CNCF Storage Landscape

White Paper

By Alex Chircop, Quinton Hoole, Clinton Kitson, Xiang Li, Luis Pabón, Xing Yang

Public link to this document: http://bit.ly/cncf-storage-whitepaper

Status: 12/18/2018 - draft has been publicly released and discussed at both Kubecon Shangai & Seattle.

1 Scope of this document

This is the first phase of documenting the storage landscape. It aims to offer clear information on terminology, usage patterns and classes of technology as defined by the goals of the document.

During phase two we might tackle the non-goals, on the basis of feedback from phase one, specifically in light of understood production use, and comparisons w.r.t. primary properties.

1.1 Goals

1. **Clarify the terminology** currently in use in the storage space, and the relationships between the various terms. Essentially a taxonomy of the storage landscape.
2. This to include anything reasonably within scope of “storage”, including block stores, key value stores, databases[1], object stores, volumes, file systems etc.
3. Provide some general information as to **how these things are currently being used in production** in public or private cloud environments.
4. **Compare and contrast** the various technology areas w.r.t. the primary properties of availability, scalability, consistency, durability, performance, API, etc.

1.2 Non-goals

1. Define what’s in-scope and out of scope for the CNCF.
2. Provide any recommendations regarding preferred storage approaches or solutions.

[1] It was decided to defer a discussion of databases to a later version of this document, given the broad scope of that topic.

2 Introduction and document layout

Multiple options were considered when defining how to present the many storage systems and services in the landscape for the document.

In order to simplify the consumption of information in a complex landscape, the document has been structured as follows:

- Definition of the **attributes** of a storage system such that an end-user can understand the appropriate capabilities that might be required by an application or architectural pattern
- Definition of the **layers** in a storage solution (or service) with a focus on **terminology** and how they impact the defined **attributes** - covering the container, orchestrator, transport, topology, virtual/physical, data protection, data services and the non-volatile layers.
- Definition of the **data access interfaces** in terms of **volume** (including block, file system and shared file system) and **application API** (including object, KV and database) as high level groupings
- Separate sections with further detail on **Block** Storage, **File systems**, **Object Storage** and **Key Value** Stores
- Definition of the **management interfaces** needed to orchestrate the storage layers to facilitate composability, dynamic provisioning and self service management.

3 Attributes of a storage interface or system

Storage systems and services have a variety of interfaces which are suitable for different use cases and tend to be composed of multiple layers which each impact different attributes of the system.

When choosing an overall storage solution, the different attributes of the desired solution need to be considered.

It is important to note that different storage systems are built with different design objectives, and may be architected to optimise for one or more storage attribute which may in turn impact another storage attribute.

3.1 Availability

Availability of a storage system defines the ability to access the data during failure conditions. The failures may be due to failures in the storage media, transport, controller or any other component in the system.

Availability defines how access to the data continues during a failure condition and also how access to the data is re-routed (or failed-over) to another access node in the event that the node that is accessing the data is unavailable.

The availability attribute can sometimes be referred to as a Recovery Time Objective (RTO) after a failure i.e. the time between a failure occurring and service being recovered.

Availability can be measured in Uptime as a % of availability (e.g. 99.9% uptime) as well as MTTF (mean time to failure) or MTTR (mean time to repair) which are measured in units of time.

3.2 Scalability

Scalability of a storage system can be measured by a number of criteria. Different criteria may be important for different use cases and each define a set of architectural patterns that will need to be implemented in a storage system.

Criteria used to measure scalability include :

A. the ability to scale the number of clients that can access the storage system

B. the ability to scale throughput (e.g. MB/sec) or number of operations (e.g. per second) of a single interface
C. the ability to scale the capacity, in terms of data stored, of a single deployment of the storage system/service. This could be with respect to storage volume (GB/TB/PB) and/or number of individual items.
D. ability to scale the number of components in a storage system to facilitate (a), (b), or (c)

3.3 Performance

Similar to scalability, the performance of a storage system can be measured against different criteria, the relative importance of each depending on the use case.

Performance of a storage system is typically measured in terms of one or more of:

- latency to perform a storage operation
- the number of storage operations that are possible per second
- the throughput of data that can be stored or retrieved per second

3.4 Consistency

Consistency attributes of a storage system refer to the ability to access newly created data or updates to the same after it has been committed and applies to both:

- “read” operations returning the correct data after a “write”, “update” or “delete” - with or without a delay.
- any delays that occur between performing the data storage operation and the data getting committed to a non-volatile store or being fully protected.

Systems that have delays between read operations returning up-to-date data, and/or delays before all data is protected after getting commited are defined as being “eventually consistent”. If there are no delays, the system is defined as being “strongly consistent”. Consistency is discussed in further detail in the Appendix

The consistency attribute can sometimes be referred to as a Recovery Point Objective (RPO) after a failure i.e. the amount of tolerated data loss (usually measured in time, based on the consistency delay) when a component or service in the storage system has suffered a failure.

3.5 Durability

Durability covers the attributes of a storage system that impact the ability for a data set to endure as opposed to just being accessible. Multiple factors can impact the durability of a storage system, including:

- the data protection layers, such as how many copies of the data are available
- the levels of redundancy of the system
- the endurance characteristics of the storage media that is holding the data (e.g. SSD vs spinning disks vs tape)
- the ability to detect corruption of data (e.g. due to component failure or wear/usage) and the ability to use data protection functions to rebuild or recover the corrupted data (sometimes referred to as “bit-rot”)

3.6 Instantiation & Deployment

A storage system can be deployed or instantiated on-premises or in a cloud environment in a variety of ways which defines where the storage solution or service can be deployed and/or consumed:

Hardware: deployed as hardware solution in a datacenter. This limits the portability of the application and generally means that such systems cannot be deployed in a public cloud environment
Software: deployed as software components on commodity hardware, appliances or cloud instances. Software solutions tend to be more platform agnostic and can be installed both on-premises as well as cloud environments. Some software defined storage systems can also be deployed as a container and deployment can be automated by an orchestrator.
Cloud Services: consumed from public cloud providers. Cloud services provide storage services in cloud environments.

4 Storage stack / layers

Any storage solution is composed of multiple layers of functionality that define how data is stored, retrieved, protected and interacts with an application, orchestrator and/or operating system. Each of these layers has the potential to influence and impact one or more of the attributes of a storage solution including availability, scalability, consistency and durability.

4.1 Storage Topology

The storage topology of a storage system defines the different arrangements of storage and compute devices and the data links between them. The topology can influence multiple attributes, including :

- Availability - in terms of the speed of failover and reconvergence following a component failure
- Performance - in terms of both latency and throughput
- Scalability - different topologies are optimised to scale in different directions (e.g. scaling vertically vs horizontally, sometimes referred to as scaling up vs out)
- Consistency and Durability - the topology often defines the consistency delay as well as the data protection options that are possible

4.1.1 Centralised

Storage systems that are deployed in a centralised topology tend to be formed of fewer nodes that maintain a tightly coupled state. Often the architecture is dependent on vendor specific hardware technology for intra-controller communication, configuration and dataplane activity (such as shared memory, cache synchronisation or shared data buses).

