployed an educational application that does exactly this, using Docker Swarm. It can be found here on GitHub.
The deployment on a swarm is actually very simple. It just need to initialise the swarm by:

```
$ docker swarm init
```

the, we use a Docker Compose file that deploys five different services: a *vote* front-end, a *result* front-end, a Redis that work with a .NET worker node and a database service. We deploy it by:

```
$ docker stack deploy --compose-file docker-stack.yml dummyAppName
```

Having done that, we can access to web app via localhost at the port 5000. We can simulate multiple nodes votes by accessing localhost on multiple browsers and once we vote, we'll see the result front-end side of the application updating in real-time, showing that we have multiple application layers reading and writing to a non-replicated database.

2.2 Technologies choices

Based on the replication options that we discussed, there a lot a valid ones but actually no one that gets us to the main objective of this project, which is reaching a 100% truly distributed database system that allow us to read **and** write on all the nodes participating in the cluster.

2.2.1 MariaDB with Galera Cluster

To do this, I choose MariaDB in combination with Galera Cluster. MariaDB is an open source relational database management system (DBMS) which is the replacement for the widely used MySQL database technology. MariaDB has high compatibility with MySQL and exact matching with MySQL APIs and commands. In fact, its API and protocol are compatible with those used by MySQL, and also some other features to support local non-blocking operations. MariaDB has a significantly high number of new features, which makes it better in terms of performance and user-orientation.

The main reason I've choosen to employ MariaDB in this project is its ability to create, deploy and manage a multi-node database cluster via Galera Cluster, which is a synchronous multi-master cluster. It is available on Linux only and only supports the XtraDB/InnoDB storage engines which was one of the initial requisites. Its main feature include: synchronous replication, active-active multi-master topology, reading and writing to any cluster node at any given time, automatic membership control, failed nodes drop from the cluster, automatic node joining, true parallel replication and direct client connections. Also, it comes with numerous benefits such as no slave lag, no lost transactions and read/write scalability.

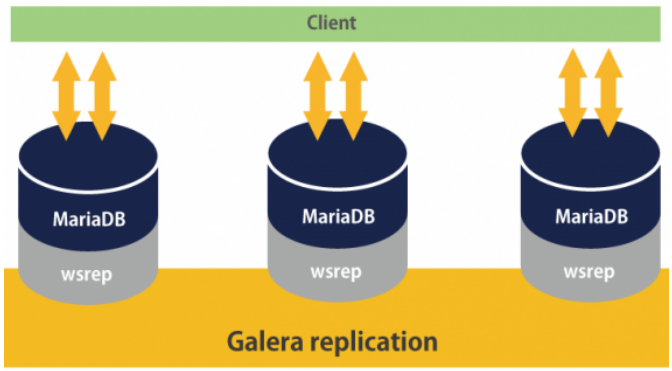

Figure 2.1: MariaDB replication with Galera Cluster

2.2.2 ClusterControl

ClusterControl is an agentless management and automation software for database clusters. It helps deploy, monitor, manage and scale database servesr/clustesr directly from ClusterControl user interface, which can be both graphical and command line. It's not mandatory to use when creating a database cluster, in fact there's no difference when in the deployment stage. The real difference comes in when there's the need to backup, restore, recovery and most of all monitor what's going on within the cluster.

2.3 Integration with Docker Swarm

Most of the work is done within the database cluster itself and its structure configuration. The use of Docker and its Swarm orchestration capabilities is used to launch new containers, each within its own replicated database replica, without needing manual configuration everytime we need to scale up (or down) in terms of nodes.

We start from an image that is engineered to work with a specific Linux distribution, called CentOS, that I choose for its lightness and excellent integration with MariaDB. This Docker Image comes with a CentOS 6 base image, an OpenSSH client and server packages, *curl* and a MySQL client. Having done that, it's now necessary to set some mandatory environment variables:

CC_HOST = {string}

the value of ClusterControl, being an IP address, hostname or service name. This container will try to connect to CC_HOST port 80 using curl to download the SSH key and register itself to CMON database if AUTO_DEPLOYMENT is on via mysql client;

AUTO_DEPLOYMENT = {0 or 1}

defaults to 1 (enabled). If set to 0, this container will only set up passwordless SSH by downloading the public key from CC_HOST and skip registering it in the CMON database. Without this registration, ClusterControl won't be able to discover the container for automatic deployment;

CMON_PASSWORD = {string}

MySQL password for user *cmon*. Defaults to *cmon*. This value will be used to register the container into the CMON database via MySQL client;

CLUSTER_NAME = {string}

It distinguishes the cluster with others from ClusterControl perspective. Must be unique;

CLUSTER_TYPE = {string}

Type of supported cluster that ClusterControl would deploy on this container. Supported values are *galera* and *replication*, which is experimental;

