

Docker Plugin for Volume Sharing

Enrico Fiumana

`enrico.fiumana2@studio.unibo.it`

March 17, 2024

In a distributed computing environment built using docker-swarm technology and used by multiple workgroups, there is a problem with separating the working areas on disk / volumes. Indeed, in this environment, knowing where the common data resides, anyone can access the data of other groups. The aim of this project is therefore to create a layer of separation between the management of the volumes by the user and the physical part of the volume itself. This can be achieved through a plugin (or driver, as they say) that allows you to abstract the physical folder from the virtual one used by docker. Furthermore, since docker swarm is indeed a distributed cluster, it is necessary to allow the containers running on the various nodes to access the data in a transparent manner regardless of the node on which they are executed. To achieve this result, we have relied on a distributed file system such as NFS (Network File System).

1 Goals/requirements

The project aims to implement some form of privacy for the data used by containers [6] running on a Swarm [3] cluster. In fact, anyone knowing where the data is stored could activate a container by mounting the volume, thus accessing another person's data.

It is therefore necessary, through a specific plugin, to ensure that anyone, even if they know the location and name of a volume, is not able to use it. This is achieved by inserting a layer of software between the Docker daemon and the filesystem.

Docker [5] itself provides this possibility through so-called plugins. In our case, the plugin works by associating a unique random code with the volume name provided by the user, which is then registered to later retrieve the volume, thus making it useless for a malicious actor to know the volume name, as depicted in Figure 1.

1.1 Scenarios

The typical use case for the plugin is the instantiation of a new service on the cluster. As a rule, a service also requires access to persistent data (remembering, in fact, that

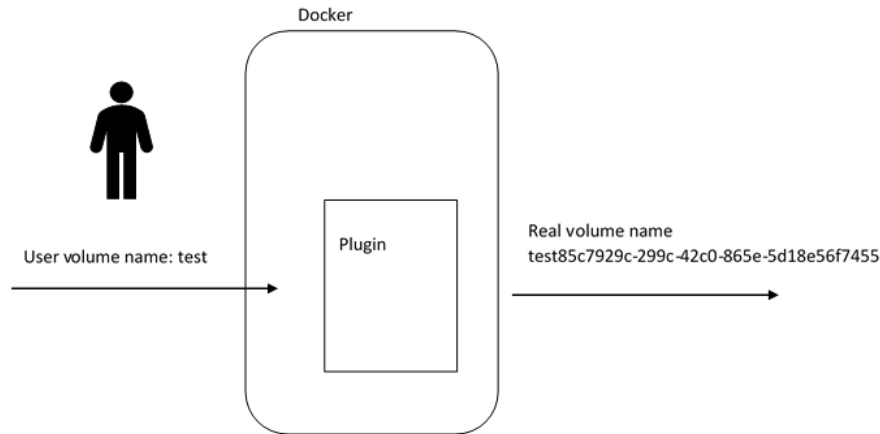


Figure 1: General Diagram

once the service is “turned off“ all the data inside, if not explicitly saved to an external volume, is lost) through Docker volume management. The plugin can be used both through the CLI, by giving the appropriate commands at the time of creating and using the volume, and through appropriate GUI platforms, such as Potainer.

2 Background

Before delving into the implementation of the component, it is necessary to introduce some notions about docker-swarm.

Docker natively supports the ability to manage a cluster of docker engines: this mode is called “Swarm“. In this mode I can literally create a “*swarm*“ of docker engines that collaborate and run services in a scalable way and leveraging HA systems.

These docker engines are installed on hosts (both virtual and physical) which are called nodes. The nodes can be manager nodes, which manage the cluster, worker nodes, on which tasks are executed, or they can be both.

Swarm also allows you to “scale“ services. That is, we can increase or decrease the instances of the single service which will then eventually be distributed among the various hosts (See Figure 2)

Swarm also takes care to keep the state of services consistent. That is, in case a node becomes unavailable, due to failure or maintenance, Swarm will automatically redistribute the load among the remaining nodes.

As anticipated, the tasks that are declared within the services, such as the creation and execution of one or more containers, are executed on the worker nodes.

An agent is executed on each worker node that reports to the manager nodes the state of the services running on the worker so that the managers know the status of all the services at all times.

