Platform Architecture

Version: 1.0

Architecture Overview

The Portworx storage and data management solution is built from the ground up for cloud native containerized applications. It is delivered as a container and at the very core, its storage architecture is distributed scale-out and scale-up block storage. The technology stack is built upon the following set of design principles to support high density massively distributed cloud native applications:

- Heterogeneous systems
- Scalability
- Data availability
- Application level granularity
- Deployment density
- Declarative management and extensibility
- Flexible deployment models
- Observability
- Orchestrated software deployment and management

Design Principles

Heterogeneous systems

Even if clusters don't start off as heterogeneous systems, they generally evolve into them. These clusters have systems with different CPU, memory, and disk profiles. Furthermore, the disk configuration within one system can be heterogeneous both in terms of size and media type (HDD/SSD/NVMe).

Portworx software runs within a container on any Linux host. Backing storage devices on these hosts can be of any size or media type, and present to Portworx as block devices. Portworx classifies these devices according to their size and performance and aggregates them into storage pools across nodes.

Portworx further aggregates and virtualizes these storage pools into virtual volumes which can then be consumed by cloud native applications that specify their storage requirements in a declarative fashion.

Scalability

As clusters grow and the number of deployed cloud native applications grow, the system must scale in terms of number of physical objects in the cluster (nodes, storage devices) and virtual objects (volumes, snapshots, schedules, etc.). Scaling up or scaling down the cluster should not be disruptive operations.

Portworx control protocols are designed for large clusters with support for an extremely high number of virtual objects. I/O operations do not incur any metadata lookup and data for any given volume is distributed across a small set of nodes allowing the cluster to scale without affecting I/O scalability.

Data availability

A storage cluster must ensure the integrity of its data. No single failure in any of the following areas should result in data unavailability:

- Drive
- Network
- Node
- Availability zone

Additionally, the blast radius of a failure should be contained as much as possible.

Portworx is topology aware and distributes replica data across fault domains. Data for a replica may be striped across a small subset of nodes within a fault domain. An intelligent provisioning algorithm distributes volumes across the cluster.

Application Level Granularity

Clusters are not just composed of different physical resources, they also run applications with different availability, performance, and lifecycle management requirements. The storage infrastructure should adapt itself to the requirements of the application. Portworx data structures and algorithms are designed to operate at the granularity of a single volume. Volume operations can also be grouped at the clustered-application level when required.

As examples with Portworx, you can do the following:

- Configure the replication level and I/O priority for individual virtual volumes
- Take snapshots of both single virtual volumes and groups of volumes
- Migrate individual virtual volumes and groups of volumes between clusters

Density

Modern server configurations are densely packed, and allow hundreds of containers to run on a single machine. Storage requirements for these systems include supporting a control plane for hundreds of distinct devices while ensuring quality of service across all devices.

Existing mechanisms such as exporting block storage over the network using iSCSI cannot support this. By comparison, Portworx can support hundreds, or even thousands, of attached volumes per host, and can perform control plane operations in the order of milliseconds.

Declarative management and extensibility

Operating large scale clusters should require minimum admin interaction. The cluster should converge to a system defined by the operator. Exception should be signaled and allow configurable actions.

Portworx converges to the state defined by the cluster operator and generates appropriate events if it is unable to. Internally, the stack operates on a set of rules that define thresholds and actions to take. These thresholds can be extended by defining external rules. The cluster self heals on the basis of these rules.

Multiple Deployment Modes

There are two primary approaches for operating a cluster in production:

- Disaggregated storage: storage and compute operations occur on separate nodes
- Hyperconverged: storage and computer operations occur on the same node

The network topology and/or physical hardware of the cluster may dictate which approach you choose. The storage infrastructure should support both modes of deployment.

Portworx consumes a minimal amount of resources, allowing it to co-exist with user applications running on the same node. Additionally, all resources (compute, networking, and memory) are tunable to a very high granularity, allowing Portworx to take advantage of resources when available. Portworx also supports both modes of replication where replication occurs over the client network or on the storage network depending on the network topology.

Metrics, Tracing, Observability

Metrics provide crucial cluster visibility and aid in troubleshooting, AI, and machine learning tasks. Particularly, metrics should be available at all levels:

- Physical
- Virtual resources
- I/O requests
- Networks
- Disks
- Applications

Portworx collects metrics with high frequency and provides tracing using LTTNG. Collecting these metrics does not impact performance and offers rich into IO patterns, application behaviors, and cluster operations.

The Portworx rules-based engine incorporates these metrics to react programmatically to events in the cluster.

Deployment and Software Management via orchestration systems

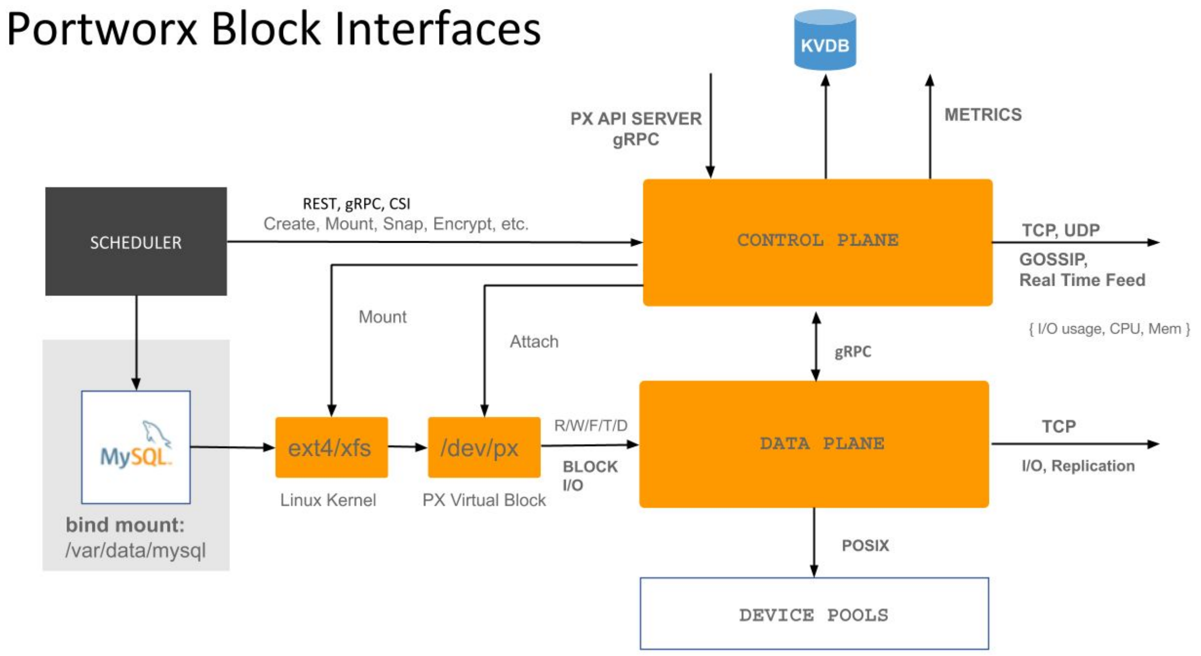
Orchestration systems, such as Kubernetes, deploy and manage cluster software at cloud scale. These orchestration systems also upgrade individual components independently. The storage infrastructure should also be deployed and managed via the orchestration system.

Portworx integrates with orchestration systems and embraces the concept of micro services to run as independent components and communicate over well defined protocols. The Portworx lifecycle is completely managed by the orchestration system that manages the cluster.

