WHITE PAPER:

A NEW STORAGE ARCHITECTURE FOR THE COMMODITIZATION ERA

portworx

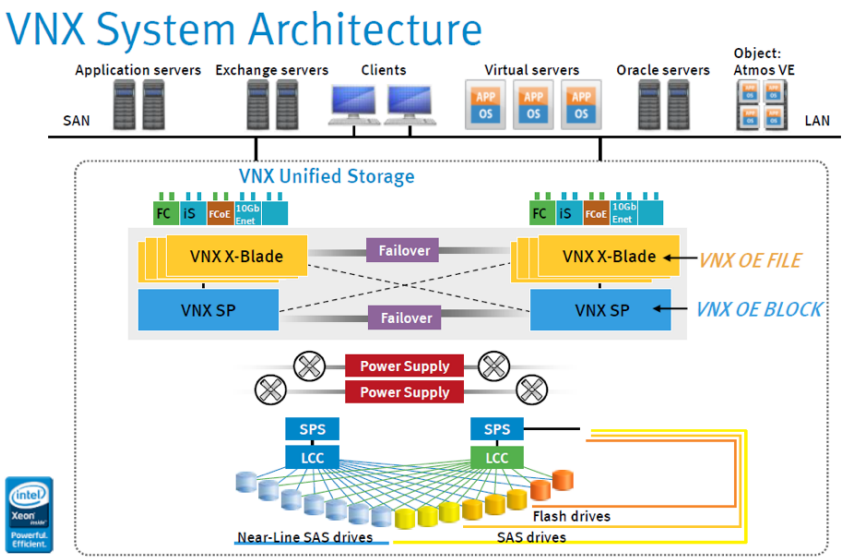
INTRODUCTION

Over time, every technology gets commoditized and democratized. Now it's time for enterprise storage to follow. Just a few years ago, if you needed tier-1 capacity storage, say about 20TB, it almost certainly had to come from your SAN or NAS vendor. These SAN and NAS systems involved custom hardware, custom switching, and a certified engineering staff to operate the equipment.

Today, it's cheaper to get that much capacity from your favorite server vendor—plus, you get to use that server for compute as well. But what is missing is a freshly designed software architecture for storage that can be deployed as software and that can take advantage of the commoditization trends in server computing. This white paper describes advances in software-defined storage that take advantage of the commoditization of tier-1 memory.

MODERN DATACENTER TRENDS

Managing storage used to require a very specialized skill set, with knowledge of fibre channel switching, iSCSI, FCoE, NFS/CIFS, etc. If you had to get tier-1 capacity, it came like this:



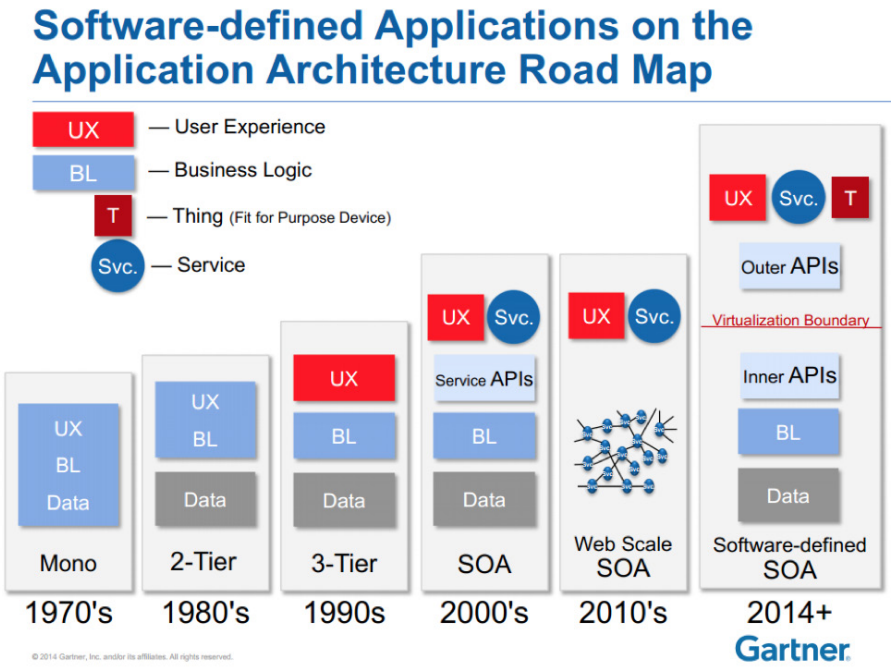
That picture says it all—incredibly complex and archaic. If you want to build an agile datacenter, this is not the picture you want to see.