When we want to submit a service to the cluster, it is entered through a manager

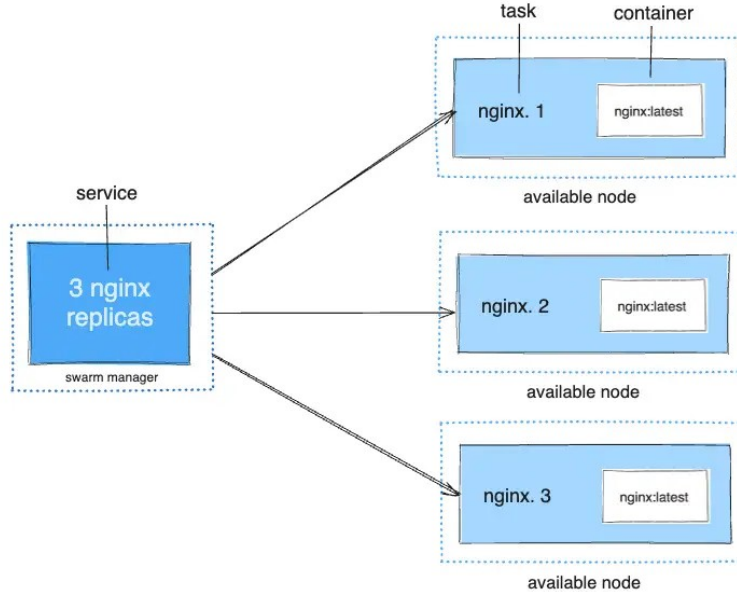


Figure 2: Docker Swarm General Diagram

node, which in turn will distribute the load to the worker nodes. The manager nodes also serve as orchestrators and cluster managers, necessary to maintain the state required by Swarm. To ensure the consistency of the managers' state, the consensus protocol[7] called "*Raft*" [4] is used as reported in Figure 3

The problem of consensus refers to the need to reach an agreement among multiple nodes or participants in a distributed system on a value or an action. This agreement is important to ensure the consistency and integrity of the data and operations in the distributed system. In a distributed system, nodes can be subject to failures or temporary disconnections, which can lead to divergences of opinions or a lack of consensus on a certain value or operation. The problem of consensus tries to solve this problem by ensuring that all nodes agree on a value or an operation, even in the presence of failures or disconnections. Raft is a distributed consensus protocol designed to ensure the consistency and atomicity of operations in a distributed system, even in the presence of node failures. It is a simpler and easier-to-understand algorithm compared to other consensus algorithms like Paxos (which under certain specific conditions is BFT unlike Raft), which makes it very suitable for the creation of practical distributed systems. The Raft protocol assumes the presence of a leader, which is responsible for receiving write requests, propagating them to follower nodes, ensuring data consistency, and ensuring that only atomic operations are executed. If the leader dies or becomes unreliable, a new leader is elected from the remaining nodes. Raft is based on a leader-follower architecture, in which the leader assumes the role of coordinator and follower nodes obey the leader's orders. The protocol uses a timeout-based election system, in which the node

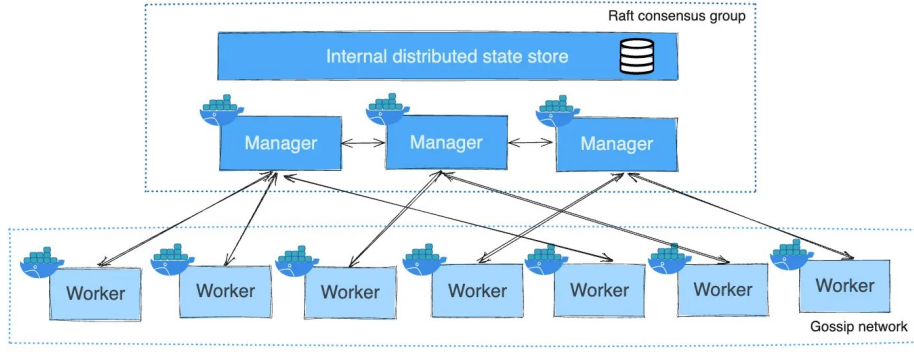


Figure 3: Docker Swarm Orchestrator Diagram

that does not receive messages from the leader for a certain period of time becomes a candidate to become the new leader.