Portworx Components

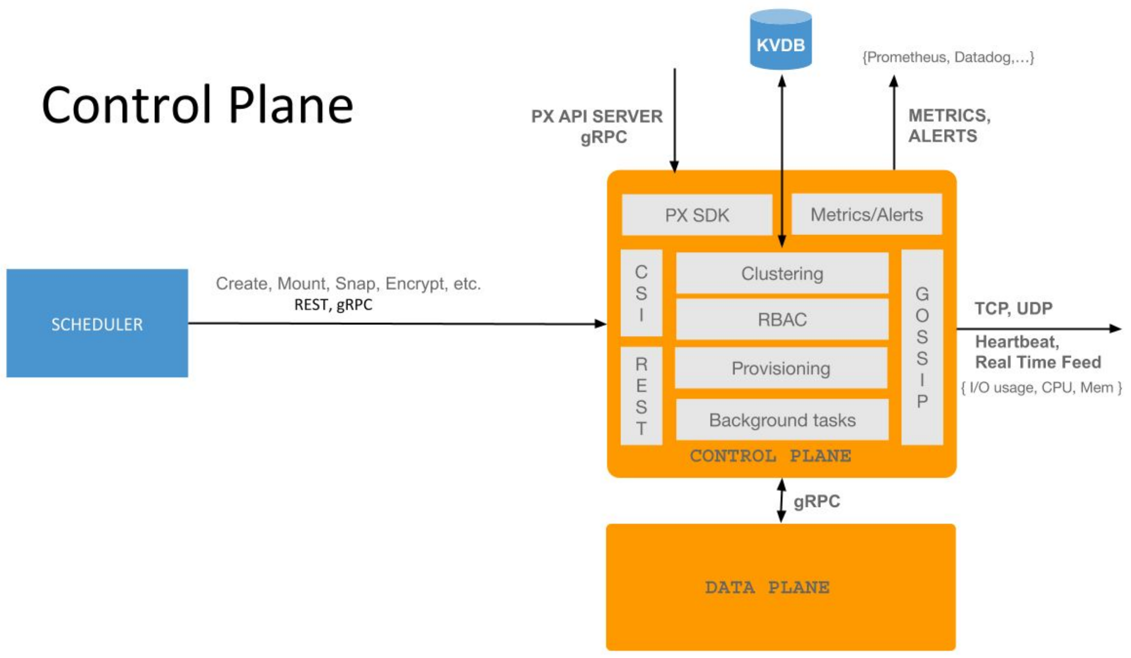
Portworx installs itself as an independent Open Container Initiative (OCI) runC container, and has no dependencies on the host. For information on Portworx installation methods, visit the Portworx documentation.

Within the container, Portworx is divided into two major components: the control plane and the data plane. These components communicate over a well defined interface over gRPC. The diagram below describes the interfaces and protocols of communication between various parts parts of the systems:



Each of these components are described in detail in the subsequent sections.

Control Plane



The Portworx control plane is responsible for clustering, provisioning, and scheduler interaction. It runs as a separate process and communicates control information with the data path over gRPC.

Clustering

At a high level, Portworx is provided with a way to talk to the KVDB (key-value database), and a cluster ID. The cluster ID is used to query the KVDB, that is either internally bootstrapped or provided externally, to form a new cluster or to join a cluster. The KVDB stores cluster membership information as well as volume configuration information.

A quorum is reached if there are a sufficient number of storage nodes that are able to talk to each other over gossip. Storageless nodes do not have voting rights to form a cluster but can join a cluster that has established quorum. Once part of a cluster that has established quorum, both storage and storage-less nodes have equal rights to perform all control plane operations.

Backing storage devices are aggregated to create storage pools. A single node may have multiple storage pools. Each pool contains drives of the same type and size. Pools are categorized based on performance which can be used during provisioning by specifying an *io_priority*. Storage pools across nodes are aggregated by the control plane.

A PX volume is created by talking over one of the available interfaces: CSI, REST, or gRPC. When a PX volume is attached, a block device appears on the host where it is attached and is mounted. The orchestration tool (e.g Kubernetes) bind mounts the mount point within the container.

The PX block devices see I/O from the filesystem over standard linux block interface and this I/O is served by the PX dataplane. Standard Posix is used to talk to the storage pools and custom messages are dispatched over raw TCP for communication with replicas. At all times, a gossip protocol monitors the health of the nodes in the cluster.

KVDB

A clustered key-value database (KVDB), *etcd* by default, serves as the single source of truth for the entire cluster. The KVDB maintains cluster membership information as well as configuration for every volume.

The portworx control plane is declarative, all configuration is first set in the KVDB and the control and datapath converges to the configuration. The KVDB also maintains a monotonically increasing cluster version number. This version number ensures update and communication order in a distributed system.

The cluster KVDB is not in the datapath. It is solely in the control path, clustered and is configured to be periodically snapshotted. The KV space from each node is also periodically saved and the entire KVDB can be reconstructed in case of a disaster.

Inter node control protocol

Internode control protocol is a scalable gossip protocol based on SWIM. It allows the cluster to scale to a large number of nodes.

At the control plane, messages are periodically propagated to all members of the cluster. These messages carry information about the following categories:

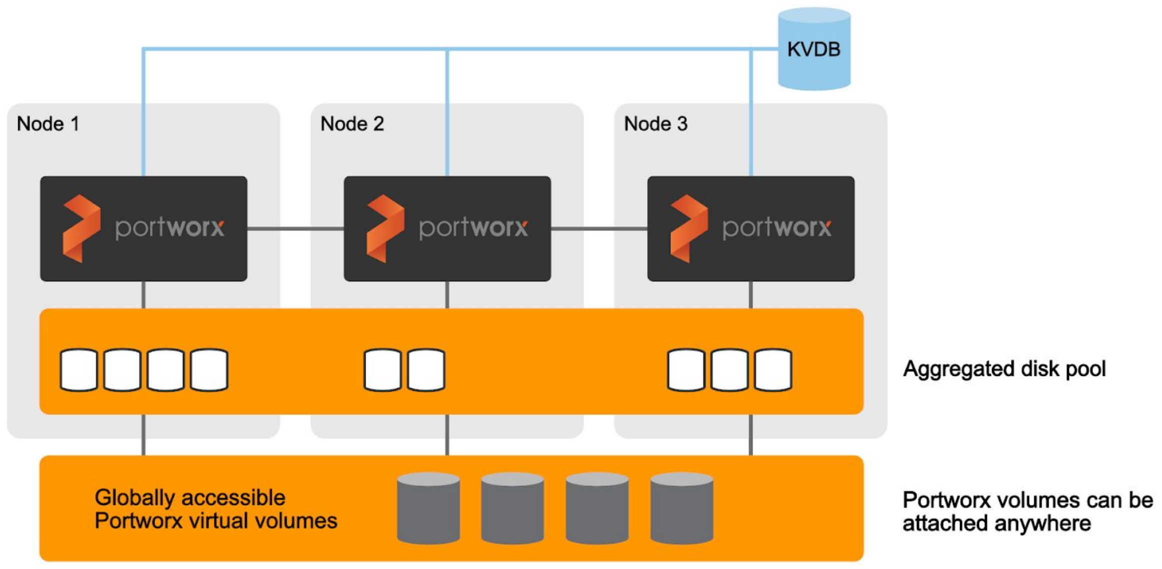
- Node health
- I/O load
- CPU load
- Memory load

This information is passed on to the datapath which uses it to make I/O scheduling decisions. When there are multiple candidates to serve an I/O request, the least loaded nodes are selected.

In an idle system, the gossip protocol also detects nodes that are down. The time taken for detecting a down node via the control path is a logarithmic function of the number of nodes in the cluster and the frequency of the messages sent. Typically, it is in the order of 20 seconds.

Storage Virtualization

Portworx is a storage overlay over a set of physical block devices spread across nodes. The virtual Portworx volumes' data is allocated on the physical block devices.


KVDB
Node 1
Node 2
Node 3
portworx
portworx
portworx
Aggregated disk pool
Globally accessible Portworx virtual volumes
Portworx volumes can be attached anywhere

Storage Pools

Storage pools are created by grouping together drives of the same size and same type. A single node with different drive sizes and/or types will have multiple pools. Within a pool, by default, the drives are written to in a RAID-0 configuration. You can configure RAID-10 for pools with at least 4 drives. There can be up to 32 pools within the same node.

At the time of pool construction, individual drives are benchmarked and are categorized as high, medium, or low based on random/sequential IOPS and latencies. These are applied as individual labels on the pools and can be used in provisioning rules. For example, a provision rule can be written to provision volumes on pools that have random I/O latencies less than 2 ms or *io_priority* high.

Storage pools are continuously monitored for health using *smartctl* utilities. Alerts are raised for discrepancies.

Topology Awareness

Portworx is topology aware. The topology information is hierarchical, encompassing the following:

- Region
- Zone
- DC
- Row
- Rack
- Hypervisor
- Node

This information is auto discovered in the cloud and from orchestration system labels. It can also be passed in as environment variables.

For more information, refer to the Region and Zone Information section of the Portworx documentation.

This information is used in volume provisioning, data rebalancing, as well as upgrades. This is covered in detail in the corresponding sections.

Provisioning

The provisioning algorithm is at the core of the Portworx stack and plays a crucial role in data placement. The goals of this algorithm are to distribute data to keep servers equally loaded(avoid hotspots), distribute replicas across fault domains, leverage capacity and horsepower from multiple nodes, and allow for scaling the cluster to a large number of nodes.

Unlike traditional SDS systems, Portworx does not stripe data across all the nodes in the cluster. Data for a volume is striped across a subset of nodes termed an aggregation set, the default number of nodes in an aggregation set is 1. This number can be higher specifically if the required capacity for a volume exceeds a node or for performance when the backplane in a rack is used for nodes that participate in an aggregation set and the top-of-rack (TOR) switch for replication traffic. Data is replicated across a replication set, which is a set of aggregation set*s*.

