BeeGFS on NetApp with E-Series Storage

BeeGFS on NetApp with E-Series Storage

NetApp
January 31, 2023

Table of Contents

BeeGFS on NetApp with E-Series Storage . . . 1
Get started . . . 2
- What's included in this site . . . 2
- Terms and concepts . . . 2

Use verified architectures . . . 4
- Overview and requirements . . . 4
- Review solution design . . . 12
- Deploy solution . . . 24

Use custom architectures . . . 70
- Overview and requirements . . . 70
- Initial Set Up . . . 71
- Define the BeeGFS file system . . . 74
- Deploy the BeeGFS file system . . . 99

Administer BeeGFS Clusters . . . 111
- Overview, key concepts, and terminology . . . 111
- When to use Ansible versus the pcs tool . . . 112
- Examine the state of the cluster . . . 112
- Reconfigure and update . . . 114
- Service and maintain . . . 118
- Troubleshoot . . . 124

Legal notices . . . 131
- Copyright . . . 131
- Trademarks . . . 131
- Patents . . . 131
- Privacy policy . . . 131
- Open source . . . 131

BeeGFS on NetApp with E-Series Storage

Get started

What's included in this site

This site documents how to deploy and manage BeeGFS on NetApp using both NetApp Verified Architectures (NVAs) and custom architectures. NVA designs are thoroughly tested and provide customers with reference configurations and sizing guidance to minimize deployment risk and accelerate time to market. NetApp also supports custom BeeGFS architectures running on NetApp hardware, giving customers and partners flexibility designing file systems to meet a wide range of requirements. Both approaches leverage Ansible for deployment, providing an appliance-like approach managing BeeGFS at any scale across a flexible range of hardware.

Terms and concepts

The following terms and concepts apply to the BeeGFS on NetApp solution.

See the Administer BeeGFS clusters section for additional details on terms and concepts specific to interacting with BeeGFS high availability (HA) clusters.

Term	Description
AI	Artificial Intelligence.
Ansible Inventory	Directory structure containing YAML files that are used to describe the desired BeeGFS HA cluster.
BMC	Baseboard management controller. Sometimes referred to as a service processor.
block nodes	Storage systems.
clients	Nodes in the HPC cluster running applications that need to utilize the file system. Sometimes also referred to as compute or GPU nodes.
DL	Deep Learning.
file nodes	BeeGFS file servers.
HA	High Availability.
HIC	Host Interface Card.
HPC	High-Performance Computing.

Term	Description
HPC-style workloads	HPC-style workloads are typically characterized by multiple compute nodes or GPUs all needing to access the same dataset in parallel to facilitate a distributed compute or training job. These datasets are often comprised of large files that should be striped across multiple physical storage nodes to eliminate the traditional hardware bottlenecks that would prevent concurrent access to a single file.
ML	Machine Learning.
NLP	Natural Language Processing.
NLU	Natural Language Understanding.
NVA	The NetApp Verified Architecture (NVA) program provides reference configurations and sizing guidance for specific workloads and use cases. These solutions are thoroughly tested, and are designed to minimize deployment risks and to accelerate time to market.
storage network / client network	Network used for clients to communicate to the BeeGFS file system. This is often the same network used for parallel Message Passing Interface (MPI) and other application communication between HPC cluster nodes.

Use verified architectures

Overview and requirements

Solution overview

The BeeGFS on NetApp solution combines the BeeGFS parallel file system with NetApp EF600 storage systems for a reliable, scalable, and cost-effective infrastructure that keeps pace with demanding workloads.

This design takes advantage of the performance density delivered by the latest enterprise server and storage hardware and network speeds, requiring file nodes that feature dual AMD EPYC 7003 “Milan” processors and support for PCIe 4.0 with direct connects using 200Gb (HDR) InfiniBand to block nodes that provide end-to-end NVMe and NVMeOF using the NVMe/IB protocol.

NVA program

The BeeGFS on NetApp solution is part of the NetApp Verified Architecture (NVA) program, which provides customers with reference configurations and sizing guidance for specific workloads and use cases. NVA solutions are thoroughly tested and designed to minimize deployment risks and to accelerate time to market.

Use cases

The following use cases apply to the BeeGFS on NetApp solution:

- Artificial Intelligence (AI) including machine learning (ML), deep learning (DL), large-scale natural language processing (NLP), and natural language understanding (NLU). For more information, see BeeGFS for AI: Fact versus fiction.
- High-performance computing (HPC) including applications accelerated by MPI (message passing interface) and other distributed computing techniques. For more information, see Why BeeGFS goes beyond HPC.
- Application workloads characterized by:
  - Reading or writing to files larger than 1GB
  - Reading or writing to the same file by multiple clients (10s, 100s, and 1000s)
- Multi-terabyte or multi-petabyte datasets.
- Environments that need a single storage namespace optimizable for a mix of large and small files.

Benefits

The key benefits of using BeeGFS on NetApp include:

- Availability of verified hardware designs providing full integration of hardware and software components to ensure predictable performance and reliability.
- Deployment and management using Ansible for simplicity and consistency at scale.
- Monitoring and observability provided using the E-Series Performance Analyzer and BeeGFS plugin. For more information, see Introducing a Framework to Monitor NetApp E-Series Solutions.
- High availability featuring a shared-disk architecture that provides data durability and availability.

- Support for modern workload management and orchestration using containers and Kubernetes. For more information, see Kubernetes meet BeeGFS: A tale of future-proof investment.

HA architecture

BeeGFS on NetApp expands the functionality of the BeeGFS enterprise edition by creating a fully integrated solution with NetApp hardware that enables a shared-disk high availability (HA) architecture.

> While the BeeGFS community edition can be used free of charge, the enterprise edition requires purchasing a professional support subscription contract from a partner like NetApp. The enterprise edition allows use of several additional features including resiliency, quota enforcement, and storage pools.

The following figure compares the shared-nothing and shared-disk HA architectures.

[beegfs design image1]

For more information, see Announcing High Availability for BeeGFS Supported by NetApp.

Ansible

BeeGFS on NetApp is delivered and deployed using Ansible automation, which is hosted on GitHub and Ansible Galaxy (the BeeGFS collection is available from Ansible Galaxy and NetApp's E-Series GitHub). Although Ansible is primarily tested with the hardware used to assemble the BeeGFS building blocks, you can configure it to run on virtually any x86-based server using a supported Linux distribution.

For more information, see Deploying BeeGFS with E-Series Storage.

Design generations

The BeeGFS on NetApp solution is currently in its second generational design.

Both the first and second generation include a base architecture that incorporates a BeeGFS file system and an NVMe EF600 storage system. However, the second generation builds on the first to include these additional benefits:

- Double the performance and capacity while adding only 2U of rack space
- High availability (HA) based on a shared-disk, two-tier hardware design
- External qualification for NVIDIA's DGX A100 SuperPOD and NVIDIA BasePOD Architectures

Second generational design

The second generation of BeeGFS on NetApp is optimized to meet the performance requirements of demanding workloads including high-performance computing (HPC) and HPC-style machine learning (ML), deep learning (DL), and similar artificial intelligence (AI) techniques. By incorporating a shared-disk high-availability (HA) architecture, the BeeGFS on NetApp solution also meets the data durability and availability requirements of enterprises and other organizations that cannot afford downtime or data loss as they look for storage that can scale to keep up with their workloads and use cases. This solution has not only been verified by NetApp, but it also passed external qualification as a storage option for the NVIDIA DGX SuperPOD and DGX BasePOD.

First generational design

The first generation of BeeGFS on NetApp was designed for machine learning (ML) and artificial intelligence (AI) workloads using NetApp EF600 NVMe storage systems, the BeeGFS parallel file system, NVIDIA DGX™ A100 systems, and NVIDIA® Mellanox® Quantum™ QM8700 200Gbps IB switches. This design also features 200Gbps InfiniBand (IB) for the storage and compute cluster interconnect fabric to provide a completely IB-based architecture for high-performance workloads.

For more information on the first generation, see NetApp EF-Series AI with NVIDIA DGX A100 Systems and BeeGFS.

Architecture overview

The BeeGFS on NetApp solution includes architectural design considerations used to determine the specific equipment, cabling, and configurations required to support validated workloads.

Building block architecture

The BeeGFS file system can be deployed and scaled in different ways depending on the storage requirements. For example, use cases primarily featuring numerous small files will benefit from extra metadata performance and capacity, whereas use cases featuring fewer large files might favor more storage capacity and performance for actual file contents. These multiple considerations impact different dimensions of the parallel file system deployment, which adds complexity to designing and deploying the file system.

To address these challenges, NetApp has designed a standard building block architecture that is used to scale out each of these dimensions. Typically, BeeGFS building blocks are deployed in one of three configuration profiles:

- A single base building block, including BeeGFS management, metadata, and storage services
- A BeeGFS metadata plus storage building block
- A BeeGFS storage only building block

The only hardware change between these three options is the use of smaller drives for BeeGFS metadata. Otherwise, all configuration changes are applied through software. And with Ansible as the deployment engine, setting up the desired profile for a particular building block makes configuration tasks straightforward.

For further details, see Verified hardware design.

File system services

The BeeGFS file system includes the following main services:

- **Management service.** Registers and monitors all other services.
- **Storage service.** Stores the distributed user file contents known as data chunk files.
- **Metadata service.** Keeps track of the file system layout, directory, file attributes, and so on.
- **Client service.** Mounts the file system to access the stored data.

The following figure shows BeeGFS solution components and relationships used with NetApp E-Series systems.

[beegfs components]

As a parallel file system, BeeGFS stripes its files over multiple server nodes to maximize read/write performance and scalability. The server nodes work together to deliver a single file system that can be simultaneously mounted and accessed by other server nodes, commonly known as *clients*. These clients can see and consume the distributed file system similarly to a local file system such as NTFS, XFS, or ext4.

The four main services run on a wide range of supported Linux distributions and communicate via any TCP/IP or RDMA-capable network, including InfiniBand (IB), Omni-Path (OPA), and RDMA over Converged Ethernet (RoCE). The BeeGFS server services (management, storage, and metadata) are user space daemons, while the client is a native kernel module (patchless). All components can be installed or updated without rebooting, and you can run any combination of services on the same node.

Verified nodes

The BeeGFS on NetApp solution includes the following verified nodes: the NetApp EF600 storage system (block node) and the Lenovo ThinkSystem SR665 Server (file node).

Block node: EF600 storage system

The NetApp EF600 all-flash array provides consistent, near real-time access to data while supporting any number of workloads simultaneously. To enable fast, continuous feeding of data to AI and HPC applications, EF600 storage systems deliver up to two million cached read IOPS, response times of under 100 microseconds, and 42GBps sequential read bandwidth in one enclosure.

File node: Lenovo ThinkSystem SR665 Server

The SR665 is a two-socket 2U server featuring PCIe 4.0. When configured to meet the requirements of this solution, it provides ample performance to run BeeGFS file services in a configuration well balanced with the availability of throughput and IOPs provided by the direct attached E-Series nodes.

For more information about the Lenovo SR665, see Lenovo's website.

Verified hardware design

The solution's building blocks (shown in the following figure) uses two dual socket PCIe 4.0-capable servers for the BeeGFS file layer and two EF600 storage systems as the block layer.

[beegfs design image2 small]

> Because each building block includes two BeeGFS file nodes, a minimum of two building blocks is required to establish quorum in the failover cluster. While you can configure a two-node cluster, this configuration has limits that might prevent a successful failover to occur. If you require a two-node cluster, you can incorporate a third device as a tiebreaker (however, that design is not covered in this site).

Each building block delivers high availability through a two-tier hardware design that separates fault domains for the file and block layers. Each tier can independently fail over, providing increased resiliency and reducing the risk of cascading failures. Using HDR InfiniBand in conjunction with NVMeOF provides high throughput and minimal latency between file and block nodes, with full redundancy and sufficient link oversubscription to avoid the disaggregated design becoming a bottleneck, even when the system is partially degraded.

The BeeGFS on NetApp solution runs across all building blocks in the deployment. The first building block deployed must run BeeGFS management, metadata, and storage services (referred to as the base building block). All subsequent building blocks are configured through software to run BeeGFS metadata and storage services, or only storage services. The availability of different configuration profiles for each building block

enables scaling of file system metadata or storage capacity and performance using the same underlying hardware platforms and building block design.

Up to five building blocks are combined into a standalone Linux HA cluster, ensuring a reasonable number of resources per cluster resource manager (Pacemaker), and reducing the messaging overhead required to keep cluster members in sync (Corosync). A minimum of two building blocks per cluster is recommended to allow enough members to establish quorum. One or more of these standalone BeeGFS HA clusters are combined to create a BeeGFS file system (shown in the following figure) that is accessible to clients as a single storage namespace.

[beegfs design image3]

Although ultimately the number of building blocks per rack depends on the power and cooling requirements for a given site, the solution was designed so that up to five building blocks can be deployed in a single 42U rack while still providing room for two 1U InfiniBand switches used for the storage/data network. Each building block requires eight IB ports (four per switch for redundancy), so five building blocks leaves half the ports on a 40-port HDR InfiniBand switch (like the NVIDIA QM8700) available to implement a fat-tree or similar nonblocking topology. This configuration ensures that the number of storage or compute/GPU racks can be scaled up without networking bottlenecks. Optionally, an oversubscribed storage fabric can be used at the recommendation of the storage fabric vendor.

The following image shows an 80-node fat-tree topology.

[beegfs design image4]

By using Ansible as the deployment engine to deploy BeeGFS on NetApp, administrators can maintain the entire environment using modern infrastructure as code practices. This drastically simplifies what would otherwise be a complex system, allowing administrators to define and adjust configuration all in one place, then ensure it is applied consistently regardless of how large the environment scales. The BeeGFS collection is available from Ansible Galaxy and NetApp's E-Series GitHub.

Technical requirements

To implement the BeeGFS on NetApp solution, make sure your environment meets the technology requirements.

Hardware requirements

The following table lists the hardware components that are required to implement a single second-generation building block design of the BeeGFS on NetApp solution.

> The hardware components used in any particular implementation of the solution might vary based on customer requirements.

Count	Hardware component	Requirements
2	BeeGFS file nodes.	Each file node should meet or exceed the following configuration to achieve expected performance. **Processors:** • 2x AMD EPYC 7343 16C 3.2 GHz. • Configured as two NUMA zones. **Memory:** • 256GB. • 16x 16GB TruDDR4 3200MHz (2Rx8 1.2V) RDIMM-A (prefer more smaller DIMMs over fewer larger DIMMs). • Populated to maximize memory bandwidth. **PCIe Expansion: Four PCE Gen4 x16 slots:** • Two slots per NUMA zone. • Each slot should provide enough power/cooling for the Mellanox MCX653106A-HDAT adapter. **Miscellaneous:** • Two 1TB 7.2K SATA drives (or comparable) configured in RAID 1 for the OS. • 10GbE OCP 3.0 adapter (or comparable) for in-band OS management. • 1GbE BMC with Redfish API for out-of-band server management. • Dual hot swap power supplies and performance fans. • Must support Mellanox optical InfiniBand cables if required to reach storage InfiniBand switches. **Lenovo SR665:** • A custom NetApp model includes the required version of the XClarity controller firmware needed to support dual-port Mellanox ConnectX-6 adapters. Contact NetApp for ordering details.
8	Mellanox ConnectX-6 HCAs (for file nodes).	• MCX653106A-HDAT Host Channel Adapters (HDR IB 200Gb, Dual-port QSFP56, PCIe4.0 x16).
8	1m HDR InfiniBand cables (for file/block node direct connects).	• MCP1650-H001E30 (1m Mellanox Passive Copper cable, IB HDR, up to 200Gbps, QSFP56, 30AWG). The length can be adjusted to account for longer distances between the file and block nodes if required.

Count	Hardware component	Requirements
8	HDR InfiniBand cables (for file node/storage switch connections)	Requires InfiniBand HDR cables (QSFP56 transceivers) of the appropriate length to connect file nodes to storage leaf switches. Possible options include: • MCP1650-H002E26 (2m Mellanox Passive Copper cable, IB HDR, up to 200Gb/s, QSFP56, 30AWG). • MFS1S00-H003E (3m Mellanox active fiber cable, IB HDR, up to 200Gb/s, QSFP56).
2	E-Series block nodes	Two EF600 controllers configured as follows: • Memory: 256GB (128GB per controller). • Adapter: 2-port 200Gb/HDR (NVMe/IB). • Drives: Configured to match desired capacity.

Software requirements

For predictable performance and reliability, releases of the BeeGFS on NetApp solution are tested with specific versions of the software components required to implement the solution.

Software deployment requirements

The following table lists the software requirements deployed automatically as part of the Ansible-based BeeGFS deployment.

Software	Version
BeeGFS	7.2.6
Corosync	3.1.5-1
Pacemaker	2.1.0-8
OpenSM	opensm-5.9.0 (from mlnx_ofed 5.4-1.0.3.0) ⓘ Only required for the direct connects to enable virtualization.