Raft tolerates up to $\frac{N-1}{2}$ unavailable manager nodes and requires a quorum of at least $\frac{N}{2+1}$ nodes to agree on a value proposed to the cluster. This means that if, for example, in a Swarm cluster there were 5 manager nodes and 3 became unavailable, the cluster would stop accepting new tasks, while those already started would continue to function. On Swarm, precisely because of how Raft is made, it is advisable to always have an odd number of Manager nodes and in any case always greater than one. In Figure 4 the lifecycle of a service is depicted, detailing how information passes through the various layers of Swarm.

Given that any container running on a cluster node can be moved to another node because the node is under maintenance or has failed, or to increase the computing resources allocated to the container, be “scaled“, that is, run on multiple nodes, it is necessary that the volume data be stored on a distributed file system (In this sense of the term “Distributed“ it means visible on different hosts and not, as a true DFS distributing data on different machines). For this purpose, we thought of using NFS (Network File System) Figure 5, a distributed file system, albeit of old generation, but reliable. NFS favors consistency over availability, in fact in case of a network partition, the filesystem is no longer usable. To provide a minimum of availability, perhaps due to short disconnections of the connection, `async_write` and `nfs_retry` mechanisms are used. NFS uses RPC systems to allow the client to communicate with the server and vice versa. However, these properties can also be dangerous in terms of data consistency. In fact, there is no assurance that data written by one node is immediately visible on another node.

To overcome this problem, the only solution would be to have synchronous writes, in order to ensure data consistency. However, this would result in performance and resilience to lack of connection being lost. Since version 3 of NFS, alternative systems have been sought to remedy this problem. One of these is the so-called close-to-open consistency, i.e., the fact that when a file is `close()`, the cache is actually written to the server’s filesystem, and another client, before accessing its own cache to read a file,

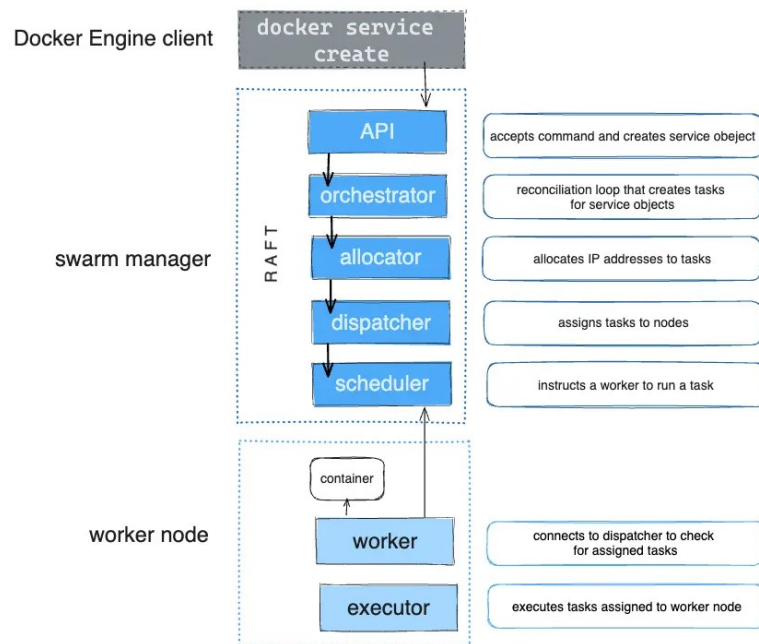


Figure 4: Service lifecycle

checks the file's timestamp on the server, so as to potentially invalidate its own cache. Fortunately, since version 4 of NFS, the protocol has become stateful (so the server keeps track of client access) and based on TCP rather than UDP.

3 Requirements Analysis

Below is an overview of the functional and non-functional requirements of the project:

3.1 Functional Requirements:

- **Hide the true volume name from the end user:** The main purpose of our plugin is precisely not to reveal the real name of the volume written on the filesystem to the end user.
- **Allow the same volume to be used by multiple nodes simultaneously:** The plugin must be transparent to the end user's ability to perform tasks on multiple nodes simultaneously using the same volume.
- **Remember the state for each volume until its destruction:** The plugin must maintain a consistent state of the use of each volume on each node in order to prevent volume deletions while they are still needed on other nodes.