INITIAL_CLUSTER_SIZE = {integer}

Default is 3. This indicates how ClusterControl should treat newly registered containers, whether they are for new deployments or for scaling. For example, if the value is 3, ClusterControl will wait for 3 containers to be running and registered into the CMON database before starting the cluster deployment job. Otherwise, it waits 30 seconds for the next cycle and retries. The next containers (4th, 5th and Nth) will fall under the "Add Node" job instead:

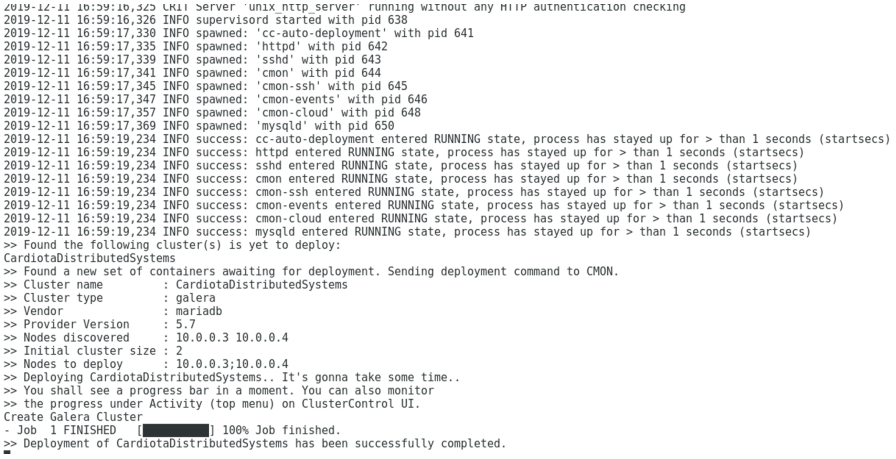

Figure 2.2: Setting an initial cluster size of 2 and sequential deploy.

VENDOR = {string}

Database vendor to use. Default is percona. Other supported values are mariadb, codership and oracle;

PROVIDER_VERSION = {string}

Defaults to 5.7, which is the database version by the chosen vendor/provider. For MariaDB is needed specify 5.5 or 10.x;

DB_ROOT_PASSWORD = {string}

The database root password for the database server. In this case, it should be MySQL root password.

Networking To be able to merge together the swarm nodes and the database replica, other than specifying it in the docker-compose file, it is necessary to make sure that they are on the same network. In particular, we can tak advantage of the *overlay network* that is created by Docker swarm as explained in the following schema.

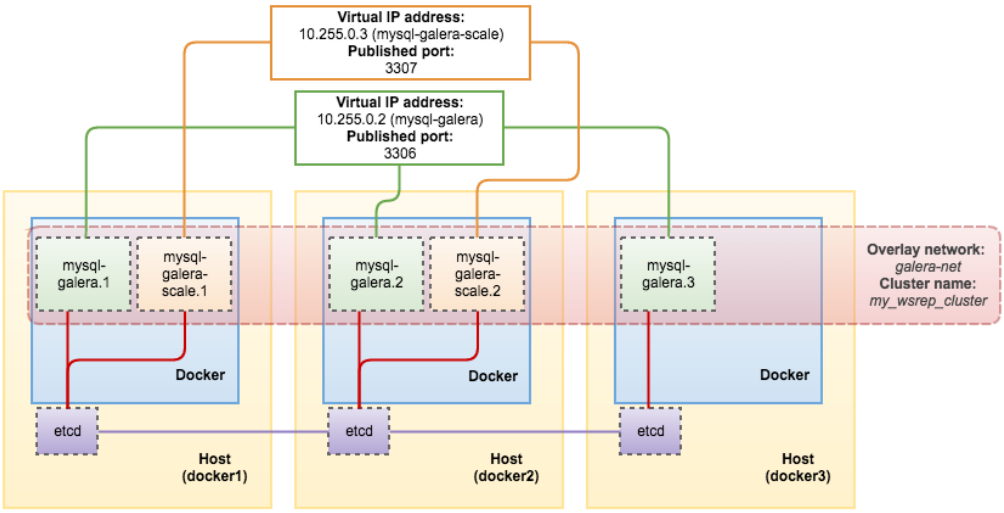

Figure 2.3: Merging database and swarm networks.

The goal is making Galera Cluster, which is a stateful service, run perfectly on Docker Swarm. So, we need to merge the two together. To run a Galera Cluster in a stable way we need to make sure to pass *health checks*, which means that each of the stateful containers must pass the Docker health checks, to ensure it achieves the correct state before being included into the active load balancing set. As for data persistency, whenever a container is replaced, it has to be booted from the last known working configuration or we might lose data, making totally useless all the work that led us to this point.