Ansible control node requirements

The BeeGFS on NetApp solution is deployed and managed from an Ansible control node. For more information, see the Ansible documentation.

The software requirements listed in the following tables are specific to the version of the NetApp BeeGFS Ansible collection listed below.

Software	Version
Ansible	2.11 When installed through pip: ansible-4.7.0 and ansible-core < 2.12,>=2.11.6

Software	Version
Python	3.9
Additional Python packages	Cryptography-35.0.0, netaddr-0.8.0
BeeGFS Ansible Collection	3.0.0

File node requirements

Software	Version
RedHat Enterprise Linux	RedHat 8.4 Server Physical with High Availability (2 socket). File nodes require a valid RedHat Enterprise Linux Server subscription and the Red Hat Enterprise Linux High Availability Add-On.
Linux Kernel	4.18.0-305.25.1.el8_4.x86_64
InfiniBand / RDMA Drivers	Inbox
ConnectX-6 HCA Firmware	FW: 20.31.1014
PXE: 3.6.0403	UEFI: 14.24.0013

EF600 block node requirements

Software	Version
SANtricity OS	11.70.2
NVSRAM	N6000-872834-D06.dlp
Drive Firmware	Latest available for the drive models in use.

Additional requirements

The equipment listed in the following table was used for the validation, but appropriate alternatives can be used as needed. In general, NetApp recommends running the latest software versions to avoid unanticipated issues.

Hardware component	Installed software
• 2x Mellanox MQM8700 200Gb InfiniBand switches	• Firmware 3.9.2110

Hardware component	Installed software
1x Ansible control node (virtualized): • Processors: Intel® Xeon® Gold 6146 CPU @ 3.20GHz • Memory: 8GB • Local storage: 24GB	• CentOS Linux 8.4.2105 • Kernel 4.18.0-305.3.1.el8.x86_64 Installed Ansible and Python versions match those in the table above.
10x BeeGFS Clients (CPU nodes): • Processor: 1x AMD EPYC 7302 16-Core CPU at 3.0GHz • Memory: 128GB • Network: 2x Mellanox MCX653106A-HDAT (one port connected per adapter).	• Ubuntu 20.04 • Kernel: 5.4.0-100-generic • InfiniBand Drivers: Mellanox OFED 5.4-1.0.3.0
1x BeeGFS Client (GPU node): • Processors: 2x AMD EPYC 7742 64-Core CPUs at 2.25GHz • Memory: 1TB • Network: 2x Mellanox MCX653106A-HDAT (one port connected per adapter). This system is based on NVIDIAs HGX A100 platform and includes four A100 GPUs.	• Ubuntu 20.04 • Kernel: 5.4.0-100-generic • InfiniBand Drivers: Mellanox OFED 5.4-1.0.3.0

Review solution design

Design overview

Specific equipment, cabling, and configurations are required to support the BeeGFS on NetApp solution, which combines the BeeGFS parallel file system with the NetApp EF600 storage systems.

Learn more:

- Hardware configuration
- Software configuration
- Design verification
- Sizing guidelines
- Performance tuning

Derivative architectures with variations in design and performance:

- High Capacity Building Block

Hardware configuration

The hardware configuration for BeeGFS on NetApp includes file nodes and network cabling.

File node configuration

File nodes have two CPU sockets configured as separate NUMA zones, which include local access to an equal number of PCIe slots and memory.

InfiniBand adapters must be populated in the appropriate PCI risers or slots, so the workload is balanced over the available PCIe lanes and memory channels. You balance the workload by fully isolating work for individual BeeGFS services to a particular NUMA node. The goal is to achieve similar performance from each file node as if it were two independent single socket servers.

The following figure shows the file node NUMA configuration.

[beegfs design image5 small]

The BeeGFS processes are pinned to a particular NUMA zone to ensure that the interfaces used are in the same zone. This configuration avoids the need for remote access over the inter-socket connection. The inter-socket connection is sometimes known as the QPI or GMI2 link; even in modern processor architectures, they can be a bottleneck when using high-speed networking like HDR InfiniBand.

Network cabling configuration

Within a building block, each file node is connected to two block nodes using a total of four redundant InfiniBand connections. In addition, each file node has four redundant connections to the InfiniBand storage network.

In the following figure, notice that:

- All file node ports outlined in green are used to connect to the storage fabric; all other file node ports are the direct connects to the block nodes.
- Two InfiniBand ports in a specific NUMA zone connect to the A and B controllers of the same block node.
- Ports in NUMA node 0 always connect to the first block node.
- Ports in NUMA node 1 connect to the second block node.

[beegfs design image6]

For storage networks with redundant switches, ports outlined in light green should connect to one switch, and ports in dark green to another switch.

The cabling configuration depicted in the figure allows each BeeGFS service to:

- Run in the same NUMA zone regardless of which file node is running the BeeGFS service.
- Have secondary optimal paths to the front-end storage network and to the back-end block nodes regardless of where a failure occurs.
- Minimize performance effects if a file node or controller in a block node requires maintenance.

Cabling to leverage bandwidth

To leverage the full PCIe bidirectional bandwidth, make sure one port on each InfiniBand adapter connects to the storage fabric, and the other port connects to a block node. The theoretical maximum speed of an HDR InfiniBand port is 25GBps (not accounting for signaling and other overhead). The maximum single direction bandwidth of a PCIe 4.0 x16 slot is 32GBps, creating a potential bottleneck when implementing file nodes that incorporate dual port InfiniBand adapters that can theoretically handle 50GBps of bandwidth.

The following figure shows the cabling design used to leverage the full PCIe bidirectional bandwidth.

[beegfs design image7]

For each BeeGFS service, use the same adapter to connect the preferred port used for client traffic with the path to the block nodes controller that is the primary owner of that services volumes. For more information, see Software configuration.

Software configuration

The software configuration for BeeGFS on NetApp includes BeeGFS network components, EF600 block nodes, BeeGFS file nodes, resource groups, and BeeGFS services.

BeeGFS network configuration

The BeeGFS network configuration consists of the following components.

- **Floating IPs**
  Floating IPs are a kind of virtual IP address that can be dynamically routed to any server in the same network. Multiple servers can own the same Floating IP address, but it can only be active on one server at any given time.

  Each BeeGFS server service has its own IP address that can move between file nodes depending on the run location of the BeeGFS server service. This floating IP configuration allows each service to fail over independently to the other file node. The client simply needs to know the IP address for a particular BeeGFS service; it does not need to know which file node is currently running that service.

- **BeeGFS server multi-homing configuration**
  To increase the density of the solution, each file node has multiple storage interfaces with IPs configured in the same IP subnet.

  Additional configuration is required to make sure that this configuration works as expected with the Linux networking stack, because by default, requests to one interface can be responded to on a different interface if their IPs are in the same subnet. In addition to other drawbacks, this default behavior makes it impossible to properly establish or maintain RDMA connections.

  The Ansible-based deployment handles tightening of the reverse path (RP) and address resolution protocol (ARP) behavior, along with ensuring when floating IPs are started and stopped; corresponding IP routes and rules are dynamically created to allow the multihomed network configuration to work properly.

- **BeeGFS client multi-rail configuration**
  Multi-rail refers to the ability of an application to use multiple independent network “rails” to increase performance.

  Although BeeGFS can use RDMA for connectivity, BeeGFS uses IPoIB to simplify discovering and establishing RDMA connections. To allow BeeGFS clients to use multiple InfiniBand interfaces, you can configure each client with an IP address located in a different subnet and then configure the preferred

interfaces for half of the BeeGFS server services in each subnet.

In the following diagram, interfaces highlighted in light green are located in one IP subnet (for example, `100.127.0.0/16`) and the dark green interfaces are located in another subnet (for example, `100.128.0.0/16`).

The following figure shows the balancing of traffic across multiple BeeGFS client interfaces.

[beegfs design image8]

Because each file in BeeGFS is typically striped across multiple storage services, the multi-rail configuration allows the client to achieve more throughput than is possible with a single InfiniBand port. For example, the following code sample shows a common file-striping configuration that allows the client to balance traffic across both interfaces:

```
root@ictad21h01:/mnt/beegfs# beegfs-ctl --getentryinfo myfile
Entry type: file
EntryID: 11D-624759A9-65
Metadata node: meta_01_tgt_0101 [ID: 101]
Stripe pattern details:
+ Type: RAID0
+ Chunksize: 1M
+ Number of storage targets: desired: 4; actual: 4
+ Storage targets:
  + 101 @ stor_01_tgt_0101 [ID: 101]
  + 102 @ stor_01_tgt_0101 [ID: 101]
  + 201 @ stor_02_tgt_0201 [ID: 201]
  + 202 @ stor_02_tgt_0201 [ID: 201]
```

Using two IPoIB subnets is a logical distinction. You can use a single physical InfiniBand subnet (storage network), if desired.

> ⓘ Multi-rail support was added in BeeGFS 7.3.0 to allow the use of multiple IB interfaces in a single IPoIB subnet. The design for the BeeGFS on NetApp solution was developed before the general availability of BeeGFS 7.3.0, and thus demonstrates the use of two IP subnets to use two IB interfaces on the BeeGFS clients. One advantage of the multiple IP subnet approach is eliminating the need to configure multihoming on BeeGFS client nodes (for more information, see BeeGFS RDMA support).

EF600 block node configuration

Block nodes are comprised of two active/active RAID controllers with shared access to the same set of drives. Typically, each controller owns half the volumes configured on the system, but can take over for the other controller as needed.

Multipathing software on the file nodes determines the active and optimized path to each volume and automatically moves to the alternate path in the event of a cable, adapter, or controller failure.

The following diagram shows the controller layout in EF600 block nodes.

[beegfs design image9]

To facilitate the shared-disk HA solution, volumes are mapped to both file nodes so that they can take over for each other as needed. The following diagram shows an example of how the BeeGFS service and preferred volume ownership is configured for maximum performance. The interface to the left of each BeeGFS service indicates the preferred interface that the clients and other services use to contact it.

[beegfs design image10]

In the previous example, clients and server services prefer to communicate with storage service 1 using interface i1b. Storage service 1 uses interface i1a as the preferred path to communicate with its volumes (storage_tgt_101, 102) on controller A of the first block node. This configuration makes use of the full bidirectional PCIe bandwidth available to the InfiniBand adapter and achieves better performance from a dual-port HDR InfiniBand adapter than would otherwise be possible with PCIe 4.0.

BeeGFS file node configuration

The BeeGFS file nodes are configured into a High-Availability (HA) cluster to facilitate failover of BeeGFS services between multiple file nodes.

The HA cluster design is based on two widely used Linux HA projects: Corosync for cluster membership and Pacemaker for cluster resource management. For more information, see Red Hat training for high-availability add-ons.

NetApp authored and extended several open cluster framework (OCF) resource agents to allow the cluster to intelligently start and monitor the BeeGFS resources.

BeeGFS HA clusters

Typically, when you start a BeeGFS service (with or without HA), a few resources must be in place:

- IP addresses where the service is reachable, typically configured by Network Manager.
- Underlying file systems used as the targets for BeeGFS to store data.

  These are typically defined in `/etc/fstab` and mounted by Systemd.

- A Systemd service responsible for starting BeeGFS processes when the other resources are ready.

  Without additional software, these resources start only on a single file node. Therefore, if the file node goes offline, a portion of the BeeGFS file system is inaccessible.

Because multiple nodes can start each BeeGFS service, Pacemaker must make sure each service and dependent resources are only running on one node at a time. For example, if two nodes try to start the same BeeGFS service, there is a risk of data corruption if they both try to write to the same files on the underlying target. To avoid this scenario, Pacemaker relies on Corosync to reliably keep the state of the overall cluster in sync across all nodes and establish quorum.

If a failure occurs in the cluster, Pacemaker reacts and restarts BeeGFS resources on another node. In some scenarios, Pacemaker might not be able to communicate with the original faulty node to confirm the resources are stopped. To verify that the node is down before restarting BeeGFS resources elsewhere, Pacemaker fences off the faulty node, ideally by removing power.

Many open-source fencing agents are available that enable Pacemaker to fence a node with a power distribution unit (PDU) or by using the server baseboard management controller (BMC) with APIs such as

Redfish.

When BeeGFS is running in an HA cluster, all BeeGFS services and underlying resources are managed by Pacemaker in resource groups. Each BeeGFS service and the resources it depends on, are configured into a resource group, which ensures resources are started and stopped in the correct order and collocated on the same node.

For each BeeGFS resource group, Pacemaker runs a custom BeeGFS monitoring resource that is responsible for detecting failure conditions and intelligently triggering failovers when a BeeGFS service is no longer accessible on a particular node.

The following figure shows the Pacemaker-controlled BeeGFS services and dependencies.

[beegfs design image11]

> So that multiple BeeGFS services of the same type are started on the same node, Pacemaker is configured to start BeeGFS services using the Multi Mode configuration method. For more information, see the BeeGFS documentation on Multi Mode.

Because BeeGFS services must be able to start on multiple nodes, the configuration file for each service (normally located at `/etc/beegfs`) is stored on one of the E-Series volumes used as the BeeGFS target for that service. This makes the configuration along with the data for a particular BeeGFS service accessible to all nodes that might need to run the service.

```
# tree stor_01_tgt_0101/ -L 2
stor_01_tgt_0101/
├── data
│   ├── benchmark
│   ├── buddymir
│   ├── chunks
│   ├── format.conf
│   ├── lock.pid
│   ├── nodeID
│   ├── nodeNumID
│   ├── originalNodeID
│   ├── targetID
│   └── targetNumID
└── storage_config
    ├── beegfs-storage.conf
    ├── connInterfacesFile.conf
    └── connNetFilterFile.conf
```

Design verification

The second-generation design for the BeeGFS on NetApp solution was verified using three building block configuration profiles.

The configuration profiles include the following:

- A single base building block, including BeeGFS management, metadata, and storage services.
- A BeeGFS metadata plus a storage building block.
- A BeeGFS storage-only building block.

The building blocks were attached to two Mellanox Quantum InfiniBand (MQM8700) switches. Ten BeeGFS clients were also attached to the InfiniBand switches and used to run synthetic benchmark utilities.

The following figure shows the BeeGFS configuration used to validate the BeeGFS on NetApp solution.

[beegfs design image12]

BeeGFS file striping

A benefit of parallel file systems is the ability to stripe individual files across multiple storage targets, which could represent volumes on the same or different underlying storage systems.

In BeeGFS, you can configure striping on a per-directory and per-file basis to control the number of targets used for each file and to control the chunksize (or block size) used for each file stripe. This configuration allows the file system to support different types of workloads and I/O profiles without the need for reconfiguring or restarting services. You can apply stripe settings using the `beegfs-ctl` command line tool or with applications that use the striping API. For more information, see the BeeGFS documentation for Striping and Striping API.

To achieve the best performance, stripe patterns were adjusted throughout testing, and the parameters used for each test are noted.

IOR bandwidth tests: Multiple clients

The IOR bandwidth tests used OpenMPI to run parallel jobs of the synthetic I/O generator tool IOR (available from HPC GitHub) across all 10 client nodes to one or more BeeGFS building blocks. Unless otherwise noted:

- All tests used direct I/O with a 1MiB transfer size.
- BeeGFS file striping was set to a 1MB chunksize and one target per file.

The following parameters were used for IOR with the segment count adjusted to keep the aggregate file size to 5TiB for one building block and 40TiB for three building blocks.

```
mpirun --allow-run-as-root --mca btl tcp -np 48 -map-by node -hostfile
10xnodes ior -b 1024k --posix.odirect -e -t 1024k -s 54613 -z -C -F -E -k
```

One BeeGFS base (management, metadata, and storage) building block

The following figure shows the IOR test results with a single BeeGFS base (management, metadata, and storage) building block.

[beegfs design image13]

BeeGFS metadata + storage building block

The following figure shows the IOR test results with a single BeeGFS metadata + storage building block.

[beegfs design image14]

BeeGFS storage-only building block

The following figure shows the IOR test results with a single BeeGFS storage-only building block.

[beegfs design image15]

Three BeeGFS building blocks

The following figure shows the IOR test results with three BeeGFS building blocks.

[beegfs design image16]

As expected, the performance difference between the base building block and the subsequent metadata + storage building block is negligible. Comparing the metadata + storage building block and a storage-only building block shows a slight increase in read performance due to the additional drives used as storage targets. However, there is no significant difference in write performance. To achieve higher performance, you can add multiple building blocks together to scale performance in a linear fashion.

IOR bandwidth tests: Single client

The IOR bandwidth test used OpenMPI to run multiple IOR processes using a single high-performance GPU server to explore the performance achievable to a single client.

This test also compares the reread behavior and performance of BeeGFS when the client is configured to use the Linux kernel page-cache (`tuneFileCacheType = native`) versus the default `buffered` setting.

The native caching mode uses the Linux kernel page-cache on the client, allowing reread operations to come from local memory instead of being retransmitted over the network.

The following diagram shows the IOR test results with three BeeGFS building blocks and a single client.

[beegfs design image17]

BeeGFS striping for these tests was set to a 1MB chunksize with eight targets per file.