Every node can run the provisioning algorithm and produce the same result. Each node has a matrix of every other node’s provisioned, used, and available capacity in every pool. Every node also knows the categorization of all pools as well as the geographical topology of every node.

When a volume is provisioned, the information that comes along amongst other volume properties are the following:

- Number of replicas
- Desired pool categories
- Pool label selectors
- Topology constraints

Based on these input parameters and the storage allocation matrix in the cluster, the algorithm provisions for the following goals:

- Replicas are distributed across failure domains, aggregation sets are allocated in the same topology unit (e.g. same rack)
- Distribute volume allocation equitably across the cluster (no hotspots)
- Limit overprovisioning
- Volumes that belong to the same instance of the application should be on the same node (volume affinity). E.g. Journal and data volumes for mysql should be on the same node.
- Volumes that belong to the same group should be across failure domains (volume anti-affinity). E.g. volumes for cassandra nodes in the same ring should be on different nodes.
- Filter for all topology, and label selectors that are specified in the input rules.

The output of the provisioning algorithm is a replication set: a set of nodes that will host the volume data. Rebalancing operations enter the provisioning rule engine with the same constraints.

The provisioning engine is implemented as a rule engine, the rules for provisioning can be specified in a language that is orchestration system specific. For Kubernetes, the rules are defined in the Volume Placement Strategies section of the Portworx documentation.

Volume Control Operations

Portworx exports a virtual block device. This block device is attached on any host that is part of the Portworx cluster by calling the Portworx attach API, over CSI or REST interface (from the orchestration tool: e.g. Kubernetes, Docker) or over the PX API (a direct call to API or via the Portworx CLI *pxctl*). When a device is attached, it is recorded in the KVDB and the Portworx control plane ensures that no other node can concurrently attach the same block device.

Every attached block device is exported in the Linux namespace under */dev/pxd*. Note that, although this is a virtual block device, it is a bonafide block device as far as Linux is concerned. In particular, it appears in *sysfs*, and all control operations available via sysfs are applicable on the Portworx block device as well.

The KVDB is the source of truth: this means that the block device will eventually be attached to the same node across restarts, reboots, upgrades.

A Portworx block device needs to be detached for it to be attached on a different node. The control plane will enforce that only one node has the device attached at any time. This means that all client containers have terminated and the filesystems on the block device are no longer mounted. Also a device can be detached if the node it was attached on is down. When a

Portworx block device is detached, all pending I/O on the device is flushed to disk before the detach call returns.

If a volume is deleted, the space reclamation is done in the background. Deletion of a volume doesn't affect the snapshots. The control plane exports operations to delete a volume along with its snapshots.

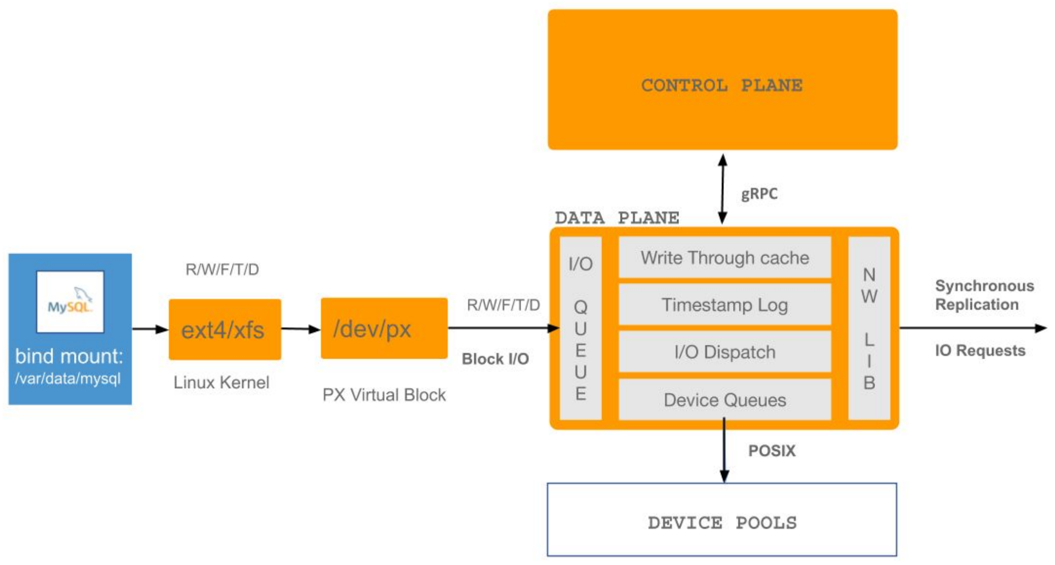
Background Operations and Cluster Coordinator

Detached volumes may be attached internally to perform background operations. Such attached volumes do not export a block device however their state is recorded in KVDB as attached. These volumes are allowed to be externally attached if required.

When a volume is internally attached, it can perform background operations such as ha-increase, raid-scan, resync, and scheduled snapshots. These operations are explained in detail in the data plane section.

Background operations consume only a set fraction of the total I/O and network bandwidth.

Data Plane



The Portworx datapath runs in the user space. The control plane and data plane establish a bi-directional control path over gRPC. The control plane transfers node and volume information to the data path on startup. During runtime, KVBD updates, along with version numbers, are transferred to the data path. These include changes in volumes or nodes in the cluster. The datapath establishes TCP connections to other nodes in the cluster. These connections are idle unless there is inflight I/O. Storage nodes perform I/O to the storage pools over POSIX interface.

Portworx Block Devices

At the time of Volume Creation, the control path provisioning algorithm decides the data placement for the volume and creates a data structure for it in the KVDB. The designated storage nodes for the volume create data structures on the appropriate device pools.

The volume must be attached and mounted to perform I/O. A volume can be attached to any node in the cluster but only one node at any given time. When a volume is attached, a portworx block device is exported in */dev/pxd/* and the node is designated as the transaction coordinator for the volume. All I/O for this volume goes through the transaction coordinator. The device ownership is stored in the cluster KVDB which ensures that no two nodes can decide that they own the device at the same time.

The semantics of a write request are honored. If the write request is a stable write request, then the response is sent only after it is written to stable medium in a quorum number of replicas. All acknowledged writes are stable and will survive crashes and reboots of one or more nodes and/or loss of storage on one of the nodes. In the case of write requests that are not yet acknowledged, they may either succeed or fail. However, once the data in such undecided case has been read back, subsequent reads will return the same data.

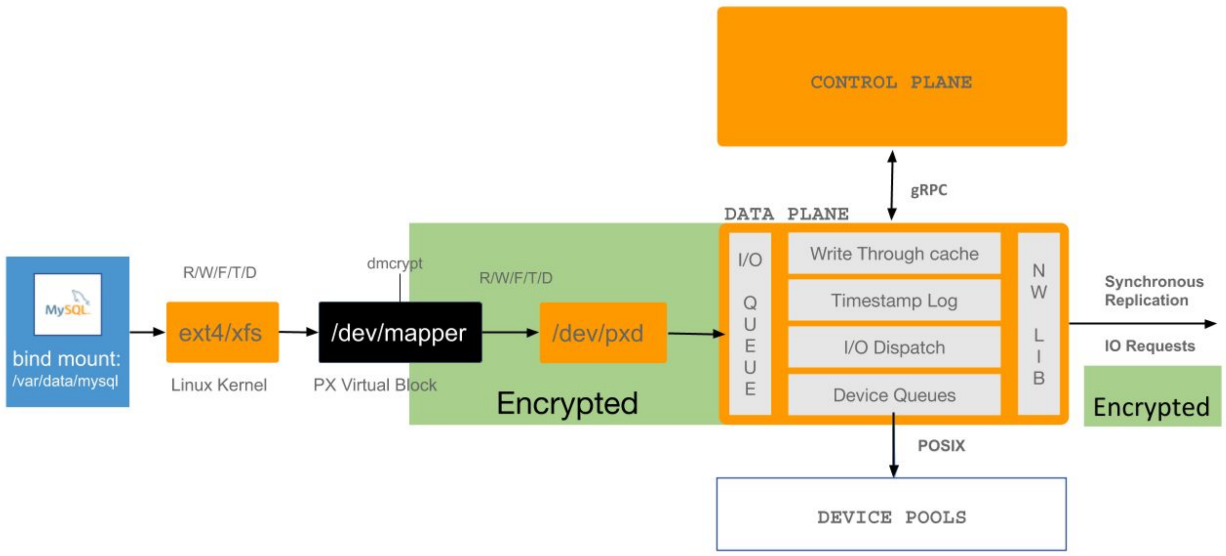
A flush request results in a write barrier, and is only acknowledged after all preceding writes are written to device is written to stable medium in a quorum number of replicas.

