PURESTORAGE

Deploying Persistent Storage in Docker Swarm® using Pure Service Orchestrator

Simon Dodsley, Director of New Stack Technologies

Version 1.0, 27 June 2018

Contents

Notices

Pure Storage, Inc.
650 Castro Street
Mountain View, CA 94041

Executive Summary

Most traditional applications are based on a single tiered monolithic architecture in which all the code is combined into a single program running on a single host platform. These applications are self-contained, feature end-to-end software coding and are independent from other computing applications. These client applications are typically more difficult to deploy, harder to scale and require additional operational overhead.

Most contemporary applications are evolving from a monolithic to a more microservice based architecture, which run as an independent service comprised of multiple code modules/segments that can communicate with each other using APIs. These 'modularized' applications are faster to test and deploy, easier to scale and can run on multiple platforms. However, microservices based applications pose new challenges for the operations team. Each microservice has its own software libraries and frameworks. Operations specialists have to maintain a different variety of hardware and software stacks to enable them. A new coding and quality assurance methodology called DevOps has arisen around microservices based applications. DevOps is based on a constant stream of code updates rather than periodic large software releases. DevOps has created its own set of unique infrastructure challenges.

Historically, virtualization solutions have tried to resolve this by abstracting the hardware away from the operating system, however virtualizing microservices can be problematic in terms of deployment and scale, whilst containerizations lightweight isolation is much better for these microservice based applications. Containers have their own isolated process space which makes them a good fit for deploying services. Containers do have some challenges, especially in the storage space as a container's data is not globally persistent and there are challenges with data persistence and data migration across multiple hosts.

This white paper describes the relationship between Pure Storage and Docker using the Pure Service Orchestrator Docker plugin and the Docker Engine plugin system that was available from Docker 1.13. The Pure Service Orchestrator Docker plugin is also compatible with other container runtime environments, such as Mesosphere, and can assist in providing advanced storage features across common cloud, virtualization and storage platforms.

Audience

This white paper is designed for developers, system architects and storage administrators who are evaluating or deploying persistent storage in a private cloud or containerized application environment.

It is assumed that the reader understands Pure Storage products and architecture as well as an understanding of Docker and Docker Swarm.

Changes in Application Development Landscape

Traditional applications tended to be monolithic in their architecture, putting all their functionality into a single process. This contrasts to the newer emerging modern applications which tend to be more loosely coupled. They are breaking down the whole large application into smaller, standalone applications and discrete code segments that communicate with each other over APIs.

Here we describe this paradigm shift in application development and how this has given rise to a need for containers and more structured DevOps strategy.

Traditional Applications

A well-designed software program should be able to remove complexity and provide a simple solution to a problem. Traditionally, applications were developed as monolithic solutions where all the services and their interactions were linked together using internal mechanisms and system calls. The application was responsible not only for a specific task but it also needed to perform every other step required to complete all the tasks for the application. Most traditional monolithic applications can run on a virtual machine infrastructure or bare metal client/server clusters and form private data silos. These monolithic solutions did not provide infrastructure modularity and were hampered by long development lifecycles and complex quality assurance methodologies.

Some of the challenges that were faced by traditional monolithic applications are:

- As the application grew, the code increased in size with it. This lead to even longer development life cycles and increased quality assurance checking, slowing down the cadence of product version releases.
- Traditional monolithic applications prone to error – it either worked as intended or it failed, usually without a simple way to recover without manual intervention or restarting.
- Being provided as a single package, it needed to be deployed all at once by a single installation process.
- Scaling of the applications was expensive and time-consuming because the same software solution had to be deployed on one or more servers. Each instance of the application used the same amount of hardware resources which might not even be fully utilised.

Modern Applications

Modern software development lifecycles have changed dramatically. Many modern applications are now based on the concept of microservices, breaking large software projects into smaller independent loosely coupled modules and/or code segments. Individual modules are now responsible for very specific services/tasks which communicate to other modules via universally accessible APIs.

Introduction to Containers

Containers are usually thought of as lightweight virtual machines, providing isolation in the Linux kernel based on namespace and resources, such as CPU, memory, network, etc.. But there are several differences between containers and virtual machines.

Let's look at each in turn.