Although write and initial read performance is higher using the default buffered mode, for workloads that reread the same data multiple times, a significant performance boost is seen with the native caching mode. This improved reread performance is important for workloads like deep learning that reread the same dataset multiple times across many epochs.

Metadata performance test

The Metadata performance tests used the MDTest tool (included as part of IOR) to measure the metadata performance of BeeGFS. The tests utilized OpenMPI to run parallel jobs across all ten client nodes.

The following parameters were used to run the benchmark test with the total number of processes scaled from 10 to 320 in step of 2x and with a file size of 4k.

```
mpirun -h 10xnodes –map-by node np $processes mdtest -e 4k -w 4k -i 3 -I
16 -z 3 -b 8 -u
```

Metadata performance was measured first with one then two metadata + storage building blocks to show how performance scales by adding additional building blocks.

One BeeGFS metadata + storage building block

The following diagram shows the MDTest results with one BeeGFS metadata + storage building blocks.

[beegfs design image18]

Two BeeGFS metadata + storage building blocks

The following diagram shows the MDTest results with two BeeGFS metadata + storage building blocks.

[beegfs design image19]

Functional validation

As part of validating this architecture, NetApp executed several functional tests including the following:

- Failing a single client InfiniBand port by disabling the switch port.
- Failing a single server InfiniBand port by disabling the switch port.
- Triggering an immediate server power off using the BMC.
- Gracefully placing a node in standby and failing over service to another node.
- Gracefully placing a node back online and failing back services to the original node.
- Powering off one of the InfiniBand switches using the PDU. All tests were performed while stress testing was in progress with the `sysSessionChecksEnabled: false` parameter set on the BeeGFS clients. No errors or disruption to I/O was observed.

> There is a known issue (see the Changelog) when BeeGFS client/server RDMA connections are disrupted unexpectedly, either through loss of the primary interface (as defined in `connInterfacesFile`) or a BeeGFS server failing; active client I/O can hang for up to ten minutes before resuming. This issue does not occur when BeeGFS nodes are gracefully placed in and out of standby for planned maintenance or if TCP is in use.

NVIDIA DGX A100 SuperPOD and BasePOD validation

NetApp validated a storage solution for NVIDIAs DGX A100 SuperPOD using a similar BeeGFS file system consisting of three building blocks with the metadata plus storage configuration profile applied. The qualification effort involved testing the solution described by this NVA with twenty DGX A100 GPU servers running a variety of storage, machine learning, and deep learning benchmarks. All storage certified for use in NVIDIA's DGX A100 SuperPOD is automatically certified for use in NVIDIA BasePOD architectures as well.

For more information, see NVIDIA DGX SuperPOD with NetApp and NVIDIA DGX BasePOD.

Sizing guidelines

The BeeGFS solution includes recommendations for performance and capacity sizing that were based on verification tests.

The objective with a building-block architecture is to create a solution that is simple to size by adding multiple building blocks to meet the requirements for a particular BeeGFS system. Using the guidelines below, you can estimate the quantity and types of BeeGFS building blocks that are needed to meet the requirements of your environment.

Keep in mind that these estimates are best-case performance. Synthetic benchmarking applications are written and utilized to optimize the use of underlying file systems in ways that real-world applications might not.

Performance sizing

The following table provides recommended performance sizing.

Configuration profile	1MiB reads	1MiB writes
Metadata + storage	62GiBps	21GiBps
Storage only	64GiBps	21GiBps

Metadata capacity sizing estimates are based on the "rule of thumb" that 500GB of capacity is sufficient for roughly 150 million files in BeeGFS. (For more information, see the BeeGFS documentation for System Requirements.)

The use of features like access control lists and the number of directories and files per directory also affect how quickly metadata space is consumed. Storage capacity estimates do account for usable drive capacity along with RAID 6 and XFS overhead.

Capacity sizing for metadata + storage building blocks

The following table provides recommended capacity sizing for metadata plus storage building blocks.

Drive size (2+2 RAID 1) metadata volume groups	Metadata capacity (number of files)	Drive size (8+2 RAID 6) storage volume groups	Storage capacity (file content)
1.92TB	1,938,577,200	1.92TB	51.77TB
3.84TB	3,880,388,400	3.84TB	103.55TB
7.68TB	8,125,278,000	7.68TB	216.74TB
15.3TB	17,269,854,000	15.3TB	460.60TB

When sizing metadata plus storage building blocks, you can reduce costs by using smaller drives for metadata volume groups versus storage volume groups.

Capacity sizing for storage-only building blocks

The following table provides rule-of-thumb capacity sizing for storage-only building blocks.

Drive size (10+2 RAID 6) storage volume groups	Storage capacity (file content)
1.92TB	59.89TB
3.84TB	119.80TB
7.68TB	251.89TB
15.3TB	538.55TB

The performance and capacity overhead of including the management service in the base (first) building block are minimal, unless global file locking is enabled.

Performance tuning

The BeeGFS solution includes recommendations for performance tuning that were based on verification tests.

Although BeeGFS provides reasonable performance out of the box, NetApp has developed a set of recommended tuning parameters to maximize performance. These parameters take into account the capabilities of the underlying E-Series block nodes and any special requirements needed to run BeeGFS in a shared-disk HA architecture.

Performance tuning for file nodes

The available tuning parameters that you can configure include the following:

1. **System settings in the UEFI/BIOS of file nodes.**
   To maximize performance, we recommend configuring the system settings on the server model you use as your file nodes. You configure the system settings when you set up your file nodes by using either the system setup (UEFI/BIOS) or the Redfish APIs provided by the baseboard management controller (BMC).

   The system settings vary depending on the server model you use as your file node. The settings must be manually configured based on the server model in use. To learn how to configure the system settings for the validated Lenovo SR665 file nodes, see Tune file node system settings for performance.

2. **Default settings for required configuration parameters.**
   The required configuration parameters affect how BeeGFS services are configured and how E-Series volumes (block devices) are formatted and mounted by Pacemaker. These required configuration parameters include the following:

   - BeeGFS Service configuration parameters

     You can override the default settings for the configuration parameters as needed. For the parameters that you can adjust for your specific workloads or use cases, see the BeeGFS service configuration parameters.

   - Volume formatting and mounting parameters are set to recommended defaults, and should only be adjusted for advanced use cases. The default values will do the following:

     - Optimize initial volume formatting based on the target type (such as management, metadata, or storage), along with the RAID configuration and segment size of the underlying volume.
     - Adjust how Pacemaker mounts each volume to ensure that changes are immediately flushed to E-series block nodes. This prevents data loss when file nodes fail with active writes in progress.

       For the parameters that you can adjust for your specific workloads or use cases, see the volume formatting and mounting configuration parameters.

3. **System settings in the Linux OS installed on the file nodes.**
   You can override the default Linux OS system settings when you create the Ansible inventory in step 4 of Create the Ansible inventory.

   The default settings were used to validate the BeeGFS on NetApp solution, but you can change them to adjust for your specific workloads or use cases. Some examples of the Linux OS system settings that you can change include the following:

   - I/O queues on E-Series block devices.

You can configure I/O queues on the E-Series block devices used as BeeGFS targets to:

- Adjust the scheduling algorithm based on the device type (NVMe, HDD, and so on).
- Increase the number of outstanding requests.
- Adjust request sizes.
- Optimize read ahead behavior.

◦ Virtual memory settings.

You can adjust virtual memory settings for optimal sustained streaming performance.

◦ CPU settings.

You can adjust the CPU frequency governor and other CPU configurations for maximum performance.

◦ Read request size.

You can increase the maximum read request size for Mellanox HCAs.

Performance tuning for block nodes

Based on the configuration profiles applied to a particular BeeGFS building block, the volume groups configured on the block nodes change slightly. For example, with a 24-drive EF600 block node:

- For the single base building block, including BeeGFS management, metadata, and storage services:
  - 1x 2+2 RAID 10 volume group for BeeGFS management and metadata services
  - 2x 8+2 RAID 6 volume groups for BeeGFS storage services
- For a BeeGFS metadata + storage building block:
  - 1x 2+2 RAID 10 volume group for BeeGFS metadata services
  - 2x 8+2 RAID 6 volume groups for BeeGFS storage services
- For BeeGFS storage only building block:
  - 2x 10+2 RAID 6 volume groups for BeeGFS storage services

As BeeGFS needs significantly less storage space for management and metadata versus storage, one option is to use smaller drives for the RAID 10 volume groups. Smaller drives should be populated in the outermost drive slots. For more information, see the deployment instructions.

These are all configured by the Ansible-based deployment, along with several other settings generally recommended to optimize performance/behavior including:

- Adjusting the global cache block size to 32KiB and adjusting demand-based cache flushing to 80%.
- Disabling autoload balancing (ensuring controller volume assignments stay as intended).
- Enabling read caching and disabling read-ahead caching.
- Enabling write caching with mirroring and requiring battery backup, so that caches persist through failure of a block node controller.
- Specifying the order that drives are assigned to volume groups, balancing I/O across available drive channels.

High capacity building block

The standard BeeGFS solution design is built with high performance workloads in mind. Customers looking for a high capacity use cases should observe the variations in design and performance characteristics outlined here.

Hardware and software configuration

Hardware and software configuration for the high capacity building block is standard except that the EF600 controllers should be replaced with a EF300 controllers with an option to attach between 1 and 7 IOM expansion trays with 60 drives each for each storage array, totaling 2 to 14 expansions trays per building block.

Customers deploying a high capacity building block design are likely to use only the base building block style configuration consisting of BeeGFS management, metadata, and storage services for each node. For cost efficiency, high capacity storage nodes should provision metadata volumes on the NVMe drives in the EF300 controller enclosure and should provision storage volumes to the NL-SAS drives in the expansion trays.

[high capacity rack diagram]

Sizing guidelines

These sizing guidelines assume high capacity building blocks are configured with one 2+2 NVMe SSD volume group for metadata in the base EF300 enclosure and 6x 8+2 NL-SAS volume groups per IOM expansion tray for storage.

Drive size (capacity HDDs)	Capacity per BB (1 Tray)	Capacity per BB (2 Trays)	Capacity per BB (3 Trays)	Capacity per BB (4 Trays)
4TB	439TB	878 TB	1317 TB	1756 TB
8 TB	878 TB	1756 TB	2634 TB	3512 TB
10 TB	1097 TB	2195 TB	3292 TB	4390 TB
12 TB	1317 TB	2634 TB	3951 TB	5268 TB
16 TB	1756 TB	3512 TB	5268 TB	7024 TB
18 TB	1975 TB	3951 TB	5927 TB	7902 TB

Deploy solution

Deployment overview

You can deploy BeeGFS on NetApp to validated file and block nodes using the second generation of NetApp’s BeeGFS building block design.

Ansible collections and roles

You deploy the BeeGFS on NetApp solution using Ansible, which is a popular IT automation engine used to automate application deployments. Ansible uses a series of files collectively known as an inventory, which models the BeeGFS file system you want to deploy.

Ansible allows companies such as NetApp to expand on built-in functionality using collections on Ansible

Galaxy (see NetApp E-Series BeeGFS collection). Collections include modules that perform some specific function or task (like create an E-Series volume) and include roles that can call multiple modules and other roles. This automated approach reduces the time needed to deploy the BeeGFS file system and the underlying HA cluster. In addition, it simplifies adding building blocks to expand the existing file systems.

For additional details, see Learn about the Ansible inventory.

> Because numerous steps are involved in deploying the BeeGFS on NetApp solution, NetApp does not support manually deploying the solution.

Configuration profiles for BeeGFS building blocks

The deployment procedures cover the following configuration profiles:

- One base building block that includes management, metadata, and storage services.
- A second building block that includes metadata and storage services.
- A third building block that includes only storage services.

These profiles demonstrate the full range of recommended configuration profiles for the NetApp BeeGFS building blocks. For each deployment, the number of metadata and storage building blocks or storage services-only building blocks may vary in the procedures, depending on capacity and performance requirements.

Overview of deployment steps

Deployment involves the following high-level tasks:

Hardware deployment

1. Physically assemble each building block.
2. Rack and cable hardware.
   For detailed procedures, see Deploy hardware.

Software deployment

1. Set up file and block nodes.
   - Configure BMC IPs on file nodes
   - Install a supported operating system and configure management networking on file nodes
   - Configure management IPs on block nodes
2. Set up an Ansible control node.
3. Tune system settings for performance.
4. Create the Ansible inventory.
5. Define Ansible inventory for BeeGFS building blocks.
6. Deploy BeeGFS using Ansible.
7. Configure BeeGFS clients.

The deploy procedures includes several examples where text needs to be copied to a file. Pay close attention to any inline comments denoted by “#” or “//” characters for anything that should or can be modified for a specific deployment. For example:

```
beegfs_ha_ntp_server_pools: # THIS IS AN EXAMPLE OF A COMMENT!
- "pool 0.pool.ntp.org iburst maxsources 3"
- "pool 1.pool.ntp.org iburst maxsources 3"
```

Derivative architectures with variations in deployment recommendations:

- High Capacity Building Block

Learn about the Ansible inventory

Before you begin deployment, make sure you understand how to use Ansible to configure and deploy the BeeGFS on NetApp solution using the second generation BeeGFS building block design.

The Ansible inventory defines the configuration for file and block nodes and represents the BeeGFS file system you want to deploy. The inventory includes hosts, groups, and variables describing the desired BeeGFS file system. Sample inventories can be downloaded from NetApp E-Series BeeGFS GitHub.

Ansible modules and roles

To apply the configuration described by the Ansible inventory, use the various Ansible modules and roles provided in the NetApp E-Series Ansible collection, in particular the BeeGFS HA 7.2 role (available from the NetApp E-Series BeeGFS GitHub) that deploys the end-to-end solution.

Each role in the NetApp E-Series Ansible collection is a complete end-to-end deployment of the BeeGFS on NetApp solution. The roles use the NetApp E-Series SANtricity, Host, and BeeGFS collections that allow you to configure the BeeGFS file system with HA (High Availability). You can then provision and map storage, and ensure the cluster storage is ready for use.

While in-depth documentation is provided with the roles, the deployment procedures describe how to use the role to deploy a NetApp Verified Architecture using the second generation BeeGFS building block design.

Although the deployment steps attempt to provide enough detail so that prior experience with Ansible is not a prerequisite, you should have some familiarity with Ansible and related terminology.

Inventory layout for a BeeGFS HA cluster

Use the Ansible inventory structure to define a BeeGFS HA cluster.

Anyone with previous Ansible experience should be aware that the BeeGFS HA role implements a custom method of discovering which variables (or facts) apply to each host. This is required to simplify building an Ansible inventory that describes resources that can run on multiple servers.

An Ansible inventory typically consists of the files in `host_vars` and `group_vars`, and an `inventory.yml` file that assigns hosts to specific groups (and potentially groups to other groups).

Don’t create any files with the content in this subsection, which is intended as an example only.

Although this configuration is predetermined based on the configuration profile, you should have a general understanding of how everything is laid out as an Ansible inventory, as follows:

```
# BeeGFS HA (High Availability) cluster inventory.
all:
  children:
    # Ansible group representing all block nodes:
    eseries_storage_systems:
      hosts:
        ictad22a01:
        ictad22a02:
        ictad22a03:
        ictad22a04:
        ictad22a05:
        ictad22a06:
    # Ansible group representing all file nodes:
    ha_cluster:
      children:
        meta_01:  # Group representing a metadata service with ID 01.
          hosts:
            file_node_01:  # This service is preferred on the first file
node.
            file_node_02:  # And can failover to the second file node.
        meta_02:  # Group representing a metadata service with ID 02.
          hosts:
            file_node_02:  # This service is preferred on the second file
node.
            file_node_01: # And can failover to the first file node.
```

For each service, an additional file is created under `group_vars` describing its configuration:

```
# meta_01 - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: 8015
  connMetaPortUDP: 8015
  tuneBindToNumaZone: 0
floating_ips:
  - i1b: <IP>/<SUBNET_MASK>
  - i4b: <IP>/<SUBNET_MASK>
# Type of BeeGFS service the HA resource group will manage.
beegfs_service: metadata # Choices: management, metadata, storage.
# What block node should be used to create a volume for this service:
beegfs_targets:
  ictad22a01:
    eseries_storage_pool_configuration:
      - name: beegfs_m1_m2_m5_m6
        raid_level: raid1
        criteria_drive_count: 4
        owning_controller: A
        common_volume_configuration:
          segment_size_kb: 128
        volumes:
          - size: 21.25
```

This layout allows the BeeGFS service, network, and storage configuration for each resource to be defined in a single place. Behind the scenes, the BeeGFS role aggregates the necessary configuration for each file and block node based on this inventory structure. For more information, see this blog post: NetApp accelerates deployment of HA for BeeGFS with Ansible.

> The BeeGFS numerical and string node ID for each service is automatically configured based on the group name. Thus, in addition to the general Ansible requirement for group names to be unique, groups representing a BeeGFS service must end in a number that is unique for the type of BeeGFS service the group represents. For example, meta_01 and stor_01 are allowed, but metadata_01 and meta_01 are not.