This type of storage is typically accessed by compute nodes via network interfaces where a number of clients consume storage from a small number of centralised nodes. Centralised storage is often characterised by scale up topologies (or scaling vertically) and is usually more consistent than distributed storage.

As a result of the small number of nodes (often just a single pair), the latency required to maintain data protection and sync consistency is very low and many block based systems use this architecture as a result.

It can be hard to scale such a system horizontally as the requirement for a tightly coupled state limits the number of nodes that can be supported.

4.1.2 Distributed

Storage systems that use a distributed topology tend to have a stronger focus on software solutions vs hardware solutions. A software solution will often be implemented with a "shared nothing" architecture where data needs to be synced across more that one node over a standard network connection.

Some distributed solutions are accessed directly in a scale out manner which allows many clients to access many server nodes in parallel. Other distributed storage systems layer other protocols on top to enable compatibility with existing environments or access transports (e.g. NFS or iSCSI) which may limit the overall scalability.

Different distributed architectures have different focuses and make design decisions that may favour performance, scalability, durability, availability or consistency. Distributed topologies typically offer better horizontal scalability as data can be distributed across many more nodes and can support many clients. This can result in systems that are also more complex to deploy and operate and therefore benefit from additional automation.

4.1.3 Sharded

Sharding is a process where a dataset or workload is partitioned based on ranges of keys across multiple instances. The shard can be computed by using the key to determine which node to access based on a range, a hash or other algorithms. Sharding is primarily used as a method for scaling database architectures.

Sharded systems provide a way to scale a storage system for both capacity and compute capability. Workload is distributed across the shards in the system allowing workloads to scale horizontally.

Sharded systems can increase operational complexity and care needs to be taken to ensure that the algorithm used to distribute the keys is balanced to the specific workload or dataset. Managing availability can also be more challenging as systems may experience more complex or partial failure modes where only parts of a data set are impacted by individual node or network failures. Although sharding enables scale, the performance of any particular request will be limited to the performance of the specific node that the shard is located on, and it is possible for individual shards to become "hot" or overloaded. Rebalancing shards when scaling a cluster can also be complex.

4.1.4 Hyperconverged

Hyperconverged topologies combine application as well as storage workloads onto the same nodes. Multiple nodes can be clustered together creating a common resource pool which is shared for both computer workloads and storage functionality. In hyperconverged

topologies, the storage layer is usually implemented as a software component on commodity compute nodes and typically shares the same attributes of a distributed system.

Hyperconverged topologies tend to be selected to maximise flexibility as the storage system can grow with the compute workload.

Hyperconverged systems can increase operational complexity as maintenance operations, or any node failure, not only impacts the workload on that node, but also the underlying storage system.

4.2 Data Protection

A key function of any storage system is to provide protection of the data that is being persisted in the system or service. This is often implemented as a transparent layer in the system.

4.2.1 RAID: Striping, Mirrors & Parity

RAID (redundant array of independent disks) uses techniques such as striping, mirroring and parity to distribute and provide redundancy for data across a set of disks:

> **Striping**: this is a process where data is spread evenly across 2 or more disks. Striping in itself does not provide redundancy or fault tolerance - in fact, striping on it's own increases the chance of failure as a failure of any of the individual disks will typically result in unavailability of the whole dataset. Instead striping is used to increase performance of a number of data protection functions by distributing the load across more disks such that a workload is not limited to the performance of a single component.
>
> **Mirrors**: a mirror maintains an identical copy of a dataset across two disks. This configuration enables the availability of data to continue in the event of a disk or component failure. It is also possible to mirror multiple disks for additional redundancy.
>
> **Parity**: when using parity, a data set is distributed across a number of disks that are grouped together. For each unit of data (typically a block, but can be as small as a byte), an algorithm is used to generate an additional set of parity data which is stored alongside the data. In the event of the failure of any individual disk, then the missing data can be regenerated using the remaining data segments and the parity data. The benefit of using parity over mirrors is that parity does not require a full copy of the data set and can therefore implement data protection with less overhead in terms of disks or backend storage capacity. The capacity benefit comes at the expense of performance overhead and using parity for data protection can impact latency and throughput.

There are four main RAID levels in common use today:

> **RAID0**: this uses a simple stripe data set and is typically only used when the only consideration is performance, as RAID0 datasets do not have any redundancy.
>
> **RAID1**: a RAID1 dataset consists of mirror. In a RAID1 dataset, read performance can be increased as the reads can be striped across both sets of the mirror, but writes will only be as fast as an individual disk as the write needs to be written to both disks in parallel. Any data set will also consume double the capacity on the disks as a result.

RAID5: this implements a distributed dataset with distributed parity. Each block is distributed across the disks in a RAID5 set together with the additional parity. This method provides a good balance between capacity utilisation and redundancy: the parity ensures that data can be recovered or rebuilt if any single disk fails, but as data is not mirrored, the capacity lost to redundancy is only 1/x (where x is the number of disks in the raid set). Performance of read operations is similar to a striped dataset and can utilise the combined speed of all the disks in the dataset, but write performance has a high penalty: every write or update needs to touch every disk in the RAID set. A RAID5 set can only survive a single disk failure and care must be taken to ensure that a rebuild of the data is completed before a second failure occurs.

RAID6: RAID6 is also a distributed dataset with distributed parity with the difference that two sets of parity is generated. This allows for two concurrent disk failures to occur in a RAID6 set without impacting the availability of data. RAID6 has similar disk performance characteristics of RAID5, but imposes a higher CPU workload - this is due to the calculation of the second parity set.

RAID sets can be striped to further spread the data across many more disks for improved performance. This is sometimes referred to as nested RAID, but more often is determined by adding a “0” to each of the RAID levels e.g. RAID10, RAID50 and RAID60 referred to stripes of RAID1, RAID5 and RAID6 respectively.

Using multiple copies of parity (such as RAID6) has become more important as the size of disks continues to grow, as the size of the disk tends to determine the time to rebuild a dataset. As a result, custom additional parity based RAID sets have been defined in some solutions (e.g. RAIDZ in ZFS) to add 3 or more sets of parity.

Although RAID is typically implemented within the set of disks in a specific node, it is also possible to distribute RAID across a network and implement redundancy across nodes AND disks at the same time. This is a technique used in some distributed storage systems.

4.2.2 Erasure Coding

Erasure coding is a method used to protect data where a data set or object is split into multiple fragments that are then encoded and stored with a configurable number of redundant parity sets. As an example a data object might be broken down into 6 data fragments and 4 parity fragments and would be referred to as (6+4) erasure coding. The ability to have many parity fragments enables very high redundancy and very high durability.

Each of the fragments can be distributed across different disks and servers/nodes in multiple locations. Erasure coding typically uses Reed-Solomon codes to perform encoding and is therefore a computationally intensive process. The primary benefit of using erasure codes is the flexibility of a user configurable balance between data distribution, capacity utilisation

and redundancy. As a result, erasure coding is utilised in many distributed storage systems and the primary building block for data protection and redundancy for many object stores.

One drawback of erasure coding is that the number of data fragments and the distribution across multiple nodes means that write and read operations on data objects can incur significant latency due to the network overhead as well as the computational overhead. As a result, erasure coding is best applied to large datasets which are optimising for either reducing overall capacity utilisation or improving redundancy and durability.