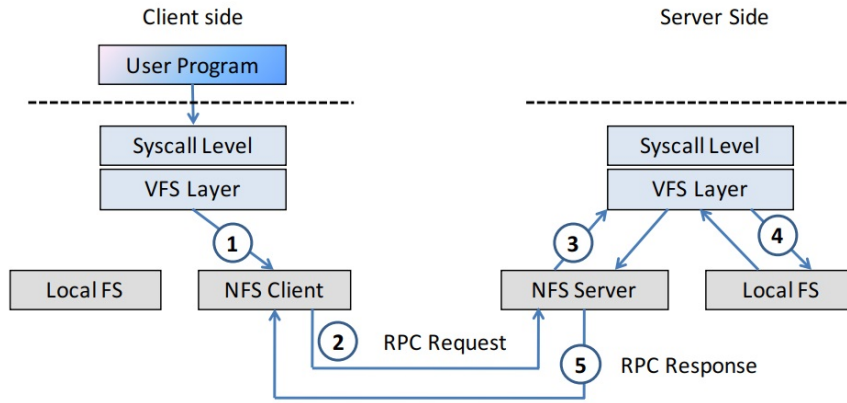


Figure 5: NFS

3.2 Non-functional Requirements

- **Meet all requirements imposed by Docker:** Docker imposes specific guidelines both for writing plugins and for interacting with them.
- **Be completely transparent to the end user:** The end user should not notice that the physical volume has a different name and where it is saved. It is not important for them to know this.
- **Ease of deployment:** Both installation and potential configuration of the plugin must be easy. It should be assumed that the person who may install the plugin has technical knowledge and it should also be easy to install it on a series of nodes without worrying about configuring it.
- **Compatibility with third-party user interfaces:** There are a number of third-party interfaces to Swarm. Our plugin must also be transparent to them.

4 Design

Since it will become a component of Docker, the rules imposed for the creation of the plugin by Docker had to be followed. These include the use of a socket and the use of ReST [1] technology to communicate between Docker and the plugin

From a logical point of view, the functioning of the plugin is very simple. The structure consists of 4 objects: the Docker engine, a Unix socket, through which communication between the engine and the plugin takes place, the plugin itself and finally the volume. A Unix Socket is an Inter-Process Communication (IPC) mechanism that allows communication between processes on the same host. The interaction between the engine and the plugin takes place using the socket and emulating TCP communication through the HTTP protocol using ReST APIs REST APIs (Representational State Transfer Application Programming Interface) are a type of software architecture based on HTTP that

allows different computer systems to communicate with each other in an efficient and scalable way.

REST APIs follow the fundamental principles of REST, which include:

- **Client-server architecture:** REST APIs follow a client-server architecture, where the client and the server are distinct and separate. The server provides the resources, and the client requests and processes the results.
- **State of resources:** The system's resources (such as files, data, or other types of information) are represented uniquely by URIs (Uniform Resource Identifiers).
- **Uniform interface:** APIs provide a uniform interface for accessing and manipulating resources through standard HTTP methods, such as GET, POST, PUT, DELETE.
- **Statelessness:** Each HTTP request between the client and the server must contain all the information necessary to fulfill that request, without the server retaining any state about previous requests. This makes REST APIs more scalable and easier to manage.

Finally, the plugin interfaces with the volume through the System Calls of the operating system to create the necessary data structures for the engine to store the required data.

Operation Name	Meaning
VolumeDriver.Create	Volume creation
VolumeDriver.Remove	Volume removal
VolumeDriver.Mount	The volume is mounted, making it available to the requesting container
VolumeDriver.Path	Returns the path of the volume
VolumeDriver.Unmount	The container that used the volume is about to be deactivated, so it requests to unmount it
VolumeDriver.Get	Returns information about the volume
VolumeDriver.List	Returns the list of volumes
VolumeDriver.Capabilities	Returns volume capabilities

Table 1: Rest API List

5 Implementation Details

The implementation follows the classic method of creating backend interfaces for a web client-server structure. In particular, it is structured around three main modules: The router: this module is responsible for interpreting ReST calls and routing execution to the relevant modules. Particular note: all engine calls are of the POST type. Then there is the controller, which is the heart of the plugin. Depending on the type of call interpreted by the router, methods of this object are called (here, both “methods“ and “objects“ are not to be interpreted in relation to OOP, but rather as simple function calls. These terms are used to give a better idea of how it works). Here, depending on the requested operation, data structures necessary for volumes can be created or deleted, information about the volumes themselves can be retrieved, and other operations can be performed. The operations are listed in the Docker manual and are reported in Table 1.

