STORAGE DEVELOPER CONFERENCE
SDC 21
BY Developers FOR Developers
Virtual Conference
September 28-29, 2021

CSI Driver Design

Bringing a Parallel File System to Containerized Workloads

Eric Weber, Software Engineer and BeeGFS CSI Driver Contributor, NetApp
Joe McCormick, Software Engineer and BeeGFS CSI Driver Contributor, NetApp

Container Storage Interface – Why?

Containers often need access to pre-provisioned or dynamically allocated external storage

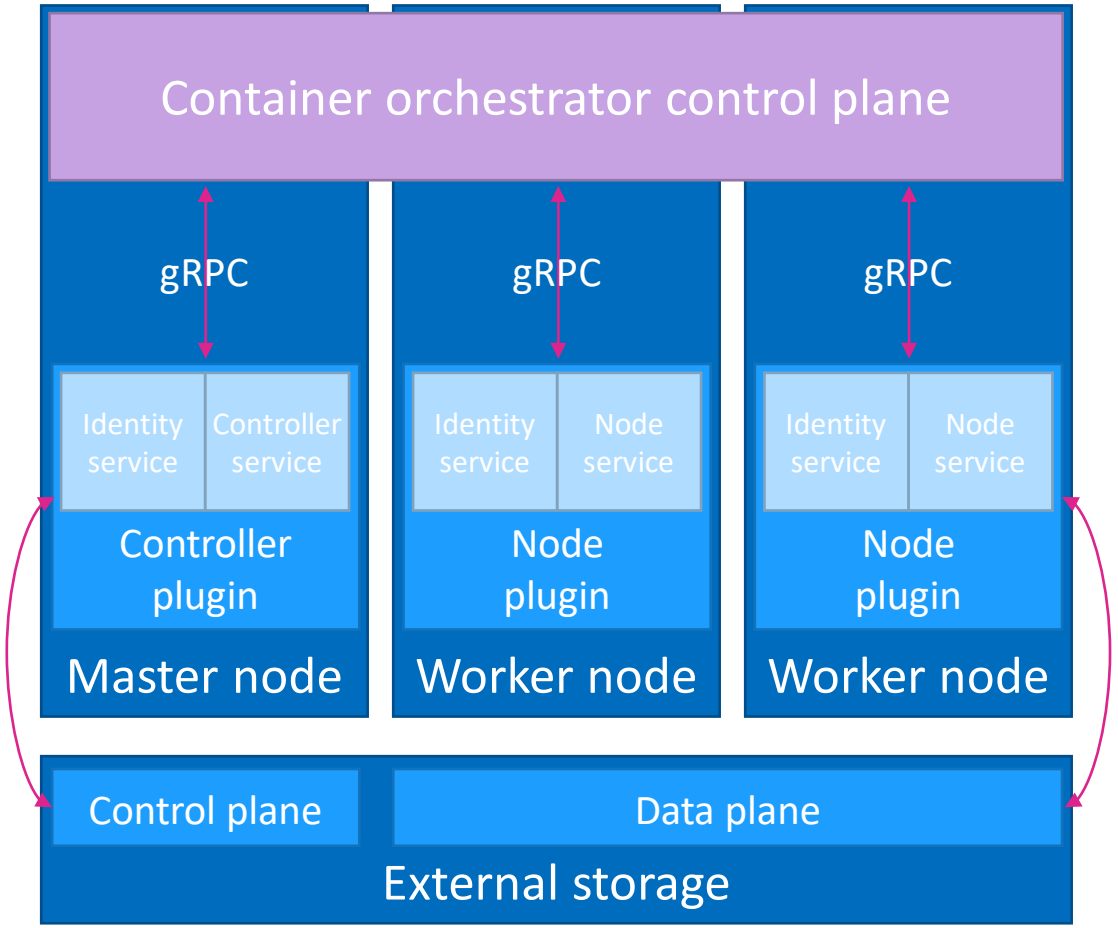
Old way

- Kubernetes-specific plugins maintained in-tree
- Limited documentation on creating dynamic provisioner

New way

- Standalone, fully-featured plugins supported by multiple orchestrators

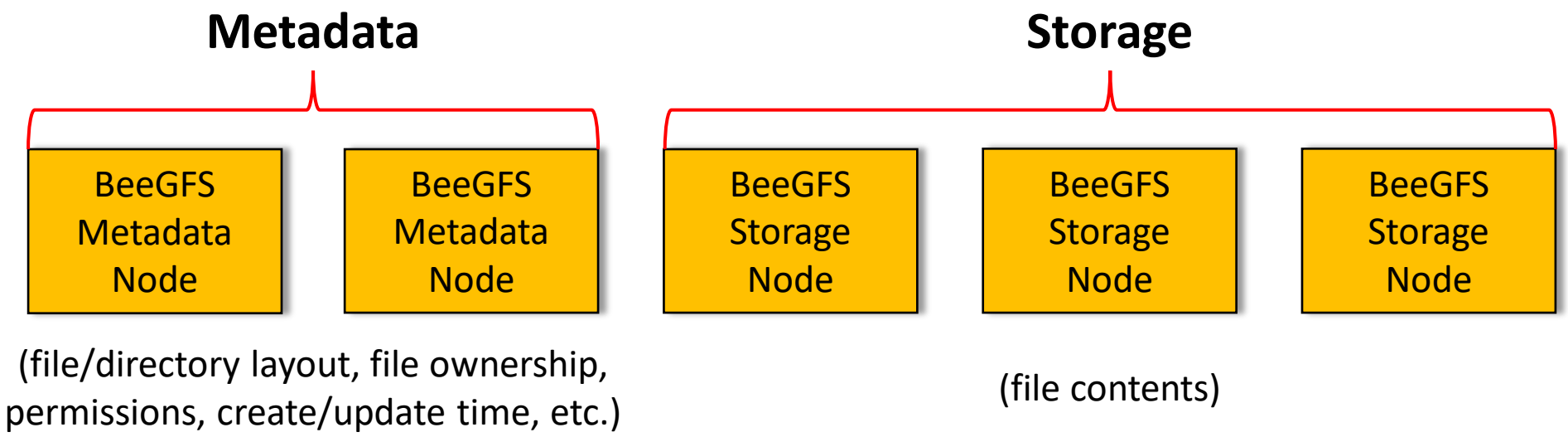
Container Storage Interface – How?