2.4 Systems behaviour

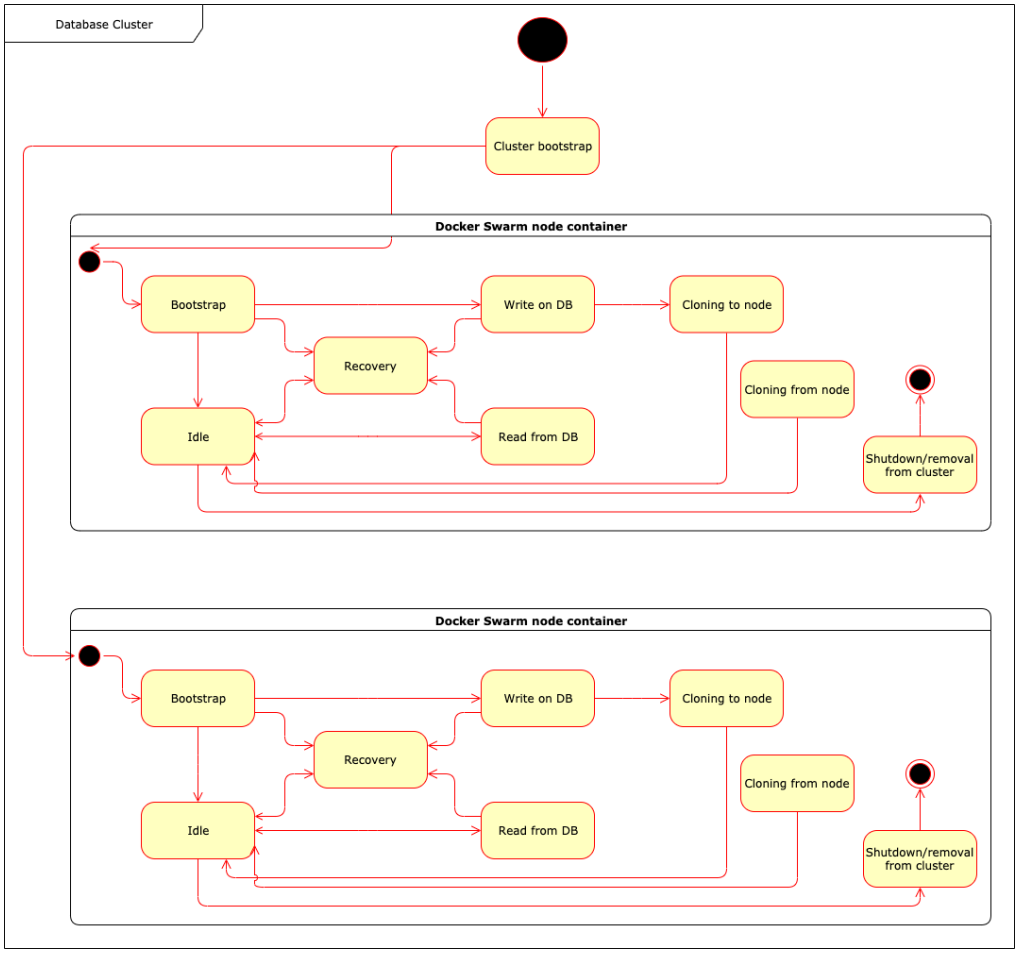

Figure 2.4: Database cluster UML state diagram. Notice that the *Cloning to node* and *Cloning from node* states don't actually have in/out transactions as this is better described in the following UML sequence diagram. This is due to the lack of semantics provided by the UML state diagram for this scenario.