A read request can be serviced by multiple replicas when the volume is in a clean state. The I/O load on replicas (as transmitted by the gossip protocol), and the contiguity of read requests on the device (to maintain pseudo sequentiality for read-ahead), the attach point of the device (a local replica is always favored) are factors that determine which replica serves the I/O.

Encryption

Portworx uses the *dm-crypt* device mapper target for encryption of the Portworx virtual block device. Encryption happens in the Linux kernel with the *dm-crypt* module with hardware-accelerated encryption on Intel CPUs - this means that the data will still be encrypted during all storage operations relating to a Portworx volume. No data is interpreted on the device mapper target. The flow of data from containerized application to local device store is illustrated below.

Encryption data flow

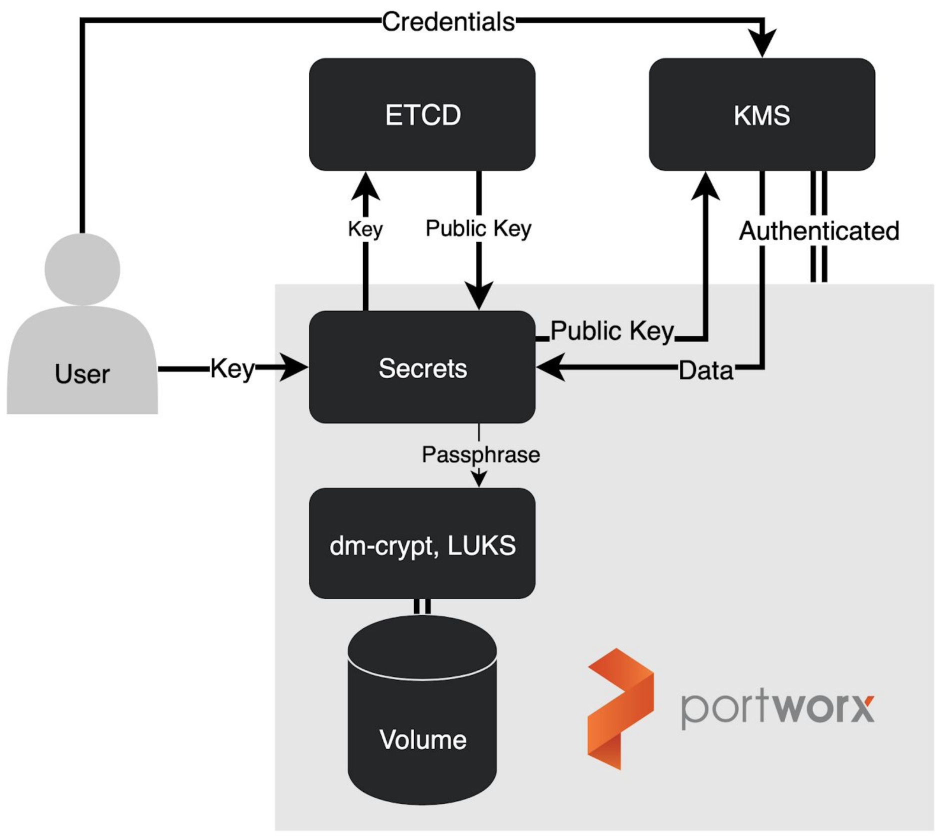


An *ext4* or *xfs* filesystem representing the Portworx volume is bind mounted into the container by the orchestration system. The underlying block device presented to the filesystem is the */dev/mapper* target (LUKS) created by *dm-crypt*. The */dev/mapper* is layered on top of the Portworx virtual block device and all writes get encrypted by *dm-crypt* before it is written to the */dev/pxd* device. In the read path, *dm-crypt* decrypts the data read from the Portworx device before and responds to the client. Portworx never sees unencrypted data: data stays encrypted at rest and over the wire. Backups of volumes are also encrypted.

In the control path, Portworx uses the libgcrypt library to interface with the dm-crypt kernel module for creating, accessing and managing encrypted devices. Portworx uses the LUKS format of *dm-crypt* and *AES-256* as the cipher with *xts-plain64* as the cipher mode. The *dm-crypt* layer requires a passphrase to create the */dev/mapper* target.

A passphrase is passed down to *dm-crypt* during the volume attach operations. When attaching a volume, Portworx first attaches the virtual *pxd* block device. Portworx then invokes the *dm-crypt* module and create */dev/mapper* device link to the *pxd* block device. The detach operation first detaches the */dev/mapper* device followed by the *pxd* block device. Portworx does not manage passphrases for the volumes but it relies on an external Key Management Store.

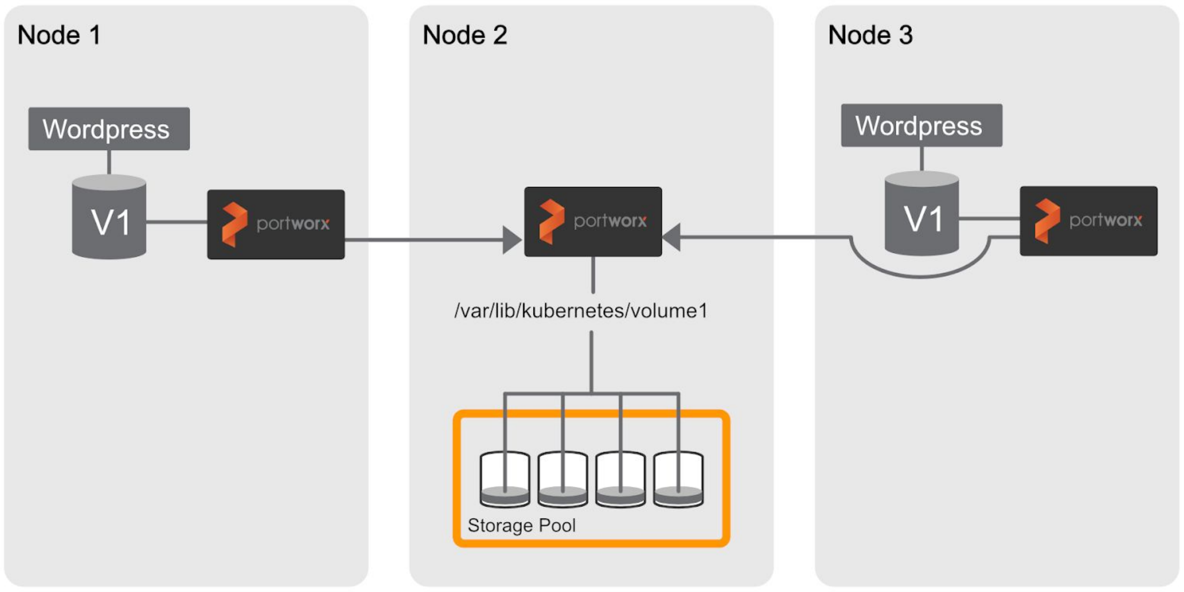
Key Management



Portworx uses envelope encryption. The encryption passphrase is retrieved from a Key Management Store (KMS) such as Hashicorp Vault, IBM Key Protect, AWS KMS.

Portworx is authenticated with KMS using the credentials provided by the user. The user creates a symmetric key pair with the KMS system. Portworx stores the public key from the key pair in its key value database and returns a unique identifier (Key) to it. When there is a need for accessing the encrypted volume, the user provides the unique identifier (Key) to Portworx. Portworx fetches the associated Public Key from its key value database. Using the Public Key, Portworx invokes the KMS APIs to fetch the DataKey which will then be used as the passphrase.

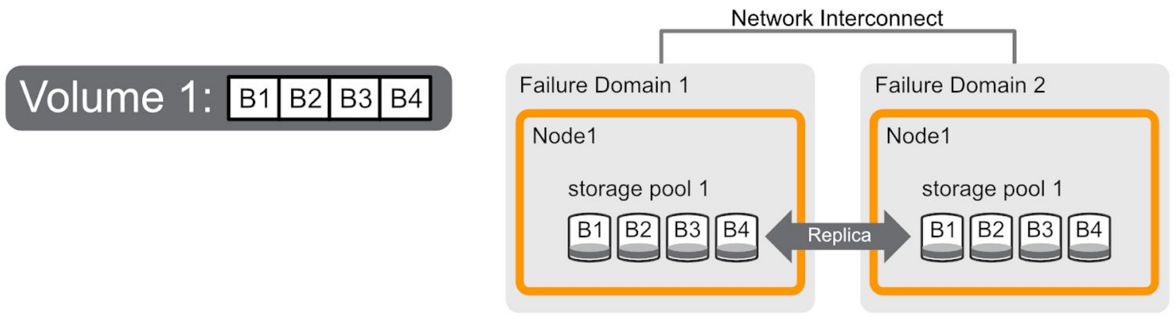
Shared Volumes



In contrast to block volumes, shared volumes allow concurrent access to a Portworx volume from multiple nodes at the same time. The example above shows wordpress containers accessing a shared file system that is mounted on a storage node that hosts data for the volume.