- A proto3 spec defines the RPC interface between the Container Orchestrator (CO) control plane and the plugin
- The plugin handles communicating with and mounts external storage

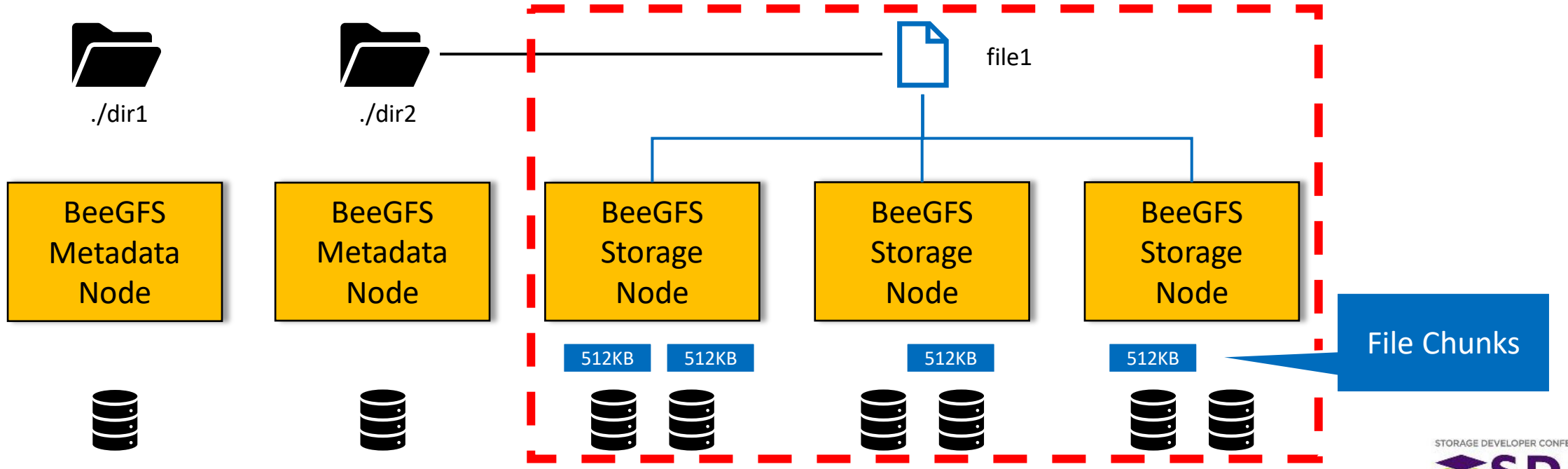
What is BeeGFS?

- Parallel File System (similar architecture as Lustre).
- Stores file/directory metadata and file contents separately.
- Stripes file contents across multiple storage nodes.
  - Designed for concurrent access to the same file(s) from 10s, 100s, or 1000s of clients.



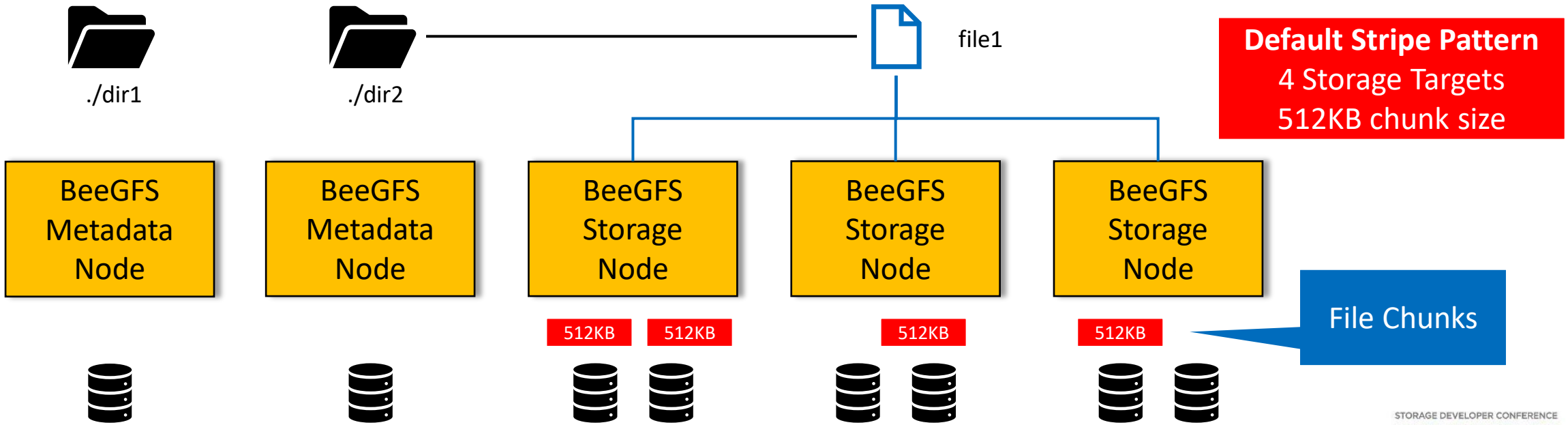
What is BeeGFS?

- Parallel File System (similar architecture as Lustre).
- Stores file/directory metadata and file contents separately.
- Stripes file contents across multiple storage nodes.
  - Designed for concurrent access to the same file(s) from 10s, 100s, or 1000s of clients.

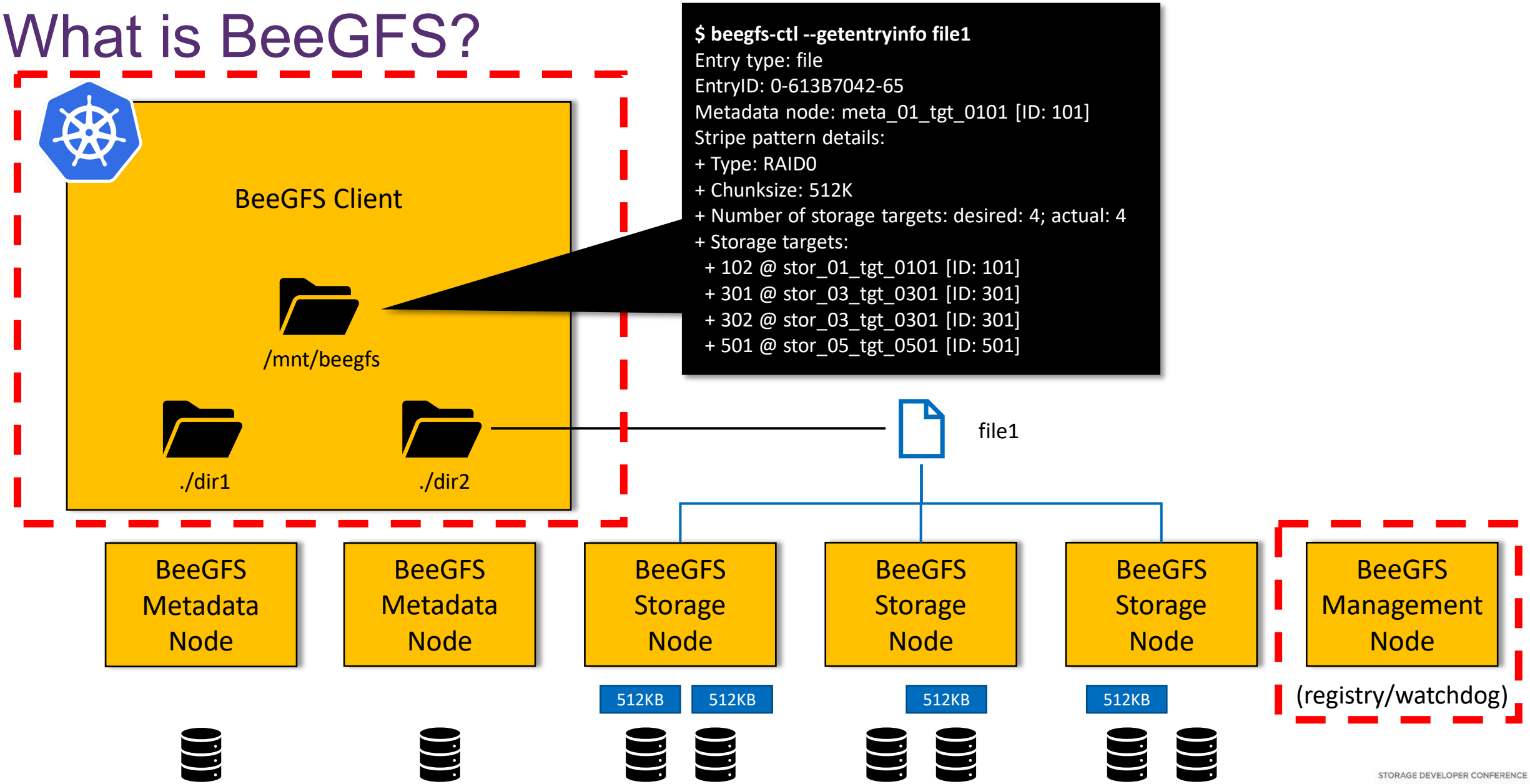


What is BeeGFS?