2.5 Systems interaction

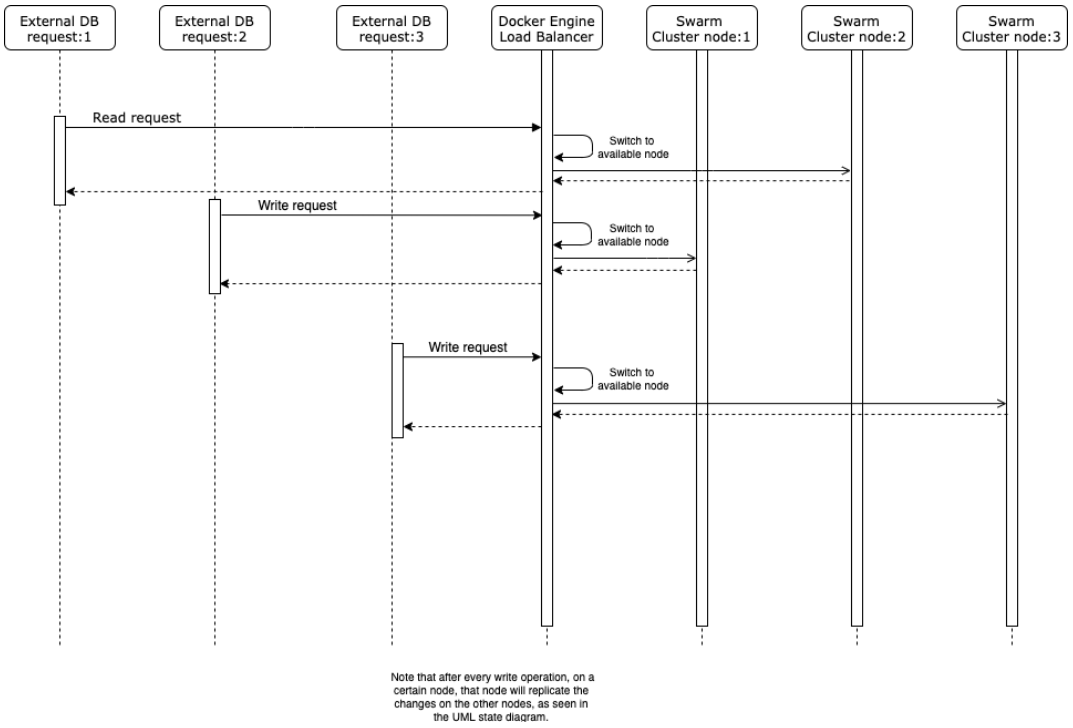

Figure 2.5: Database cluster UML sequence diagram.

2.6 Deployment procedure

The following steps show how we can deploy a Galera Cluster automatically, which means we only need to execute the *docker run* commands and wait until the deployment completes. There's a shorter method, using Docker Compose, that we'll see later on.
First thing first, we need to run th ClusterControl container, which is not an actual node partecipating in the cluster, via the command:

```
$ docker run -d --name clustercontrol -p 5000:80 severalnines/clustercontrol
```

Then, having already found the ClusterControl container's IP address, we'll run the database containers that will take part in the cluster as nodes; in this case, we just have two replicas:

```
docker run -d --name node1 -p 6661:3306 -e CC_HOST=${CC_IP} -e
    CLUSTER_TYPE=galera -e CLUSTER_NAME=CardiotaDistributedSystems -e
    INITIAL_CLUSTER_SIZE=3 severalnines/centos-ssh

docker run -d --name node2 -p 6662:3306 -e CC_HOST=${CC_IP} -e
    CLUSTER_TYPE=galera -e CLUSTER_NAME=CardiotaDistributedSystems -e
    INITIAL_CLUSTER_SIZE=3 severalnines/centos-ssh
```

There's a better way to use the swarm and its containers to deploy database nodes inside them, and that is using docker services, such as:

```
$ docker service create --name cc_clustercontrol -p 5000:80
    --replicas 1 severalnines/clustercontrol
```

for the ClusterControl node and sequentially similar commands to run the database nodes. Alternately, we can combine the ClusterControl service together with the database container service and form a stack in a Docker Compose (docker-compose.yml) file, which looks like this:

```
version: '3'

services:

  galera:
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failure
        delay: 10s
    image: severalnines/centos-ssh
    ports:
      - 3306:3306
    environment:
      CLUSTER_TYPE: "galera"
      CLUSTER_NAME: "CardiotaDistributedSystems"
      VENDOR: "mariadb"
      PROVIDER_VERSION: "10.1"
      INITIAL_CLUSTER_SIZE: 2
      DB_ROOT_PASSWORD: "mysqlpass"
```

```
    networks:
      - galera_cc

  clustercontrol:
    deploy:
      replicas: 1
    image: severalnines/clustercontrol
    ports:
      - 5000:80
    networks:
      - galera_cc

networks:
  galera_cc:
    driver: overlay
```

To deploy it, as we've seen at the beginning of this chapter, we need just one command, which is:

```
$ docker stack deploy --compose-file=docker-compose.yml dummyName
```

Docker Swarm will then deploy one container for ClusterControl (replicas:1) which it'll be used as a control room for the whole infrastructure, and then two more containers with the database nodes inside of them (replicas:2). The database container will then register itself into the CMON database for deployment.

After the deployment phase is completed, which takes a fair amount of time, it's also possibile - and suggested - to check the status of each container to actually confirm that the expected amount of nodes has been deployed with the right service running within themselves. To do that, run the command:

```
$ docker service ls
```

which will list the services deployed with their own status.

```
[parallels@localhost Desktop]$ sudo docker service ls
ID            NAME               MODE        REPLICAS  IMAGE
fgqewiarsyhw  cc_clustercontrol  replicated  1/1       severalnines/clustercontrol:latest
hzq0sdoje765  cc_galera          replicated  2/2       severalnines/centos-ssh:latest
[parallels@localhost Desktop]$ sudo docker ps
CONTAINER ID        IMAGE                                                                                               COMMAND
  CREATED             STATUS                    PORTS                                                        NAMES
630a61921019        severalnines/centos-ssh@sha256:98d293966ea0603bdb2a81a4ddbbd869f4da08469f20c585ecb4873adad8e418       "/entrypoint.sh"
  13 minutes ago      Up 13 minutes             22/tcp, 3306/tcp, 5432/tcp, 9999/tcp, 27107/tcp              cc_galera.2.mkjco18m7o4ele7mavv2ie
yiu
5bc20a58ec02        severalnines/centos-ssh@sha256:98d293966ea0603bdb2a81a4ddbbd869f4da08469f20c585ecb4873adad8e418       "/entrypoint.sh"
  13 minutes ago      Up 13 minutes             22/tcp, 3306/tcp, 5432/tcp, 9999/tcp, 27107/tcp              cc_galera.1.bh39f55pmr3mx3cb21nht2
5lb
00237b527bf2        severalnines/clustercontrol@sha256:3a08f1314336b7e09959ca519679be33243701708e3a7d59beaef7460bd33198   "/entrypoint.sh"
  13 minutes ago      Up 13 minutes (healthy)   80/tcp, 443/tcp, 9500-9501/tcp, 9510-9511/tcp, 9999/tcp      cc_clustercontrol.1.sofoi70e1n17ib
kwor6b2weon
```

Figure 2.6: Double checking node deployment via console.