The principles of a shared volume are the same as that of a block volume. A block device is attached on one of the nodes that hosts a replica for the volume. User containers can run on any host in the cluster, several containers accessing the same volume can run concurrently across the cluster. All the I/O for user containers is routed to the Portworx node where the volume is attached. In the event of a failure of a Portworx replica, the clients seamlessly switch over to another replica.

Replication

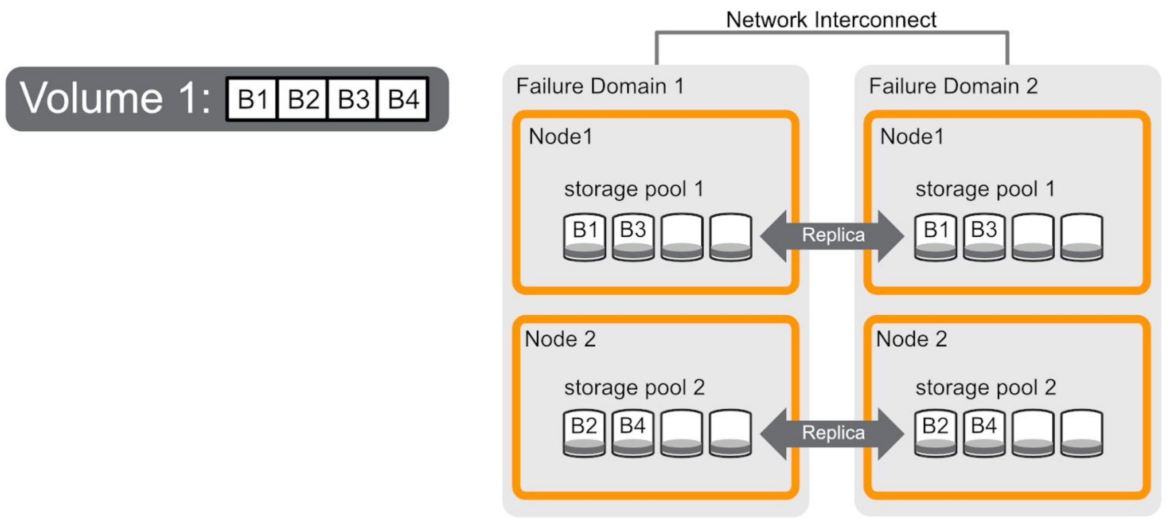


The set of nodes a block device is replicated to is defined by the replication sets. Each storage node has a complete copy of block device maps in memory. This allows it to perform I/O operations directly to nodes that own the data without going through an additional metadata server.

Portworx implements synchronous replication. By the time a write operation is acknowledged the data will be stable on a sufficient set of nodes such that the write is recoverable in the event a node goes down. For example, in two-node replication, the data is written to both nodes before the write is acknowledged. For three-node replication, the data is written to at least two nodes before the write is acknowledged.

Replication can occur in one of two ways: The transaction coordinator transmits I/Os to all the replicas (the transaction coordinator may itself host a replica) or the transaction coordinator transmits I/Os to one of the replicas and then the replica transmits to the other replicas. From the perspective of the durability of the I/O, both modes are equivalent. However, depending upon the network configuration, one mode may be more performant than the other. If the client network is a different network than the storage replication network, it may be desirable to restrict client traffic to one network and the replication traffic to another.

Aggregation



Portworx does not stripe a replica across all the nodes in the cluster but a small subset of nodes termed an aggregation set. This allows for both hyperconverged workloads as well as leveraging the combined horsepower of multiple nodes, avoiding hotspots, and limiting the blast radius in case of failure.

Hyperconverged deployments have 1 node in an aggregation set. The number of nodes in these deployments can be higher in certain situations: if a volume's required capacity exceeds a node's available storage, or in specific topologies permitting higher performance by splitting replicas across multiple nodes.

Recovery

To support crash recovery, each write operation has an associated timestamp which is stored to stable storage on each node in the replication set before the write is acknowledged by that node. During reads, we compare timestamps received from different nodes and return the data with the most recent timestamp to the application.

The timestamp includes the cluster database version. This ensures total order on the timestamps in the event of nodes going up and down and device ownership changes, since the cluster database version is changed whenever a device moves between the nodes.

To limit the amount of storage for the timestamps, the timestamps are retired when all the nodes in the replication set has completed the write. During normal operation, when all nodes are up, the active timestamps set will be very small and can be completely cached in memory. The timestamp cache spills over to disk in the event a node goes down and is missing I/Os. When the failed node comes back online, the missing I/O is replayed from the metadata recorded in the timestamp file.

Resync

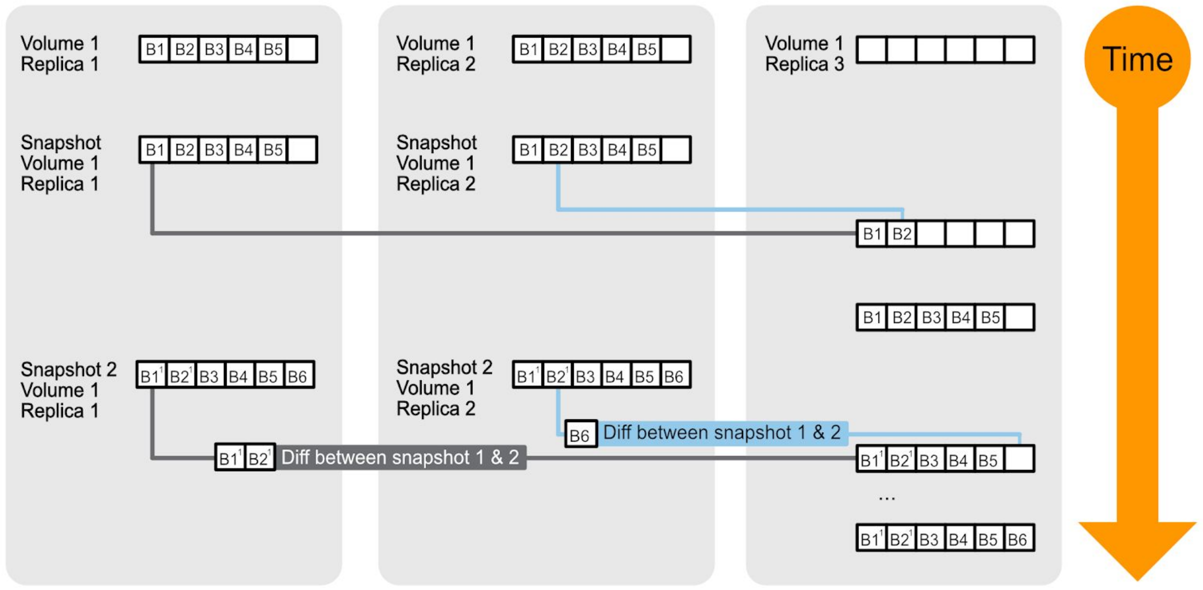
Resync is the process that reconciles replica states in the event a replica fails. Each write records a persistent timestamp entry as described in the replication section. The timestamp entries in each storage node are purged as soon as all the replicas acknowledge the write. In the event that a replica is down, the timestamp entries persist.

When the failed replica comes back online, the replicas engage in the resync process. The first step in the resync process is the exchange of timestamp entries. At the most basic level, for each block, the entry with the most recent version is chosen and this is propagated to the replica that is missing this entry. This process continues until all the timestamp entries for a volume have been resynced.

The collection of timestamp entries is assumed transient and the number of entries per volume is bounded by both time and size. Thresholds are set in place such that if the timestamp entries go past time or size then they are abandoned in favor of a Repl-Add process. The Repl-Add process is described in the Replication Level Modification section.

Note that the resync process occurs when the volume is online and that the replica that is being resynced ingests new writes while it is receiving the writes that it missed. This allows the replica to catch up to the current state of the replicas without diverging further.

Replication Level modification (HA/Repl Add/Reduce)


Volume 1 Replica 1
Volume 1 Replica 2
Volume 1 Replica 3
Time
Snapshot Volume 1 Replica 1
Snapshot Volume 1 Replica 2
Snapshot 2 Volume 1 Replica 1
Snapshot 2 Volume 1 Replica 2
Diff between snapshot 1 & 2
Diff between snapshot 1 & 2

Portworx allows you to increase and decrease the number of replicas of virtual volumes in the cluster. These are online operations and do not involve any disruption to user I/O.