Today, you can get comparable tier-1 capacity in a 2U form factor on a single server. Google, Facebook, and other large-scale datacenters realized this and hired teams of engineers with the specialized skill sets to build a distributed

fault-tolerant storage layer that takes advantage of x86 architecture for both compute and storage. If we look to these companies as bellwethers of modern datacenter architectures, we see a physical infrastructure that contains scaled-out x86 servers with a decent amount of local tier-1 capacity.

In parallel with datacenter server advances, the software industry has been making advances, as well:

We live in an era of service-oriented, scalable architectures. These services are now being deployed as isolated, fully functional containers that let you build bigger solutions using smaller solution components.



Acknowledging the major shifts happening in datacenter server and software technology, this article focuses on datacenter storage—and more specifically, a new software architecture for stateful infrastructure services purpose-built for the future.

EXAMPLES OF SOFTWARE-DEFINED STORAGE TECHNIQUES FOR LINUX CONTAINERS

Based on the the trajectory of modern datacenter trends, we have focused on building a new storage architecture from the ground up to take advantage of the following properties:

1. **Service-oriented storage provisioning:** Provisioning storage volumes per service as opposed to via a physical server or a virtual machine means that

the old techniques that relied on FCoE,or iSCSI can be booted out the door.

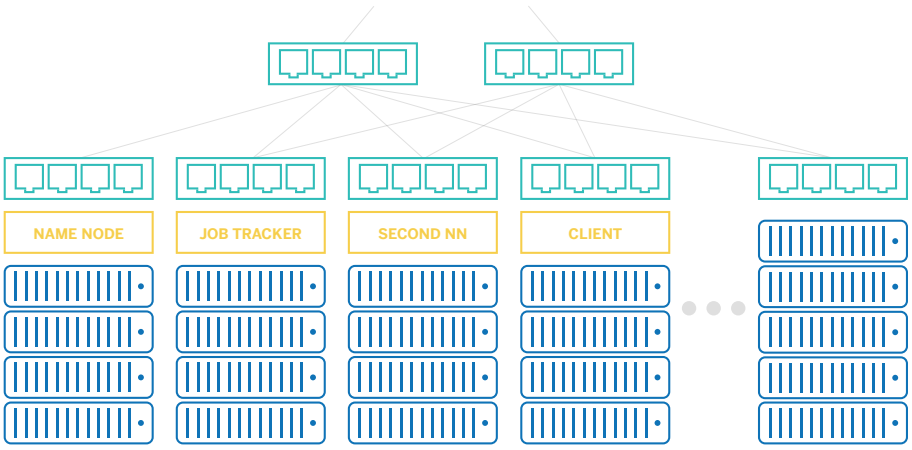
2. **Higher granularity number of volumes per service:** Most modern services, such as NoSQL and message queues, are designed for scalability. That is, they are deployed in higher numbers with a smaller local capacity. Contrast this with databases from a few years ago having just one compute node and a larger number of terabytes for that node.

3. **Multiple local tiers of storage:** Today's servers have at least two tiers of storage. Given the realities of global server procurement, you are likely to have multiple servers from different vendors, with different internal local capacity tiers. You want your storage architecture to understand these variations and automatically take advantage of what is provided to it.

4. **A desire for hyperconvergence:** As newer stateful services are architected, they prefer that data be co-located with the compute. Take Hadoop as an example, where compute is dispatched to the service running on a node that hosts the data.

5. **Multi-service granular storage operations:** Your applications are not one monolithic stack. Instead, they are deployed via distributed orchestration software across multiple nodes. Enterprise storage operations such as snapshots, checkpoint restore, replication, quotas, and so on need to be made available at a service stack level.