Virtual Machines

As server processing power and capacity increased, bare metal applications weren't able to utilise the new abundance of resources. Thus VMs were born, designed by running software on top of physical servers in

order to emulate a particular hardware system. A hypervisor, or a virtual machine monitor, is software, firmware, or hardware that creates and runs VMs. It's what sits between the OS and hardware and is necessary to virtualize the server.

Each virtual machine runs its own unique operating system. VMs with different operating systems can be run on the same physical server, so a Windows VM can happily live alongside a Linux-based VM. Each VM has its own binaries/libraries and application(s) that it services, and each VM may be gigabytes in size.

The biggest benefit server virtualization brought about was the ability to consolidate multiple monolithic applications with different operating system and library requirements onto a single system. This removed the need for single application, single server solutions, and so virtualization brought about massive cost savings through reduced footprint and faster server provisioning. Application development also felt the benefit from this consolidation because greater utilisation on larger, faster servers freed up unused servers to be used for QA, development, or lab gear.

Containers

Over the last few years, the concept of application process isolation has grown in popularity. This provides a means of enabling software to run predictably when moved from one server environment to another. A single server with a single operating system can now provide containers as a way to run these isolated microservices.

Containers sit on top of a physical server and its host operating system, whether that be Linux or Windows. Each container shares the host OS kernel and, usually, binaries and libraries, too. These shared components are read-only, but with each container being able to be written to through a unique mount point. This makes containers exceptionally "light". Containers can be only megabytes in size and take only seconds to start, versus gigabytes and many minutes for a virtual machine.

The benefits of containers often derive from their speed and lightweight nature; many more containers can be put onto a server than onto a traditional VM. Containers are "shareable" and can be used on a variety of public and private cloud deployments, accelerating dev and test by quickly packaging applications along with their dependencies. Additionally, containers reduce management overhead. Because they share a common operating system, only a single operating system needs care and feeding (bug fixes, patches, etc). This concept is similar to what we experience with hypervisor hosts; fewer management points but slightly higher fault domain. Additionally, you cannot run a container with an operating system distribution that differs from that of the host because of the shared kernel so, at the moment, no Windows containers sitting on a Linux-based host.