Finally, there is the last module, called by me as *db*, which handles the management of the state file, that file in which the name of the volume, the associated random number, and the number of mounts in all the docker engines of the swarm cluster are recorded. The file is rewritten each time to ensure consistency, and locking systems are used to manage concurrency.

6 Self-assessment / Validation

Given the particular nature of the developed software system, while on one hand it would certainly be possible to design self-assessment systems, on the other hand, their implementation appears extremely complicated and non-trivial. Therefore, it was considered to create specific shell scripts that perform a series of checks. The checks we thought of carrying out are:

- Creation of “n” containers, each of which writes on its volume:
I expect n different volumes both on the filesystem and in the state, and a single data within the volume. 1
- Creation of “n” containers, each of which writes on the same volume:
I expect one volume both on the filesystem and in the state, and different data within the volume. 2
- Creation of a service scaled on 2 hosts that uses the same volume:
I expect only one volume but in the state a reference counter value equal to two and on the volume different data. 3
- I decrease the scaling of the previous container, and delete the volume from a node:
I expect that the volume is not deleted and the reference counter equal 1 (I only observe a decrease in the counter).
- I decrease the scaling of the container even more and delete the volume:
I expect: the volume deleted from both the filesystem and from the state too.

7 Deployment Instructions

The plugin, through a series of scripts and makefiles, is encapsulated within a special container image (special because it is “branded” as a plugin) and then uploaded to Docker Hub, which is a registry where anyone can upload their containers to make them usable by anyone. This technique makes the installation procedure very simple, using just a few docker commands.

8 Usage Examples

The driver can be used in two distinct modes: **standalone** or **Swarm**.

In both cases, the first thing to do is to create the volume using the command:

```
docker volume create <VolumeName> -d enricofiumana2/mydriver
```

If we want to use the volume in standalone mode, that is, using a simple Docker container, we use the normal command to run a Docker image specifying the use of the previously created volume. For example:

```
docker run -q -d --name container1 -h container1
-v myVolume:/mnt/MyVolume alpine sh -c 'echo "Hello from $HOSTNAME" > /mnt/vol/
$HOSTNAME'
```

In Swarm mode, on the other hand, the command to use will be, always having previously created the volume:

```
docker service create -d \newline
--replicas=2 \newline
--name testSwarm \newline
--hostname testSwarm \newline
--label testDriver \newline
--mount type=volume,volume-driver=enricofiumana2/mydriver,source=myVol,target=/mnt
/VolShared\newline
alpine tail -f /dev/null
```

It is also possible to run a Swarm service using the ‘docker stack’ command, which utilizes configuration files in YAML format. In this case, we can leverage two different configurations. In the first one 4, it is necessary to create the volume beforehand using the ‘docker volume create’ command, indicating within the YAML configuration file that the volume is “external.” In the second case 5, however, we use another YAML configuration file where we instruct the system to create the volume, specifying the name of the driver to be used.

9 Conclusions

Summarizing the activities carried out and arranging them in chronological order, I started with the study of how a Docker plugin works and how it interfaces with the engine, looking at the APIs, their input parameters, and their output parameters; then, since the final environment on which the software had to run was a Swarm cluster, I tried to understand how this tool works, both theoretically and practically. At this point, I merged the acquired knowledge and started to draft a preliminary project, proceeding step by step. In parallel, I tried to understand the basic features of NFS and see if it could suit our needs. Once the design phase was completed, I started developing the software and thinking about how to deploy it. Initially, I had thought of the classic deployment method of Unix daemons, that is, distributing the executables and having the code run at system startup. This deployment/execution mode of the plugin offers some advantages, such as the ability to create all the necessary folders and files for the plugin’s operation and the possible parameterization of certain data, such as the location of the folders where the volumes are created. On the other hand, however, there is another possible deployment mode, which is that of the container as mentioned above. Through this method, the installation and update of the plugin by the swarm cluster administrator is extremely simple. In the end, we decided to pursue this latter mode.