4.2.3 Replicas

Replicas are mirrored data sets that are distributed across multiple servers/nodes. A replica is a full copy of the dataset and therefore the number of replicas for a data set multiplies the capacity needed to store a particular data set. Each individual replica is useable as a standalone copy and therefore rebuild operations are extremely quick as it is both simple and can be implemented as a point to point transaction.

Replicas have a much lower compute and network distribution overhead and are therefore preferred when lower latency is important.

4.3 Data Services

Storage systems often implement a number of data services which complement the core storage function by providing additional functionality that may be implemented at different layers of the stack.

4.3.1 Replication

This service provides the capability to replicate a set of data (e.g. a volume or a bucket) to improve the availability and durability of the data. Note - this data service is often separate from the core data protection function (such as mirrors or replicas) and is generally used to replicate data between independent storage systems often in different locations.

Replication can be performed synchronously where a request to persist data is only acknowledged to the application after the replica target has also acknowledged it. This provides a strongly consistent model with a low time to recover from failure, but can impact latency and performance. Due to the time taken for data to traverse a network, latency increases with distance, and synchronous replication is typically only feasible when the source and target systems are within 100km of each other.

Replication can also be performed asynchronously where data to be replicated is queued and is transferred to the target replica out of band of the actual storage persist operations. This means that asynchronous replication is eventually consistent and has a lower impact to overall performance. Asynchronous replication can support replication over long distances but adequate bandwidth must be available to be able to transfer the deltas that change between the source and target system in an acceptable time frame.

4.3.2 Snapshots and Point in Time (PIT) copies

Snapshots or point in time copies of data improve the availability of a dataset and provide the capability to backup and further protect the data. A snapshot is a view of the data set at a given point in time (when the snapshot was taken) and this provides the ability to access this data consistently at that data point.

Snapshots can be implemented in a space efficient manner using techniques such as “copy-on-write” (COW), which provides a virtualisation layer where snapshots only contain the delta between what was written since the snapshot was taken and the original data set. This provides the capability of taking multiple snapshots at different intervals whilst minimising the amount of capacity needed to store the snapshots.

Many storage systems also allow the creation of a point in time copy of the data which includes a full copy of the data set. This is often referred to as a “clone” and utilises

additional capacity in the storage system, but creates an independent copy of the data set. This can be useful when the data set is to be utilised in a manner which might impact the performance of the original data set if a snapshot was taken.

Processing snapshots and data copies often means maintaining data structures and metadata which may impact the CPU, memory or disk overhead and performance. Whilst the creation of space efficient snapshots is often a low overhead function, the creation of a clone requires the creation of a full copy of the data set which will impact performance and utilise bandwidth to move the data from the original data set to the copy.

4.4 Encryption

Storage systems can provide methods ensure that data is protected by encrypting data. Data encryption can be implemented for data in transit or data at rest and can ensure that the encryption function is implemented independently to the application.

Encryption can have an impact on performance as it implies a compute overhead, but acceleration options are available on many systems which can reduce the overhead.

The encryption function will often depend on integration with a key management system which may add complexity to a storage system.

4.5 Physical / Non-Volatile Layer – terminology

Storage systems will ultimately persist data on some form of physical storage layer which is generally non-volatile. The choice of the physical layer impacts the overall performance of the storage system and defines the long term durability of the stored dataset.

Cloud services often use similar terminology for service classes to define the performance characteristics and SLAs of the service.

Some of the most commonly used systems include:

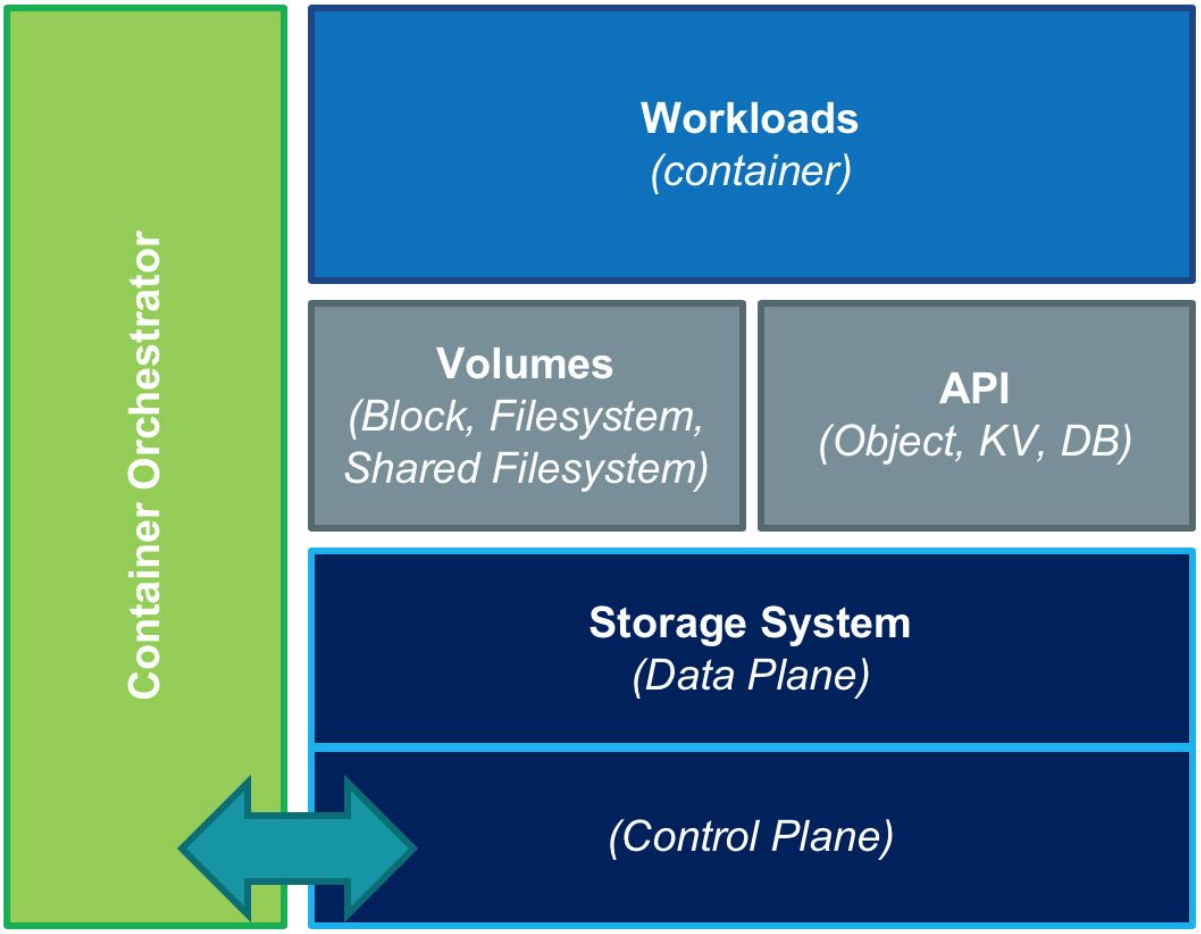
- **Spinning / magnetic disk (e.g. SATA, SAS & SCSI)** - magnetic media are traditional harddisks and are mechanical devices in that they have spinning magnetic disks that are read by a read/write head. Latency is a combination of the rotational latency of the disk, the seek time for the head to move into place to read/write the data and the electronics/bus. SATA, SAS and SCSI are transports used by the operating system to access the device through a host bus adapter (HBA). Latency per operation is measured in a number of milliseconds and throughput is generally under 250MB/sec. Magnetic media generally offers the lowest cost per GB of capacity.