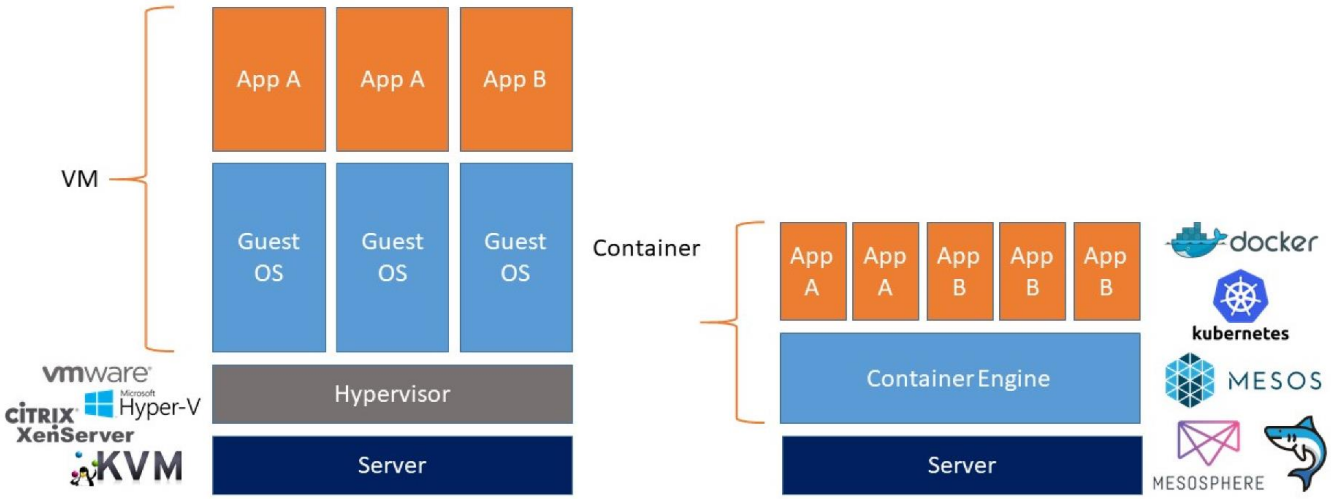


Figure 1 Comparison of VMs with Containers

Docker

Docker is an open source platform for developers and sysadmins to build, ship and run distributed applications. It is a container orchestration engine that separates the application from the underlying operating system just like Linux containers (LXC). Docker is also viewed as an LXC. The current version of Docker uses its own default library `libcontainer` for running an application, accessed through `runC`, instead of using LXC as the default engine. By doing so, Docker has decoupled itself from the Linux dependency and thus it has the capability of running on any operating system.

Docker provides all the container benefits which are discussed above, but in addition, it provides the following on top of LXC:

- Docker is optimised for the deployment of applications as opposed to machines.
- Docker enables bundling an application and all its dependencies into a single object that can be transferred to any Docker enabled host. The bundled environment, also known as the Docker image, ensures that the exposed environment is always the same.
- Docker provides versioning capabilities with all the history being maintained. It provides an option to commit new versions or rollback to a previous version.
- Docker hosts a large number of container templates on the Docker Hub open community.
- Containers isolate applications from one another and the underlying infrastructure while providing an added layer of protection for the application.
- Docker containers are based on open standards, enabling containers to run on all major Linux distributions and on Microsoft Windows and on top of any infrastructure.

Docker Swarm

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts. From Docker 1.12 Swarm mode is integrated directly into the Docker engine, however prior to that Docker Swarm was a standalone product that needed to be configured separately to the Docker Engine.

For the purposes of this whitepaper, we will only concern ourselves with the integrated version of Docker Swarm.

Introduction to Pure Service Orchestrator

Since 2017 Pure Storage has been building seamless integrations with container platforms and orchestration engines using the plugin model, allowing persistent storage to be leveraged by environments such as Docker Swarm.

As adoption of container environments move forward the device plugin model is not sufficient to deliver the cloud experience developers are expecting. This is amplified by the fluid nature of modern containerized environments, where stateless containers are spun up and spun down within seconds and stateful containers have much longer lifespans. Where some applications require block storage, whilst others require file storage, and a container environment can rapidly scale to 1000s of containers. These requirements can easily push past the boundaries of any single storage system.

Pure Service Orchestrator was designed to provide a similar experience to your developers that they expect they can only get from the public cloud. Pure Service Orchestrator can provide a seamless container-as-a-service environment that is:

- **Simple, Automated and Integrated:** Provisions storage on demand automatically via policy, and integrates seamlessly enabling DevOps and Developer friendly ways to consume storage
- **Elastic:** Allows you to start small and scale your storage environment with ease and flexibility, mixing and matching varied configurations as your Swarm environment grows/
- **Multi-protocol:** Support for both file and block
- **Enterprise-grade:** Deliver the same Tier1 resilience, reliability and protection that your mission-critical applications depend upon, for stateful applications in your Docker environments
- **Shared:** Makes shared storage a viable and preferred architectural choice for the next generation, containerized datacenters by delivering a vastly superior experience relative to direct-attached storage alternatives.

Pure Service Orchestrator integrates seamlessly with your Docker Swarm orchestration environment and functions as a control-plane virtualization layer that enables container-as-a-service rather than storage-as-a-service.

Deployment Example

In this section, we will deploy a small Docker Swarm, implement the Pure Service Orchestrator Docker volume plugin, and install a distributed SQL database, specifically CockroachDB, with the containers created using Pure Storage presented persistent volumes.

Environment

The environment being used is based on three servers running Ubuntu 16.04 LTS connected via iSCSI[1] to two Pure Storage FlashArrays and via NFS to one Pure Storage FlashBlade™, however for the purposes on this document we will only be using the FlashArrays for storage provisioning.

The Ubuntu installations are base server deployments with both multipathing and iSCSI initiator software installed. To ensure that we are getting the best performance and connectivity abilities from the Pure Storage FlashArrays, each server has been configured according to the Pure Storage Best Practices for both iSCSI connectivity and multipathing. The latest versions of these can be found on the Pure Storage Support website.