Review best practices

Follow the best practice guidelines when deploying the BeeGFS on NetApp solution.

Standard conventions

When physically assembling and creating the Ansible inventory file, follow these standard conventions (for more information, see Create the Ansible inventory).

- File node host names are sequentially numbered (h01-hN) with lower numbers at the top of the rack and higher numbers at the bottom.

  For example, the naming convention `[location][row][rack]hN` looks like: `ictad22h01`.

- Each block node is comprised of two storage controllers, each with their own host name.

  A storage array name is used to refer to the whole block storage system as part of an Ansible inventory. The storage array names should be sequentially numbered (a01 - aN), and the host names for individual controllers are derived from that naming convention.

  For example, a block node named `ictad22a01` typically can have host names configured for each controller like `ictad22a01-a` and `ictad22a01-b`, but be referred to in an Ansible inventory as `ictad22a01`.

- File and block nodes within the same building block share the same numbering scheme and are adjacent to each other in the rack with both file nodes on top and both block nodes directly underneath them.

  For example, in the first building block, file nodes h01 and h02 are both directly connected to block nodes a01 and a02. From top to bottom, the host names are h01, h02, a01, and a02.

- Building blocks are installed in sequential order based on their host names, so that lower numbered host names are at the top of the rack and higher numbered host names are at the bottom.

  The intent is to minimize the length of cable running to the top of rack switches, and to define a standard deployment practice to simplify troubleshooting. For datacenters where this is not allowed due to concerns around rack stability, the inverse is certainly allowed, populating the rack from the bottom up.

InfiniBand storage network configuration

Half the InfiniBand ports on each file node are used to connect directly to block nodes. The other half are connected to the InfiniBand switches and are used for BeeGFS client-server connectivity. When determining the size of the IPoIB subnets that are used for BeeGFS clients and servers, you must consider the anticipated growth of your compute/GPU cluster and BeeGFS file system. If you must deviate from the recommended IP ranges, keep in mind that each direct connect in a single building block has a unique subnet and there is no overlap with subnets used for client-server connectivity.

Direct connects

File and block nodes within each building block always use the IPs in the following table for their direct connects.

> This addressing scheme adheres to the following rule: The third octet is always odd or even, which depends on whether the file node is odd or even.

File node	IB port	IP address	Block node	IB port	Physical IP	Virtual IP
Odd (h1)	i1a	192.168.1.10	Odd (c1)	2a	192.168.1.100	192.168.1.101
Odd (h1)	i2a	192.168.3.10	Odd (c1)	2a	192.168.3.100	192.168.3.101
Odd (h1)	i3a	192.168.5.10	Even (c2)	2a	192.168.5.100	192.168.5.101
Odd (h1)	i4a	192.168.7.10	Even (c2)	2a	192.168.7.100	192.168.7.101
Even (h2)	i1a	192.168.2.10	Odd (c1)	2b	192.168.2.100	192.168.2.101
Even (h2)	i2a	192.168.4.10	Odd (c1)	2b	192.168.4.100	192.168.4.101
Even (h2)	i3a	192.168.6.10	Even (c2)	2b	192.168.6.100	192.168.6.101

File node	IB port	IP address	Block node	IB port	Physical IP	Virtual IP
Even (h2)	i4a	192.168.8.10	Even (c2)	2b	192.168.8.100	192.168.8.101

BeeGFS client-server IPoIB addressing scheme (two subnets)

To allow BeeGFS clients to use two InfiniBand ports, two IPoIB subnets are required with half the BeeGFS server services configured with a preferred IP on each subnet to ensure clients use two InfiniBand ports to maximize redundancy and possible throughput to the file system.

Each file node runs multiple BeeGFS server services (management, metadata, or storage). To allow each service to fail over independently to the other file node, each is configured with unique IP addresses that can float between both nodes (sometimes referred to as a logical interface or LIF).

While not mandatory, this deployment presumes the following IPoIB subnet ranges are in use for these connections and defines a standard addressing scheme that applies the following rules:

- The second octet is always odd or even, based on whether the file node InfiniBand port is odd or even.
- BeeGFS cluster IPs are always `xxx. 127.100.yyy` or `xxx.128.100.yyy`.

> In addition to the interface used for in-band OS management, additional interfaces can be used by Corosync for cluster heart beating and synchronization. This ensures that the loss of a single interface does not bring down the entire cluster.

- The BeeGFS Management service is always at `xxx.yyy.101.0` or `xxx.yyy.102.0`.
- BeeGFS Metadata services are always at `xxx.yyy.101.zzz` or `xxx.yyy.102.zzz`.
- BeeGFS Storage services are always at `xxx.yyy.103.zzz` or `xxx.yyy.103.zzz`.
- Addresses in the range `100.xxx.1.1` through `100.xxx.99.255` are reserved for clients.

Subnet A: 100.127.0.0/16

The following table provides the range for Subnet A: 100.127.0.0/16.

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP	i1b	100.127.100.1 - 100.127.100.255
BeeGFS Management	i1b	100.127.101.0
BeeGFS Metadata	i1b or i3b	100.127.101.1 - 100.127.101.255
BeeGFS Storage	i1b or i3b	100.127.103.1 - 100.127.103.255
BeeGFS Clients	(varies by client)	100.127.1.1 - 100.127.99.255

Subnet B: 100.128.0.0/16

The following table provides the range for Subnet B: 100.128.0.0/16.

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP	i4b	100.128.100.1 - 100.128.100.255
BeeGFS Management	i2b	100.128.102.0

Purpose	InfiniBand port	IP address or range
BeeGFS Metadata	i2b or i4b	100.128.102.1 - 100.128.102.255
BeeGFS Storage	i2b or i4b	100.128.104.1 - 100.128.104.255
BeeGFS Clients	(varies by client)	100.128.1.1 - 100.128.99.255

Not all IPs in the above ranges are used in this NetApp Verified Architecture. They demonstrate how IP addresses can be pre-allocated to allow easy file system expansion using a consistent IP addressing scheme. In this scheme, BeeGFS file nodes and service IDs correspond with the fourth octet of a well-known range of IPs. The file system could certainly scale beyond 255 nodes or services if needed.

Deploy hardware

Each building block consists of two validated x86 file nodes directly connected to two block nodes using HDR (200Gb) InfiniBand cables.

Because each building block includes two BeeGFS file nodes, a minimum of two building blocks is required to establish quorum in the failover cluster. While it is possible to configure a two-node cluster, there are limitations to this configuration that can prevent a successful failover to occur in some scenarios. If a two-node cluster is required, it is also possible to incorporate a third device as a tiebreaker, though that is not covered in this deployment procedure.

Unless otherwise noted, the following steps are identical for each building block in the cluster regardless of whether it is used to run BeeGFS metadata and storage services or storage services only.

Steps

1. Configure each BeeGFS file node with four PCIe 4.0 ConnectX-6 dual port Host Channel Adapters (HCAs) in InfiniBand mode and install them in PCIe slots 2, 3, 5, and 6.
2. Configure each BeeGFS block node with a dual-port 200Gb Host Interface Card (HIC) and install the HIC in each of its two storage controllers.

   Rack the building blocks so the two BeeGFS file nodes are above the BeeGFS block nodes. The following figure shows the correct hardware configuration for the BeeGFS building block (rear view).

   [buildingblock]

   The power supply configuration for production use cases should typically use redundant PSUs.

3. If needed, install the drives in each of the BeeGFS block nodes.
   a. If the building block will be used to run BeeGFS metadata and storage services and smaller drives are used for metadata volumes, verify that they are populated in the outermost drive slots, as shown in the figure below.
   b. For all building block configurations, if a drive enclosure is not fully populated, make sure that an equal number of drives are populated in slots 0–11 and 12–23 for optimal performance.

      [driveslots]

4. To cable the file and block nodes, use 1m InfiniBand HDR 200Gb direct attach copper cables, so that they match the topology shown in the figure below.

   [directattachcable]

   > The nodes across multiple building blocks are never directly connected. Each building block should be treated as a standalone unit and all communication between building blocks occurs through network switches.

5. Use 2m (or the appropriate length) InfiniBand HDR 200Gb direct attach copper cables to cable the remaining InfiniBand ports on each file node to the InfiniBand switches that will be used for the storage network.

   If there are redundant InfiniBand switches in use, cable the ports highlighted in light green in the following figure to different switches.

   [networkcable]

6. As needed, assemble additional building blocks following the same cabling guidelines.

   > The total number of building blocks that can be deployed in a single rack depends on the available power and cooling at each site.

Deploy software

Set up file nodes and block nodes

While most software configuration tasks are automated using the NetApp-provided Ansible collections, you must configure networking on the baseboard management controller (BMC) of each server and configure the management port on each controller.

Set up file nodes

1. Configure networking on the baseboard management controller (BMC) of each server.

   To learn how to configure networking for the validated Lenovo SR665 file nodes, see the Lenovo ThinkSystem Documentation.

   > A baseboard management controller (BMC), sometimes referred to as a service processor, is the generic name for the out-of-band management capability built into various server platforms that can provide remote access even if the operating system is not installed or accessible. Vendors typically market this functionality with their own branding. For example, on the Lenovo SR665, the BMC is referred to as the *Lenovo XClarity Controller (XCC)*.

2. Configure the system settings for maximum performance.

   You configure the system settings using the UEFI setup (formerly known as the BIOS) or by using the Redfish APIs provided by many BMCs. The system settings vary based on the server model used as a file node.

   To learn how to configure the system settings for the validated Lenovo SR665 file nodes, see Tune system settings for performance.

3. Install Red Hat 8.4 and configure the host name and network port used to manage the operating system including SSH connectivity from the Ansible control node.

   Do not configure IPs on any of the InfiniBand ports at this time.

   > While not strictly required, subsequent sections presume that host names are sequentially numbered (such as h1-hN) and refer to tasks that should be completed on odd versus even numbered hosts.

4. Use RedHat Subscription Manager to register and subscribe the system to allow installation of the required packages from the official Red Hat repositories and to limit updates to the supported version of Red Hat: `subscription-manager release --set=8.4`. For instructions, see How to register and subscribe a RHEL system and How to limit updates.
5. Enable the Red Hat repository containing the packages required for high availability.

   ```
   subscription-manager repo-override --repo=rhel-8-for-x86_64
   -highavailability-rpms --add=enabled:1
   ```

6. Update all ConnectX-6 HCA firmware to the version recommended in Technology requirements.

   This update can be done by downloading and running a version of the mlxup tool that bundles the recommended firmware. You can download this tool from mlxup - Update and Query Utility (user guide).

Set up block nodes

Set up the EF600 block nodes by configuring the management port on each controller.

1. Configure the management port on each EF600 controller.

   For instructions on configuring ports, go to the E-Series Documentation Center.

2. Optionally, set the storage array name for each system.

   Setting a name can make it easier to refer to each system in subsequent sections. For instructions on setting the array name, go to the E-Series Documentation Center.

   > While not strictly required, subsequent topics presume storage array names are sequentially numbered (such as c1 - cN) and refer to the steps that should be completed on odd versus even numbered systems.

Set up an Ansible control node

To set up an Ansible control node, you must identify a virtual or physical machine with network access to the management ports of all file and block nodes that can be used to configure the solution.

The following steps were tested on CentOS 8.4. For steps specific to your preferred Linux distribution, see the Ansible documentation.

1. Install Python 3.9 and ensure that the correct version of `pip` is installed.

```
sudo dnf install python3.9 -y
sudo dnf install python39-pip
sudo dnf install sshpass
```

2. Create symbolic links, ensuring that the Python 3.9 binary is used whenever `python3` or `python` is called.

```
sudo ln -sf /usr/bin/python3.9 /usr/bin/python3
sudo ln -sf /usr/bin/python3 /usr/bin/python
```

3. Install the Python packages required by the NetApp BeeGFS collections.

```
python3 -m pip install ansible cryptography netaddr
```

To ensure that you are installing a supported version of Ansible and all required Python packages, refer to the BeeGFS collection's Readme file. Supported versions are also noted in Technical requirements.

4. Verify that the correct versions of Ansible and Python are installed.

```
ansible --version
ansible [core 2.11.6]
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/local/lib/python3.9/site-
packages/ansible
  ansible collection location =
/root/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/local/bin/ansible
  python version = 3.9.2 (default, Mar 10 2021, 17:29:56) [GCC 8.4.1
20200928 (Red Hat 8.4.1-1)]
  jinja version = 3.0.2
  libyaml = True
```

5. Store the Ansible inventories used to describe the BeeGFS deployment in source control systems such as Git or BitBucket, and then install Git to interact with those systems.

```
sudo dnf install git -y
```

6. Set up passwordless SSH. This is the easiest way to allow Ansible to access the remote BeeGFS file nodes from the Ansible control node.

a. On the Ansible control node, if needed, generate a pair of public keys using `ssh-keygen`

b. Set up passwordless SSH to each of the file nodes using `ssh-copy-id <ip_or_hostname>`

   Do **not** set up passwordless SSH to the block nodes. This is neither supported nor required.

7. Use Ansible Galaxy to install the version of the BeeGFS collection listed in Technical requirements.

   This installation includes additional Ansible dependencies, such as the NetApp SANtricity software and host collections.

```
ansible-galaxy collection install netapp_eseries.beegfs:==3.0.1
```

Create the Ansible inventory

To define the configuration for file and block nodes, you create an Ansible inventory that represents the BeeGFS file system you want to deploy. The inventory includes hosts, groups, and variables describing the desired BeeGFS file system.

Step 1: Define configuration for all building blocks

Define the configuration that applies to all building blocks, regardless of which configuration profile you may apply to them individually.

Before you begin

- Use a source control system like BitBucket or Git to store the contents of the directory containing the Ansible inventory and playbook files.
- Create a `.gitignore` file that specifies the files that Git should ignore. This helps avoid storing large files in Git.

Steps

1. On your Ansible control node, identify a directory that you want to use to store the Ansible inventory and playbook files.

   Unless otherwise noted, all files and directories created in this step and following steps are created relative to this directory.

2. Create the following subdirectories:

   `host_vars`

   `group_vars`

   `packages`

Step 2: Define configuration for individual file and block nodes

Define the configuration that applies to individual file nodes and individual building block nodes.

1. Under `host_vars/`, create a file for each BeeGFS file node named `<HOSTNAME>.yml` with the following

content, paying special attention to the notes regarding content to populate for BeeGFS cluster IPs and host names ending in odd versus even numbers.

Initially, the file node interface names do match what is listed here (such as ib0 or ibs1f0). These custom names are configured in Step 4: Define configuration that should apply to all file nodes.

```
ansible_host: “<MANAGEMENT_IP>”
eseries_ipoib_interfaces:  # Used to configure BeeGFS cluster IP
addresses.
  - name: i1b
    address: 100.127.100. <NUMBER_FROM_HOSTNAME>/16
  - name: i4b
    address: 100.128.100. <NUMBER_FROM_HOSTNAME>/16
beegfs_ha_cluster_node_ips:
  - <MANAGEMENT_IP>
  - <i1b_BEEGFS_CLUSTER_IP>
  - <i4b_BEEGFS_CLUSTER_IP>
# NVMe over InfiniBand storage communication protocol information
# For odd numbered file nodes (i.e., h01, h03, ..):
eseries_nvme_ib_interfaces:
  - name: i1a
    address: 192.168.1.10/24
    configure: true
  - name: i2a
    address: 192.168.3.10/24
    configure: true
  - name: i3a
    address: 192.168.5.10/24
    configure: true
  - name: i4a
    address: 192.168.7.10/24
    configure: true
# For even numbered file nodes (i.e., h02, h04, ..):
# NVMe over InfiniBand storage communication protocol information
eseries_nvme_ib_interfaces:
  - name: i1a
    address: 192.168.2.10/24
    configure: true
  - name: i2a
    address: 192.168.4.10/24
    configure: true
  - name: i3a
    address: 192.168.6.10/24
    configure: true
  - name: i4a
    address: 192.168.8.10/24
    configure: true
```

If you have already deployed the BeeGFS cluster, you must stop the cluster before adding or changing statically configured IP addresses, including cluster IPs and IPs used for NVMe/IB. This is required so these changes take effect properly and do not disrupt cluster operations.

2. Under `host_vars/`, create a file for each BeeGFS block node named `<HOSTNAME>.yml` and populate it with the following content.

   Pay special attention to the notes regarding content to populate for storage array names ending in odd versus even numbers.

   For each block node, create one file and specify the `<MANAGEMENT_IP>` for one of the two controllers (usually A).

```
eseries_system_name: <STORAGE_ARRAY_NAME>
eseries_system_api_url: https://<MANAGEMENT_IP>:8443/devmgr/v2/
eseries_initiator_protocol: nvme_ib
# For odd numbered block nodes (i.e., a01, a03, ..):
eseries_controller_nvme_ib_port:
  controller_a:
    - 192.168.1.101
    - 192.168.2.101
    - 192.168.1.100
    - 192.168.2.100
  controller_b:
    - 192.168.3.101
    - 192.168.4.101
    - 192.168.3.100
    - 192.168.4.100
# For even numbered block nodes (i.e., a02, a04, ..):
eseries_controller_nvme_ib_port:
  controller_a:
    - 192.168.5.101
    - 192.168.6.101
    - 192.168.5.100
    - 192.168.6.100
  controller_b:
    - 192.168.7.101
    - 192.168.8.101
    - 192.168.7.100
    - 192.168.8.100
```

Step 3: Define configuration that should apply to all file and block nodes

You can define configuration common to a group of hosts under `group_vars` in a file name that corresponds with the group. This prevents repeating a shared configuration in multiple places.