- **SSD (with traditional interfaces such as SATA, SAS or SCSI)** - a solid state disk does not have any moving parts and stores data in non-volatile memory (typically some type of flash). This allows for much lower latency operations - typically small

fractions of a millisecond and allow for tens of thousands of I/O operations per second. Throughput is usually limited only to the transport utilised and measured in hundreds of MB per second. Different classes of flash are available which impact the performance as well as the durability - SSD flash wears out and can fail after a given number of cell overwrites. Storage systems that are optimised for SSDs will therefore generally attempt to minimise write amplification to minimise wear.

- **Non Volatile Memory (e.g. SSD/NVMe)** - flash based devices are generally faster than the current generation of transports. NVMe is a faster transport that minimises the protocol overhead by treating the flash more like memory where data can be accessed randomly rather than in block format as defined in disk transport protocols like SCSI. This allows for much lower latency - typically a few tens of microseconds - and much faster throughput - typically measured in GB per second.

5 Data Access Interface



The data access interface defines how applications or workloads store or consume data that is persisted by the storage system or service.

The interface is an important factor in the choice of a storage solution as often, different workloads or applications will have a pre-defined or preferred access method.

Different interfaces also influence a number of attributes such as:

- availability – in terms of failover and moving access between nodes
- performance – in terms of latency and throughput
- scalability - in terms of the number of clients that can access a given pool of storage

In addition to the attributes, in practice, the choice of access interface has a large impact on the management interfaces available and therefore the ability of orchestrators to manage and provision storage. In particular, Volume interfaces currently have more mature integrations with orchestrators.

5.1 Data Access Interface: Volumes

5.1.1 Block

A block device is the fundamental building block of many volumes. A disk device is represented as a block device to an operating system and represents a contiguous set of blocks that are ultimately stored in the disk (or other non-volatile storage). Blocks are typically represented as a 4KB unit of data to the operating system, although different disk systems may actually store blocks internally in either smaller or larger units. Read and write operations are performed in units of individual blocks.

A block device can be a representation of a local disk but can also be a representation of a virtual or remote disk that is either connected to or provided by a storage system.

Block devices are rarely consumed by applications directly and are often used as a device that underlies a filesystem. Some databases can be configured to consume raw block devices directly in order to improve performance. Permissions and access control of block devices are typically reserved to admin users of the operating system.

Further details are available in this section.

5.1.2 Filesystem

A filesystem defines how data is persisted and retrieved by the operating system, by structuring the data in terms of files and directories. A filesystem will often use a block device to persist the data to a non-volatile storage medium such as a disks

Permission attributes in a filesystem can be allocated to both files and directories, allowing granular access to users and groups, as well as defining the type of access (e.g read, write or execute access). Some filesystems support more extended attributes that improve the flexibility and levels of control and access. Filesystems can also support locking semantics that allow an application to mark a file as locked for exclusive use. The supported locking capabilities vary between filesystems and may operate differently when used on a remote or distributed filesystem.

Filesystem code is typically run within the kernel of the operating system to maximise performance, which means that the filesystems available to an application will be dependent on the particular operating system distribution. It is also possible to run filesystems at the user level (FUSE), which are often used to provide a filesystem representation of datasets other than those stored in a native block device.

Further details are available in this section.

5.1.3 Shared Filesystem

Filesystems are typically limited to an individual server or node and can therefore only be accessed by one node at a time. A shared filesystem is a filesystem that can be mounted on more than one node at a time. This provides additional flexibility and supports patterns where applications are distributed between multiple servers and need to access a common set of data.

A shared filesystem can be consumed from a point-to-point service endpoint, where a server node exposes a local filesystem to other servers - this is limited to the performance (and sometimes the availability) of a single node. Alternatively, a shared filesystem can be distributed across multiple nodes and systems in a distributed filesystem - this allows for datasets and scalability beyond what can be supported on a single node.

Clustered filesystems can provide similar functionality to shared filesystems but are rarely utilised in a cloud native context and they use shared block devices which are available on multiple nodes.

5.2 Data Access Interface: Application API

5.2.1 Object Stores

Object stores use an API to store and retrieve objects or blobs. The API for the most popular object stores utilise a HTTP interface. Object stores are typically based on a distributed architecture that is optimised for capacity, durability and scalability allowing thousands of clients to connect to PB buckets of storage.
The overhead required to commit multiple copies for availability and durability, and the use of a HTTP API tends to lead to a higher latency overhead per operation, but can maintain high levels of throughput through parallel access from multiple clients.

Further details are available in this section.

5.2.2 Key Value Stores

A key value store is accessed by an API and use a key as an identifier to store and retrieve values from the store. Key value stores can be implemented in a library, a local system or a distributed system.

Key value stores are often used to store metadata and configuration and are often implemented with strong consistency. As a result they are often utilised as a method for storing state, configuration and indexes for distributed systems and applications.

Further details are available in this section.

5.2.3 Databases

Databases are outside of the scope of the initial draft of this document.

5.3 Orchestrator, host and operating system level interactions

A number of virtualization and access layers are often overlaid or interposed on a Data Access Interface as part of the integration of the storage solution into an orchestrated environment, and can influence Availability, Scalability and Performance of the overall end to end solution.

Often a hypervisor may also be providing access to resources and may be performing a variety of functions including mapping storage resources, pooling multiple resources which are shared between workloads, managing connectivity to resources and handling failover and data protection functions.

5.3.1 Volumes

Some interactions that may apply to volume access interfaces include :

- A volume manager (e.g. lvm) which may provide functionality to pool resources, provide data protection and even take an active role in failover and recovery
- Bind mounts and overlay filesystems which provide functionality to layer filesystems and image layers to provide integration with orchestrators and container runtimes.