2.6.1 Finding services parameters

When deploying a new cluster, node or service, it is very useful to monitor the whole process so we can be sure of what's going on within the cluster. To do that, once we discovered the service/node ID, we can use the

```
$ docker inspect ID
```

command, which will return to us a number of the object parameters, IP address included. This is very useful when we need to know the new nodes IP address or, most importantly, the controller IP address to log-in in the cluster control panel.

2.7 Load balancing

Docker Swarm comes with its own load balancer, to distribute traffic to all containers, using a round-robin scheduling. It lacks several useful configurable options to handle routing of stateful applications, for example persistent connection (so source will always reach the same destination) and support for other balancing algorithm, like least connection, weighted round-robin or random. Since in MySQL some connections might take longer to process before it returns the output back to the clients, Galera Cluster load distribution should use a *least connection* algorithm, so the load is equally distributed to all database containers. As a workaround, relying on other load balancers, such as HAProxy, ProxySQL and MaxScale, in front of the provided service is the recommended way.

2.8 Managing the cluster

As sais before, it's possibile to do everywhing that was here tested, just by using the command line interface. Although this is of course possible, it is far easier using some sort of graphical interface. ClusterControl provides us with just that, giving us the options of monitoring and deploying all sorts of security measures, which is very welcomed when handling and securing data of any kind. Following are some snapshots of some ClusterControl features, which are way too much to be studied in this project. Bare in mind, some cluster names and IP addresses might be slightly different from the one we've seen before this point. This is due to many trial and errors previously performed to study all the technology mechanisms.

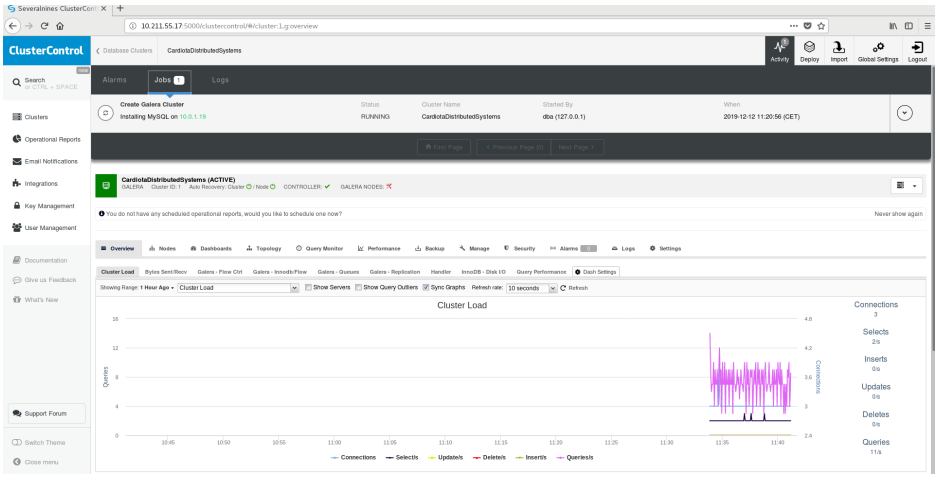

Figure 2.7: ClusterControl main management panel.

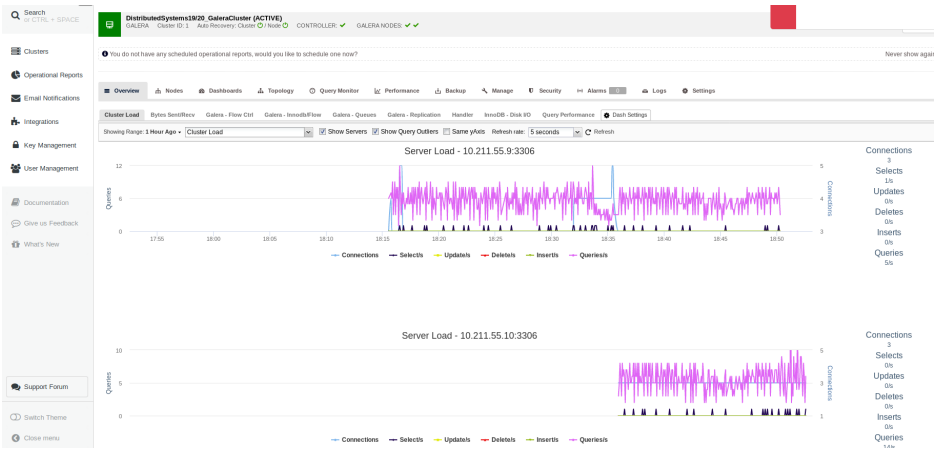

Figure 2.8: Multiple nodes load-balance monitoring.