About this task

Hosts can be in more than one group, and at runtime, Ansible chooses what variables apply to a particular host based on its variable precedence rules. (For more information on these rules, see the Ansible documentation for Using variables.)

Host-to-group assignments are defined in the actual Ansible inventory file, which is created towards the end of this procedure.

Step

In Ansible, any configuration you want to apply to all hosts can be defined in a group called `All`. Create the file `group_vars/all.yml` with the following content:

```
ansible_python_interpreter: /usr/bin/python3
beegfs_ha_ntp_server_pools:  # Modify the NTP server addressess if
desired.
  - "pool 0.pool.ntp.org iburst maxsources 3"
  - "pool 1.pool.ntp.org iburst maxsources 3"
```

Step 4: Define configuration that should apply to all file nodes

The shared configuration for file nodes is defined in a group called `ha_cluster`. The steps in this section build out the configuration that should be included in the `group_vars/ha_cluster.yml` file.

Steps

1. At the top of the file, define the defaults, including the password to use as the `sudo` user on the file nodes.

```
### ha_cluster Ansible group inventory file.
# Place all default/common variables for BeeGFS HA cluster resources
below.
### Cluster node defaults
ansible_ssh_user: root
ansible_become_password: <PASSWORD>
eseries_ipoib_default_hook_templates:
  - 99-multihoming.j2 # This is required when configuring additional
static IPs (for example cluster IPs) when multiple IB ports are in the
same IPoIB subnet.
# If the following options are specified, then Ansible will
automatically reboot nodes when necessary for changes to take effect:
eseries_common_allow_host_reboot: true
eseries_common_reboot_test_command: "systemctl --state=active,exited |
grep eseries_nvme_ib.service"
```

Particularly for production environments, do not store passwords in plain text. Instead, use the Ansible Vault (see Encrypting content with Ansible Vault) or the `--ask-become-pass` option when running the playbook. If the `ansible_ssh_user` is already `root`, then you can optionally omit the `ansible_become_password`.

2. Optionally, configure a name for the high-availability (HA) cluster and specify a user for intra-cluster communication.

   If you are modifying the private IP addressing scheme, you must also update the default `beegfs_ha_mgmtd_floating_ip`. This must match what you configure later for the BeeGFS Management resource group.

   Specify one or more emails that should receive alerts for cluster events using `beegfs_ha_alert_email_list`.

```
### Cluster information
beegfs_ha_firewall_configure: True
eseries_beegfs_ha_disable_selinux: True
eseries_selinux_state: disabled
# The following variables should be adjusted depending on the desired
configuration:
beegfs_ha_cluster_name: hacluster                  # BeeGFS HA cluster
name.
beegfs_ha_cluster_username: hacluster              # BeeGFS HA cluster
username.
beegfs_ha_cluster_password: hapassword             # BeeGFS HA cluster
username's password.
beegfs_ha_cluster_password_sha512_salt: randomSalt # BeeGFS HA cluster
username's password salt.
beegfs_ha_mgmtd_floating_ip: 100.127.101.0         # BeeGFS management
service IP address.
# Email Alerts Configuration
beegfs_ha_enable_alerts: True
beegfs_ha_alert_email_list: ["email@example.com"]  # E-mail recipient
list for notifications when BeeGFS HA resources change or fail.  Often a
distribution list for the team responsible for managing the cluster.
beegfs_ha_alert_conf_ha_group_options:
      mydomain: “example.com”
# The mydomain parameter specifies the local internet domain name. This
is optional when the cluster nodes have fully qualified hostnames (i.e.
host.example.com).
# Adjusting the following parameters is optional:
beegfs_ha_alert_timestamp_format: "%Y-%m-%d %H:%M:%S.%N" #%H:%M:%S.%N
beegfs_ha_alert_verbosity: 3
#  1) high-level node activity
#  3) high-level node activity + fencing action information + resources
(filter on X-monitor)
#  5) high-level node activity + fencing action information + resources
```

While seemingly redundant, `beegfs_ha_mgmtd_floating_ip` is important when you scale the BeeGFS file system beyond a single HA cluster. Subsequent HA clusters are deployed without an additional BeeGFS management service and point at the management service provided by the first cluster.

3. Configure a fencing agent. (For more details, see Configure fencing in a Red Hat High Availability cluster.) The following output shows examples for configuring common fencing agents. Choose one of these options.

   For this step, be aware that:

   - By default, fencing is enabled, but you need to configure a fencing *agent*.
   - The `<HOSTNAME>` specified in the `pcmk_host_map` or `pcmk_host_list` must correspond with the hostname in the Ansible inventory.
   - Running the BeeGFS cluster without fencing is not supported, particularly in production. This is largely to ensure when BeeGFS services, including any resource dependencies like block devices, fail over due to an issue, there is no risk of concurrent access by multiple nodes that result in file system corruption or other undesirable or unexpected behavior. If fencing must be disabled, refer to the general notes in the BeeGFS HA role's getting started guide and set `beegfs_ha_cluster_crm_config_options["stonith-enabled"]` to false in `ha_cluster.yml`.
   - There are multiple node-level fencing devices available, and the BeeGFS HA role can configure any fencing agent available in the Red Hat HA package repository. When possible, use a fencing agent that works through the uninterruptible power supply (UPS) or rack power distribution unit (rPDU), because some fencing agents such as the baseboard management controller (BMC) or other lights-out devices that are built into the server might not respond to the fence request under certain failure scenarios.

```
### Fencing configuration:
# OPTION 1: To enable fencing using APC Power Distribution Units
(PDUs):
beegfs_ha_fencing_agents:
 fence_apc:
   - ipaddr: <PDU_IP_ADDRESS>
     login: <PDU_USERNAME>
     passwd: <PDU_PASSWORD>
     pcmk_host_map:
"<HOSTNAME>:<PDU_PORT>,<PDU_PORT>;<HOSTNAME>:<PDU_PORT>,<PDU_PORT>"
# OPTION 2: To enable fencing using the Redfish APIs provided by the
Lenovo XCC (and other BMCs):
redfish: &redfish
  username: <BMC_USERNAME>
  password: <BMC_PASSWORD>
  ssl_insecure: 1 # If a valid SSL certificate is not available
specify “1”.
beegfs_ha_fencing_agents:
  fence_redfish:
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
# For details on configuring other fencing agents see
https://access.redhat.com/documentation/en-
us/red_hat_enterprise_linux/8/html/configuring_and_managing_high_avai
lability_clusters/assembly_configuring-fencing-configuring-and-
managing-high-availability-clusters.
```

4. Enable recommended performance tuning in the Linux OS.

   While many users find the default settings for the performance parameters generally work well, you can optionally change the default settings for a particular workload. As such, these recommendations are included in the BeeGFS role, but are not enabled by default to ensure users are aware of the tuning applied to their file system.

   To enable performance tuning, specify:

   ```
   ### Performance Configuration:
   beegfs_ha_enable_performance_tuning: True
   ```

5. (Optional) You can adjust the performance tuning parameters in the Linux OS as needed.

   For a comprehensive list of the available tuning parameters that you can adjust, see the Performance

Tuning Defaults section of the BeeGFS HA role in E-Series BeeGFS GitHub site. The default values can be overridden for all nodes in the cluster in this file or the `host_vars` file for an individual node.

6. To allow full 200Gb/HDR connectivity between block and file nodes, use the Open Subnet Manager (OpenSM) package from the Mellanox Open Fabrics Enterprise Distribution (MLNX_OFED). (The inbox `opensm` package does not support the necessary virtualization functionality.) Although deployment using Ansible is supported, you must first download the desired packages to the Ansible control node used to run the BeeGFS role.

   a. Using `curl` or your desired tool, download the packages for the version of OpenSM listed in the technology requirements section from Mellanox's website to the `packages/` directory. For example:

   ```
   curl -o packages/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
   0.1.54103.x86_64.rpm
   https://linux.mellanox.com/public/repo/mlnx_ofed/5.4-
   1.0.3.0/rhel8.4/x86_64/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
   0.1.54103.x86_64.rpm

   curl -o packages/opensm-5.9.0.MLNX20210617.c9f2ade-
   0.1.54103.x86_64.rpm
   https://linux.mellanox.com/public/repo/mlnx_ofed/5.4-
   1.0.3.0/rhel8.4/x86_64/opensm-5.9.0.MLNX20210617.c9f2ade-
   0.1.54103.x86_64.rpm
   ```

   b. Populate the following parameters in `group_vars/ha_cluster.yml` (adjust packages as needed):

```
### OpenSM package and configuration information
eseries_ib_opensm_allow_upgrades: true
eseries_ib_opensm_skip_package_validation: true
eseries_ib_opensm_rhel_packages: []
eseries_ib_opensm_custom_packages:
  install:
    - files:
        add:
          "packages/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm": "/tmp/"
          "packages/opensm-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm": "/tmp/"
    - packages:
        add:
          - /tmp/opensm-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
          - /tmp/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
  uninstall:
    - packages:
        remove:
          - opensm
          - opensm-libs
      files:
        remove:
          - /tmp/opensm-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
          - /tmp/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
eseries_ib_opensm_options:
  virt_enabled: "2"
```

7. Configure the `udev` rule to ensure consistent mapping of logical InfiniBand port identifiers to underlying PCIe devices.

   The `udev` rule must be unique to the PCIe topology of each server platform used as a BeeGFS file node.

   Use the following values for verified file nodes:

```
### Ensure Consistent Logical IB Port Numbering
# OPTION 1: Lenovo SR665 PCIe address-to-logical IB port mapping:
eseries_ipoib_udev_rules:
  "0000:41:00.0": i1a
  "0000:41:00.1": i1b
  "0000:01:00.0": i2a
  "0000:01:00.1": i2b
  "0000:a1:00.0": i3a
  "0000:a1:00.1": i3b
  "0000:81:00.0": i4a
  "0000:81:00.1": i4b

# Note: At this time no other x86 servers have been qualified.
Configuration for future qualified file nodes will be added here.
```

8. (Optional) Update the metadata target selection algorithm.

```
beegfs_ha_beegfs_meta_conf_ha_group_options:
  tuneTargetChooser: randomrobin
```

> In verification testing, `randomrobin` was typically used to ensure that test files were evenly distributed across all BeeGFS storage targets during performance benchmarking (for more information on benchmarking, see the BeeGFS site for Benchmarking a BeeGFS System). With real world use, this might cause lower numbered targets to fill up faster than higher numbered targets. Omitting `randomrobin` and just using the default `randomized` value has been shown to provide good performance while still utilizing all available targets.

Step 5: Define the configuration for the common block node

The shared configuration for block nodes is defined in a group called `eseries_storage_systems`. The steps in this section build out the configuration that should be included in the `group_vars/eseries_storage_systems.yml` file.

Steps

1. Set the Ansible connection to local, provide the system password, and specify if SSL certificates should be verified. (Normally, Ansible uses SSH to connect to managed hosts, but in the case of the NetApp E-Series storage systems used as block nodes, the modules use the REST API for communication.) At the top of the file, add the following:

```
### eseries_storage_systems Ansible group inventory file.
# Place all default/common variables for NetApp E-Series Storage Systems
here:
ansible_connection: local
eseries_system_password: <PASSWORD>
eseries_validate_certs: false
```

> Listing any passwords in plaintext is not recommended. Use Ansible vault or provide the `eseries_system_password` when running Ansible using `--extra-vars`.

2. To ensure optimal performance, install the versions listed for block nodes in Technical requirements.

   Download the corresponding files from the NetApp Support site. You can either upgrade them manually or include them in the `packages/` directory of the Ansible control node, and then populate the following parameters in `eseries_storage_systems.yml` to upgrade using Ansible:

```
# Firmware, NVSRAM, and Drive Firmware (modify the filenames as needed):
eseries_firmware_firmware: "packages/RCB_11.70.2_6000_61b1131d.dlp"
eseries_firmware_nvsram: "packages/N6000-872834-D06.dlp"
```

3. Download and install the latest drive firmware available for the drives installed in your block nodes from the NetApp Support site. You can either upgrade them manually or include them in the `packages/` directory of the Ansible control node, and then populate the following parameters in `eseries_storage_systems.yml` to upgrade using Ansible:

```
eseries_drive_firmware_firmware_list:
  - "packages/<FILENAME>.dlp"
eseries_drive_firmware_upgrade_drives_online: true
```

> Setting `eseries_drive_firmware_upgrade_drives_online` to `false` will speed up the upgrade, but should not be done until after BeeGFS is deployed. This is because that setting requires stopping all I/O to the drives before the upgrade to avoid application errors. Although performing an online drive firmware upgrade before configuring volumes is still quick, we recommend you always set this value to `true` to avoid issues later.

4. To optimize performance, make the following changes to the global configuration:

```
# Global Configuration Defaults
eseries_system_cache_block_size: 32768
eseries_system_cache_flush_threshold: 80
eseries_system_default_host_type: linux dm-mp
eseries_system_autoload_balance: disabled
eseries_system_host_connectivity_reporting: disabled
eseries_system_controller_shelf_id: 99 # Required.
```

5. To ensure optimal volume provisioning and behavior, specify the following parameters:

```
# Storage Provisioning Defaults
eseries_volume_size_unit: pct
eseries_volume_read_cache_enable: true
eseries_volume_read_ahead_enable: false
eseries_volume_write_cache_enable: true
eseries_volume_write_cache_mirror_enable: true
eseries_volume_cache_without_batteries: false
eseries_storage_pool_usable_drives:
"99:0,99:23,99:1,99:22,99:2,99:21,99:3,99:20,99:4,99:19,99:5,99:18,99:6,
99:17,99:7,99:16,99:8,99:15,99:9,99:14,99:10,99:13,99:11,99:12"
```

The value specified for `eseries_storage_pool_usable_drives` is specific to NetApp EF600 block nodes and controls the order in which drives are assigned to new volume groups. This ordering ensures that the I/O to each group is evenly distributed across backend drive channels.

Define Ansible inventory for BeeGFS building blocks

After defining the general Ansible inventory structure, define the configuration for each building block in the BeeGFS file system.

These deployment instructions demonstrate how to deploy a file system that consists of a base building block including management, metadata, and storage services; a second building block with metadata and storage services; and a third storage-only building block.

These steps are intended to show the full range of typical configuration profiles that you can use to configure NetApp BeeGFS building blocks to meet the requirements of the overall BeeGFS file system.

In this and subsequent sections, adjust as needed to build the inventory representing the BeeGFS file system that you want to deploy. In particular, use Ansible host names that represent each block or file node and the desired IP addressing scheme for the storage network to ensure it can scale to the number of BeeGFS file nodes and clients.

Step 1: Create the Ansible inventory file

Steps

1. Create a new `inventory.yml` file, and then insert the following parameters, replacing the hosts under `eseries_storage_systems` as needed to represent the block nodes in your deployment. The names should correspond with the name used for `host_vars/<FILENAME>.yml`.

```
# BeeGFS HA (High Availability) cluster inventory.
all:
  children:
    # Ansible group representing all block nodes:
    eseries_storage_systems:
      hosts:
        ictad22a01:
        ictad22a02:
        ictad22a03:
        ictad22a04:
        ictad22a05:
        ictad22a06:
    # Ansible group representing all file nodes:
    ha_cluster:
      children:
```

   In the subsequent sections, you will create additional Ansible groups under `ha_cluster` that represent the BeeGFS services you want to run in the cluster.