Adding a new replica

Consider the following scenario: increasing a volume's replication factor from 2 to 3. The volume in this scenario has a simple replication set: *{{A}, {B}}* where one replica resides in *node A* and the other in *node B*.

1. The volume spec enters the provisioning system and an aggregation set is selected based on the criteria specified in the spec. In this scenario, the new replication set becomes *{{A}, {B}, {C}}.* Data now needs to be moved to *node C.*
2. A snapshot S1 is taken on existing replicas *{A}* and *{B}*. This snapshot is transferred to node *{C}* by iterating over the extents in the snapshots. Both *{A}* and *{B}* are sourced for the reads depending upon their IO loads. The transfers are checkpointed and if interrupted, they can resume from where they left off.
3. After *{S1}* is completely transferred to *{C}*, a second snapshot *{S2}* is taken.

4. The delta between *{S1}* and *{S2}* is then transferred to *{C}* by iterating over the diffs in the extents and reading data from *{A}* and *{B}* and writing to *{C}.*
5. The process in step 4 repeats until the delta between *{Si}* and *{Si+1}* is small enough (less than 100MB). At this point the new node *{C}* is added to the replication set and the system switches to timestamp based resync as described in the resync section.

Removing a replica

Consider the following scenario: decreasing a volume's replication factor from 3 to 2. The volume in this scenario has a simple *replication set: {{A}, {B}, {C}}* where one replica each resides in *node A*, *B*, and *C*.

1. The volume spec enters the provisioning system and an aggregation set is unselected based on disk spaced provisioned and used in the cluster. In this scenario, the new replication set becomes *{{A}, {B}}.*
2. The replica {C} is removed from the replication set and marked for garbage collection. The space consumed by the replica is returned to the free pool.

Snapshots

Snapshots are supported at the volume granularity. Snapshots are Redirect-On-Write (ROW) based. Both read-only and read-write snapshots are supported. Snapshot creation is near instantaneous and incurs little overhead. Diffs between snapshot and volume trees are supported, this allows calculation of space taken up by a snapshot.

Snapshots are crash consistent, the data in the page cache is flushed via a call to *fs_freeze*. This is followed by inserting a quiesce barrier for the volume and taking a snapshot across all replicas. This guarantees that snapshots are point-in-time consistent across all replicas. In the event that not all replicas are available when a snapshot is triggered, the snapshot is taken across replicas that are online.

Consistent groups for volumes is supported. This means that a snapshot can be taken across a set of volumes that are consistent across all volumes. This is done by first flushing data from the page cache and quiescing I/Os across all replicas of volumes that are part of a consistent group. A snapshot is then taken on the replicas, and the volume I/O is resumed.

Snapshots are by default read-only, they can also be read/write. Read-only snapshots can also be promoted to read/write. Restoring a volume (or a consistent volume group) from snapshots is also supported.

Snapshots of snapshots are supported. Snapshots can be deleted in any order. The parent volume can be deleted while preserving the snapshot. It is possible to see how many unique blocks a snapshot references and therefore how much space will be reclaimed if a snapshot is deleted, which can influence a lot of space management workflows.

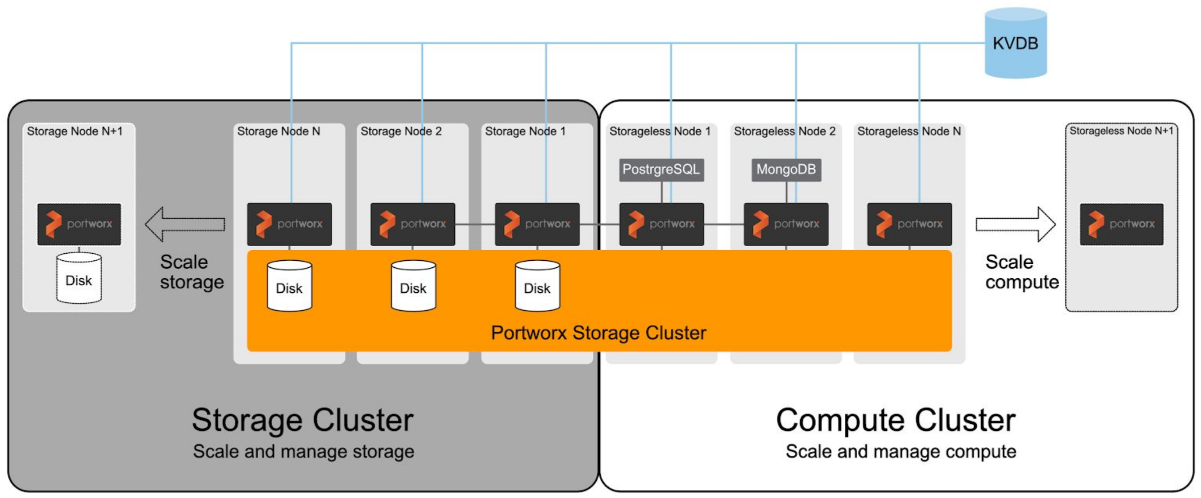
Storage Deployment Models

Portworx supports both the disaggregated deployment model or a hyperconverged model. In general, nodes in a cluster can contribute storage or be storageless. You can choose to run applications on nodes that contribute storage or not. The system does not require you to choose the mode of operation before hand and it is very much possible to mix and match modes (i.e have user applications run on a subset of storage nodes).

Further Reading

The Deployment architectures for Portworx article provides more information about the decision points on choosing one model over the other.

Disaggregated Deployment



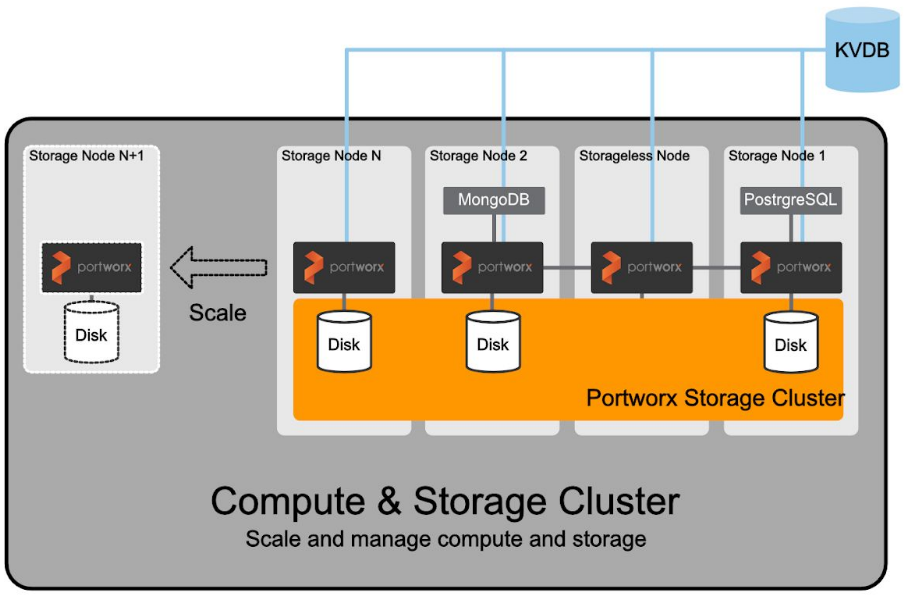
In the disaggregated model, user applications do not run on storage nodes. Typically, this is useful in cloud environments where instances are autoscaled up to a high number to account for bursts and then scaled back down. These operations are not well suited for storage nodes. The disaggregated model is also useful when server architectures are very different in the cluster and there are nodes that are CPU and memory intensive but do not offer any storage.

The principal consequences of a disaggregated storage model is two-fold:

- Portworx scales down resources that it consumes with the assumption that it shares the resources with user applications. In the disaggregated mode, the resource consumption can be that of the system in the storage nodes resulting in better performance.

- Client networks can be different than storage networks. It might be beneficial in these cases to have all replication traffic go over the storage network.

Hyperconverged Deployment



In the hyperconverged model, user applications run on storage nodes. This mode benefits from limiting traffic on the network when the application is scheduled on the same node where one of the replicas resides. Portworx resources are scaled back to share resources with client applications. Even in this mode, it is possible to choose if the replication traffic goes from client to replicas or just between replicas.

High Availability and Self Healing Cluster

Failures can occur at various levels: software, disks, node, power, and network. Some failures may be transient, others intermittent or permanent. Portworx enables a resilient, self healing cluster with the right controls for redundancy, monitoring and runtime detection of failures, alerting on failures, programmable hooks for actions, and auto-correcting when possible.

Data Integrity

Portworx ensures data integrity in the entire I/O path. Specifically, it protects against memory corruption, corruption over the network, and bit rot. Where possible, Portworx will seamlessly repair invaild blocks.