Docker Installation

Following the standard Docker installation instructions for Ubuntu, which are available from the Docker documentation, we have installed Docker CE 18.03.1 which included Swarm capability.

Firstly, we will select one of the servers to act as the swarm manager. In this case, we are using the server called *docker-1*. Configure the new swarm as follows:

```
# docker swarm init --advertise-addr 10.234.112.26
Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token --token SWMTKN-1-2do3ljrt9v0ut4uz00slg1jxqkns4rdwt924s6p4yji1gg1dem-2j9sr53x9rc9avntjiuulg5d5 \
    10.234.112.26:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Execute the command provided in the output during the swarm creation on each of the remaining servers to enable them to join the swarm.

After running this on all other nodes we can look at the current state of the swarm:

```
# docker info
Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
```

[1] Fibre Channel connectivity is also available for Pure Storage FlashArrays. Use the PURE_FLASHARRAY_SAN_TYPE parameter and the `docker plugin set` command.

```
Server Version: 18.03.1-ce
  ...snip...
Swarm: active
 NodeID: zlksbko5y9yk1gda78qesjwoa
 Is Manager: true
 ClusterID: zgeg39smunkk5jqtprpjku5c2
 Managers: 1
 Nodes: 3
  ...snip...
```

And we can also see that all three nodes are configured correctly in the swarm:

```
# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
2wqf820tqc454oawzyterkjh7     docker-2            Ready               Active
3uuqrko9wpyk6lib6rtkbyif7     docker-3            Ready               Active
zlksbko5y9yk1gda78qesjwoa *   docker-1            Ready               Active              Leader
```

Pure Service Orchestrator Docker Plugin Installation

Installation and configuration of the Pure Service Orchestrator Docker plugin is fully automated using the new `docker plugin install` command and is described in the Docker Store location of the Pure Storage plugin.

However, there are a couple of actions that need to be performed on every docker node in your swarm before running the install command:

- ensure the directory `/etc/multipath` exists
- ensure the `/etc/multipath.conf` file exists and contains the Pure Storage stanza as described in the Linux Best Practices referenced above
- a populated configuration file (`/etc/pure-docker-plugin/pure.json`) exists. This is described below.

Create plugin configuration file

To enable the docker plugin to communicate with your Pure Storage backend storage providers, be they FlashArray, FlashBlade, or a combination of these, it is required that every node in the swarm has a configuration containing access information for these backend storage providers.

The file is called `/etc/pure-docker-plugin/pure.json` and contains management IP addresses and API tokens for you FlashArray and FlashBlades, as well as NFS endpoint information for FlashBlades.

The format of the configuration file, using two FlashArrays and one FlashBlade in this example, is as follows:

```
{
    "FlashArrays": [
        {
            "MgmtEndPoint": "10.21.200.210",
            "APIToken": "c6033033-fe69-2515-a9e8-966bb7fe4b40",
            "Labels":{
                "env": "prod"
            }
```

```
        },
        {
            "MgmtEndPoint": "10.21.200.8",
            "APIToken": "9c0b56bc-f941-f7a6-9f85-dcc3e9a8f7d6",
            "Labels":{
                "env": "dev"
                "rack": "7a"
            }
        }
    ],
    "FlashBlades": [
        {
            "MgmtEndPoint": "10.21.200.5",
            "NFSEndPoint": "10.21.200.4",
            "APIToken": "T-9f276a18-50ab-446e-8a0c-666a3529a1b6",
            "Labels":{
                "env": "uat"
                "rack": "6c"
            }
        }
    ]
}
```

Ensure that the values you enter are correct for your own backends.

Labels

You will see in the above example that some entries have one or more labels assigned to them. Labels can be used to filter the list of backends. Labels are arbitrary (key, value) pairs that can be added to any backend as seen in the example above. More than one backend can have the same (key, value) pair. When creating a new volume, label (key = value) pairs can be specified to filter the list of backends to a given set. The plugin also provides the following well known labels that can be used:

- `purestorage.com/backend`: Holds the value `file` for FlashBlades and `block` for FlashArrays.
- `purestorage.com/hostname`: Holds the host name of the backend.
- `purestorage.com/id`: Holds the ID of the backend.
- `purestorage.com/family`: Holds either `FlashArray` or `FlashBlade`

Plugin Installation

When all of the above items have been completed the actual plugin can be installed on each swarm node. In the example below, we are using an alias to enable easier utilisation of the plugin.