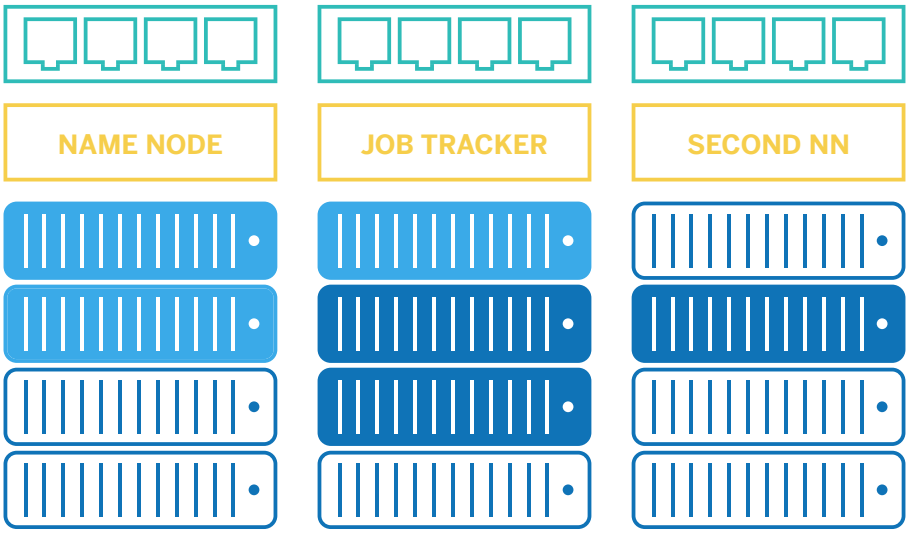
We architected a storage fabric that is aware of a distributed set of x86 servers tied together by an Ethernet fabric.



It is important to note that the hardware fabric in the datacenter is subject to:

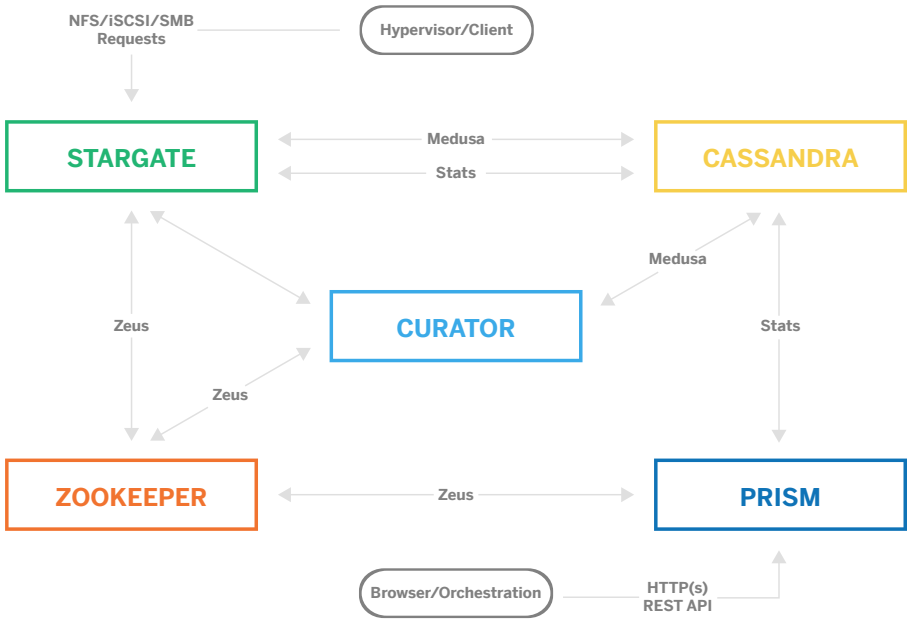
1. Servers from different vendors
2. Servers of varying storage capacity and tier types
3. Top-of-rack switchWes with different capabilities
4. Software running on each node at a higher granularity due to its decomposition as a set of containerized services

So how do you scale a storage layer to work with a large number of relatively thinner servers? The answer is to split the stack into separate data-availability zones and compute zones.



The scale at which container services are deployed straddle many data placement zones. The figure above shows how a container-aware storage architecture segregates the data placement zones across container placement boundaries. This architecture lets you build a much larger-cluster compute fabric for your containers, while keeping storage aggregation to the desired level of IO performance. For example, you can build a t1,000-node WordPress architecture while still guaranteeing a minimum of a “top-of-rack” switch-capped IO latency on storage.

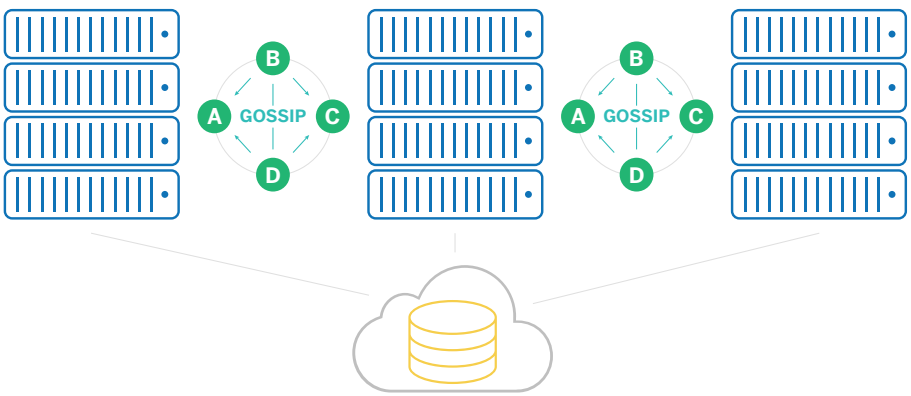
Another core architectural pattern in building these scale-out storage fabrics is to avoid having any central metadata server. A typical scale-out storage architecture traditionally has looked like this:



These systems aim to achieve monolithic cluster scale. That means that they treat entire application stacks as a monolithic entity, such as a VM or a machine and attempt to scale to thousands of nodes—assuming that any one entity should be able to span the entire physical cluster. Such architectures, while not scalable, more importantly introduce multiple layers of discovery and metadata schemes, such as shown in the diagram above.

This is the antithesis of the more fine-grained, self-recoverable, and discoverable service-oriented architectures for which today's software is written.

The diagram below shows a more elegant storage fabric that does not require any core metadata server component that is in the data path for data discovery and relies on a more social and democratic protocol for achieving some notion of a working cluster, namely a gossip protocol.



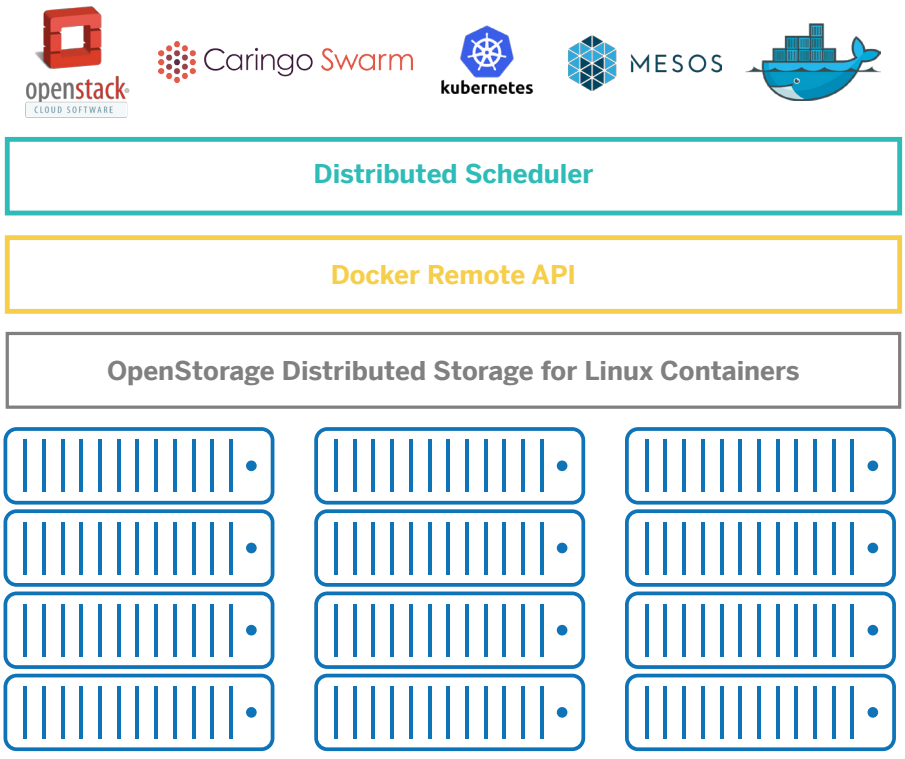
The diagram above shows a more scalable distributed computing model where the cluster capacity and compute are coordinated loosely via a gossip protocol In the rare event of a disagreement on the datacenter topology, a cloud-accessible key value system is consulted.

These are a few examples of modern infrastructure architectural traits that we leveraged when designing a software-defined storage stack for the cloud- and service-enabled era.