Data Integrity on Write

When data is written from the application to the block device, the blocks are checksummed. The data along with the checksums is transferred all along the I/O path, it is revalidated to protect against memory corruption and written to stable medium at along with the checksums.

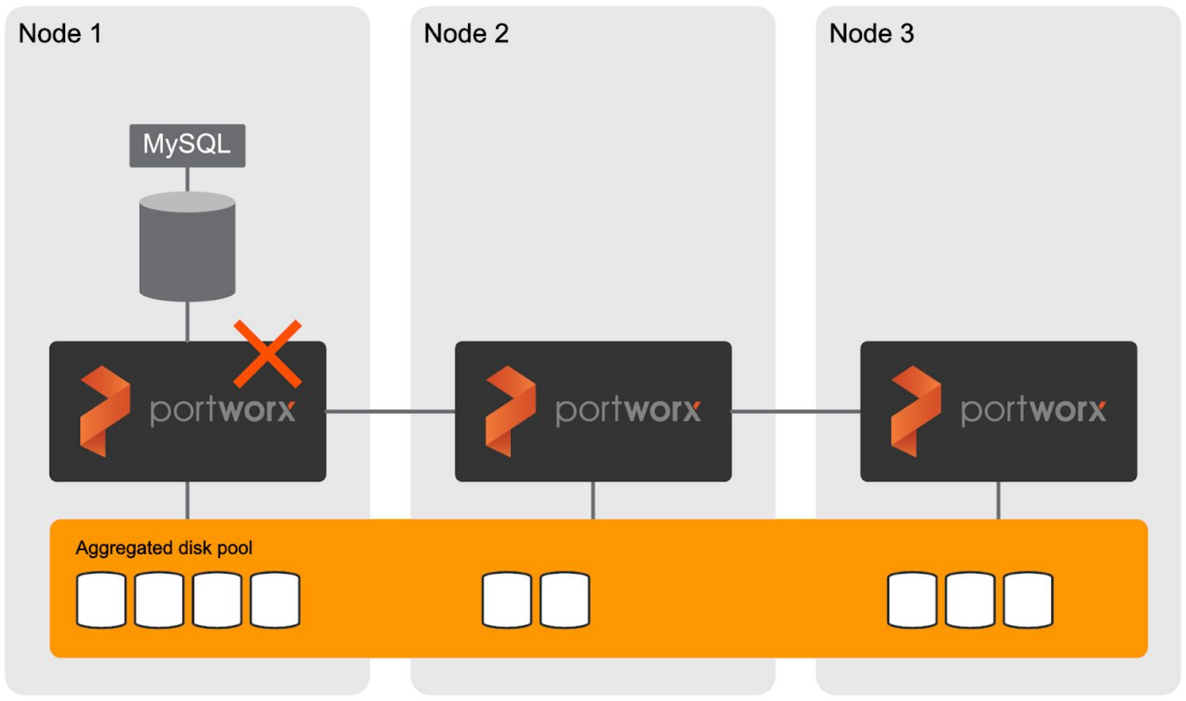
Data integrity and repair on Read

In the read path, the data is read along with checksums. If the checksum does not match, the block is fetched from a replica and re-written. The checksum is computed again before writing the response to the client to protect against memory or network corruption.

Background Scrub

In addition to detecting and repairing media errors on the fly, periodic background scrub can be run against volumes. These can publish reports, raise alerts, and repair errors in a similar manner as repairing on read.

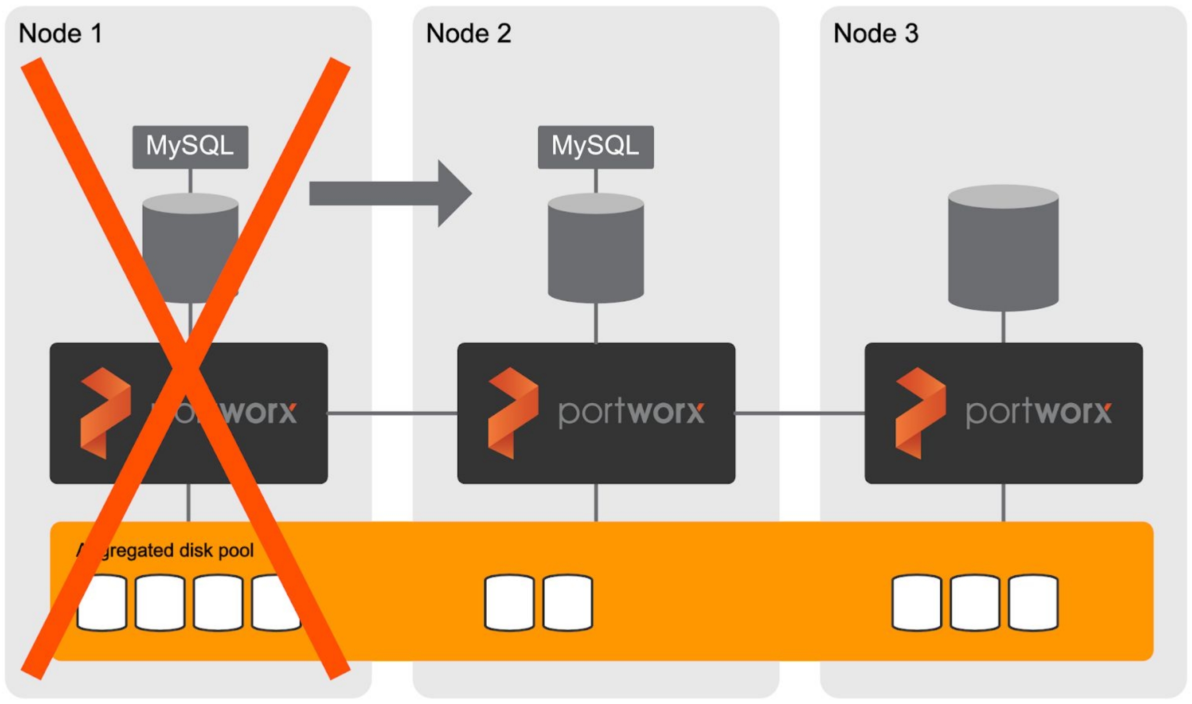
Software failures



A transient Portworx software failure is detected by another node in the cluster either via missing heartbeats in the control path (via gossip in approximately 20 seconds) or if there was any I/O in transit, the datapath detects the node failure (in approximately 2 seconds). An alert is raised in such an event. Locally an internal watchdog will restart any process that crashed.

A Portworx crash does not lead to I/O failures. For volumes that are attached on the node, these volumes continue to be attached, the I/Os are buffered in the kernel and unacknowledged writes are replayed. For volumes that are attached on a different node but the failed node holds a replica, the failed node is removed from the active replication set, the I/Os continue on the replicas. When the failed node comes back up, the I/Os are replayed via resync.

Recovery from Node failures

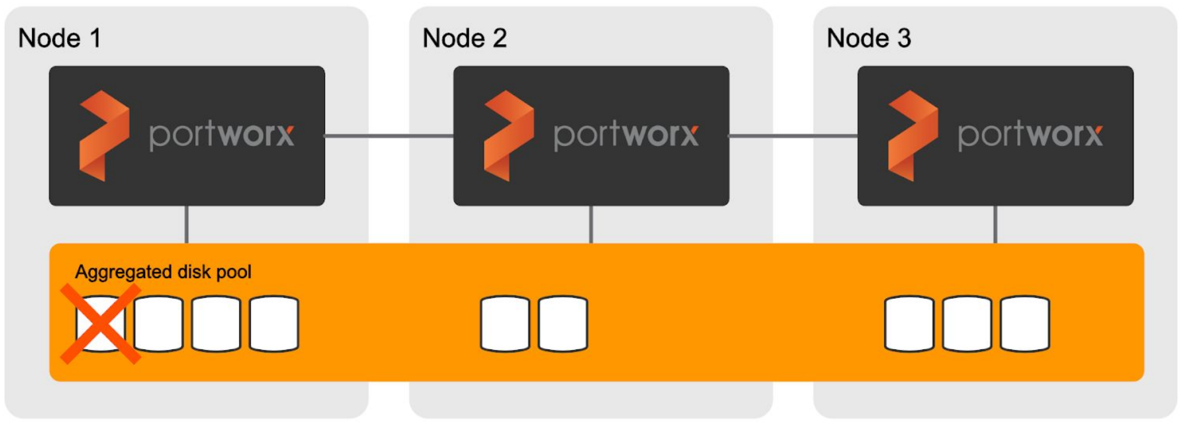


A node failure is detected exactly as though Portworx software failure. Failure detection as well as alerting is handled in the same way as though it was a Portworx software failure.