5.3.2 Application API

Some interactions that may apply to application API interfaces include :

- Discovery to provide functionality to identify resources in a cluster or a network
- Meshes, ingress end-points and load balancers that can provide functionality to route requests to store and retrieve data based on content or resource availability

5.4 Comparison between Object Stores, File Systems and Block Stores

Data Access Interface	Most suited	Least suited
Block	• Availability • Low latency performance • Good throughput performance for individual workloads	• Capacity scaling • Sharing data with multiple workloads simultaneously
Filesystem	• Sharing data with multiple workloads simultaneously • Optimised throughput for aggregated workloads	• Strong file locking integrity when filesystems are shared
Object Store	• Availability • Large capacities (PB scale) • Durability • Sharing data with multiple workloads simultaneously • Optimised throughput for parallelised workloads	• Low Latency performance

*** The information in this table are generally accepted attributes and measurements for object stores, file systems and block stores.*

5.5 Comparison between Local, Remote and Distributed Systems

	Local	Remote	Distributed
Availability	Limited by failure of components locally and ability to failover. If a node fails, the local storage is isolated to the local node.	May be limited by single points of failure. Workloads can move to another node and reconnect to the remote storage.	Clients may access numerous nodes, and any storage node failures can be mitigated. The additional complexity of distributed systems may add operational complexity which may in turn affect availability or the ability to recover errors.
Scalability	Limited by local architecture (1 node; typically TB)	Limited by monolithic architecture (2-16 nodes; typically 10s-100s of TB)	Scale by adding additional systems. Topology enables scale of both nodes and supported capacities. (3-1000s nodes; often supports PB)
Consistency	Yes (storage system implementation is easy)	Yes (storage system implementation is harder with more nodes)	Yes (storage system implementation is hardest)
Durability	Limited by local components (less)	Limited by monolithic architecture (more)	Scaling out to additional systems increases durability (most)
Performance	Limited by local components, can benefit low-latency applications (100us-5ms, GB/sec)	Similar to local, but additional overhead in network transport (500us-5ms, GB/sec)	Scaling out to additional systems increases performance (500us-5ms, TB/sec)

*** The information in this table are generally accepted attributes and measurements among local, remote, and distributed storage systems.*

6 Block Stores

Block stores are a persistence target where data is stored in blocks in local, remote, or distributed locations. The blocks are typically numerically addressed using a method called Logical Block Addressing (LBA) and accessed as a client through a device interface provided by a Kernel. The location (local/remote/distributed) is determined by the physical persistence location of the blocks and serves as a method to group and categorize different stores.

It is possible to transparently augment or enhance numerous characteristics of block stores such as availability, scalability, consistency, durability, and performance by adding additional software-based storage layers (ie. RAID) along with physical devices, networking, and nodes. Please refer to the Capacity, Availability, and Partition-tolerance (CAP) theorem overview in the appendix for more details.

Virtualization adds another perspective which is important to consider. Operating systems may or may not be aware of the type of block store being used. Virtual machines and machine instances are likely not storing any blocks locally but completely leveraging remote or distributed block stores. In this case, instances provide virtualized hardware that store data remotely and emulate the connectivity and behavior of local physical storage devices. This storage would not be considered a local block store due to non-locality of stored data.

Most applications do not directly store data in block format, but instead interface with file systems supported by block devices (ie. application -> local EXT4 filesystem -> local block device -> local/remote/distributed block store). See the filesystems section below for more details.

The following categories include examples solely with the intent of providing context to the category being described. Examples are intended to be widely known to the readers.

6.1 Local Block Stores

Local block stores are built on Direct Attached Storage (DAS) where data is persisted locally on hardware devices. Since all data is stored locally, the scale is limited to the local resource capabilities. The availability of the data is a major consideration when applications are interacting directly with local block stores. Logical volume management (LVM) and similar techniques can be used to augment and concatenate the capabilities that discrete hardware devices provide. These stores tend to be focused on specific use cases where latency is critical or to support other storage services.

Generally accepted example terms, platforms, and protocols: ATA, IDE, logical volumes, LVM, physical volumes, physical storage devices, RAID, SCSI, volume groups

6.2 Remote Block Stores

Remote block stores provide storage attached by a network where data is persisted remotely across a network. This is different from local because there is a separation of application from storage. Generally this has the ability to increase capacity, and performance. The availability is also increased since high availability design patterns can be implemented. Without detailed information and assurances and intentional design, service levels are likely to be driven by this category.

Generally accepted example terms, platforms, and protocols: AWS EBS, FC, FCoE, iSCSI, SAN

6.3 Distributed Block Stores

Distributed block stores are similar to remote block stores but data is persisted across many nodes, possibly in conjunction with the local node, and clients are able to rely on many nodes to provide redundancy and horizontal scalability. When compared with local and remote block stores, distributed block stores require additional control and data access layers to manage data distribution (and often also replication). This added complexity can provide improved scalability, availability, and durability.

Generally accepted example terms, platforms, and protocols: Ceph, OpenEBS, hyper-converged

7 File Systems

A File system is a logical persistence layer organized around storing and retrieving data referenced by files. They provide a richer set of primitives than block stores. These primitives include access control, concurrency control and locking, naming and directory structure, sequential file access, and other features. This makes them more suitable for direct use by applications than block stores. The actual persistence function is performed by supporting layers where the file system may translate files to logical block addresses. File systems can be local, remote, or distributed (independent of underlying block store locality). There are numerous types of file systems which tend to differentiate to optimize for many characteristics including storage medium, read/write expectations, performance, durability, and access patterns.

The following categories include examples solely with the intent of providing context to the category being described. Examples are intended to be widely known to the readers.

7.1 Local File Systems

Local file systems are typically built from local, remote, or distributed block stores. They are commonly used by operating systems to store dependent files.

Generally accepted example terms, platforms, and protocols: EXT4, file, inode, XFS

7.2 Remote File Systems

Remote file systems are also referred to as as the category network file systems. They consist of a specialized client that presents local data structures and stores data across a network in remote locations. Through separating client from server a remote file system's capabilities expand beyond the limits of the local system.

There are numerous types of remote file systems with their own specializations. For example, remote file systems are not inherently optimized for safe multi-client access. Applications sometimes solve for this by introducing additional locking mechanisms or they embrace clustered file systems.

Generally accepted example terms, platforms, and protocols: CIFS, cluster, file locks, NFS, VMFS

7.3 Distributed File Systems

Distributed file systems are a type of remote file system that provide the ability for clients to seamlessly store and retrieve files across clusters of servers. The scale is elastic because files are stored in a distributed manner and are globally addressable.

Generally accepted example terms, platforms, and protocols: Gluster, HDFS, Lustre

7.4 Comparison

Comparing file systems requires also considering the interaction with the underlying storage layers. The following table describes the optimal/neutral/non-optimal combination generally accepted understanding of the interaction of these layers.

	Local File System on..	Remote File System on..	Distributed File System on..
Local Block Store	Optimal	Optimal	Optimal
Remote Block Store	Optimal	Neutral	Non-Optimal
Distributed Block Store	Optimal	Neutral	Non-Optimal

8 Object Stores

Unlike file systems and block stores, where there is a general understanding of the implementation behind the interface, object stores are quite heterogeneous in their implementations. In general an object store system is an *atomic* key-value store, where the key and value are defined by the implementer of the storage system. An atomic key value store guarantees that a request to *set* or *get* a value is either fully committed or not at all.

There are many examples of object stores. From how an internet browser gets HTML content and sends, or posts, data back to the web server, to how an operating system gets and sets data from an LBA on a block store.

8.1 HTTP Based Object Storage

Due to the large range of object store implementations, this paper will be focusing on HTTP based object stores as defined by Amazon Web Services S3, Google Cloud, and OpenStack Swift. Typically, these type of object stores are used for large opaque values, as a result, they are often used to store images, videos, and data backups.

These types of object store systems have defined a set of methods based on the HTTP protocol where the key is a URL and the value is a set of data. This interface makes it simple to access content since there is no need to mount or attach an object store. Due to the nature of this model, these type of object store systems are always remote to the requester node.