- Parallel File System (similar architecture as Lustre).
- Stores file/directory metadata and file contents separately.
- Stripes file contents across multiple storage nodes.
  - Designed for concurrent access to the same file(s) from 10s, 100s, or 1000s of clients.



What is BeeGFS?

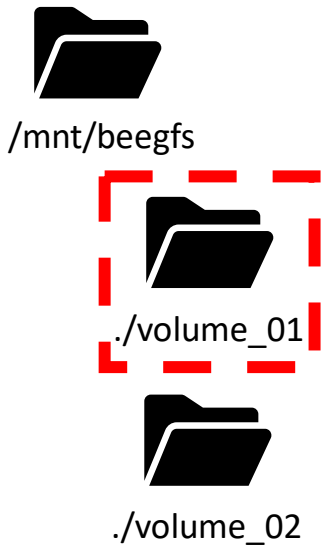


Major design decisions

- What is a “volume”?
- Do we support dynamic provisioning?
- Which volume lifecycle model fits?
- How do we package our driver?
- How do we handle configuration?
- How do we get started with implementation?

What is a “volume”?

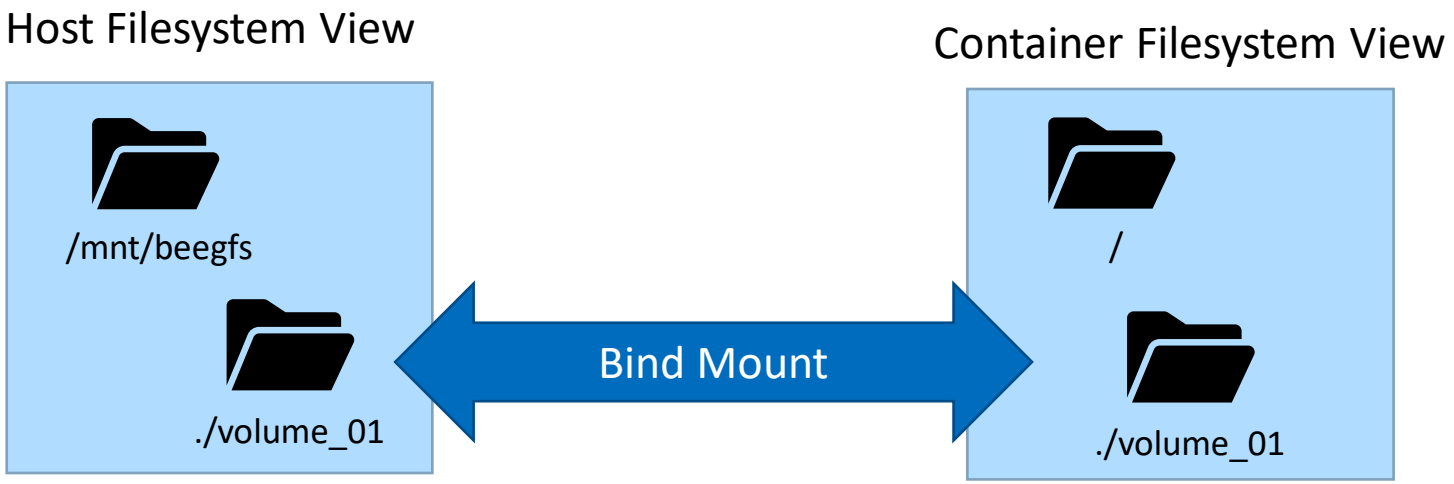
- CSI specification defines a volume as:
  - A unit of storage that will be made available inside of a container orchestrator (CO) managed container, via the CSI.
- CSI volumes are directories in a BeeGFS filesystem.
  - Capacity not enforced – all BeeGFS volumes are essentially “thin” provisioned.



https://github.com/NetApp/beegfs-csi-driver/blob/release-1.1/docs/usage.md#capacity

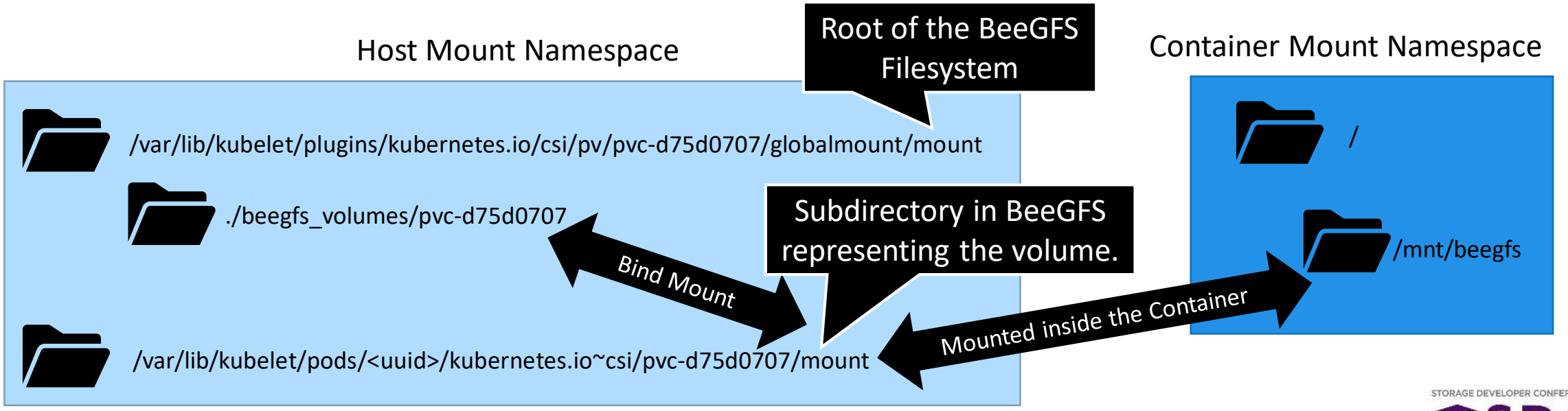
What is a “volume”?

- CSI specification defines a volume as:
  - A unit of storage that will be made available inside of a container orchestrator (CO) managed container, via the CSI.
- CSI volumes are directories in a BeeGFS filesystem.
  - Capacity not enforced – all BeeGFS volumes are essentially “thin” provisioned.
  - Isolation achieved by creating a bind mount of the directory representing a specific volume that is then exposed inside the container.