A node failure does not lead to IO failures. For volumes that are attached on the node, the user containers are also running on the same node and they die along with the node. During the period while the node is down, the user containers may be scheduled on a different node (depending on the amount of time the node is down and the thresholds set in the orchestration system). For volumes that are attached on a different node but the failed node holds a replica, the failed node is removed from the active replication set, the IOs continue on the replicas.When the failed node comes back up, Portworx is restarted via the orchestration system or *systemd*. The volumes that were attached on the node are reattached (unless the orchestration system moved them to a different node). The volumes that hosted replicas on the failed node get the data resynced.

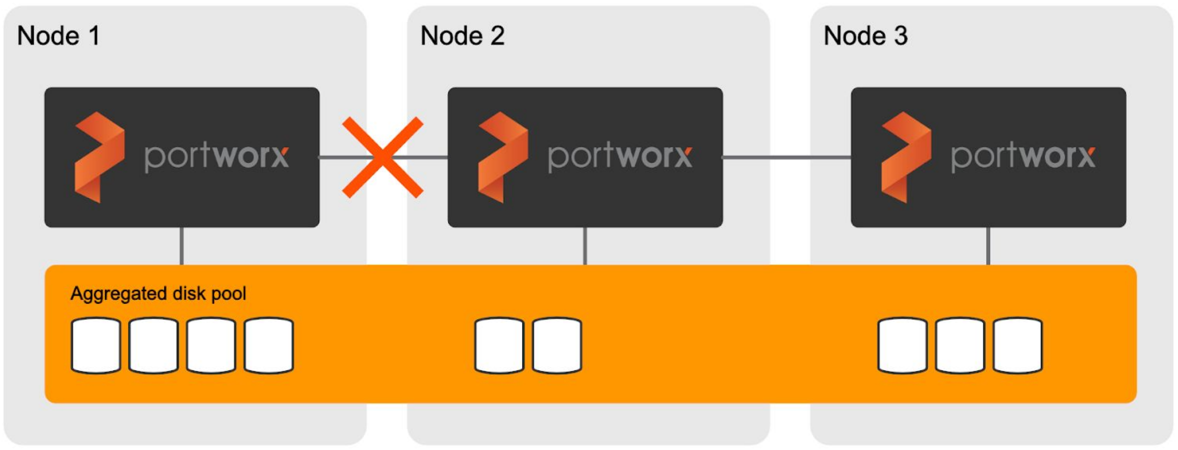
If the node fails permanently, then the recovery procedure varies. Portworx, by default, assumes that a failed node is going to recover and will not relocate data for replicas on the failed host. However, if the node fails to come back up within a configurable amount of time (24 hours by default), the replicas hosted on the node will be relocated to a different node. The provisioning algorithm is invoked to select new nodes as replacement for the failed node. The repl-add procedure is used to restore replica count for volumes that were hosted on the dead node. If, and when, the failed node comes back online, the replicas it hosted are wiped clean.

Disk failures



Smartctl tools are used for constantly monitoring disk errors. Alerts are raised when disks begin exhibiting a high error rate (Predictive Failure Analysis), failing disks should be replaced. In the event that a disk I/O fails continuously, alerts are raised, and the pool that the disk belongs to is marked offline. This is because disks in a pool are, by default, configured as RAID-0. If the disks are configured into RAID-10, the pool goes into degraded mode. If the pool stays down for a configurable period of time (24 hours by default), it is treated similarly to a node failure and the volumes located on the pool are migrated to a different node.

Network failures



Network failures come in many flavors:

- A local network device failure
- A virtual network software failure
- A rack backplane failure
- A switch failure

In all cases, a failure manifests itself as an interrupted network connection or an inability to connect. Alerts will be generated in both cases.

If there is a persistent network connectivity failure, then the node that is unable to communicate will be marked offline. The rules that apply to a node failure will apply here as well. If the failure is intermittent or connections are dropped intermittently then Portworx will re-establish network connections and retransmit messages that were not acknowledged. From the perspective of the end user traffic, this will be transparent except that alerts will be generated from the cluster.

Security and RBAC

Overview

From version 2.1, Portworx supports security. Portworx security centers around the ubiquitous JSON Web Token (JWT) based authentication and authorization model. This technology is currently used by most major internet systems, providing a proven secure model for user and account identification.

A token is generated by a token authority (TA) and signed using either a private key or a shared secret. Then, the user should provide the token to Portworx for identification. No passwords are ever sent to Portworx. This secure model enables Portworx to only have to verify the validity of the token to authenticate the user. Portworx then destroys the token, ensuring tokens are never saved on a Portworx system.

The token contains a section called claims which not only identifies the user but also provides authorization information in the form of role-based access control (RBAC). Portworx uses the RBAC information to determine if the user is authorized to make the request.

Token Authorization

Portworx security supports two types of token generation models: OpenID Connect (or OIDC) and self-generated tokens.

OIDC is a standard model for user authentication and management and is a great solution for enterprise customers due to its integration with SAML 2.0, Active Directory, and/or LDAP.

The second model is a self-generated token, where the administrator generates a token using their own TA application, like a command line tool, for example. For convenience, Portworx provides a method of generating tokens using *pxctl*.

For Portworx to verify the tokens are valid, they must be signed with one of the following:

- Shared secret
- An RSA private key
- An ECDSA private key

Portworx would validate tokens using the respective shared secret or public key.

Security Models

Portworx security is composed of three models:

- Authentication: A model for verifying the token is correct and generated from a trusted issuer.
- Authorization: A model for verifying access to a particular request type according to the role or roles applied to the user.
- Ownership: A model for determining access to resources.

Authentication

Portworx will determine the validity of a user through a token-based model. The token will be created by the TA and will contain information about the user in the claims section. When Portworx receives a request from the user, it will check the token validity by verifying its signature, using either a shared secret or public key provided during configuration.

Authorization

Once the token has been determined to be valid, Portworx then checks if the user is authorized to make the request. The *roles* value in the claims section of token must contain the name of an existing default or customer registered role in the Portworx system. A role is the name given to a set of RBAC rules which enable access to certain SDK calls.

Ownership

Ownership is the model used for resource control. The model is composed of the owner and a list of groups and collaborators with access to the resource. Groups and collaborators can also have their access to a resource constrained by their access type. The following table defines the three access types supported:

Type	Description
Read	Has access to view or copy the resource. Cannot affect or mutate the resource.
Write	Has *Read* access plus permission to change the resource.
Admin	Has *Write* access plus the ability to delete the resource.

For example, *user1* could create a volume and give Read access to *group1*. This means that only *user1* can mount the volume. However, *group1* can clone the volume. When a volume is cloned, it is owned by the user who made the request.

Administrator Role

Portworx uses the concept of a system administrator, a role similar to the *root* user in a Linux system. This role is not constrained by RBAC and participates in every group. The built-in RBAC role for system administrators is called *system.admin* and gives the caller access to every API call. To have access to every resource, a user must be in group *, which means they are part of every group.

Monitoring, Logging, and Tracing

Portworx exports a rich set of metrics available over a REST endpoint and exported to Prometheus. These are documented here. These come with Grafana templates and offer detailed insights on storage I/O in the cluster.

In addition to metrics, Portworx uses LTTNG to trace I/O requests. Using Portworx tools and traces, it is possible to plot the path of an I/O across the cluster. It is also possible to plot the I/O patterns of an application as well as pinpoint potential bottlenecks in the cluster.

Definitions and Key Concepts

KVDB refers to key-value database (*etcd*/*consul*). This can be hosted externally or Portworx can be instructed to run KVDB internally.

Gossip refers to a control plane protocol between nodes to periodically exchange heartbeats and node specific information.

Storage nodes are nodes that were provided backing storage. These contribute storage to the aggregated storage pool.

Storageless nodes or Compute nodes do not have any backing storage but can access storage from the Portworx cluster.

Hyperconverged deployments run compute and provide storage from the same nodes.

Disaggregated deployments refer to deployments where storage nodes do not run any applications. These nodes usually have a different hardware profile from compute nodes.

Replication level is the number of copies of volume data in a cluster.

Aggregation level is equivalent to *stripe width* across cluster. This determines the number of nodes across which a replica is distributed.

Aggregation set is a set of nodes across which data for a replica is distributed. The minimum number of nodes in an aggregation set is 1.

Replication set is a set of *Aggregation sets* across which data is mirrored.

Transaction Coordinator for a volume is the node where the volume is attached. All IO for the volume transits via the transaction coordinator.

Journal is an entity that resides at a storage node log that is used to recover local node consistency.

Timestamp is a per node data structure that tracks incomplete writes on a *replication set* in a volume.

Resync is a per volume operation that replays data to bring replicas in sync.

HA-add also called Repl-add is a per volume operation increases the replication factor of a volume.

HA-reduce also called Repl-reduce is a per volume operation reduces the replication factor of a volume.