An HTTP based object store access model is largely contructured of an *account* in which it has a set of *buckets* as they are called by S3 or *containers* as they are called by OpenStack Swift. Each of these buckets or containers can then contain objects. The *key* is based on the combination of these values loosely based on the format:

```
http(s)://<server/account/container>/[...object]
```

where the *object* is a unique identifier reference to an object. For example:

```
https://server.io/v1/admin/pictures/path/to/the/picture.jpg
```

shows the object is *`path/to/the/picture.jpg`*.

One of the advantages of these HTTP based systems over simple object stores is the metadata management. These storage systems make it possible for the requester to attach custom metadata to the objects, which can then be used to to list, fetch, or group objects. Another advantage is in access control, where access can be placed on anything from a set of containers or buckets to a single object.

8.2 Scalability, Availability, Durability, Performance

HTTP object stores are designed for scalability and durability, but not for low latency performance as compared to block or file based storage systems. Instead they are designed for supporting extremely large amounts of data spread over not only a single data center, but over many regions all over the world. HTTP object stores are also designed for durability, supporting many methods of maintaining the data integrity of their objects. They may maintain multiple copies of their objects or use erasure coding to maintain object durability. These methods provide an unprecedented object durability service level agreement. As an example, Amazon Web Services S3 claims object store service is designed for a durability of 99.999999999%.

Due to their nature, HTTP based object store systems are not suited for latency sensitive applications. On the other hand, unlike block and file, an object store system can provide data to clients on behalf of an application! For example, when a web browser requests data from an application which stores data in an object store, instead of having the application return the data itself, it can send the web browser pointers to the data on the object store. The object store system then returns the data *directly* to the client from a region closest to it, reducing the network IO requirements of the application.

9 Key-Value Stores

A key-value store is a storage system designed for storing, retrieving, and managing key-value pairs. Values are identified and accessed via a key, which is similar to a hash table. In a key-value store, there is no predefined schema and the value of the data is usually opaque. It is a very flexible data model because the application has complete control over what is stored in the value.

A key-value store system might store its data fully in memory, partially in memory or fully on disk. It might be only locally accessible or remotely accessible. It might only run on a single node or might be distributed and scalable. Many more complex storage systems like databases, block storage, file systems, logging systems are usually built on top of key-value stores or key-value abstraction.

9.1 Local Key-value Stores

A local key-value store is usually accessed by single application through inter process communication or direct intra-process API calls. It stores the data in local memory or local filesystem. The local key-value store is designed for low latency access and the ease of use and operation. Many distributed applications or distributed storage systems use one or more local key-value stores as their basic storage unit for further replication. Berkeley-DB, InnoDB, LMDB, RocksDB are the best examples of this category.

9.2 Remote Shared Key-value Stores

A remote shared key-value store is usually accessed by a number of applications through networking protocols (HTTP, RPC or customized ones). It stores the data in local memory or local filesystem. The shared key-value store is designed for efficiency and flexibility. Some remote key-value stores also provide additional data structures API for the ease of use. A traditional relational database can also be used as a remote key-value store with a simple two columns (key, value) table when reliability and durability is the first priority.

Redis, memcached are the best examples of this category.

9.3 Distributed Key-value Stores

A distributed key-value store replicates its data to one or more nodes in the system for high availability and durability, and might shard its data to different replication groups for scale out. Some distributed key-value stores trade off latency or scalability for linearizability and

serializability consistency guarantees over entire key-value space to reduce the risk of potential conflict updates. Some provide weaker consistency guarantees (either eventual consistency or stronger consistency within one single partition) but better latency guarantees.

etcd, ZooKeeper, Consul, etc. provide distributed key-value store API for handling metadata or coordination. They only implement the data replication, but no sharding, to simplify the overall design and improve reliability. These systems provide strong consistency guarantees over the entire key space.

Cassandra, HBase, etc. provide distributed key-value store API for managing massive amounts of data with low latency. They are similar since they are all Wide-Row key value stores. They implement both data replication and sharding. Strong consistency can be achieved for mutations within a row or within a partition, sometimes with limited availability. They do not provide strong consistency guarantees over mutations over different partitions through the entire key space.

Spanner, CockroachDB ,TiKV, FaunaDB, etc., provide distributed key-value store API for managing massive amounts of data and strong consistency guarantees. They implement both the data replication and sharding features. Additionally, they implement distributed transactional protocols across multiple shards to support global transaction either through clocks (high accuracy physical clock or HCL) or through a single master (Calvin or similar protocols). The distributed transaction protocol typically introduces additional latency for cross shard transactions. Even with high accuracy physical clocks, the latency can be as high as several milliseconds.

9.4 Comparison

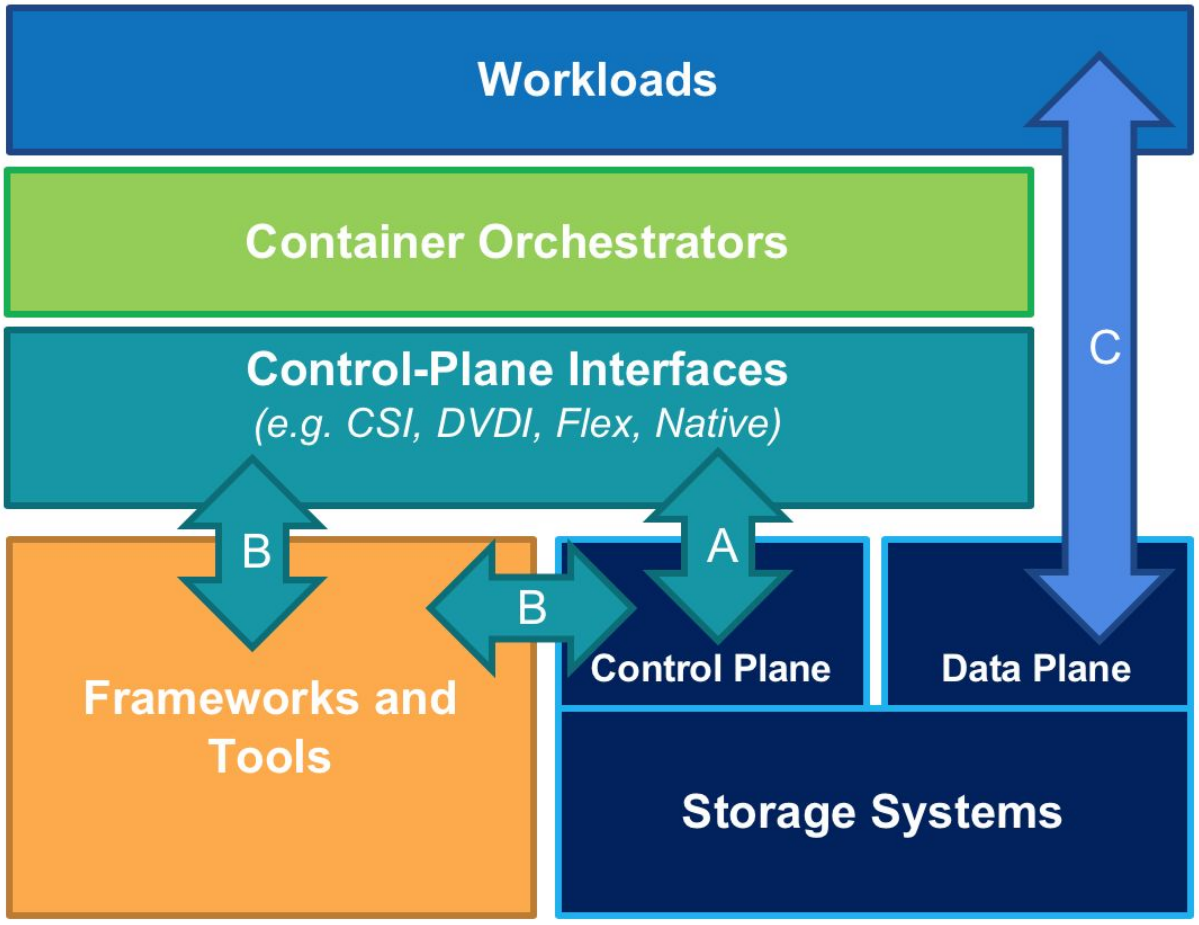
	Local	**Remote**	**Distributed and non-global-transactional**	**Distributed and global-transactional**
Availability	Limited by local components failures	Limited by remote components failures	Partial failures do not affect availability or only limited key-space	Partial failures do not affect availability or only limited key-space
Scalability	Limited by local resources	Limited by remote resources	Scale out as adding more capacities	Scale out as adding more capacities. API scalability is often limited by a single-master.
Global consistency	Strong	Strong	Weak	Strong