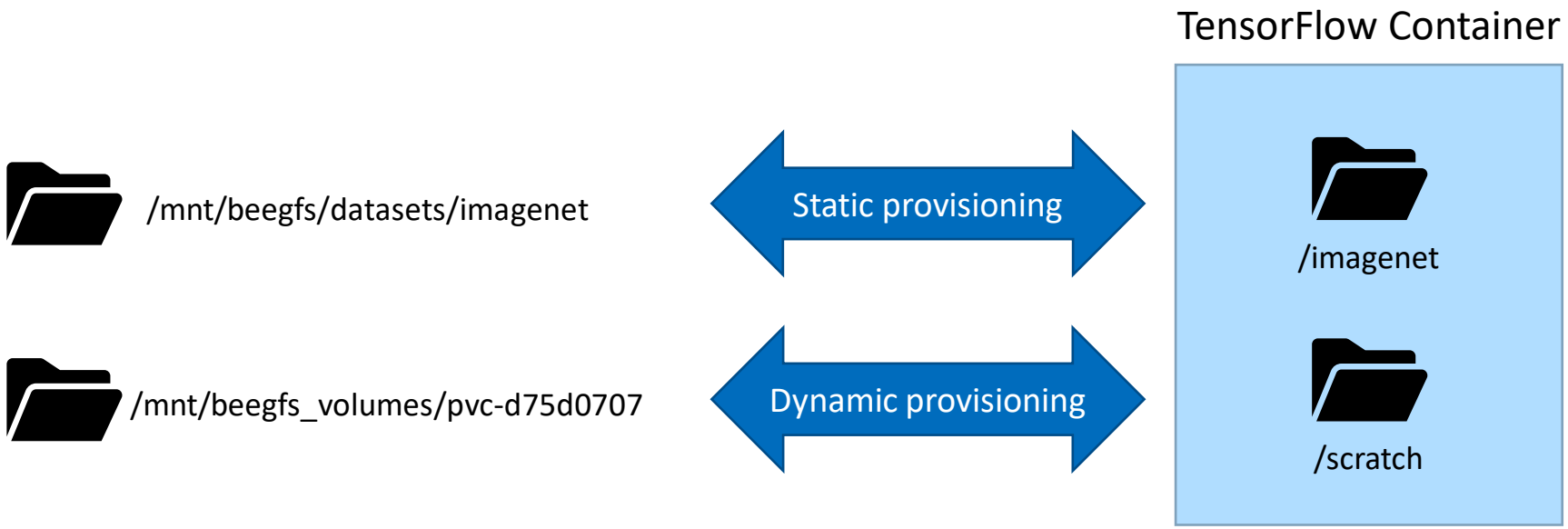
What is a “volume”?

- CSI specification defines a volume as:
  - A unit of storage that will be made available inside of a container orchestrator (CO) managed container, via the CSI.
- CSI volumes are directories in a BeeGFS filesystem.
  - Capacity not enforced – all BeeGFS volumes are essentially “thin” provisioned.
  - Isolation achieved by creating a bind mount of the directory representing a specific volume that is then exposed inside the container by the CO.



Do we support dynamic provisioning?
(i.e., Do we need a controller plugin?)

- According to the spec:
  - A controller plugin is NOT required.
  - A controller plugin with the CREATE_DELETE_VOLUME capability can be implemented to dynamically provision volumes.



Do we support dynamic provisioning?
(i.e., Do we need a controller plugin?)

- According to the spec:
  - A controller plugin is NOT required.
  - A controller plugin with the CREATE_DELETE_VOLUME capability can be implemented to dynamically provision volumes.
- BeeGFS CSI driver design considerations:
  - BeeGFS does not have an easily consumed REST API.
- BeeGFS CSI driver decision:
  - The controller component of our driver uses beegfs-ctl commands AND mount commands to create and delete directories.
  - Administrators must preinstall the BeeGFS client and utilities packages on nodes that run both the controller and node plugins.
    - Use CO features (i.e., labels, node selectors, affinities/anti-affinities) if BeeGFS cannot or shouldn't be mounted to all nodes in the cluster.

Which volume lifecycle model fits?

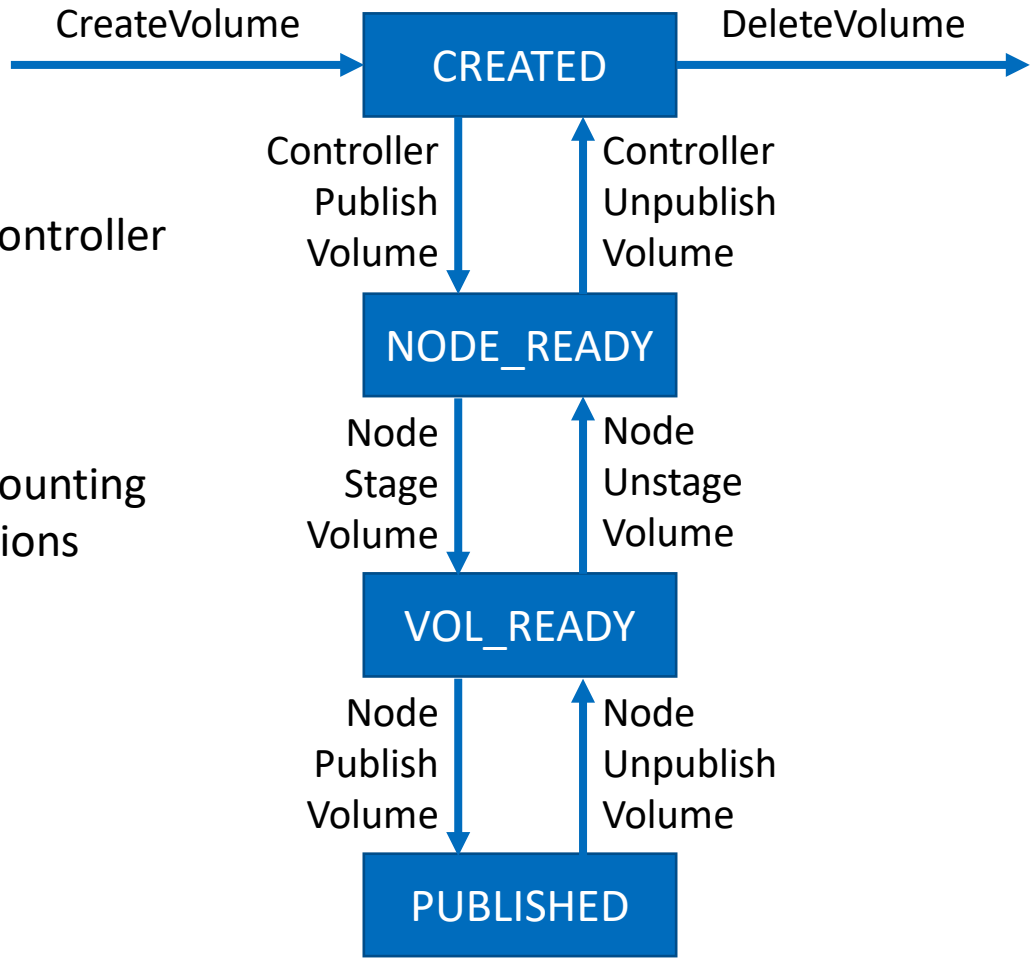
- ControllerPublishVolume:
  - Called when the CO wants to place a workload on a node.
  - The controller plugin makes the volume available on that node in response.
  - OPTIONALLY implemented (subject to the controller plugin PUBLISH_UNPUBLISH_VOLUME capability).
- NodeStageVolume:
  - Called prior to a workload running on a node.
  - The node plugin stages the volume at a global location on that node in response.
  - OPTIONALLY implemented (subject to the node plugin STAGE_UNSTAGE_VOLUME capability).