Step 2: Configure the inventory for a management, metadata, and storage building block

The first building block in the cluster or base building block must include the BeeGFS management service along with metadata and storage services:

Steps

1. In `inventory.yml`, populate the following parameters under `ha_cluster: children`:

```
      # ictad22h01/ictad22h02 HA Pair (mgmt/meta/storage building
block):
        mgmt:
          hosts:
            ictad22h01:
            ictad22h02:
        meta_01:
          hosts:
            ictad22h01:
            ictad22h02:
        stor_01:
          hosts:
            ictad22h01:
            ictad22h02:
        meta_02:
```

```
      hosts:
        ictad22h01:
        ictad22h02:
    stor_02:
      hosts:
        ictad22h01:
        ictad22h02:
    meta_03:
      hosts:
        ictad22h01:
        ictad22h02:
    stor_03:
      hosts:
        ictad22h01:
        ictad22h02:
    meta_04:
      hosts:
        ictad22h01:
        ictad22h02:
    stor_04:
      hosts:
        ictad22h01:
        ictad22h02:
    meta_05:
      hosts:
        ictad22h02:
        ictad22h01:
    stor_05:
      hosts:
        ictad22h02:
        ictad22h01:
    meta_06:
      hosts:
        ictad22h02:
        ictad22h01:
    stor_06:
      hosts:
        ictad22h02:
        ictad22h01:
    meta_07:
      hosts:
        ictad22h02:
        ictad22h01:
    stor_07:
      hosts:
        ictad22h02:
```

```
            ictad22h01:
        meta_08:
          hosts:
            ictad22h02:
            ictad22h01:
        stor_08:
          hosts:
            ictad22h02:
            ictad22h01:
```

2. Create the file `group_vars/mgmt.yml` and include the following:

```
# mgmt - BeeGFS HA Management Resource Group
# OPTIONAL: Override default BeeGFS management configuration:
# beegfs_ha_beegfs_mgmtd_conf_resource_group_options:
#  <beegfs-mgmt.conf:key>:<beegfs-mgmt.conf:value>
floating_ips:
  - i1b: 100.127.101.0/16
  - i2b: 100.128.102.0/16
beegfs_service: management
beegfs_targets:
  ictad22a01:
    eseries_storage_pool_configuration:
      - name: beegfs_m1_m2_m5_m6
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb:  128
        volumes:
          - size: 1
            owning_controller: A
```

3. Under `group_vars/`, create files for resource groups `meta_01` through `meta_08` using the following template, and then fill in the placeholder values for each service referencing the table below:

```
# meta_0X - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: <PORT>
  connMetaPortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET> # Example: i1b:192.168.120.1/16
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: metadata
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb:  128
        volumes:
          - size: 21.25 # SEE NOTE BELOW!
            owning_controller: <OWNING CONTROLLER>
```

The volume size is specified as a percentage of the overall storage pool (also referred to as a volume group). NetApp highly recommends that you leave some free capacity in each pool to allow room for SSD overprovisioning (for more information, see Introduction to NetApp EF600 array). The storage pool, `beegfs_m1_m2_m5_m6`, also allocates 1% of the pool's capacity for the management service. Thus, for metadata volumes in the storage pool, `beegfs_m1_m2_m5_m6`, when 1.92TB or 3.84TB drives are used, set this value to `21.25`; for 7.65TB drives, set this value to `22.25`; and for 15.3TB drives, set this value to `23.75`.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_01.yml	8015	i1b:100.127.101.1/16 i2b:100.128.102.1/16	0	ictad22a01	beegfs_m1_m2_m5_m6	A
meta_02.yml	8025	i2b:100.128.102.2/16 i1b:100.127.101.2/16	0	ictad22a01	beegfs_m1_m2_m5_m6	B
meta_03.yml	8035	i3b:100.127.101.3/16 i4b:100.128.102.3/16	1	ictad22a02	beegfs_m3_m4_m7_m8	A

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_04.yml	8045	i4b:100.128.102.4/16 i3b:100.127.101.4/16	1	ictad22a02	beegfs_m3_m4_m7_m8	B
meta_05.yml	8055	i1b:100.127.101.5/16 i2b:100.128.102.5/16	0	ictad22a01	beegfs_m1_m2_m5_m6	A
meta_06.yml	8065	i2b:100.128.102.6/16 i1b:100.127.101.6/16	0	ictad22a01	beegfs_m1_m2_m5_m6	B
meta_07.yml	8075	i3b:100.127.101.7/16 i4b:100.128.102.7/16	1	ictad22a02	beegfs_m3_m4_m7_m8	A
meta_08.yml	8085	i4b:100.128.102.8/16 i3b:100.127.101.8/16	1	ictad22a02	beegfs_m3_m4_m7_m8	B

4. Under `group_vars/`, create files for resource groups `stor_01` through `stor_08` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource
Groupbeegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 10
        common_volume_configuration:
          segment_size_kb: 512        volumes:
          - size: 21.50 # See note below!             owning_controller:
<OWNING CONTROLLER>
          - size: 21.50           owning_controller: <OWNING
CONTROLLER>
```

For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_01.yml	8013	i1b:100.127.103.1/16 i2b:100.128.104.1/16	0	ictad22a01	beegfs_s1_s2	A
stor_02.yml	8023	i2b:100.128.104.2/16 i1b:100.127.103.2/16	0	ictad22a01	beegfs_s1_s2	B
stor_03.yml	8033	i3b:100.127.103.3/16 i4b:100.128.104.3/16	1	ictad22a02	beegfs_s3_s4	A
stor_04.yml	8043	i4b:100.128.104.4/16 i3b:100.127.103.4/16	1	ictad22a02	beegfs_s3_s4	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_05.yml	8053	i1b:100.127.103.5/16 i2b:100.128.104.5/16	0	ictad22a01	beegfs_s5_s6	A
stor_06.yml	8063	i2b:100.128.104.6/16 i1b:100.127.103.6/16	0	ictad22a01	beegfs_s5_s6	B
stor_07.yml	8073	i3b:100.127.103.7/16 i4b:100.128.104.7/16	1	ictad22a02	beegfs_s7_s8	A
stor_08.yml	8083	i4b:100.128.104.8/16 i3b:100.127.103.8/16	1	ictad22a02	beegfs_s7_s8	B

Step 3: Configure the inventory for a Metadata + storage building block

These steps describe how to set up an Ansible inventory for a BeeGFS metadata + storage building block.

Steps

1. In `inventory.yml`, populate the following parameters under the existing configuration:

```
        meta_09:
          hosts:
            ictad22h03:
            ictad22h04:
        stor_09:
          hosts:
            ictad22h03:
            ictad22h04:
        meta_10:
          hosts:
            ictad22h03:
            ictad22h04:
        stor_10:
          hosts:
            ictad22h03:
            ictad22h04:
        meta_11:
          hosts:
            ictad22h03:
            ictad22h04:
```

```
        stor_11:
          hosts:
            ictad22h03:
            ictad22h04:
        meta_12:
          hosts:
            ictad22h03:
            ictad22h04:
        stor_12:
          hosts:
            ictad22h03:
            ictad22h04:
        meta_13:
          hosts:
            ictad22h04:
            ictad22h03:
        stor_13:
          hosts:
            ictad22h04:
            ictad22h03:
        meta_14:
          hosts:
            ictad22h04:
            ictad22h03:
        stor_14:
          hosts:
            ictad22h04:
            ictad22h03:
        meta_15:
          hosts:
            ictad22h04:
            ictad22h03:
        stor_15:
          hosts:
            ictad22h04:
            ictad22h03:
        meta_16:
          hosts:
            ictad22h04:
            ictad22h03:
        stor_16:
          hosts:
            ictad22h04:
            ictad22h03:
```

2. Under `group_vars/`, create files for resource groups `meta_09` through `meta_16` using the following

template, and then fill in the placeholder values for each service referencing the example:

```
# meta_0X - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: <PORT>
  connMetaPortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: metadata
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb: 128
        volumes:
          - size: 21.5 # SEE NOTE BELOW!
            owning_controller: <OWNING CONTROLLER>
```

For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_09.yml	8015	i1b:100.127.101.9/16 i2b:100.128.102.9/16	0	ictad22a03	beegfs_m9_m10_m13_m14	A
meta_10.yml	8025	i2b:100.128.102.10/16 i1b:100.127.101.10/16	0	ictad22a03	beegfs_m9_m10_m13_m14	B
meta_11.yml	8035	i3b:100.127.101.11/16 i4b:100.128.102.11/16	1	ictad22a04	beegfs_m11_m12_m15_m16	A
meta_12.yml	8045	i4b:100.128.102.12/16 i3b:100.127.101.12/16	1	ictad22a04	beegfs_m11_m12_m15_m16	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_13.yml	8055	i1b:100.127.101.13/16 i2b:100.128.102.13/16	0	ictad22a03	beegfs_m9_m10_m13_m14	A
meta_14.yml	8065	i2b:100.128.102.14/16 i1b:100.127.101.14/16	0	ictad22a03	beegfs_m9_m10_m13_m14	B
meta_15.yml	8075	i3b:100.127.101.15/16 i4b:100.128.102.15/16	1	ictad22a04	beegfs_m11_m12_m15_m16	A
meta_16.yml	8085	i4b:100.128.102.16/16 i3b:100.127.101.16/16	1	ictad22a04	beegfs_m11_m12_m15_m16	B

3. Under `group_vars/`, create files for resource groups `stor_09` through `stor_16` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource Group
beegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 10
        common_volume_configuration:
          segment_size_kb: 512        volumes:
          - size: 21.50 # See note below!
            owning_controller: <OWNING CONTROLLER>
          - size: 21.50            owning_controller: <OWNING
CONTROLLER>
```

For the correct size to use, see Recommended storage pool overprovisioning percentages..

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_09.yml	8013	i1b:100.127.103.9/16 i2b:100.128.104.9/16	0	ictad22a03	beegfs_s9_s10	A
stor_10.yml	8023	i2b:100.128.104.10/16 i1b:100.127.103.10/16	0	ictad22a03	beegfs_s9_s10	B
stor_11.yml	8033	i3b:100.127.103.11/16 i4b:100.128.104.11/16	1	ictad22a04	beegfs_s11_s12	A
stor_12.yml	8043	i4b:100.128.104.12/16 i3b:100.127.103.12/16	1	ictad22a04	beegfs_s11_s12	B
stor_13.yml	8053	i1b:100.127.103.13/16 i2b:100.128.104.13/16	0	ictad22a03	beegfs_s13_s14	A
stor_14.yml	8063	i2b:100.128.104.14/16 i1b:100.127.103.14/16	0	ictad22a03	beegfs_s13_s14	B
stor_15.yml	8073	i3b:100.127.103.15/16 i4b:100.128.104.15/16	1	ictad22a04	beegfs_s15_s16	A
stor_16.yml	8083	i4b:100.128.104.16/16 i3b:100.127.103.16/16	1	ictad22a04	beegfs_s15_s16	B

Step 4: Configure the inventory for a storage-only building block

These steps describe how to set up an Ansible inventory for a BeeGFS storage-only building block. The major difference between setting up the configuration for a metadata + storage versus a storage-only building block is the omission of all metadata resource groups and changing `criteria_drive_count` from 10 to 12 for each storage pool.

Steps

1. In `inventory.yml`, populate the following parameters under the existing configuration:

```
# ictad22h05/ictad22h06 HA Pair (storage only building block):
  stor_17:
    hosts:
      ictad22h05:
      ictad22h06:
  stor_18:
    hosts:
      ictad22h05:
      ictad22h06:
  stor_19:
    hosts:
      ictad22h05:
      ictad22h06:
  stor_20:
    hosts:
      ictad22h05:
      ictad22h06:
  stor_21:
    hosts:
      ictad22h06:
      ictad22h05:
  stor_22:
    hosts:
      ictad22h06:
      ictad22h05:
  stor_23:
    hosts:
      ictad22h06:
      ictad22h05:
  stor_24:
    hosts:
      ictad22h06:
      ictad22h05:
```

2. Under `group_vars/`, create files for resource groups `stor_17` through `stor_24` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource Group
beegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 12
        common_volume_configuration:
          segment_size_kb: 512
        volumes:
          - size: 21.50 # See note below!
            owning_controller: <OWNING CONTROLLER>
          - size: 21.50
            owning_controller: <OWNING CONTROLLER>
```

For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_17.yml	8013	i1b:100.127.103.17/16 i2b:100.128.104.17/16	0	ictad22a05	beegfs_s17_s18	A
stor_18.yml	8023	i2b:100.128.104.18/16 i1b:100.127.103.18/16	0	ictad22a05	beegfs_s17_s18	B
stor_19.yml	8033	i3b:100.127.103.19/16 i4b:100.128.104.19/16	1	ictad22a06	beegfs_s19_s20	A
stor_20.yml	8043	i4b:100.128.104.20/16 i3b:100.127.103.20/16	1	ictad22a06	beegfs_s19_s20	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_21.yml	8053	i1b:100.127.103.21/16 i2b:100.128.104.21/16	0	ictad22a05	beegfs_s21_s22	A
stor_22.yml	8063	i2b:100.128.104.22/16 i1b:100.127.103.22/16	0	ictad22a05	beegfs_s21_s22	B
stor_23.yml	8073	i3b:100.127.103.23/16 i4b:100.128.104.23/16	1	ictad22a06	beegfs_s23_s24	A
stor_24.yml	8083	i4b:100.128.104.24/16 i3b:100.127.103.24/16	1	ictad22a06	beegfs_s23_s24	B

Deploy BeeGFS

Deploying and managing the configuration involves running one or more playbooks that contain the tasks Ansible needs to execute and bring the overall system to the desired state.

While all tasks can be included in a single playbook, for complex systems, this quickly becomes unwieldy to manage. Ansible allows you to create and distribute roles as a way of packaging reusable playbooks and related content (for example: default variables, tasks, and handlers). For more information, see the Ansible documentation for Roles.

Roles are often distributed as part of an Ansible collection containing related roles and modules. Thus, these playbooks primarily just import several roles distributed in the various NetApp E-Series Ansible collections.

> ⓘ Currently, at least two building blocks (four file nodes) are required to deploy BeeGFS, unless a separate quorum device is configured as a tiebreaker to mitigate any issues when establishing quorum with a two-node cluster.

Steps

1. Create a new `playbook.yml` file and include the following:

```
# BeeGFS HA (High Availability) cluster playbook.
- hosts: eseries_storage_systems
  gather_facts: false
  collections:
    - netapp_eseries.santricity
  tasks:
    - name: Configure NetApp E-Series block nodes.
```

```
      import_role:
        name: nar_santricity_management
- hosts: all
  any_errors_fatal: true
  gather_facts: false
  collections:
    - netapp_eseries.beegfs
  pre_tasks:
    - name: Ensure a supported version of Python is available on all
file nodes.
      block:
        - name: Check if python is installed.
          failed_when: false
          changed_when: false
          raw: python --version
          register: python_version
        - name: Check if python3 is installed.
          raw: python3 --version
          failed_when: false
          changed_when: false
          register: python3_version
          when: 'python_version["rc"] != 0 or (python_version["stdout"]
| regex_replace("Python ", "")) is not version("3.0", ">=")'
        - name: Install python3 if needed.
          raw: |
            id=$(grep "^ID=" /etc/*release* | cut -d= -f 2 | tr -d '"')
            case $id in
              ubuntu) sudo apt install python3 ;;
              rhel|centos) sudo yum -y install python3 ;;
              sles) sudo zypper install python3 ;;
            esac
          args:
            executable: /bin/bash
          register: python3_install
          when: python_version['rc'] != 0 and python3_version['rc'] != 0
          become: true
        - name: Create a symbolic link to python from python3.
          raw: ln -s /usr/bin/python3 /usr/bin/python
          become: true
          when: python_version['rc'] != 0
      when: inventory_hostname not in
groups[beegfs_ha_ansible_storage_group]
    - name: Verify any provided tags are supported.
      fail:
        msg: "{{ item }} tag is not a supported BeeGFS HA tag. Rerun
your playbook command with --list-tags to see all valid playbook tags."
```

```
      when: 'item not in ["all", "storage", "beegfs_ha",
"beegfs_ha_package", "beegfs_ha_configure",
"beegfs_ha_configure_resource", "beegfs_ha_performance_tuning",
"beegfs_ha_backup", "beegfs_ha_client"]'
      loop: "{{ ansible_run_tags }}"
  tasks:
    - name: Verify before proceeding.
      pause:
        prompt: "Are you ready to proceed with running the BeeGFS HA
role? Depending on the size of the deployment and network performance
between the Ansible control node and BeeGFS file and block nodes this
can take awhile (10+ minutes) to complete."
    - name: Verify the BeeGFS HA cluster is properly deployed.
      import_role:
        name: beegfs_ha_7_2
```

This playbook runs a few `pre_tasks` that verify Python 3 is installed on the file nodes and check that the Ansible tags provided are supported.

2. Use the `ansible-playbook` command with the inventory and playbook files when you're ready to deploy BeeGFS.

   The deployment will run all `pre_tasks`, and then prompt for user confirmation before proceeding with the actual BeeGFS deployment.

   Run the following command, adjusting the number of forks as needed (see the note below):

   ```
   ansible-playbook -i inventory.yml playbook.yml --forks 20
   ```

   Especially for larger deployments, overriding the default number of forks (5) using the `forks` parameter is recommended to increase the number of hosts that Ansible configures in parallel. (For more information, see Ansible Performance Tuning and Controlling playbook execution.) The maximum value setting depends on the processing power available on the Ansible control node. The above example of 20 was run on a virtual Ansible control node with 4 CPUs (Intel® Xeon® Gold 6146 CPU @ 3.20GHz).

   Depending on the size of the deployment and network performance between the Ansible control node and BeeGFS file and block nodes, deployment time might vary.

Configure BeeGFS clients

You must install and configure the BeeGFS client on any hosts that need access to the BeeGFS file system, such as compute or GPU nodes. For this task, you can use Ansible and the BeeGFS collection.

Steps

1. If needed, set up passwordless SSH from the Ansible control node to each of the hosts you want to configure as BeeGFS clients:

   ```
   ssh-copy-id <user>@<HOSTNAME_OR_IP>
   ```

2. Under `host_vars/`, create a file for each BeeGFS client named `<HOSTNAME>.yml` with the following content, filling in the placeholder text with the correct information for your environment:

   ```
   # BeeGFS Client
   ansible_host: <MANAGEMENT_IP>
   # OPTIONAL: If you want to use the NetApp E-Series Host Collection's
   IPoIB role to configure InfiniBand interfaces for clients to connect to
   BeeGFS file systems:
   eseries_ipoib_interfaces:
     - name: <INTERFACE>
       address: <IP>/<SUBNET_MASK> # Example: 100.127.1. 1/16
     - name: <INTERFACE>0
       address: <IP>/<SUBNET_MASK>
   ```

   Currently, two InfiniBand interfaces must be configured on each client, one in each of the two storage IPoIB subnets. If using the example subnets and recommended ranges for each BeeGFS service listed here, clients should have one interface configured in the range of `100.127.1. 0` to `100.127.99.255` and the other in `100.128.1. 0` to `100.128. 99.255`.

3. Create a new file `client_inventory.yml`, and then populate the following parameters at the top:

   ```
   # BeeGFS client inventory.
   all:
     vars:
       ansible_ssh_user: <USER> # This is the user Ansible should use to
   connect to each client.
       ansible_become_password: <PASSWORD> # This is the password Ansible
   will use for privilege escalation, and requires the ansible_ssh_user be
   root, or have sudo privileges.
   The defaults set by the BeeGFS HA role are based on the testing
   performed as part of this NetApp Verified Architecture and differ from
   the typical BeeGFS client defaults.
   ```

   Do not store passwords in plain text. Instead, use the Ansible Vault (see the Ansible documentation for Encrypting content with Ansible Vault) or use the `--ask-become-pass` option when running the playbook.

4. In the `client_inventory.yml` file, list all hosts that should be configured as BeeGFS clients under the `beegfs_clients` group, and then specify any additional configuration required to build the BeeGFS client kernel module.