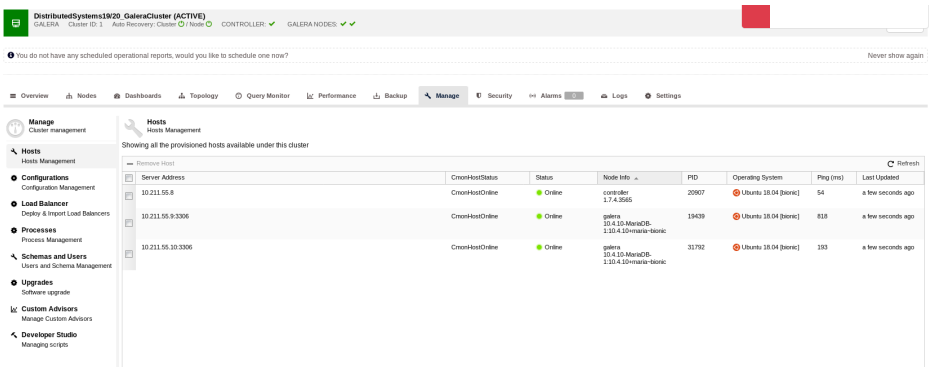

Figure 2.9: Hosts status monitoring.

Cluster topology One of the features that is provided by the ClusterControl management panel is the ability of graphically view the cluster topology in one single page, also providing useful status information, eventual nodes recovery and bootstrapping options.

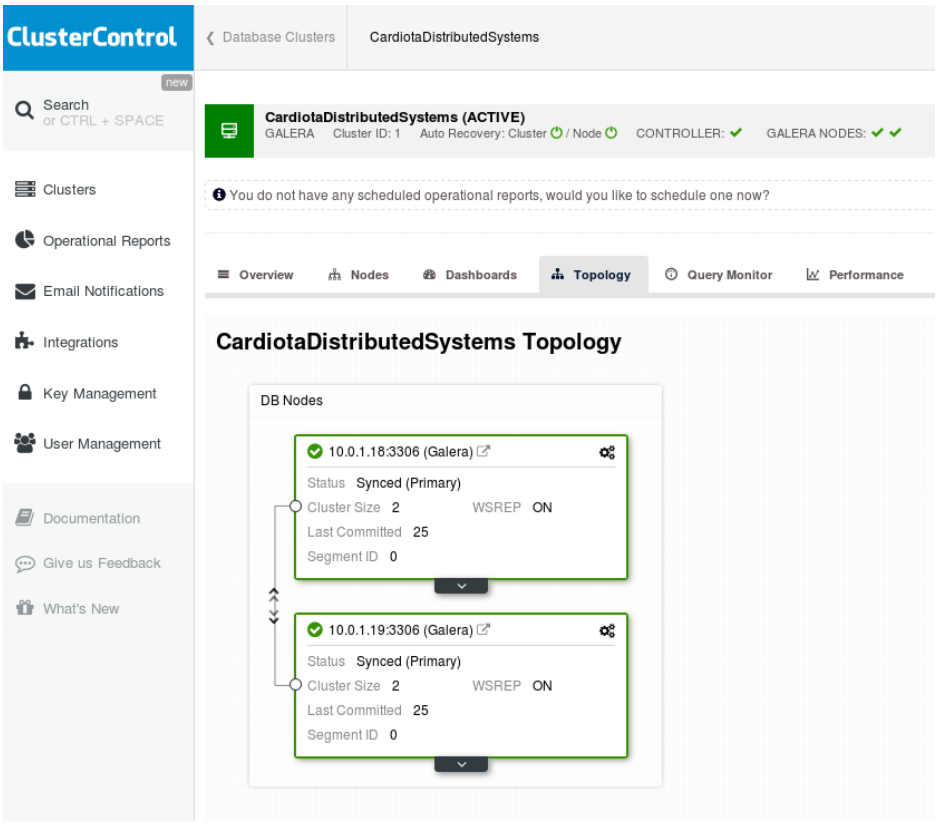

Figure 2.10: Cluster topology.