9.1 Future Works

Considering that the basic project in my opinion has been developed exhaustively, one thing that surely can, indeed must be done, is to develop it using OOP technologies and a higher-level language than the one used (NodeJS[2]). Another improvement could be to use a real Database to save the state, perhaps something like Sqlite.

9.2 What did we learned

The project has been stimulating and enjoyable. Confronting myself with a cluster system like Swarm has given me the opportunity to see how these systems, which I use daily in my work, work “under the skin“, i.e., to understand, for example, why consensus systems like Raft are used and how they work. I have also had the opportunity to see, always from the designer’s point of view rather than from the user’s, how distributed file systems work, both NFS and, as mentioned, even though having a middleware layer, it is not a true distributed file system, both the real DFS and the implications of managing concurrency by seeing the practical implementation of the CAP theorem.

Thank you!

Listing 1: Test N. 1

```
1  #!/bin/bash
2  NumContainer=2
3  for (( Counter=0; Counter<=NumContainer; Counter++ ))
4  do
5      VolName=Volume$Counter
6      docker volume create $VolName -d enricofiumana2/mydriver
7      docker run -q -d --name container-$Counter -h container-$Counter -v $VolName:/
          mnt/vol alpine sh -c 'echo "Hello from $HOSTNAME" > /mnt/vol/$HOSTNAME'
8  done
9  echo "==> State content"
10 cat /mnt/shared/state
11 echo "==> Volume List"
12 ls -l /mnt/shared/vols
13 for fileName in $(ls /mnt/shared/vols)
14 do
15     echo "==> Volume Files in $fileName"
16     echo -e "\t$(ls /mnt/shared/vols/$fileName)"
17     echo "==> Volume Files Content"
18     echo -e "\t$(cat /mnt/shared/vols/$fileName/*)"
19 done
20 ### Cleanup
21 for (( Counter=0; Counter<=NumContainer; Counter++ ))
22 do
23     VolName=Volume$Counter
24     docker rm container-$Counter
25     docker volume rm $VolName
26 done
```

```
administrator@swarm1-enrico:~/tests$ ./test1.sh
Volume0
3c44aaff7ad9fc817a1c0b53bf86019654ef903e6ec0b2fbacfb1601e5638b41
Volume1
9e19153e08eac9e6569208e454a8878bdd5f6a225a6660a5902a7163c6e4cd79
Volume2
569601e5ee6b3763ee5c991f7529a45836af36bcfb42497f50b06ff4ab0611af
==> State content
Volume0,3f6b0fd9-7289-4176-8504-0daf578d7d2e,0
Volume1,3f351171-d57f-4292-994f-a2f95f7f3a51,0
Volume2,137726bf-d00d-4af4-9a87-3d2d01b1cf5a,1
==> Volume List
total 12
drwxr-xr-x 2 root root 4096 Mar 17 12:32 137726bf-d00d-4af4-9a87-3d2d01b1cf5a
drwxr-xr-x 2 root root 4096 Mar 17 12:32 3f351171-d57f-4292-994f-a2f95f7f3a51
drwxr-xr-x 2 root root 4096 Mar 17 12:32 3f6b0fd9-7289-4176-8504-0daf578d7d2e
==> Volume Files in 137726bf-d00d-4af4-9a87-3d2d01b1cf5a
    container-2
==> Volume Files Content
    Hello from container-2
==> Volume Files in 3f351171-d57f-4292-994f-a2f95f7f3a51
    container-1
==> Volume Files Content
    Hello from container-1
==> Volume Files in 3f6b0fd9-7289-4176-8504-0daf578d7d2e
    container-0
==> Volume Files Content
    Hello from container-0
Cleaning Up
container-0
Volume0
container-1
Volume1
container-2
Volume2
administrator@swarm1-enrico:~/tests$
```