Which volume lifecycle model fits?

Example use case:
LUN provisioning

Example use case:
LUN masking on a storage array controller

Example use case:
Discovering a block device and mounting a filesystem from one of its partitions



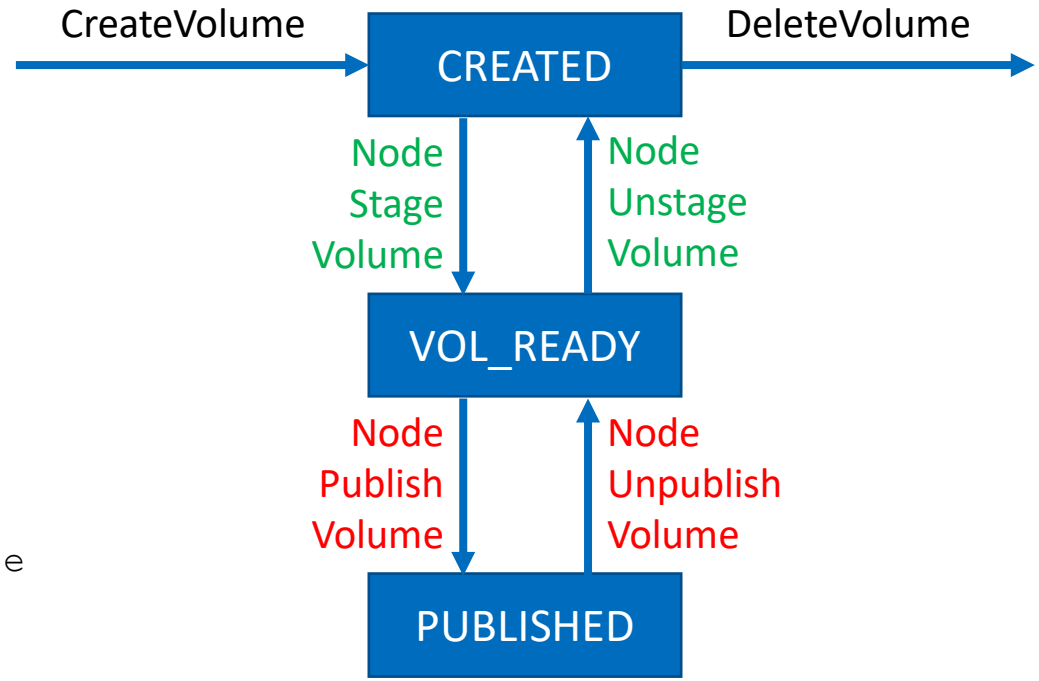
Which volume lifecycle model fits?

Design considerations:

- BeeGFS is designed for parallel access
- BeeGFS can’t be mounted to a host on a per-directory basis
- Different workloads may want to mount BeeGFS with different options

Which volume lifecycle model fits?

```
/ (BeeGFS file system)
|-- unrelated_dir
|-- csi_dir
    |-- csi_vol_1
    |-- csi_vol_2
        |-- some_file


/ (node file system)
|-- ...
    |-- staging_target_path
        |-- unrelated_dir
        |-- csi_dir
            |-- csi_vol_1
            |-- csi_vol_2
                |-- some_file


/ (node file system)
|-- ...
    |-- target_path
        |-- some_file
```



How do we package our driver?

- According to the spec:
  - For Plugins packaged in software form, Plugin Packages SHOULD use a well-documented container image format (e.g., Docker, OCI).
  - Plugin Supervisor SHALL guarantee that plugins will have CAP_SYS_ADMIN capability on Linux when running on nodes.
- According to the Kubernetes CSI Developer Documentation:
  - The controller component can be deployed as a Deployment or a StatefulSet on any node in the cluster.
  - The node component should be deployed on every node in the cluster through a DaemonSet.
  - Each component consists of a driver container and sidecars.

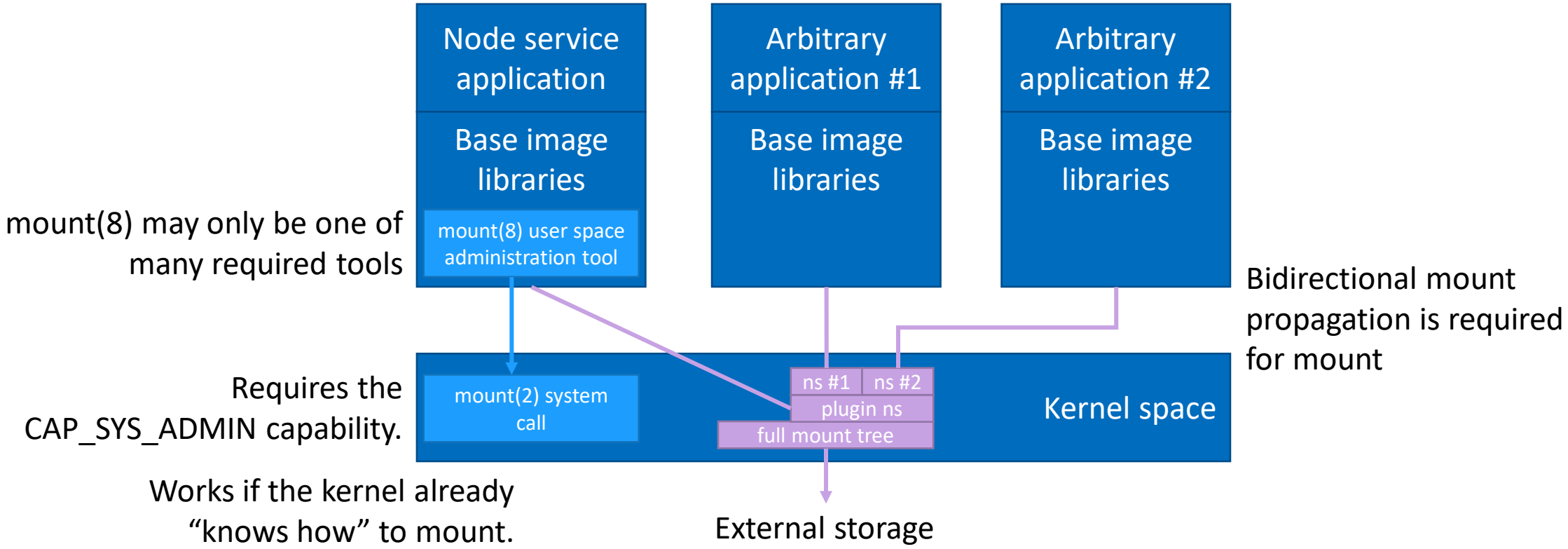
How do we package our driver?

Design considerations:

- The BeeGFS client packages are not available by default in industry standard Linux distributions.
- The BeeGFS client and RDMA connectivity function as kernel modules.
- The BeeGFS client license does not clearly allow for redistribution.
- It is generally preferable (especially when being careful about 3rd party licensing) to limit the number of distributed packages and components.
- The BeeGFS client must communicate over UDP on a different port for each file system mount. (This requires host instead of container networking.)

How do we package our driver?