2.9 System and replication testing

I initially developed a basic PHP web application that read and wrote some SQL queries inside the database to test out the replication procedure. I then realized that I could do just that by testing exactly the same queries via the integrated MariaDB command line interface on each node. This was not only faster, but is algo gave me a more accurate rappresentation on both what I was doing and what was happening inside each node and the cluster as a whole.
So, to test the whole system and check that is actually does what it's supposed to do, I configured two virtual machines, each running a node that was, at this point, pre-registered to the database nodes cluster and into the swarm. As we can see from the picture below, there's no initial database and therefore no initial tables to work with. On the first node some operation, such as database creation, table creation and some dummy write queries, are performed. At this point, the cluster starts its own replication operation behind the scenes, de facto replicating itself on the others Docker swarm nodes. This operation, due to the fact of the database footprint, took very little time. To double check that everyting worked corretly, we can now login to the second node's database command line interface and try pulling all the data from the table - and database - that we previously created on the other node. Those rows, table and database

are actually there and the data, after many runs, is consistenly the same, so we now know that this kind of replication actually works even when integrated in a swarm provided by Docker.

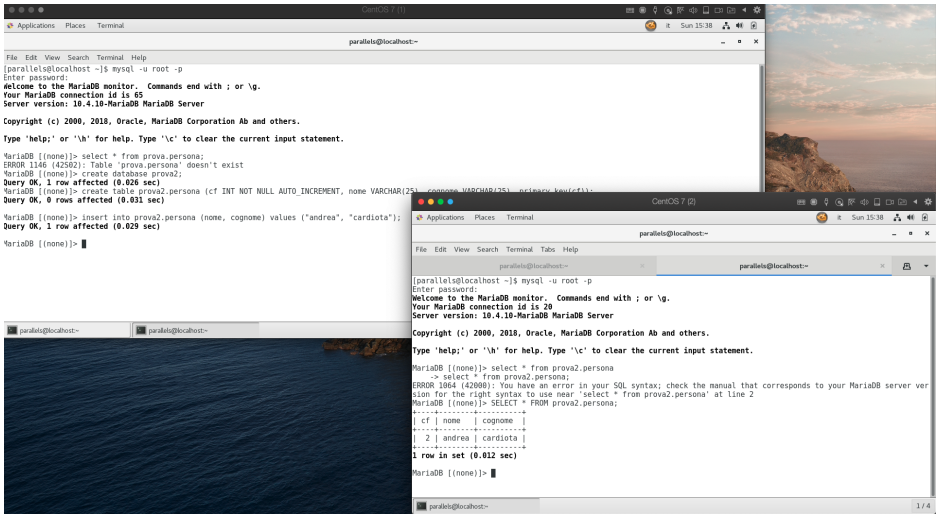

Figure 2.11: Database replication taking place in-between Docker Swarm nodes.

2.10 Removing ClusterControl

ClusterControl performs a critical function within the system we've built, but that doesn't mean it totally dependent upon it. We can actually remove it from the equation, but this usually means that we need to find another way to manage both the cluster and swarm services. The idea is generally the same as the one we kept in mind when we *had* ClusterControl helping us. Like a RAID setup, the data gets replicated across the cluster. If one node fails, another node will be bootstrapped and the data will be replicated and initialized. Of course, not having ClusterControl anymore means the we need to find another way to load-balance the requests coming from the application layer. To do just that, we can use Docker swarm integrated load-balancer but, since it has not been tested deeply enough for this case of scenarios, and there are really high-performing ones available, the safest bet is relying on external load-balancers such as MaxScale. MaxScale provides both some additional features regarding load-balancing database traffic and its ways to get information on the status of the cluster, all in the same package. If the backend servers are running within a service controlled Docker swarm, we can start MaxScale as service as well and let it autodiscover each database node; this is done by querying the internal DNS service and processing the DNS round robin result to generate the maxscale.cnf file at startup.

After it is correctly initialized, we can check the status with the *docker exec* command and the result will be something like this, depending on the setup:

```
+-----------+-----------+------+-------------+-----------------------------+
| Server    | Address   | Port | Connections | Status                      |
+-----------+-----------+------+-------------+-----------------------------+
| 10.0.0.3  | 10.0.0.3  | 3306 |           0 | Slave, Synced, Running      |
| 10.0.0.4  | 10.0.0.4  | 3306 |           0 | Slave, Synced, Running      |
| 10.0.0.5  | 10.0.0.5  | 3306 |           0 | Master, Synced, Running     |
+-----------+-----------+------+-------------+-----------------------------+
```

Of course, this would also mean that we'd lose all ClusterControl provided functionalities, such as automatic backup, restore and bootstrapping of nodes, monitoring and alerts and we should find a new solution so automate the said operations.

Chapter 3

Conclusions

Coming to an end, it's proven that such systems can work, as long as they are well thought through and have solid foundations. Of course, this has to be considered as a **proof of concept** and not as a production-ready system.