Durability	Limited by local storage failure	Limited by remote components failures	Tolerant to partial failures	Tolerant to partial failures
Performance	Limited by I/O access latency	Limited by I/O access latency and network latency	Limited by I/O access latency and network latency	Limited by I/O access latency, network latency, and usually a single-master. Multiple rounds of network latency for cross shards transactions.

10 Orchestration and Management Interfaces

This section defines how Container Orchestration Systems interact with the Storage Systems to associate workloads with Data from the Storage Systems. Depending on the Data Access Interfaces, different layers may be involved.

10.1 Volumes - block stores and filesystems



A Container Orchestration System (CO) such as Kubernetes can support multiple interfaces to interact with the Storage System.

The Storage System can:

- **(A)** support control-plane interface API directly and interact directly with the orchestrator or
- **(B)** interact with the orchestrator via an API Framework layer or other Tools.

The orchestrator can use the control-plane interfaces **(A)** or **(B)** to support the request for a volume and may also be able to use the interface to dynamically provision a volume.

Workloads consume **(C)** storage from storage systems over various data access interfaces.

The underlying storage infrastructure layer can be software-based commodity storage, cloud storage, or enterprise storage. The management layer provides abstraction over the complexity of various storage systems.

Whether to use **(A)** or **(B)** depends on user requirement and capabilities supported by the storage system. **(A)** is focusing on dynamically provisioning storage (or pre-provisioning storage) for workloads. **(B)** may also support discovery, automation, and other data services such as data protection, data migration, or data replication in addition to provisioning.

10.1.1 Control-Plane Interfaces

"Control-Plane Interfaces" refers to storage interfaces for CO. It includes Native Interfaces such as Kubernetes Native Drivers and Docker Volume Driver Interface and External Interfaces such as Kubernetes Flexvolume and Container Storage Interface.

10.1.1.1 K8S Native Drivers

This refers to Kubernetes in-tree volume drivers that extend Kubernetes volume interfaces to support block and file storage systems. Kubernetes has the following concepts for storage:

- Persistent Volume (PV) is a piece of storage provisioned by an administrator on the storage system.
- Persistent Volume Claim (PVC) is the storage requested by a user. Kubernetes cluster will try to find a matching PV that matches the PVC request.
- PV can be pre-provisioned or dynamically provisioned. Dynamic provisioning is done using a Storage Class created by an administrator. Storage Class defines different levels of services that a storage system can provide. Kubernetes manages the life cycle of PVs and PVCs. Data on a volume can persistent beyond the lifetime of a pod that consumes the volume.

Kubernetes in-tree volume drivers can support the following functionalities: create and delete volume, attach and detach volume, mount and unmount volume, and expand volume.

https://kubernetes.io/docs/concepts/storage/
https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes

Kubernetes Storage SIG is in the process of moving these in-tree drivers out of tree, in favor of CSI drivers. There is a design spec aiming to seamlessly migrating from in-tree drivers to

CSI which would allow CSI drivers to handle volume provisioning requests as a proxy for in-tree drivers.

https://github.com/kubernetes/community/blob/master/contributors/design-proposals/storage/csi-migration.md

10.1.1.2 Docker Volume Driver Interface

Docker volumes can be used to persist data in Docker. Docker provides a mechanism for storage vendors to write a volume driver so that remote storage systems such as Amazon EBS can be used to provide volumes for a docker container. This allows data volumes to persist beyond the lifetime of a single Docker host. If a plugin registers itself as a VolumeDriver when activated, it must provide the Docker Daemon with writeable paths on the host filesystem. The Docker daemon provides these paths to containers to consume. The Docker daemon makes the volumes available by bind-mounting the provided paths into the containers.

Supported Docker volume driver interfaces include Create, Remove, Mount, Unmount, Path, Get, List, and Capabilities.

https://docs.docker.com/storage/
https://docs.docker.com/engine/extend/plugins_volume/

10.1.1.3 K8S Flexvolume

Kubernetes Flexvolume provides interfaces to initialize a driver, to attach/detach a volume to/from a host, and to mount/unmount a volume on/from the host. These functions will be executed by the Kubelet component in Kubernetes.

Flexvolume does not provide interfaces to provision/deprovision a volume. A dynamic provisioner can be developed to provision/deprovision a volume and be used together with the Flexvolume plugin.

Flexvolume is also an out-of-tree plugin. Since the trend is to use out-of-tree CSI drivers in the future, new features will not be added to support Flexvolume although existing features in Flexvolume will still be supported.

https://github.com/kubernetes/community/blob/master/contributors/devel/flexvolume.md

10.1.1.4 Container Storage Interface

Container Storage Interface (CSI) is an industry standard to define a set of storage interfaces so that a storage vendor can write one plugin and have it work across a range of Container Orchestration (CO) Systems. COs supporting this specification include Kubernetes, Mesos, Docker, and Cloud Foundry. Other companies, including storage vendors, have also been helping with the design. It has evolved to become the volume driver interface of the future for Container Orchestration Systems.

CSI has three gRPC services: controller, node, and identity services. Identity service provides info and capabilities of a plugin. Controller service supports create and delete volume, create and delete snapshot, and attach and detach volume. Node service supports mount and unmount volume. Expand volume support is also coming soon. For more details, see the spec here: https://github.com/container-storage-interface/spec

CSI v1.0.0 was released in November 2018. At the time of the v1.0.0 release, support for CSI in Kubernetes is Beta, both Mesos and Cloud Foundry have experimental CSI drivers, and Docker's CSI driver is work in progress.

10.1.2 Frameworks and other tools

"Frameworks and other tools" are extensions of CO's *"Control-Plane Interfaces"*. In addition to provisioning and managing storage, this extended control plane can also support discovery, automation, data protection, data migration, disaster recovery, monitoring, analytics, performance tuning, and data lifecycle management, etc.