A “typical” CSI driver mount operation:

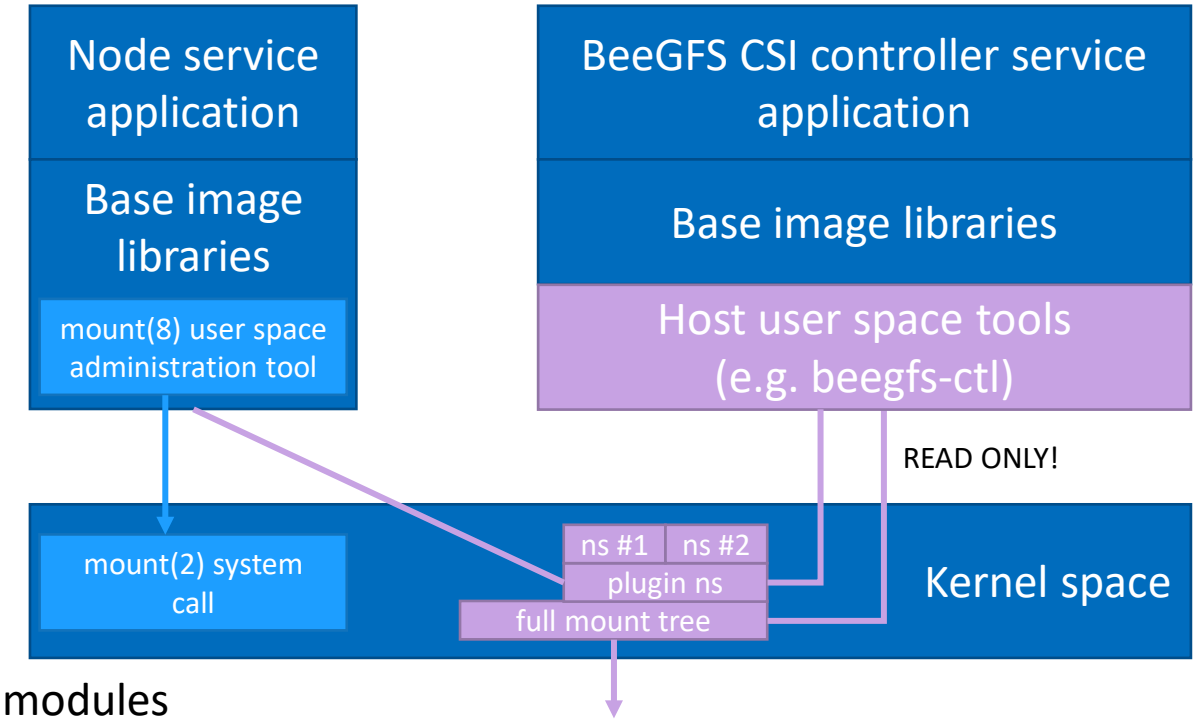


How do we package our driver?

Extra considerations for BeeGFS (and others):

Advantage:
Ship a driver image with NO additional libraries/tools (use the node's own iscsiadm, mount, multipath, etc. tools).

Disadvantage:
Gives the driver additional access to the node.



Chroot to the host's mounted root directory to access tools.

Additional privileges are required.

BeeGFS packages and modules MUST already be installed.

How do we handle configuration?

- What might users want/have to configure for BeeGFS?
  - Client Configuration: /etc/beegfs/beegfs-client.conf (default path).
    - E.g., enable/disable RDMA, tuning, retries/timeouts, preferred interfaces.

```
# mount
beegfs_nodev on /mnt/beegfs type beegfs (rw,relatime,cfgFile=/etc/beegfs/beegfs-client.conf)
beegfs_nodev on /mnt/beegfs_1 type beegfs (rw,relatime,cfgFile=/etc/beegfs/beegfs-client_1.conf)
```

  - Dynamic Configuration: beegfs-ctl
    - E.g., influence where/how files are distributed across available storage targets.

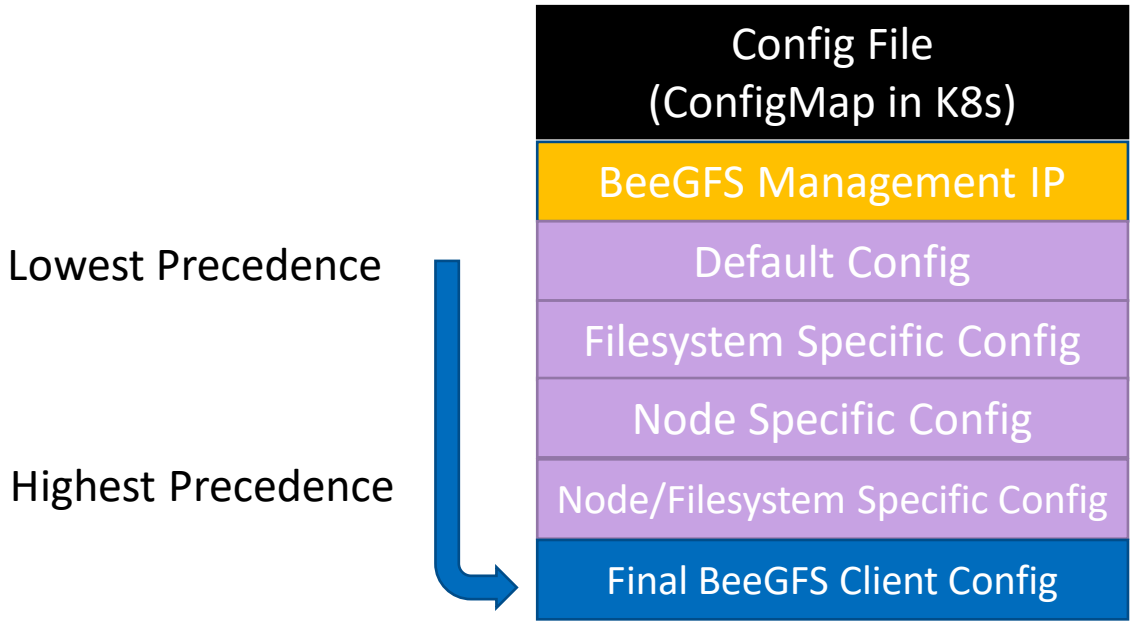
```
beegfs-ctl --getentryinfo datasets/
Entry type: directory
EntryID: 0-60818F4B-64
Metadata node: meta_01_tgt_0100 [ID: 100]
Stripe pattern details:
+ Type: RAID0
+ Chunksize: 512K
+ Number of storage targets: desired: 4
+ Storage Pool: 2 (ictm1626c1-ef600)
```


How do we handle configuration?

- According to the spec:
  - CreateVolume accepts a parameters map and returns a volume_id string and a volume_context map.
  - NodeStageVolume and NodePublishVolume accept the volume_id string and the volume_context map.
  - NodeUnpublishVolume, NodeUnstageVolume, and DeleteVolume only accept the volume_id string.
  - The volume_id alone should be sufficient to uniquely identify the volume.
- BeeGFS CSI driver design considerations:
  - Our design does not include a source of truth accessible by the controller plugin and all node plugins. The information passed in each RPC must be sufficient to complete the task designated by the RPC.
  - A sysMgmtdHost and directory path are enough to uniquely identify a BeeGFS volume.
  - Certain BeeGFS client configuration options may be necessary to connect to a particular BeeGFS file system (containing any number of volumes) from a particular BeeGFS node.
  - Certain striping patterns or permissions may increase the performance of a particular volume for a particular workload.