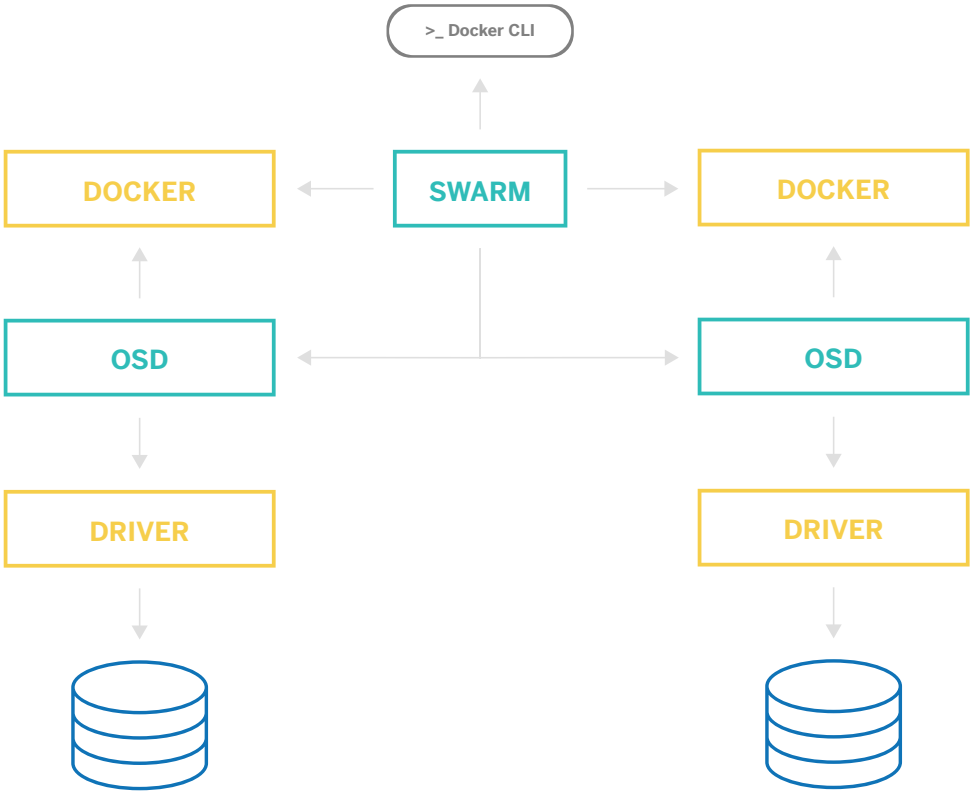
SCHEDULER AND CONTAINER ENGINE INTEGRATION

As application software has matured into a more service-oriented architecture, application software orchestration and scheduling have also evolved. We no longer want to manually manage the lifecycle of a process; we want to automate the start, stop, and lifecycle of when software runs. We also want to automate where the software runs and what its SLA requirements are.

That's where the modern schedulers—such as Mesosphere, Kubernetes, Swarm, Spark, and so on—enter the picture.



Today's application architects and DevOps teams are writing to these schedulers directly. As a result, it makes most sense for the infrastructure software, such as your SDS layer, to integrate directly with these orchestration layers.



As can be seen above, the SDS layer will integrate directly with the scheduling software to:

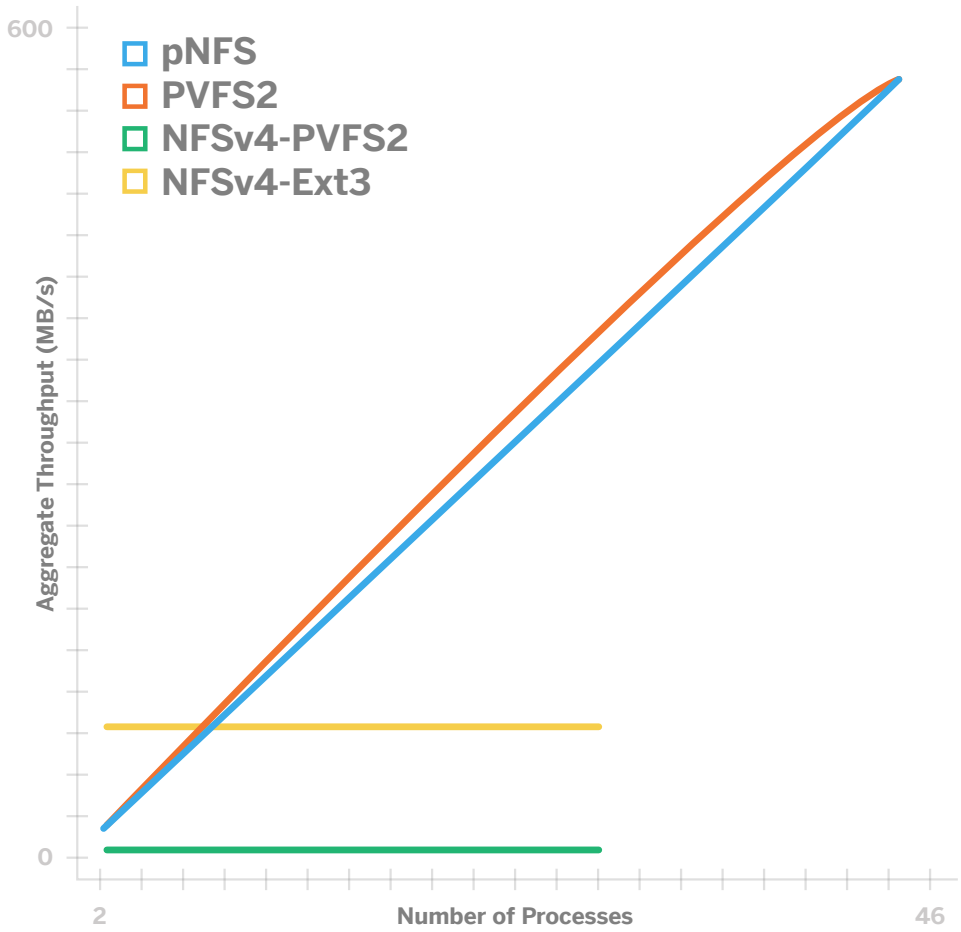
a) Influence container (service) placement decisions