```
# docker plugin install store/purestorage/docker-plugin:3.0 --alias pure --grant-all-permissions
3.0: Pulling from store/purestorage/docker-plugin
fbdfb42ec27e: Download complete
Digest: sha256:a5c7f5c34d57cbd90d33ded4ba5553d48204a3231bf1041a9d6068a66e1d94b2
Status: Downloaded newer image for store/purestorage/docker-plugin:3.0
Installed plugin store/purestorage/docker-plugin:3.0
```

The plugin is now installed and enabled:

```
# docker plugin ls
ID                  NAME                DESCRIPTION                         ENABLED
9e2bfcf4b92c        pure:latest         Pure Storage plugin for Docker      true
```

Namespace configuration

As we are going to be using the plugin in a clustered environment, it is necessary to configure the Pure Storage namespace. This should be a cluster-wide unique string, set as an environment variable. This will be used to ensure that container volumes failover correctly in the event of a swarm node failure.

To set an environment variable the docker volume plugin must be in a disabled state. The following commands must be issued on all swarm nodes with the Pure Service Orchestrator Docker plugin installed:

```
# docker plugin disable pure
pure
# docker plugin set pure PURE_DOCKER_NAMESPACE=PURE-DEMO
# docker plugin enable pure
pure
```

You can examine the environment variable to ensure it is set as required by running

```
# docker plugin inspect --format "{{ .Settings.Env }}" pure
[PURE_DOCKER_NAMESPACE=PURE-DEMO]
```

The Docker Swarm environment is now configured with the Pure Service Orchestrator Docker plugin and we are ready to use this for our demonstration of our distributed SQL database.

CockroachDB Installation

To simplify the installation and configuration of the CockroachDB cluster across the Docker swarm a script has been provided that will perform all the necessary steps. This script must be run on the Docker manager node, in this case, *docker-1*.

The Docker service created to control the CockroachDB deployment uses Pure Storage based container volumes, allowing for full failover and scalability.

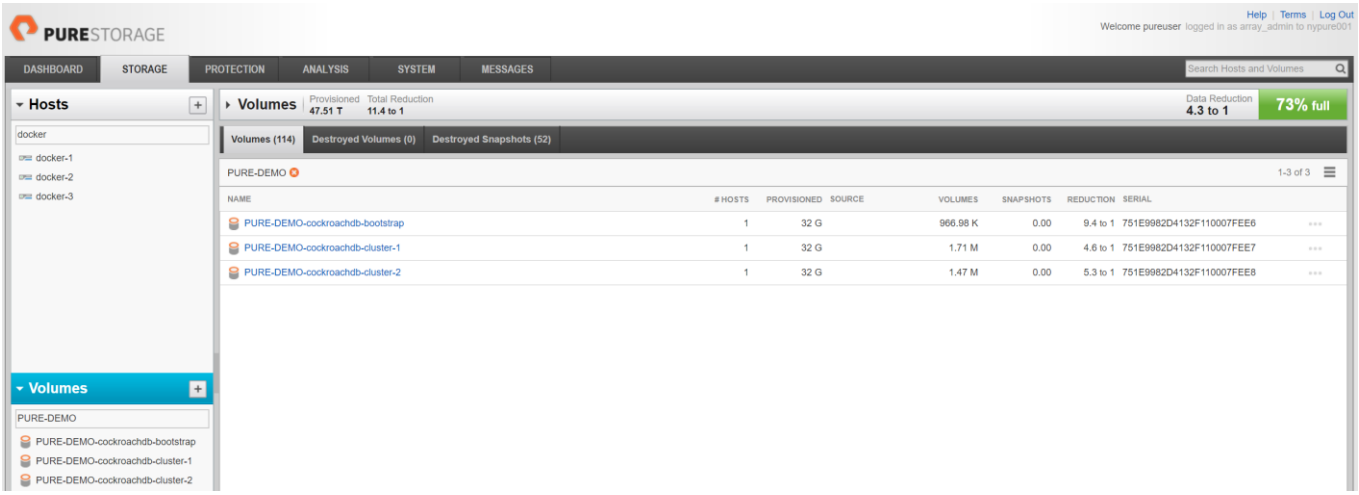
The steps the script perform are inspired by a CockroachDB deployment and orchestration described in their official documentation.