3.1 Outcome and technical limits

The Docker stateless conventions doesn't marry persistency the way we would like it to. It boots up, does its job and gets destroyed if the job is done or it doesn't pass health checks. So, what happens if one of the Swarm nodes goes down? The cluster state will get demoted into *non-primary* and the Galera node state will turn to *initialized*. This situation turns the containers into an unhealthy state which results in a down state for the network, meaning those database containers will be destroyed and replaced with new containers by Docker swarm according to the "docker service create" command. We'll then have a new cluster starting and bootstrapping, losing the whole dataset. There are workarounds to this problem but they are not ideal and they unavoidably complicate the system. So what does this mean? It means that, as of today, it's not advisable to run a database in a container since they are designed to consume all memory, CPU and I/O available to them. So, what's the proper solution in terms of architecture for production systems? The following couple of options might be good alternatives. One is not creating our own database containers but rather use an externally hosted database service, such as Amazon RDS. Secondly, we could let Docker Swarm do what it does best, handle stateless applications. For the database layer, we could use a *Database as a Service* system such as *AWS RDS*, unloading all data management problem on the hosting platform; this is safer in pretty much every way but it's also more expensive and we lose some control over our infrasctructure. Another way would be using *Firecamp*, which is a project that aims to solve the data persistence issue in Docker swarm, "dockerizing" the stateful services and run them on top of container orchestration frameworks such as Docker swarm. Unfortunately though, it has been discontinued.

3.2 Dockerizing a Galera Cluster without external controllers

As we said earlier, utilizing the sole Galera Cluster on a Docker swarm configuration it's possible, but only as a proof of concept because of the many limitations. So what if we don't want to use external controllers? The obvious alternative is *Kubernetes*, a production-grade container orchestration tool which is integrated into Docker and represents a viable alternative to Swarm. So why it can be considered *production-ready* when Swarn can't? Firstly, we need to understand which are the main differences between the two tools. Kubernetes supports two types of health check: *liveness* and *readiness*. This is important when running a Galera Cluster on containers, because a live Galera container doesn't mean it's ready to serve requests and should be included in the load balancing set. Docker Swarm only supports one health check type similar to Kubernetes' liveness, a container is either healthy and keeps running or unhealthy and gets rescheduled. So, why Kubernetes? Mainly because Kubernetes supports two types of controllers: **ReplicaSet** and **StatefulSet**. Obviously the StatefulSet is the best way to deploy Galera Cluster in production, because:

- Deleting and/or scaling down a StatefulSet will not delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources;
- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from 0 .. N-1;
- When Pods are being deleted, they are terminated in reverse order, from N-1 .. 0;
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready;
- Before a Pod is terminated, all of its successors must be completely shut down.

So, StatefulSet provides state-of-the-art support for stateful containers. It provides a deployment and scaling functionalities.

3.3 Future works

It would be interesting to implement a system with the same requirements but using the *GlusterFS* framework. A good design pattern for highly available applications is to deploy the application as a container on a Docker swarm cluster, same as we did, but with persistent storage provided, in fact, by GlusterFS, which is a fast shared filesystem that can keep the containers in sync between multiple VMs running the Docker swarm cluster. In other words, it's a software defined distributed storage that can scale to a very large amount of data without a hitch.

This pattern ensures high availability; in fact, if a VM would die, Docker swarm would bootstrap the container on another VM and GlusterFS will make

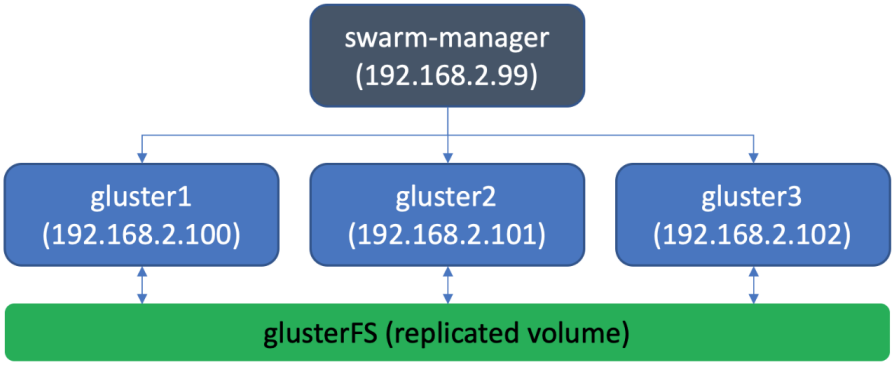

Figure 3.1: Docker swarm cluster with shared GlusterFS storage.

sure that the container has access to the same data as the old container did, strengthening data consistency and availability.

Bibliography

[1] Docker documentation
https://docs.docker.com

[2] Maria DB documentation
https://mariadb.com/kb/en/library/documentation

[3] PostgreSQL documentation
https://www.postgresql.org/docs

[4] Galera Cluster documentation
https://mariadb.com/kb/en/library/what-is-mariadb-galera-cluster

[5] ClusterControl documentation
https://severalnines.com/docs

[6] GitHub 3-tierd architecture web-application demo
https://github.com/dockersamples/example-voting-app

[7] Severalnines Docker image
https://hub.docker.com/r/severalnines/mariadb

[8] Swarm database cluster best practices
https://vmware.github.io/vsphere-storage-for-docker/documentation/demo-ha-swarm.html