b) Move data close to where the service has been dispatched

These are two critical pieces of information that are missing in any pre-service dated storage software.

CONTAINER-CONVERGED DATA ACCELERATION TECHNIQUES

By designing the SDS layer assuming a multi-server distributed architecture, and knowing the Ethernet capabilities and topologies (especially with the advent of 10+G Ethernet datacenters), it is possible to purposely distribute data across multiple nodes to "parallel source" them from multiple servers during IO. This ensures cluster-wide balancing of both IO and compute and avoids creating either IO or compute bottlenecks.



The graph above shows parallel scaling of IO in an inherently multi-sourced IO fabric. This configuration takes advantage of the sub-millisecond latency between servers and distribute the IO workload.

THE IMPORTANCE OF UNIFIED STORAGE ACCESS

Service-oriented applications can broadly be classified into three types:

1. **Stateless services:** These are usually ephemeral compute jobs that have little or no scratch storage needs. They rely on other stateful services to do something meaningful.
2. **Stateful database services:** These typically require block storage or non-shared storage access.
3. **Stateful services that require file or object access:** Examples include a global file or object namespace.

A distributed scale-out storage architecture designed for the service-oriented datacenter needs to keep these distinctions in mind and provide the right type of storage volume to each service. In other words, a block-only or file-only solution is inadequate.

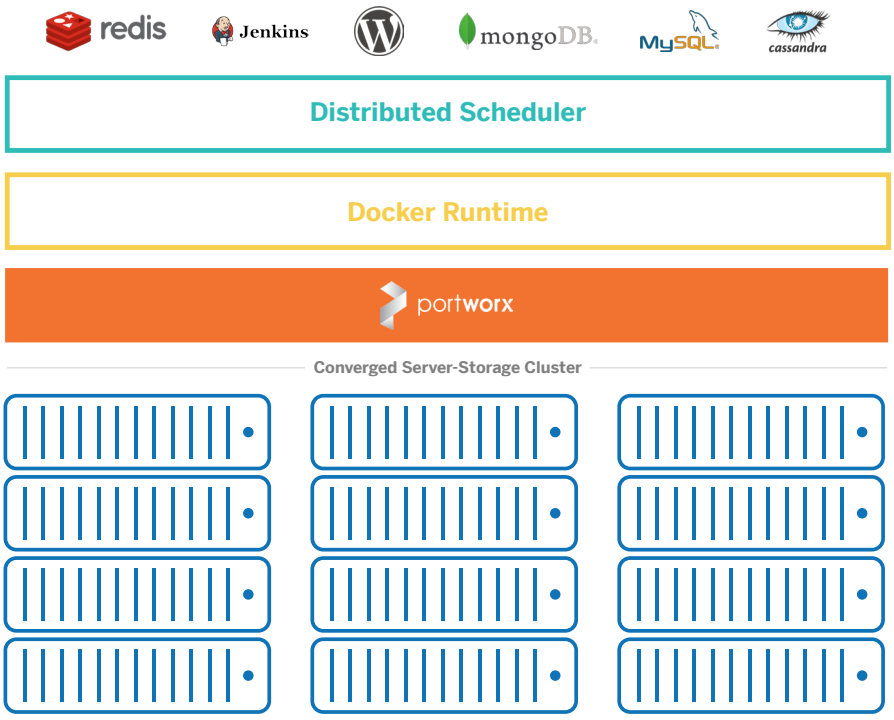
When dealing with services that require file or object access, the storage is being accessed as a global namespace from many different containers running on different machines—and potentially on different datacenters or even across clouds. Any storage fabric that wants to cater to a cloud-native application architecture should be architected to handle these different scenarios.