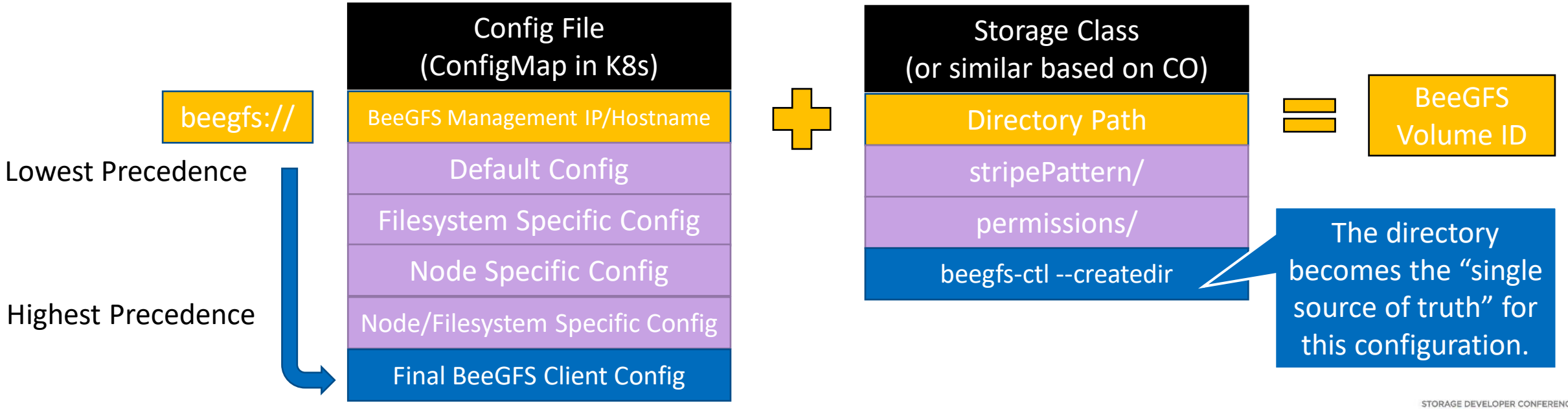
How do we handle configuration?

- BeeGFS CSI driver decision:
  - Default, file system-specific, and node-specific BeeGFS client configuration options can be specified in a configuration file loaded on startup.
    - These govern how an entire BeeGFS file system is mounted/accessed from a particular node.



How do we handle configuration?

- BeeGFS CSI driver decision:
  - Default, file system-specific, and node-specific BeeGFS client configuration options can be specified in a configuration file loaded on startup.
    - These govern how an entire BeeGFS file system is mounted/accessed from a particular node.
  - Striping and permissions parameters are passed in the CreateVolume parameters map.
  - volume_id is a URI composed of a sysMgmtdHost and a directory path (beegfs://192.168.3.100/datasets/imagenet)
    - In combination with the startup configuration, this is enough information for all RPCs to operate.



Handling BeeGFS Client configuration

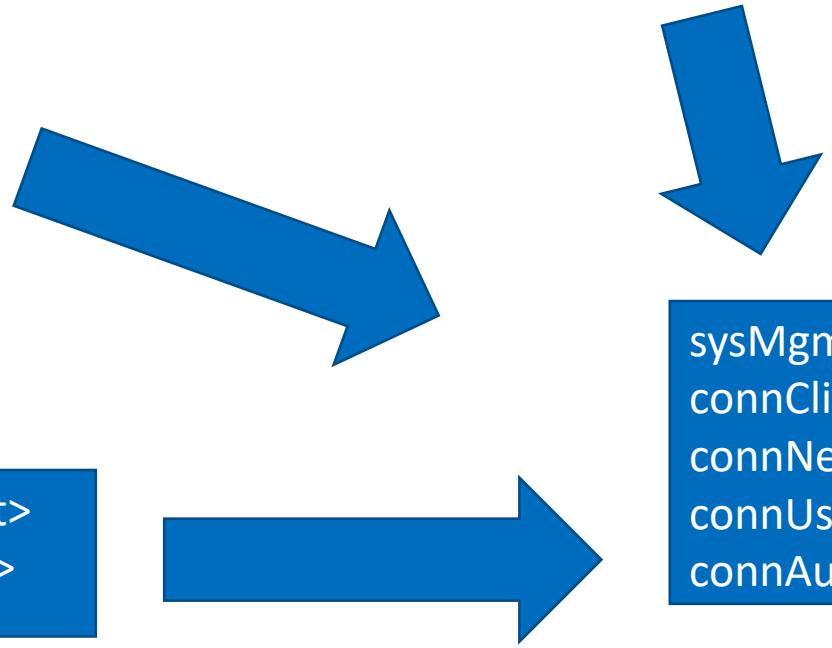
csi-beegfs-config.yaml

```
config:
   connInterfaces:
      - ib0
      - eth0
   beegfsClientConf:
      connUseRDMA: true
```

csi-beegfs-connauth.yaml

```
- sysMgmtdHost: <sysMgmtdHost>
  connAuth: <some_secret_value>
```

getEphemeralPortUDP() (port int, err error)



```
sysMgmtdHost        = <Parsed from BeeGFS URI>
connClientPortUDP   = 58616
connNetFilterFile   = <PATH>/connNetFilterFile
connUseRDMA         = True
connAuthFile        = <PATH>/connAuthFile
```


Handling dynamic BeeGFS configuration

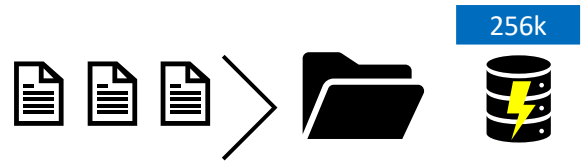
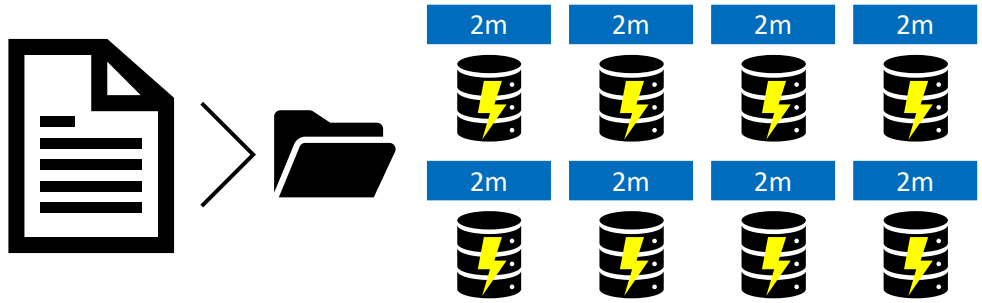
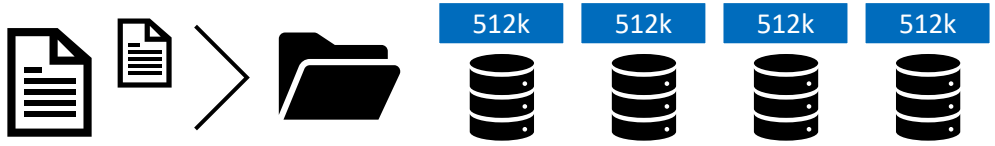
- Introducing Storage Classes in Kubernetes:
  - Describe “classes” of storage available in the cluster.
  - May map to different quality-of-service, backup policies, etc.
  - Kubernetes is unopinionated about what they represent.
- Required BeeGFS Storage Class parameters
  - sysMgmtdHost: Management IP for the BeeGFS filesystem
  - volDirBasePath: Parent directory to create BeeGFS volumes under.
- Optional BeeGFS Storage Class Parameters
  - stripePattern/storagePoolId
  - stripePattern/chunkSize
  - stripePattern/numTargets
  - permissions/uid
  - permissions/gid
  - permissions/mode

Kubernetes Terminology:

- Pod: One or more containers with shared storage/network and runtime specification.
- Storage Class: Way to describe “classes” of storage available in a Kubernetes cluster.
- Persistent Volume (PV): Piece of storage with a lifecycle independent of individual pods.
- Persistent Volume Claim (PVC): Request for storage by a user/pod (consumes PVs).