Some examples of frameworks and other tools described in this section include OpenSDS, REX-Ray, Rook and Heptio Ark.

10.2 Application API

Which management interface options are relevant for object stores, KV and databases?

Currently the Control-Plane Interfaces, the storage interfaces supported by COs, do not include object stores, key value stores, and databases. Some Frameworks and Tools have support for object stores, key value stores, and databases. Some examples will be given in the following section.

Note that there is an extension API called Service Catalog that enables applications running in Kubernetes clusters to use external managed software offerings, such as a datastore service offered by a cloud provider.

https://kubernetes.io/docs/concepts/extend-kubernetes/service-catalog/

10.2.1 Object Stores

Some management interfaces provide a way to directly deploy object storage and allow object storage to be consumed by containers through the object interface (usually S3). Rook's support of Minio is a good example of this. REX-Ray has integration with object storage as well. OpenSDS is also building object storage support via S3 APIs.

There are also ways to connect persistent volumes provisioned for containers to object store on premise or in the cloud.

- For cloud storage such as Google Cloud Persistent Disks or Amazon Elastic Block Storage, a snapshot of a PVC for block storage will be uploaded to the object store somewhere in the cloud as part of the snapshot creation process.
- Some other management interfaces provide a similar approach that uploads a snapshot created for a block storage to an object store on premise or in the cloud.
- There is ongoing work in CSI to allow the topology to be specified for a snapshot to be uploaded somewhere in the cloud. Topology describes information such as region, zone, bucket, rack, etc.
- Some management interfaces also provide a separate backup API that takes a volume or snapshot from a block storage and backs it up to an object store.

10.2.2 Key Value Stores

It is possible for a management interface to provide a way to deploy and manage key value stores, similar to how databases can be deployed and managed by the management interface.

10.2.3 Databases

Management interface can provide a way to deploy and manage databases. For example, Rook provides an operator to deploy and manage CockroachDB clusters. Another CNCF storage project Vitess also provides an operator to manage MySQL clusters.

https://vitess.io

11 Appendix

11.1 Document History

Initially, the document was structured based on classes of storage type which are categorised by the way the storage is consumed e.g. block, file or object. This did not provide a useful way to compare and contrast their attributes and how they are utilised in production as most storage systems have many layers and are formed of multiple components. While the data access interface (like block or file) might affect how the data is consumed and how it might failover between nodes, it does not effectively define attributes such as data protection, consistency, or durability.

As a further complication, many commonly used systems are layered storage systems where, for example, a filesystem may be built on an object store (e.g. CephFS), or a block store may be built on a distributed filesystem (e.g. gluster block storage). This meant that the way the storage is accessed did not usefully define the attributes that an application cared about (such as the durability, data protection or some of the performance characteristics of the overall system), as those attributes are defined at other layers in the stack.

11.2 Consensus Protocols

Consensus protocols provide reliable agreement among a group of potentially faulty distributed processes on a single data value or a replicated log. They are commonly used to decide whether to commit a data change transaction, for leader election, state machine replication, load balancing, clock synchronization and others in distributed systems. The two

most popular (families of) consensus algorithms are Multi-Paxos and Raft, both of which have been formally proven correct (for practical uses, with some caveats). Both rely on a single elected leader, and (typically) agreement by a strict majority of participants (e.g. for 5 participants, at least 3 must explicitly agree). Raft is considered simpler to understand and implement than Multi-Paxos. Other ad-hoc attempts at consensus algorithms are notoriously prone to edge case failures.

11.2.1 Paxos

Paxos is arguably the oldest formally studied family of consensus algorithms. It is considered highly robust when implemented properly, but challenging to implement correctly for practical uses.

11.2.2 Raft

Raft was developed about a decade after Paxos, to address the issues mentioned above. It has become widely used, and forms the basis of, amongst others, the popular etcd cloud-native key-value store, and Consul distributed service mesh.

11.2.3 Two-phase Commit ("2PC")

2PC is a specialized form of consensus protocol used for coordination between participants in a distributed atomic transaction to decide on whether to commit or abort (roll back) the transaction. 2PC is not resilient to all possible failures, and in some cases, outside (e.g. human) intervention is needed to remedy failures. Also, it is a blocking protocol. All participants block between sending in their vote (see below), and receiving the outcome of the transaction from the co-ordinator. If the co-ordinator fails permanently, participants may block indefinitely, without outside intervention. In normal, non-failure cases, the protocol consists of two phases, whence it derives its name:

1. The commit-request phase (or voting phase), in which a coordinator requests all participants to take the necessary steps for either committing or aborting the transaction and to vote, either "Yes" (on success) , or "No" (on failure)
2. The commit phase, in which case the coordinator decides whether to commit (if all participants have voted "Yes") or abort, and notifies all participants accordingly.

11.2.4 Three-phase Commit ("3PC")

3PC adds an additional phase to the 2PC protocol to address the indefinite blocking issue mentioned above. But 3PC still cannot recover from network segmentation, and due to the additional phase, requires more network round-trips, resulting in higher transaction latency.

11.3 Consistency, Coherence and Isolation

The above three terms are commonly used in various different contexts to mean different things in the fields of data stores and distributed systems. Without going into detail here, suffice to say that, consistency in particular, is a widely misunderstood term, so it's worth thinking twice before assuming that you understand exactly what's meant by a particular use

of the term. For example, ACID (Atomicity, Consistency, Isolation, Durability) properties and the CAP Theorem (concerning Consistency, Availability and Partition-tolerance) are both widely used terms, and many people assume that they understand what these terms mean. But considerably fewer people realise that “Consistency” means quite different things in those two contexts. For further details, Wikipedia and Irene Zhang’s musing provide good starting points.

11.3.1 ACID

With the above caveats, for data storage systems, Atomicity, Consistency, Isolation and Durability are generally considered to mean:

1. Atomicity: a guarantee that each transaction across multiple data items is treated as a single "unit", which either succeeds completely, or fails completely, even in the case of various failures including machine crashes and network errors.
2. Consistency: Usually understood to mean guarantees about whether a transaction started in the future can necessarily see the effects of all transactions committed in the past. Also sometimes understood to be a guarantee that a transaction can only bring the data from one valid state to another, while maintaining invariants (for example that stock count cannot be less than zero, or that two customers with the same id number cannot exist).
3. Isolation: guarantees that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially, in some order.
4. Durability: guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

11.3.2 The CAP Theorem

The CAP Theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

1. Consistency: Every read receives the most recent write or an error
2. Availability: Every request receives a response that is not an error
3. Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

In the absence of network failure both availability and consistency can be satisfied. CAP is frequently misunderstood to mean that one has to choose to abandon one of the three guarantees at all times. In fact, the choice is really between consistency and availability only when a network partition or failure happens; at all other times, no trade-off has to be made.

Database systems designed with traditional ACID guarantees in mind such as RDBMS choose consistency over availability, whereas systems designed around the BASE

philosophy, common in the NoSQL movement for example, choose availability over consistency.

The PACELC theorem builds on CAP by stating that even in the absence of partitioning, another trade-off between latency and consistency occurs.