WHAT PORTWORX DOES

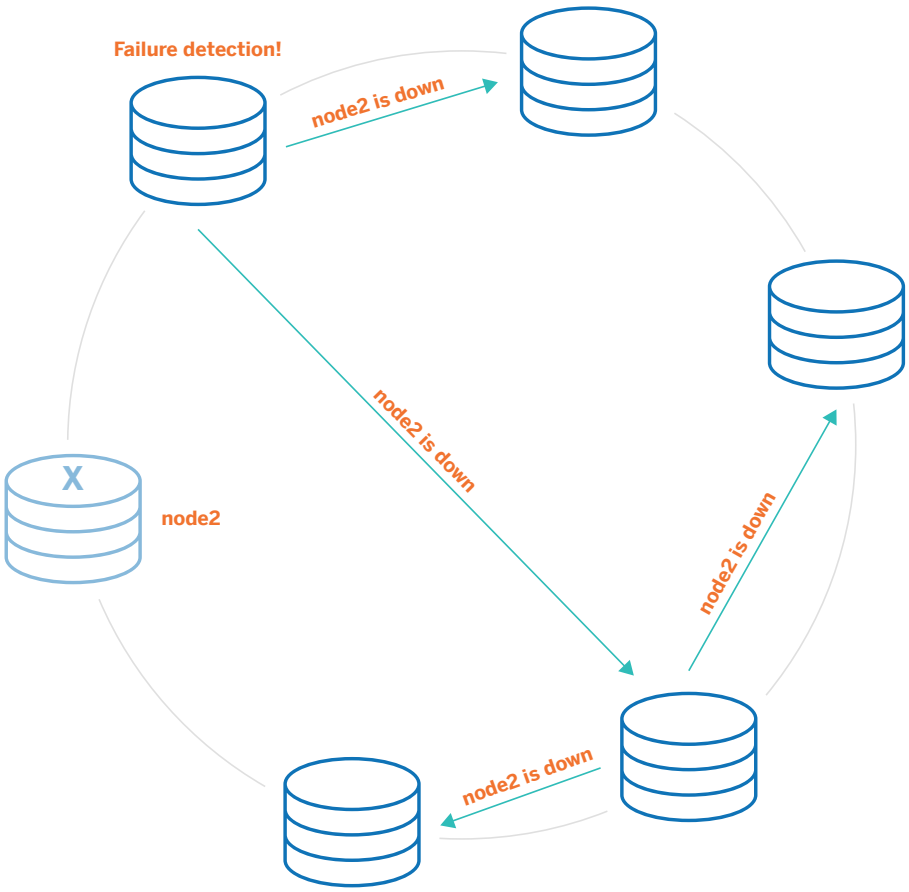
Portworx has built a new elastic scale-out unified storage stack, from the ground up, based on the observations made in this article.

At the core, it is a shared-nothing, loosely distributed metadata-based block storage layer. Portworx itself deploys as a containerized service known as PX, on every node in the cluster. Once deployed, every PX node participates in the following stages:

1. **Fingerprinting:** PX nodes fingerprint the server's hardware. detecting the type of drives, capacity, and overall server capabilities. This capacity is used for scheduling and IO participation.
2. **Discovery:** PX nodes discover other nodes in the cluster based on authorization certificates and a cluster ID. At this point, every PX node is aware of the overall topology and participating nodes in the datacenter (and across datacenters). The overall topology conveys region and rack awareness, as well as awareness of the capabilities of every other node in the cluster.



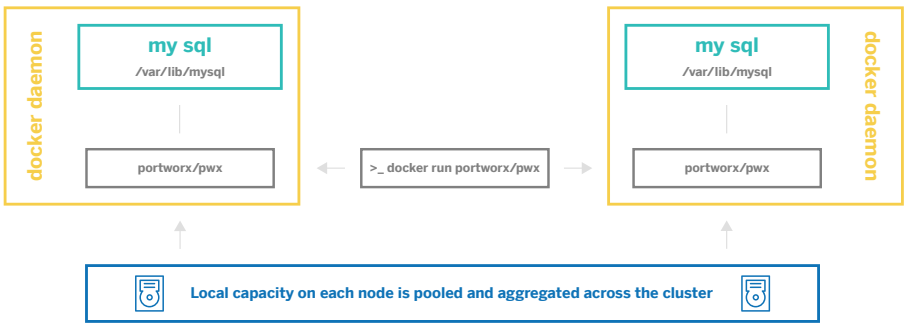
3. **Cluster formation via gossip:** Once the PX nodes discover each other, they start communicating among themselves in a highly efficient gossip protocol. This ensures consistency in the overall state of the cluster without creating a scalability bottleneck.