```
#!/usr/bin/env bash
set -x

docker network create --driver overlay cockroachdb

# Start the bootstrap node
docker service create \
--replicas 1 \
--name cockroachdb-bootstrap \
--network cockroachdb \
--mount type=volume,source=cockroachdb-bootstrap,target=/cockroach/cockroach-data,volume-driver=pure,\
volume-opt=volume_label_selector='purestorage.com/backend=block' \
--stop-grace-period 60s \
cockroachdb/cockroach:v1.0 start \
```

```
--advertise-host=cockroachdb-bootstrap \
--logtostderr \
--insecure

until [[ "$(docker inspect $(docker service ps -q cockroachdb-bootstrap | head -1) --format '{{ .Status.State }}')"
== *"running"* ]]; do sleep 5; done

# Start the cluster, point it to the bootstrap node
docker service create \
--replicas 2 \
--name cockroachdb-cluster \
--hostname "{{.Task.Name}}" \
--network cockroachdb \
--endpoint-mode dnsrr \
--mount type=volume,source="cockroachdb-cluster-{{.Task.Slot}}",target=/cockroach/cockroach-data, \
volume-driver=pure,volume-opt=volume_label_selector='purestorage.com/backend=block' \
--stop-grace-period 60s \
cockroachdb/cockroach:v1.0 start \
--join=cockroachdb-bootstrap:26257 \
--logtostderr \
--insecure

until [[ "$(docker inspect $(docker service ps cockroachdb-cluster | grep cockroachdb-cluster.1 | head -1 | awk
'{print $1}') --format '{{ .Status.State }}')" == *"running"* ]]; do sleep 5; done

# Kill the bootstrap node, we don't want it creating new clusters if it restarts
docker service rm cockroachdb-bootstrap

# Join it back to the custer (they need it to add nodes), and point it at the
# cluster dns name.
docker service create \
--replicas 1 \
--name cockroachdb-bootstrap \
--network cockroachdb \
--mount type=volume,source=cockroachdb-bootstrap,target=/cockroach/cockroach-data,volume-driver=pure,\
volume-opt=volume_label_selector='purestorage.com/backend=block' \
--stop-grace-period 60s \
cockroachdb/cockroach:v1.0 start \
--advertise-host=cockroachdb-bootstrap \
--join=cockroachdb-cluster.1:26257 \
--logtostderr \
--insecure
```

If you look at the script you can see that the `service create` commands use a switch, `volume-driver`, that is set to **pure** and `volume-opt` is set to **volume_label_selector='purestorage.com/backend=block'**. These are the instructions for Docker to use the Pure Service Orchestrator, but specifically FlashArrays, as the location for the container volumes.

After running this script, a Pure Storage FlashArray shows the automatically created hosts and volumes for the CockroachDB services



Notice that each of the volumes has been created with a size of 32GB. This is the default size for volumes, but you can override this if required using the `volume-opt=size=<VALUE>` parameter in the `--mount` switch with the `volume-driver=pure` parameter.

We can see the CockroachDB service running through following Docker command:

```
# docker service ls
ID                  NAME                    MODE                REPLICAS            IMAGE
PORTS
j05mueka560p        cockroachdb-cluster     replicated          2/2                 cockroachdb/cockroach:v1.0
mj2w1uekb1l2        cockroachdb-bootstrap   replicated          1/1                 cockroachdb/cockroach:v1.0
```

With the distributed SQL database running in the Swarm we can now show the failover of a persistent volume from one server to another, maintaining the integrity of the Docker service.

Failover of Volumes

Initially, let's look at the persistent volumes that were created by starting the service and see which servers they are connected to.