Listing 2: Test N. 2

```
1  #!/bin/bash
2  NumContainer=2
3  VolName=Shared-Volume
4  docker volume create $VolName -d enricofiumana2/mydriver
5  for (( Counter=0; Counter<=NumContainer; Counter++ ))
6  do
7      docker run -q -d --name container-$Counter -h container-$Counter -v $VolName:/
        mnt/vol alpine sh -c 'echo "Hello from $HOSTNAME" > /mnt/vol/$HOSTNAME'
8  done
9  echo "==> State content"
10 cat /mnt/shared/state
11 echo "==> Volume List"
12 ls -l /mnt/shared/vols
13 echo "==> Files in volume /mnt/shared/vols$(ls /mnt/shared/vols)"
14 ls -l /mnt/shared/vols/*/
15 echo "==> Volume Files Content"
16 for fileName in /mnt/shared/vols/*/
17 do
18     echo -e "\t$(cat $fileName)"
19 done
20 ### Cleanup
21 echo Cleaning Up
22 for (( Counter=0; Counter<=2; Counter++ ))
23 do
24     docker rm container-$Counter
25 done
26 docker volume rm $VolName
```

Result

```
administrator@swarm1-enrico:~/tests$ ./test2.sh
Shared-Volume
f5a980a31830ba29c4b5b6e6d429b3477b0b4185f6d6222aa0ce1f3082eb516e
f2b0c03c34ae04414f80563cd9205df3a08014be02ce8228db41a333d73ae75a
a59da04d8b1ed608233a8dce611774dd427a2a4bfc41bf268db7c0b9f264ab3a
==> State content
Shared-Volume,b9a80df2-9d27-4457-a24c-fd6f5bed4ade,1
==> Volume List
total 4
drwxr-xr-x 2 root root 4096 Mar 17 12:32 b9a80df2-9d27-4457-a24c-fd6f5bed4ade
==> Files in volume /mnt/shared/volsb9a80df2-9d27-4457-a24c-fd6f5bed4ade
total 12
-rw-r--r-- 1 root root 23 Mar 17 12:32 container-0
-rw-r--r-- 1 root root 23 Mar 17 12:32 container-1
-rw-r--r-- 1 root root 23 Mar 17 12:32 container-2
==> Volume Files Content
    Hello from container-0
    Hello from container-1
    Hello from container-2
Cleaning Up
container-0
container-1
container-2
Shared-Volume
administrator@swarm1-enrico:~/tests$
```

Listing 3: Test N. 3

```

1  #!/bin/bash
2  NumScale=2
3  VolName=Shared-Volume
4  docker volume create $VolName -d enricofiumana2/mydriver
5  docker service create -d \
6  --replicas=$NumScale \
7  --name testSwarm \
8  --hostname testSwarm \
9  --label testDriver \
10 --mount type=volume,volume-driver=enricofiumana2/mydriver,source=${VolName},target
    =/mnt/VolShared \
11 alpine tail -f /dev/null
12 if docker service ls | grep testSwarm > /dev/null
13 then
14     sec=0
15     echo Waiting for service starting
16     while true
17     do
18         echo -ne "$sec\033[0K\r"
19         REPLICAS=$(docker service ls --filter label=testDriver | tail -1 | awk '{
                print $4}')
20         if [ $REPLICAS = "2/2" ]
21         then break
22         fi
23         let "sec=sec+1"
24         sleep 1
25     done
26 fi
27 cat <<EOM
28 =====
29 ==>> Normal condition <<===
30 =====
31 EOM
32 echo "==> State content"
33 cat /mnt/shared/state
34 echo "==> Volume List"
35 ls -l /mnt/shared/vols
36 #####
37 #
38 # Decrease Replicas by one
39 #
40 #####
41 cat <<EOM
42 =====
43 ==>> Decreasing replicas by one <<===
44 =====
45 EOM
46 docker service scale testSwarm=1
47 echo waiting cluster to converge
48 sleep 15
49 echo "==> State content"
50 cat /mnt/shared/state
51 echo "==> Volume List"
52 ls -l /mnt/shared/vols
53 #####
54 #
55 # Now i get an error because the volume is still in use
56 #
57 #####
58 cat <<EOM
59 =====