4. **Storage provisioning via scheduler integration:** After the PX cluster is in quorum, storage can be provisioned to instances of containers via the orchestration fabric. PX is integrated with various schedulers such as Kubernetes, Swarm, Mesosphere, Spark, and others. PX handles the control plane protocol to establish the correct type of storage volume into each running instance of a container. Based on the container's IO spec, the volumes can be provisioned to deliver a certain level of SLA and CoS. Here is an example of a PX volume being provisioned via a Kubernetes POD spec:

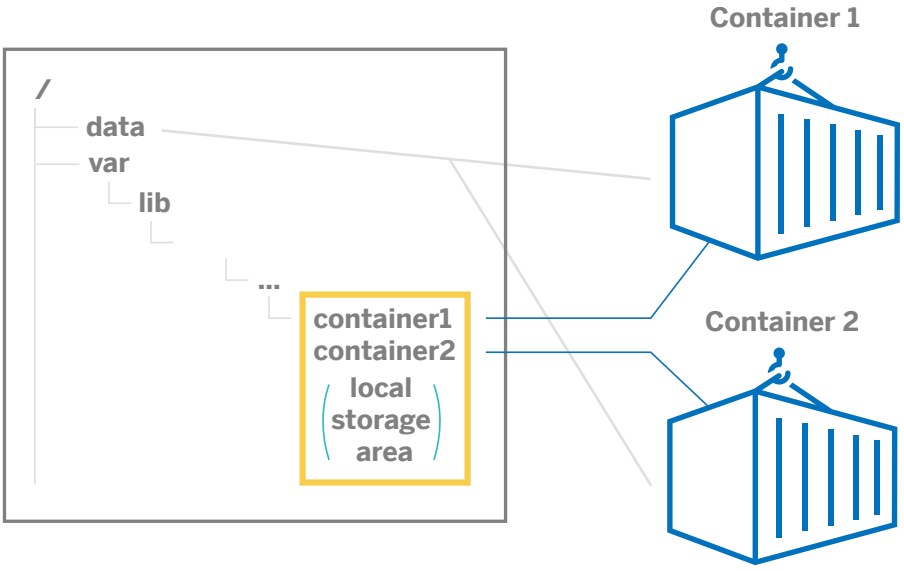
```
apiVersion: v1
kind: Pod
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  containers:
    - resources:
        limits:
          cpu: 0.5
      image: mysql
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          # Change this password!
          value: yourpassword
      ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
          # This name must match the volumes.name below.
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
  volumes:
    - name: mysql-persistent-storage
      pwxDisk:
        # This disk will be created if it does not already exist
        pwxDiskName: mysql-disk
        fsType: ext4
        cos: 2
        snapshots: 12h
        haFactor: 3
        size: 4TB
```

The volumes are thinly provisioned, and the actual physical capacity comes from the global pool of capacity available across all server nodes. Other properties such as snapshot intervals, block sizes, etc. can be configured on a container-granular volume basis.



5. **Data placement:** Once the volume has been provisioned to a running container, PX is in the data path. Depending on the type of volume, either a block device or a shared global namespace volume is wired into

the container. As the container writes data out, the blocks are strategically placed on other servers to ensure high availability of the data. Metadata associated with the volume is made part of the volume itself, avoiding costly and hard-to-maintain central metadata servers. These volumes can be thought of as container-addressable volumes, thereby making this a very scalable and easy-to-maintain architecture. This process has the same advantages of content-addressable storage, but these patent-pending algorithms operate at a distributed container granularity called "container-addressable storage."



6. **Lifecycle management:** PX maintains the lifecycle of a volume past the container's lifecycle. These volumes can be cloned, archived to other tiers, or even moved to other services such as S3. PX maintains a rich history of IO activity on each volume and exposes this via a RESTFul interface and the PX CLI, pxctl.

FOR FURTHER INFORMATION

PX is being used to solve use cases ranging from genomics research, movie production and rendering, and scaling database services such as NoSQL to standard DevOps-centric SoA provisioning. We expect this new era of commoditized distributed computing to continue to disrupt how the modern datacenter looks for years to come.

For more information, try out our developer edition of PX at

https://github.com/portworx/px-dev/

Portworx, Inc.
900 Veterans Boulevard, Suite 230
Redwood City, CA 94063
Tel: 650-241-3222 | info@portworx.com | www.portworx.com

PB-PXE-7-1-16