Handling dynamic BeeGFS configuration

Note: The striping and storage pool configuration on directories in BeeGFS is like a “template” for how any new files or subdirectories created in that directory are written (by default).

If a user wants to…	They use a storage class like:	Resulting in this directory configuration:
Optimize my volume for small files needing fast access:	Hot Small Files stripePattern/storagePoolId: “1” stripePattern/chunkSize: 256k stripePattern/numTargets: “1”	
Optimize my volume for large files needing fast access:	Hot Large Files stripePattern/storagePoolId: “1” stripePattern/chunkSize: 2m stripePattern/numTargets: “8”	
Optimize my volume for mixed files that just need to be archived:	Cold Storage stripePattern/storagePoolId: “2” stripePattern/chunkSize: 512k stripePattern/numTargets: “4”	

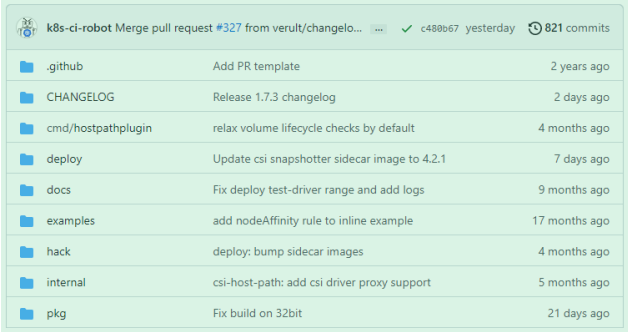
How do we start the implementation?

```
service Controller {
  rpc CreateVolume (CreateVolumeRequest)
    returns (CreateVolumeResponse) {}

  rpc DeleteVolume (DeleteVolumeRequest)
    returns (DeleteVolumeResponse) {}

  rpc ControllerPublishVolume (ControllerPublishVolumeRequest)
    returns (ControllerPublishVolumeResponse) {}

  rpc ControllerUnpublishVolume (ControllerUnpublishVolumeRequest)
    returns (ControllerUnpublishVolumeResponse) {}
```

Start from the proto3 spec and stub out a server in one of a large selection of languages.

```
// ControllerServer is the server API for Controller service.
type ControllerServer interface {
        CreateVolume(context.Context, *CreateVolumeRequest) (*CreateVolumeResponse, error)
        DeleteVolume(context.Context, *DeleteVolumeRequest) (*DeleteVolumeResponse, error)
        ControllerPublishVolume(context.Context, *ControllerPublishVolumeRequest) (*ControllerPublishVolumeResponse, error)
        ControllerUnpublishVolume(context.Context, *ControllerUnpublishVolumeRequest) (*ControllerUnpublishVolumeResponse, error)
        ValidateVolumeCapabilities(context.Context, *ValidateVolumeCapabilitiesRequest) (*ValidateVolumeCapabilitiesResponse, error)
        ListVolumes(context.Context, *ListVolumesRequest) (*ListVolumesResponse, error)
        GetCapacity(context.Context, *GetCapacityRequest) (*GetCapacityResponse, error)
        ControllerGetCapabilities(context.Context, *ControllerGetCapabilitiesRequest) (*ControllerGetCapabilitiesResponse, error)
        CreateSnapshot(context.Context, *CreateSnapshotRequest) (*CreateSnapshotResponse, error)
        DeleteSnapshot(context.Context, *DeleteSnapshotRequest) (*DeleteSnapshotResponse, error)
        ListSnapshots(context.Context, *ListSnapshotsRequest) (*ListSnapshotsResponse, error)
        ControllerExpandVolume(context.Context, *ControllerExpandVolumeRequest) (*ControllerExpandVolumeResponse, error)
        ControllerGetVolume(context.Context, *ControllerGetVolumeRequest) (*ControllerGetVolumeResponse, error)
}
```

Import the precompiled GoLang stub into a new GoLang project.



Start with the HostPath example driver and strip out/modify code as necessary.

Additional design decisions

- What volume capabilities do we support?
  - E.g., block vs. filesystem, single vs. multi-node, reader and/or writer.
  - There is no concept of a “block” volume to the BeeGFS client.
  - BeeGFS is designed for highly parallel access from multiple nodes.
  - There is no downside to supporting MULTI_NODE_xxx capabilities.
- What CO do we focus on for deployment/testing?
  - Kubernetes, Hashicorp Nomad, Apache Mesos, and others support CSI.
  - Kubernetes is far-and-away the most popular CO.
  - Kubernetes has a well-documented deployment model for CSI drivers.
  - BeeGFS CSI driver v1.2.0 will include an example Nomad deployment.
- What CSI features do we support?
  - E.g., snapshots, cloning, volume expansion.
  - BeeGFS is heavily focused on performance, so there isn’t a lot of native feature support.
  - The BeeGFS CSI driver v1.0.0 was designed to be an MVP.

Additional design decisions

- How should the driver be deployed (kustomize vs something else)?
  - E.g., straight YAML manifests, Kustomize, Helm, the operator pattern.
  - The hostpath example driver uses straight manifests integrated with release tools.
  - Kustomize provides a similar feature set with an added layer of customization.
  - BeeGFS CSI driver v1.2.0 will release with an operator for native OpenShift/okd deployments.
- How do we handle permissions?
  - BeeGFS is a POSIX compliant file system.
  - Administrators should be able to control the permissions on new directories.
  - The recursive process kicked off by fsGroup doesn't work well for a shared file system.
- How do we test the driver?
  - Tools like csi-sanity enable testing outside the context of a container orchestrator.
  - The Kubernetes end-to-end tests have basic CSI integration.
  - The Kubernetes end-to-end tests can be extended for custom use-cases.
  - The Kubernetes storage APIs have evolved; testing on multiple versions is advised.

Contact us!

- Create an issue on our GitHub page
  - https://github.com/netapp/beegfs-csi-driver
- E-mail us
  - ng-beegfs-csi-driver@netapp.com
  - eric.weber2@netapp.com
  - joe.mccormick@netapp.com
- Read our blogs
  - https://www.netapp.com/blog/kubernetes-meet-beegfs/
  - https://netapp.io/?s=beegfs+csi

Important resources

- CSI specification
  https://github.com/container-storage-interface/spec/blob/master/spec.md
- Precompiled Golang stub
  https://github.com/container-storage-interface/spec/blob/master/lib/go/csi/csi.pb.go
- Kubernetes CSI developer documentation
  https://kubernetes-csi.github.io/docs/
- List of production CSI drivers (including BeeGFS)
  https://kubernetes-csi.github.io/docs/drivers.html
- Kubernetes CSI GA announcement
  https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/

Please take a moment to rate this session.

Your feedback is important to us.