```
pureuser@nypure001> purevol list --connect PURE-DEMO-cockroachdb-*
Name                              Size  LUN  Host Group  Host
PURE-DEMO-cockroachdb-cluster-1   32G   1    -           docker-1
PURE-DEMO-cockroachdb-cluster-2   32G   1    -           docker-3
PURE-DEMO-cockroachdb-bootstrap   32G   1    -           docker-2
```

Each node has one volume connected to it, as would be expected in this post-install configuration.

To simulate a node failure we will drain one of the servers running one of the cockroachdb-cluster replicas. This will force the replica to be recreated on one of the remaining nodes in the swarm.

In this example, docker-1 is running one of the replicas:

```
root@docker-1:~# docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS               NAMES
785397f80052        cockroachdb/cockroach:v1.0     "/cockroach/cockro..."   3 hours ago         Up 3 hours          8080/tcp, 26257/tcp   cockroachdb-cluster.1.jjt7qqgo5i0pdwg8rvrr988sj
```

so we will drain this node:

```
root@docker-1:~# docker node ls
ID                           HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
2wqf820tqc454oawzyterkjh7    docker-2            Ready               Active
3uuqrko9wpyk6lib6rtkbyif7    docker-3            Ready               Active
zlksbko5y9yk1gda78qesjwoa *  docker-1            Ready               Active              Leader
root@docker-1:~# docker node update --availability drain zlksbko5y9yk1gda78qesjwoa
zlksbko5y9yk1gda78qesjwoa
```

Now we check that the service replica has restarted correctly:

```
root@docker-1:~# docker service ls
ID                  NAME                    MODE                REPLICAS            IMAGE
PORTS
j05mueka560p        cockroachdb-cluster     replicated          2/2                 cockroachdb/cockroach:v1.0
mj2w1uekb1l2        cockroachdb-bootstrap   replicated          0/1                 cockroachdb/cockroach:v1.0
```

and if we look back at the Pure Storage FlashArray, the volume that was associated with *docker-1* (`PURE-DEMO-cockroachdb-cluster-1`) has now been remounted on *docker-2*, with no volumes now on *docker-1*:

```
pureuser@nypure001> purevol list --connect PURE-DEMO-cockroachdb-*
Name                          Size  LUN  Host Group  Host
PURE-DEMO-cockroachdb-cluster-2  32G   1    -           docker-3
PURE-DEMO-cockroachdb-bootstrap  32G   1    -           docker-2
PURE-DEMO-cockroachdb-cluster-1  32G   2    -           docker-2
```

Conclusion

As applications evolve from monolithic architectures to those based on more agile microservices, operations teams need to meet the needs of these developers on both a server and storage based front.

With the evolution of containers these, initially, seemed to solve the infrastructure solution for developers, but as applications that require a more persistent nature are being developed and ported to containers, the ephemeral and stateless nature of containers has caused issues.

By providing the ability to provide external persistent volumes for containers, Docker has expanded its reach into the DevOps world and allowed further growth of containers across clustered environments. This has allowed more traditional applications, such as databases, to be deployed in these new infrastructures.

The Pure Service Orchestrator Docker volume plugin now allows the DevOps community to fully utilize the enterprise-class capabilities of a Pure Storage backend storage providers, including the FlashArray's state of the art data reduction capabilities, solid-state speed and seamless expansion and upgrade functionality. Together with the simplicity inherent in Pure Storage solutions and the simple integration with Docker and Docker Swarm environments, the Pure Service Orchestrator Docker volume plugin has expanded the

capabilities of the DevOps community's deployment and development functions to fully maximize their infrastructure solutions.

About the Author

As Director of New Stack, Simon is responsible for developing the technical direction of Pure Storage pertaining to Open Source technologies including OpenStack, Containers, and associated orchestration and automation toolsets. His responsibilities also include developing best practices, reference architectures and configuration guides.

With over 30 years of storage experience across all aspects of the discipline, from administration to architectural design, Simon has worked with all major storage vendors' technologies and organisations, large and small, across Europe and the USA as both customer and service provider.

Blog: http://blog.purestorage.com/author/simon

PURESTORAGE

Pure Storage, Inc.
Twitter: @purestorage
www.purestorage.com

650 Castro Street, Suite #260
Mountain View, CA 94041

T: 650-290-6088
F: 650-625-9667

Sales: sales@purestorage.com
Support: support@purestorage.com
Media: pr@purestorage.com
General: info@purestorage.com