```

```

60 | ==> Try to remove volume but ... <===
61 | =====
62 | EOM
63 | docker volume rm Shared-Volume
64 | #####
65 | #
66 | # Decrease Replicas to zero (Service is still alive, but no container is running in
    |   the cluster)
67 | #
68 | #####
69 | cat <<EOM
70 | =====
71 | ==> Decreasing replicas to zero <===
72 | =====
73 | EOM
74 | docker service scale testSwarm=0
75 | echo waiting cluster to converge
76 | sleep 15
77 | echo "==> State content"
78 | cat /mnt/shared/state
79 | echo "==> Volume List"
80 | ls -l /mnt/shared/vols
81 | #####
82 | #
83 | # Now I can delete the volume because it is not used
84 | #
85 | #####
86 | cat <<EOM
87 | =====
88 | ==> Now the volume can be deleted <===
89 | =====
90 | EOM
91 | docker volume rm Shared-Volume
92 | echo "==> State content"
93 | cat /mnt/shared/state
94 | echo "==> Volume List"
95 | ls -l /mnt/shared/vols
96 | docker service rm testSwarm

```

```

administrator@swarm1-enrico:~/tests$ ./test3.sh
Shared-Volume
abxmd66v0fpl33rgdr3gngz3l
Waiting for service starting
=====
====>> Normal condition <<====
=====
==> State content
Shared-Volume,f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97,2
==> Volume List
total 4
drwxr-xr-x 2 root root 4096 Mar 17 12:33 f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97
=====
====>> Decreasing replicas by one <<====
=====
testSwarm scaled to 1
overall progress: 1 out of 1 tasks
1/1: running [=====>]
verify: Service converged
waiting cluster to converge
==> State content
Shared-Volume,f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97,1
==> Volume List
total 4
drwxr-xr-x 2 root root 4096 Mar 17 12:33 f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97
=====
====>> Try to remove volume but ... <<====
=====
Shared-Volume
=====
====>> Decreasing replicas to zero <<====
=====
testSwarm scaled to 0
overall progress: 0 out of 0 tasks
verify: Service converged
waiting cluster to converge
==> State content
Shared-Volume,f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97,0
==> Volume List
total 4
drwxr-xr-x 2 root root 4096 Mar 17 12:33 f0076b4d-5ca2-42a4-b0c4-2b4d24edbf97
=====
====>> Now the volume can be deleted <<====
=====
Shared-Volume
==> State content
==> Volume List
total 0
testSwarm

```

```
administrator@swarm1-enrico:~/tests$
```

Listing 4: Docker stack test N. 1

```
1 version: '3.2'
2 services:
3   testSwarm:
4     image: alpine
5     volumes:
6       - Shared-Volume:/mnt/VolShared
7     entrypoint: [ "tail", "-f", "/dev/null" ]
8     deploy:
9       mode: replicated
10      replicas: 2
11 volumes:
12   Shared-Volume:
13     external: true
```

Listing 5: Docker stack test N. 2

```
1 version: '3.2'
2 services:
3   testSwarm:
4     image: alpine
5     volumes:
6       - Shared-Volume:/mnt/VolShared
7     entrypoint: [ "tail", "-f", "/dev/null" ]
8     deploy:
9       mode: replicated
10      replicas: 2
11 volumes:
12   Shared-Volume:
13     driver: enricofiumana2/mydriver
```

References

- [1] Bruno Costa, Paulo F. Pires, Flávia C. Delicato, and Paulo Merson. Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture? In *2014 IEEE/IFIP Conference on Software Architecture*, pages 105–114, 2014.
- [2] Xiaoping Huang. Research and application of node.js core technology. In *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, pages 1–4, 2020.
- [3] Nikhil Marathe, Ankita Gandhi, and Jaimeel M Shah. Docker swarm and kubernetes in cloud computing environment. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 179–184, 2019.
- [4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, page 305–320, USA, 2014. USENIX Association.
- [5] Pankaj Saha, Angel Beltré, Piotr Uminski, and Madhusudhan Govindaraju. Evaluation of docker containers for scientific workloads in the cloud. *CoRR*, abs/1905.08415, 2019.
- [6] Tamanna Siddiqui, Shadab Alam Siddiqui, and Najeeb Ahmad Khan. Comprehensive analysis of container technology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 218–223, 2019.
- [7] Abdul Wahab and Waqas Mehmood. Survey of consensus protocols. *CoRR*, abs/1810.03357, 2018.