```
  children:
    # Ansible group representing all BeeGFS clients:
    beegfs_clients:
      hosts:
        ictad21h01:
        ictad21h02:
        ictad21h03:
        ictad21h04:
        ictad21h05:
        ictad21h06:
        ictad21h07:
        ictad21h08:
        ictad21h09:
        ictad21h10:
      vars:
        # OPTION 1: If you're using the Mellanox OFED drivers and they
are already installed:
        eseries_ib_skip: True # Skip installing inbox drivers when using
the IPoIB role.
        beegfs_client_ofed_enable: True
        beegfs_client_ofed_include_path:
"/usr/src/ofa_kernel/default/include"
        # OPTION 2: If you're using inbox IB/RDMA drivers and they are
already installed:
        eseries_ib_skip: True # Skip installing inbox drivers when using
the IPoIB role.
        # OPTION 3: If you want to use inbox IB/RDMA drivers and need
them installed/configured.
        eseries_ib_skip: False # Default value.
        beegfs_client_ofed_enable: False # Default value.
```

When using the Mellanox OFED drivers, make sure that `beegfs_client_ofed_include_path` points to the correct "header include path" for your Linux installation. For more information, see the BeeGFS documentation for RDMA support.

5. In the `client_inventory.yml` file, list the BeeGFS file systems you want mounted at the bottom of any previously defined `vars`.

```
        beegfs_client_mounts:
          - sysMgmtdHost: 100.127.101.0 # Primary IP of the BeeGFS
management service.
            mount_point: /mnt/beegfs    # Path to mount BeeGFS on the
client.
            connInterfaces:
              - <INTERFACE> # Example: ibs4f1
              - <INTERFACE>
            beegfs_client_config:
              # Maximum number of simultaneous connections to the same
node.
              connMaxInternodeNum: 128 # BeeGFS Client Default: 12
              # Allocates the number of buffers for transferring IO.
              connRDMABufNum: 36 # BeeGFS Client Default: 70
              # Size of each allocated RDMA buffer
              connRDMABufSize: 65536 # BeeGFS Client Default: 8192
              # Required when using the BeeGFS client with the shared-
disk HA solution.
              # This does require BeeGFS targets be mounted in the
default "sync" mode.
              # See the documentation included with the BeeGFS client
role for full details.
              sysSessionChecksEnabled: false
```

The `beegfs_client_config` represents the settings that were tested. See the documentation included with the `netapp_eseries.beegfs` collection's `beegfs_client` role for a comprehensive overview of all options. This includes details around mounting multiple BeeGFS file systems or mounting the same BeeGFS file system multiple times.

6. Create a new `client_playbook.yml` file, and then populate the following parameters:

```
# BeeGFS client playbook.
- hosts: beegfs_clients
  any_errors_fatal: true
  gather_facts: true
  collections:
    - netapp_eseries.beegfs
    - netapp_eseries.host
  tasks:
    - name: Ensure IPoIB is configured
      import_role:
        name: ipoib
    - name: Verify the BeeGFS clients are configured.
      import_role:
        name: beegfs_client
```

Omit importing the `netapp_eseries.host` collection and `ipoib` role if you have already installed the required IB/RDMA drivers and configured IPs on the appropriate IPoIB interfaces.

7. To install and build the client and mount BeeGFS, run the following command:

```
ansible-playbook -i client_inventory.yml client_playbook.yml
```

8. Before you place the BeeGFS file system in production, we **strongly** recommend that you log in to any clients and run `beegfs-fsck --checkfs` to ensure that all nodes are reachable and there are no issues reported.

Scale beyond five building blocks

You can configure Pacemaker and Corosync to scale beyond five building blocks (10 file nodes). However, there are drawbacks to larger clusters, and eventually Pacemaker and Corosync do impose a maximum of 32 nodes.

NetApp has only tested BeeGFS HA clusters for up to 10 nodes; scaling individual clusters beyond this limit is not recommended or supported. However, BeeGFS file systems still need to scale far beyond 10 nodes, and NetApp has accounted for this in the BeeGFS on NetApp solution.

By deploying multiple HA clusters containing a subset of the building blocks in each file system, you can scale the overall BeeGFS file system independently of any recommended or hard limits on the underlying HA clustering mechanisms. In this scenario, do the following:

- Create a new Ansible inventory representing the additional HA cluster(s), and then omit configuring another management service. Instead, point the `beegfs_ha_mgmtd_floating_ip` variable in each additional cluster `ha_cluster.yml` to the IP for the first BeeGFS management service.
- When adding additional HA clusters to the same file system, ensure the following:
  - The BeeGFS node IDs are unique.

    - The file names corresponding with each service under `group_vars` is unique across all clusters.
    - The BeeGFS client and server IP addresses are unique across all clusters.
    - The first HA cluster containing the BeeGFS management service is running before trying to deploy or update additional clusters.
- Maintain inventories for each HA cluster separately in their own directory tree.

  Trying to mix the inventory files for multiple clusters in one directory tree might cause issues with how the BeeGFS HA role aggregates the configuration applied to a particular cluster.

> There is no requirement that each HA cluster scale to five building blocks before creating a new one. In many cases, using fewer building blocks per cluster is easier to manage. One approach is to configure the building blocks in each single rack as an HA cluster.

Recommended storage pool overprovisioning percentages

When following the standard four volumes per storage pool configuration for second generation building blocks, refer to the following table.

This table provides recommended percentages to use as the volume size in the `eseries_storage_pool_configuration` for each BeeGFS metadata or storage target:

Drive size	Size
1.92TB	18
3.84TB	21.5
7.68TB	22.5
15.3TB	24

> The above guidance does not apply to the storage pool containing the management service, which should reduce the sizes above by .25% to allocate 1% of the storage pool for management data.

To understand how these values were determined, see TR-4800: Appendix A: Understanding SSD endurance and overprovisioning.

High capacity building block

The standard BeeGFS solution deployment guide outlines procedures and recommendations for high performance workload requirements. Customers looking to meet high capacity requirements should observe the variations in deployment and recommendations outlined here.

[high capacity rack diagram]

Controllers

For high capacity building blocks EF600 controllers should be replaced with EF300 controllers, each with a Cascade HIC installed for SAS expansion. Each block node will have a minimal number of NVMe SSDs

populated in the array enclosure for BeeGFS metadata storage and will be attached to expansion shelves populated with NL-SAS HDDs for BeeGFS storage volumes.

File node to Block node configuration remains the same.

Drive placement

A minimum of 4 NVMe SSD’s are required in each block node for BeeGFS metadata storage. These drives should be placed in the outermost slots of the enclosure.

[high capacity drive slots diagram]

Expansion trays

The high capacity building block can be sized with 1-7, 60 drive expansion trays per storage array.

For instructions to cable each expansion tray, refer to EF300 cabling for drive shelves.

Use custom architectures

Overview and requirements

Use any NetApp E/EF-Series storage systems as BeeGFS block nodes and x86 servers as BeeGFS file nodes when deploying BeeGFS high availability clusters using Ansible.

Definitions for terminology used throughout this section can be found on the terms and concepts page.

Introduction

While NetApp verified architectures provide predefined reference configurations and sizing guidance, some customers and partners may prefer to design custom architectures better suited to particular requirements or hardware preferences. One of the primary benefits of choosing BeeGFS on NetApp is the ability to deploy BeeGFS shared-disk HA clusters using Ansible, simplifying cluster management and improving reliability with NetApp authored HA components. The deployment of custom BeeGFS architectures on NetApp is still done using Ansible, maintaining an appliance-like approach on a flexible range of hardware.

This section outlines the general steps needed to deploy BeeGFS file systems on NetApp hardware and use of Ansible to configure BeeGFS file systems. For details on best practices surrounding the design of BeeGFS file systems and optimized examples please refer to the NetApp verified architectures section.

Deployment Overview

Generally deploying a BeeGFS file system involves the following steps:

- Initial set up:
  - Install/cable hardware.
  - Set up file and block nodes.
  - Set up an Ansible control node.
- Define the BeeGFS file system as an Ansible inventory.
- Run Ansible against file and block nodes to deploy BeeGFS.
  - Optionally to set up clients and mount BeeGFS.

Subsequent sections will cover these steps in more detail.

Ansible handles all software provisioning and configuration tasks including:

- Creating/mapping volumes on block nodes.
- Formatting/tuning volumes on file nodes.
- Installing/configuring software on file nodes.
- Establishing the HA cluster and configuring BeeGFS resources and file system services.

Requirements

Support for BeeGFS in Ansible is released on Ansible Galaxy as a collection of roles and modules that automate the end-to-end deployment and management of BeeGFS HA clusters.

BeeGFS itself is versioned following a <major>.<minor>.<patch> versioning scheme and the collection maintains roles for each supported <major>.<minor> version of BeeGFS, for example BeeGFS 7.2 or BeeGFS 7.3. As updates to the collection are released the patch version in each role will be updated to point at the latest available BeeGFS version for that release branch (example: 7.2.8). Each version of the collection is also tested and supported with specific Linux distributions and versions, currently Red Hat for file nodes, and RedHat and Ubuntu for clients. Running other distributions is not supported, and running other versions (especially other major versions) is not recommended.

Ansible Control Node

This node will contain the inventory and playbooks used to manage BeeGFS. It requires:

- Ansible 6.x (ansible-core 2.13)
- Python 3.6 (or later)
- Python (pip) packages: ipaddr and netaddr

It is also recommend you setup passwordless SSH from the control node to all BeeGFS file nodes and clients.

BeeGFS File Nodes

File nodes must run RedHat 8.4 and have access to the HA repository containing required packages (pacemaker, corosync, fence-agents-all, resource-agents). For example the following command can be executed to enable the appropriate repository on RedHat 8:

```
subscription-manager repo-override repo=rhel-8-for-x86_64-
highavailability-rpms --add=enabled:1
```

BeeGFS Client Nodes

A BeeGFS client Ansible role is available to install the BeeGFS client package and manage BeeGFS mount(s). This role has been tested with RedHat 8.4 and Ubuntu 22.04.

If you are not using Ansible to setup the BeeGFS client and mount BeeGFS, any BeeGFS supported Linux distribution and kernel can be used.

Initial Set Up

Install and Cable Hardware

Steps needed to install and cable hardware used to run BeeGFS on NetApp.

Plan the Installation

Each BeeGFS file system will consist of some number of file nodes running BeeGFS services using backend storage provided by some number of block nodes. The file nodes are configured into one or more high availability clusters to provide fault tolerance for BeeGFS services. Each block node is a already an

active/active HA pair. The minimum number of supported file nodes in each HA cluster is three, and the maximum number of supported file nodes in each cluster is ten. BeeGFS file systems can scale beyond ten node by deploying multiple independent HA clusters that work together to provide a single file system namespace.

Commonly each HA cluster is deployed as a series of "building blocks" where some number of file nodes (x86 servers) are directly connected to some number of block nodes (typically E-Series storage systems). This configuration creates an asymmetrical cluster, where BeeGFS services are only able to run on certain file nodes that have access to the backend block storage used for the BeeGFS targets. The balance of file-to-block nodes in each building block and the storage protocol in use for the direct-connects depend on the requirements of a particular installation.

An alternative HA cluster architecture uses a storage fabric (also known as a storage area network or SAN) between the file and block nodes to establish a symmetrical cluster. This allows BeeGFS services to run on any file node in a particular HA cluster. As generally symmetrical clusters are not as cost effective due to the extra SAN hardware, this documentation presumes use of an asymmetrical cluster deployed as a series of one or more building blocks.

Ensure the desired file system architecture for a particular BeeGFS deployment is well understood before proceeding with the installation.

Rack Hardware

When planning the installation it is important all equipment in each building block is racked in adjacent rack units. Best practice is for file nodes to be racked immediately above block nodes in each building block. Follow the documentation for the model(s) of file and block nodes you are using as you install rails and hardware into the rack.

Example of a single building block:

[example building block]

Example of a large BeeGFS installation where there are multiple building blocks in each HA cluster, and multiple HA clusters in the file system:

[example BeeGFS deployment]

Cable File and Block Nodes

Typically you will direct-connect the HIC ports of the E-Series block nodes to the designated host channel adapter (for InfiniBand protocols) or host bus adapter (for fibre channel and other protocols) ports of the file nodes. The exact way to establish these connections will depend on the desired file system architecture, here is an example based on the second-generation BeeGFS on NetApp verified architecture:

[example BeeGFS file to block node cabling]

Cable File Nodes to the Client Network

Each file node will have some number of InfiniBand or Ethernet ports designated for BeeGFS client traffic. Depending on the architecture each file node will have one or more connections to a high performance client/storage network, potentially to multiple switches for redundancy and increased bandwidth. Here is an example of client cabling using redundant network switches, where ports highlighted in dark green versus light green are connected to separate switches:

[example BeeGFS client cabling]

Connect Management Networking and Power

Establish any network connections needed for in-band and out-of-band network.

Connect all power supplies ensuring each file and block node has connections to multiple power distribution units for redundancy (if available).

Set Up File and Block Nodes

Manual steps required to set up file and block nodes before running Ansible.

File Nodes

Configure the Baseboard Management Controller (BMC)

A baseboard management controller (BMC), sometimes referred to as a service processor, is the generic name for the out-of-band management capability built into various server platforms that can provide remote access even if the operating system is not installed or accessible. Vendors typically market this functionality with their own branding. For example, on the Lenovo SR665, the BMC is referred to as the Lenovo XClarity Controller (XCC).

Follow the server vendor's documentation to enable any licenses needed to access this functionality and ensure the BMC is connected to the network and configured appropriately for remote access.

If BMC based fencing using Redfish is desired, ensure Redfish is enabled and the BMC interface is accessible from the OS installed on the file node. Special configuration may be required on the network switch if the BMC and operating share the same physical network interface.

Tune System Settings

Using the system setup (BIOS/UEFI) interface, ensure settings are set to maximize performance. The exact settings and optimal values will vary based on the server model in use. Guidance is provided for verified file node models, otherwise refer to the server vendor's documentation and best practices based on your model.

Install an Operating System

Install a supported operating system based on the file node requirements listed here. Refer to any additional steps below based on your Linux distribution.

RedHat

Use RedHat Subscription Manager to register and subscribe the system to allow installation of the required packages from the official Red Hat repositories and to limit updates to the supported version of Red Hat: `subscription-manager release --set=<MAJOR_VERSION>.<MINOR_VERSION>`. For instructions, see How to register and subscribe a RHEL system and How to limit updates.

Enable the Red Hat repository containing the packages required for high availability:

```
subscription-manager repo-override --repo=rhel-8-for-x86_64
-highavailability-rpms --add=enabled:1
```

Configure Management Network

Configure any network interfaces needed to allow in-band management of the operating system. The exact steps will depend on the specific Linux distribution and version in use.

> Ensure SSH is enabled and all management interfaces are accessible from the Ansible control node.

Update HCA and HBA Firmware

Ensure all HBAs and HCAs are running supported firmware versions listed on the NetApp Interoperability Matrix and upgrade if necessary. Additional recommendations for NVIDIA ConnectX adapters can be found here.

Block Nodes

Follow the steps to get up and running with E-Series to configure the management port on each block node controller and optionally set the storage array name for each system.

> No additional configuration beyond ensuring all block nodes are accessible from the Ansible control node is necessary. The remaining system configuration will be applied/maintained using Ansible.

Set Up Ansible Control Node

Set up an Ansible control node to deploy and manage the file system.

Overview

An Ansible control node is a physical or virtual Linux machine used to manage the cluster. It must meet the following requirements:

- Meet the requirements for the BeeGFS HA role including the installed versions of Ansible, Python, and any additional Python packages.
- Meet the official Ansible control node requirements including operating system versions.
- Have SSH and HTTPS access to all file and block nodes.

Detailed installation steps can be found here.

Define the BeeGFS file system

Ansible Inventory Overview

The Ansible inventory is a set of configuration files that define the desired BeeGFS HA cluster.

Overview

It is recommended to follow standard Ansible practices for organizing your inventory, including the use of sub-directories/files instead of storing the entire inventory in one file.

The Ansible inventory for a single BeeGFS HA cluster is organized as follows:

[Ansible Inventory Overview]

> Since a single BeeGFS file system can span multiple HA clusters, it is possible for large installations to have multiple Ansible inventories. Generally it is not recommend to try and define multiple HA clusters as a single Ansible inventory to avoid issues.

Steps

1. On your Ansible control node create an empty directory that will contain the Ansible inventory for the BeeGFS cluster you want to deploy.
   a. If your file system will/may eventually contain multiple HA clusters, it is recommended to first create a directory for the file system, then sub-directories for the inventory representing each HA cluster. For example:

```
beegfs_file_system_1/
    beegfs_cluster_1/
    beegfs_cluster_2/
    beegfs_cluster_N/
```

2. In the directory containing the inventory for the HA cluster you want to deploy, create two directories `group_vars` and `host_vars` and two files `inventory.yml` and `playbook.yml`.

The following sections walk through defining the contents of each of these files.

Plan the File System

Plan the file system deployment before building out the Ansible inventory.

Overview

Before deploying the file system, you should define what IP addresses, ports, and other configuration will be required by all file nodes, block nodes, and BeeGFS services running in the cluster. While the exact configuration will vary based on the architecture of the cluster, this section defines best practices and steps to follow that are generally applicable.

Steps

1. If you are using an IP based storage protocol (such as iSER, iSCSI, NVMe/IB, or NVMe/RoCE) to connect file nodes to block nodes, fill out the following worksheet for each building block. Each direct connect in a single building block should have a unique subnet, and there should be no overlap with subnets used for client-server connectivity.

File node	IB port	IP address	Block node	IB port	Physical IP	Virtual IP (for EF600 with HDR IB only)
<HOSTNAME>	<PORT>	<IP/SUBNET>	<HOSTNAME>	<PORT>	<IP/SUBNET>	<IP/SUBNET>

If the file and block nodes in each building block are directly connected you can often reuse the same IPs/scheme for multiple building blocks.

2. Regardless if you are using InfiniBand or RDMA over Converged Ethernet (RoCE) for the storage network, fill out the following worksheet to determine the IP ranges that will be used for HA cluster services, BeeGFS file services, and clients to communicate:

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP(s)	<INTERFACE(s)>	<RANGE>
BeeGFS Management	<INTERFACE(s)>	<IP(s)>
BeeGFS Metadata	<INTERFACE(s)>	<RANGE>
BeeGFS Storage	<INTERFACE(s)>	<RANGE>
BeeGFS Clients	<INTERFACE(s)>	<RANGE>

   a. If you are using a single IP subnet only one worksheet is needed, otherwise also fill out a worksheet for the second subnet.

3. Based on the above, for each building block in the cluster, fill out the following worksheet defining what BeeGFS services it will run. For each service specify the preferred/secondary file node(s), network port, floating IP(s), NUMA zone assignment (if required), and what block node(s) will be used for its targets. Refer to the following guidelines when filling out the worksheet:

   a. Specify BeeGFS services as either `mgmt.yml`, `meta_<ID>.yml`, or `storage_<ID>.yml` where ID represents a unique number across all BeeGFS services of that type in this file system. This convention will simplify referring back to this worksheet in subsequent sections while creating files to configure each service.

   b. Ports for BeeGFS services only need to be unique across a particular building block. Ensure services with the same port number cannot ever run on the same file node to avoid port conflicts.

   c. If necessary services can use volumes from more than one block node and/or storage pool (and not all volumes need to be owned by the same controller). Multiple services can also share the same block node and/or storage pool configuration (individual volumes will be defined in a later section).

BeeGFS service (file name)	File Nodes	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
<SERVICE TYPE>_<ID>.yml	<PREFERRED FILE NODE> <SECONDARY FILE NODE(s)>	<PORT>	<INTERFACE>:<IP/SUBNET> <INTERFACE>:<IP/SUBNET>	<NUMA NODE/ZONE>	<BLOCK NODE>	<STORAGE POOL/VOLUME GROUP>	<A OR B>

For more details on standard conventions, best practices, and filled out example worksheets refer to the best practices and define BeeGFS building blocks sections of the BeeGFS on NetApp Verified Architecture.

Define File and Block Nodes

Configure Individual File Nodes

Specify configuration for individual file nodes using host variables (host_vars).

Overview

This section walks through populating a `host_vars/<FILE_NODE_HOSTNAME>.yml` file for each file node in the cluster. These files should only contain configuration unique to a particular file node. This commonly includes:

- Defining the IP or hostname Ansible should use to connect to the node.
- Configuring additional interfaces and cluster IPs used for HA cluster services (Pacemaker and Corosync) to communicate to other file nodes. By default these services use the same network as the management interface, but additional interfaces should be available for redundancy. Common practice is to define additional IPs on the storage network, avoiding the need for an additional cluster or management network.
  - The performance of any networks used for cluster communication is not critical for file system performance. With the default cluster configuration generally at least a 1Gbps network will provide sufficient performance for cluster operations such as synchronizing node states and coordinating cluster resource state changes. Slow/busy networks may cause resource state changes to take longer than usual, and in extreme cases could result in nodes being evicted from the cluster if they cannot send heartbeats in a reasonable time frame.
- Configuring interfaces used for connecting to block nodes over the desired protocol (for example: iSCSI/iSER, NVMe/IB, NVMe/RoCE, FCP, etc.)

Steps

Referencing the IP addressing scheme defined in the Plan the File System section, for each file node in the cluster create a file `host_vars/<FILE_NODE_HOSTNAME>/yml` and populate it as follows:

1. At the top specify the IP or hostname Ansible should use to SSH to the node and manage it:

```
ansible_host: "<MANAGEMENT_IP>"
```

2. Configure additional IPs that can be used for cluster traffic:
   a. If the network type is InfiniBand (using IPoIB):

```
eseries_ipoib_interfaces:
- name: <INTERFACE>  # Example: ib0 or i1b
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

   b. If the network type is RDMA over Converged Ethernet (RoCE):

```
eseries_roce_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

c. If the network type is Ethernet (TCP only, no RDMA):

```
eseries_ip_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

3. Indicate what IPs should be used for cluster traffic, with preferred IPs listed higher:

```
beegfs_ha_cluster_node_ips:
- <MANAGEMENT_IP> # Including the management IP is typically but not
required.
- <IP_ADDRESS>    # Ex: 100.127.100.1
- <IP_ADDRESS>    # Additional IPs as needed.
```

IPs configured in step two will not be used as cluster IPs unless they are included in the `beegfs_ha_cluster_node_ips` list. This allows you to configure additional IPs/interfaces using Ansible that can be used for other purposes if desired.

4. If the file node needs to communicate to block nodes over an IP-based protocol, IPs will need to be configured on the appropriate interface, and any packages required for that protocol installed/configured.

a. If using iSCSI:

```
eseries_iscsi_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
```

b. If using iSER:

```
eseries_ib_iser_interfaces:
- name: <INTERFACE>  # Example: ib0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
  configure: true # If the file node is directly connected to the
block node set to true to setup OpenSM.
```

c. If using NVMe/IB:

```
eseries_nvme_ib_interfaces:
- name: <INTERFACE>  # Example: ib0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
  configure: true # If the file node is directly connected to the
block node set to true to setup OpenSM.
```

d. If using NVMe/RoCE:

```
eseries_nvme_roce_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
```

e. Other Protocols:

i. If using NVMe/FC, configuring individual interfaces is not required. The BeeGFS cluster deployment will automatically detect the protocol and install/configure requirements as needed. If you are using a fabric to connect file and block nodes, ensure switches are properly zoned following NetApp and your switch vendor's best practices.

ii. Use of FCP or SAS do not require installing or configuring additional software. If using FCP, ensure switches are properly zoned following NetApp and your switch vendor's best practices.

iii. Use of IB SRP is not recommended at this time. Use NVMe/IB or iSER depending on what your E-Series block nodes support.

Click here for an example of a complete inventory file representing a single file node.

Advanced: Toggling NVIDIA ConnectX VPI Adapters between Ethernet and InfiniBand Mode

NVIDIA ConnectX-Virtual Protocol Interconnect® (VPI) adapters support both InfiniBand and Ethernet as the transport layer. Switching between modes is not automatically negotiated, and must be configured using the <code>mstconfig</code> tool included in <code>mstflint</code>, an open source package that is part of the Mellanox Firmare Tools (MFT). Changing the mode of the adapters only need to be done once. This can be done manually, or included in the Ansible inventory as part of any interfaces configured using the <code>eseries-[ib|ib_iser|ipoib|nvme_ib|nvme_roce|roce]_interfaces:</code> section of the inventory, to have it checked/applied automatically.

For example to change an interface current in InfiniBand mode to Ethernet so it can be used for RoCE:

1. For each interface you want to configure specify `mstconfig` as a mapping (or dictionary) that specifies `LINK_TYPE_P<N>` where `<N>` is determined by the HCA's port number for the interface. The `<N>` value can be determined by running `grep PCI_SLOT_NAME /sys/class/net/<INTERFACE_NAME>/device/uevent` and adding 1 to the last number from the PCI slot name and converting to decimal.

   a. For example given `PCI_SLOT_NAME=0000:2f:00.2` (2 + 1 → HCA port 3) → `LINK_TYPE_P3: eth`:

```
eseries_roce_interfaces:
- name: <INTERFACE>
  address: <IP/SUBNET>
  mstconfig:
    LINK_TYPE_P3: eth
```

For additional details refer to the NetApp E-Series Host collection's documentation for the interface type/protocol you are using.

Configure Individual Block Nodes

Specify configuration for individual block nodes using host variables (host_vars).

Overview

This section walks through populating a `host_vars/<BLOCK_NODE_HOSTNAME>.yml` file for each block node in the cluster. These files should only contain configuration unique to a particular block node. This commonly includes:

- The system name (as displayed in System Manager).
- The HTTPS URL for one of the controllers (used to manage the system using its REST API).
- What storage protocol file nodes use to connect to this block node.
- Configuring host interface card (HIC) ports, such as IP addresses (if needed).

Steps

Referencing the IP addressing scheme defined in the Plan the File System section, for each block node in the cluster create a file `host_vars/<BLOCK_NODE_HOSTNAME>/yml` and populate it as follows:

1. At the top specify the system name and the HTTPS URL for one of the controllers:

```
eseries_system_name: <SYSTEM_NAME>
eseries_system_api_url:
https://<MANAGEMENT_HOSTNAME_OR_IP>:8443/devmgr/v2/
```

2. Select the protocol file nodes will use to connect to this block node:
   a. Supported Protocols: `auto`, `iscsi`, `fc`, `sas`, `ib_srp`, `ib_iser`, `nvme_ib`, `nvme_fc`, `nvme_roce`.

```
eseries_initiator_protocol: <PROTOCOL>
```

3. Depending on the protocol in use, the HIC ports may require additional configuration. When needed, HIC port configuration should be defined so the top entry in the configuration for each controller corresponds with with the physical left-most port on each controller, and the bottom port the right-most port. All ports require valid configuration even if they are not currently in use.

Also see the section below if you are using HDR (200Gb) InfiniBand or 200Gb RoCE with EF600 block nodes.

a. For iSCSI:

```
eseries_controller_iscsi_port:
  controller_a:          # Ordered list of controller A channel
definition.
    - state:             # Whether the port should be enabled.
Choices: enabled, disabled
      config_method:     # Port configuration method Choices: static,
dhcp
      address:           # Port IPv4 address
      gateway:           # Port IPv4 gateway
      subnet_mask:       # Port IPv4 subnet_mask
      mtu:               # Port IPv4 mtu
    - (...)              # Additional ports as needed.
  controller_b:          # Ordered list of controller B channel
definition.
    - (...)              # Same as controller A but for controller B

# Alternatively the following common port configuration can be
defined for all ports and omitted above:
eseries_controller_iscsi_port_state: enabled         # Generally
specifies whether a controller port definition should be applied
Choices: enabled, disabled
eseries_controller_iscsi_port_config_method: dhcp    # General port
configuration method definition for both controllers. Choices:
static, dhcp
eseries_controller_iscsi_port_gateway:               # General port
IPv4 gateway for both controllers.
eseries_controller_iscsi_port_subnet_mask:           # General port
IPv4 subnet mask for both controllers.
eseries_controller_iscsi_port_mtu: 9000              # General port
maximum transfer units (MTU) for both controllers. Any value greater
than 1500 (bytes).
```

b. For iSER:

```
eseries_controller_ib_iser_port:
  controller_a:    # Ordered list of controller A channel address
definition.
    -              # Port IPv4 address for channel 1
    - (...)        # So on and so forth
  controller_b:    # Ordered list of controller B channel address
definition.
```

c. For NVMe/IB:

```
eseries_controller_nvme_ib_port:
  controller_a:    # Ordered list of controller A channel address
definition.
    -              # Port IPv4 address for channel 1
    - (...)        # So on and so forth
  controller_b:    # Ordered list of controller B channel address
definition.
```

d. For NVMe/RoCE:

```
eseries_controller_nvme_roce_port:
  controller_a:         # Ordered list of controller A channel
definition.
    - state:            # Whether the port should be enabled.
      config_method:    # Port configuration method Choices: static,
dhcp
      address:          # Port IPv4 address
      subnet_mask:      # Port IPv4 subnet_mask
      gateway:          # Port IPv4 gateway
      mtu:              # Port IPv4 mtu
      speed:            # Port IPv4 speed
  controller_b:         # Ordered list of controller B channel
definition.
    - (...)             # Same as controller A but for controller B

# Alternatively the following common port configuration can be
defined for all ports and omitted above:
eseries_controller_nvme_roce_port_state: enabled         # Generally
specifies whether a controller port definition should be applied
Choices: enabled, disabled
eseries_controller_nvme_roce_port_config_method: dhcp    # General
port configuration method definition for both controllers. Choices:
static, dhcp
eseries_controller_nvme_roce_port_gateway:               # General
port IPv4 gateway for both controllers.
eseries_controller_nvme_roce_port_subnet_mask:           # General
port IPv4 subnet mask for both controllers.
eseries_controller_nvme_roce_port_mtu: 4200              # General
port maximum transfer units (MTU). Any value greater than 1500
(bytes).
eseries_controller_nvme_roce_port_speed: auto            # General
interface speed. Value must be a supported speed or auto for
automatically negotiating the speed with the port.
```

e. FC and SAS protocols do not require additional configuration. SRP is not correctly recommended.

For additional options to configure HIC ports and host protocols including the ability to configure iSCSI CHAP refer to the documentation included with the SANtricity collection. Note when deploying BeeGFS the storage pool, volume configuration, and other aspects of provisioning storage will be configured elsewhere, and should not be defined in this file.

Click here for an example of a complete inventory file representing a single block node.

Using HDR (200Gb) InfiniBand or 200Gb RoCE with NetApp EF600 block nodes:

To use HDR (200Gb) InfiniBand with the EF600, a second "virtual" IP must be configured for each physical port. THis is an example of the correct way to configure an EF600 equipped with the dual port InfiniBand HDR

HIC:

```
eseries_controller_nvme_ib_port:
  controller_a:
    - 192.168.1.101 # Port 2a (physical)
    - 192.168.2.101 # Port 2a (virtual)
    - 192.168.1.100 # Port 2b (physical)
    - 192.168.2.100 # Port 2b (virtual)
  controller_b:
    - 192.168.3.101 # Port 2a (physical)
    - 192.168.4.101 # Port 2a (virtual)
    - 192.168.3.100 # Port 2b (physical)
    - 192.168.4.100 # Port 2b (virtual)
```

Specify Common File Node Configuration

Specify common file node configuration using group variables (group_vars).

Overview

Configuration that should apple to all file nodes is defined at `group_vars/ha_cluster.yml`. It commonly includes:

- Details on how to connect and login to each file node.
- Common networking configuration.
- Whether automatic reboots are allowed.
- How firewall and selinux states should be configured.
- Cluster configuration including alerting and fencing.
- Performance tuning.
- Common BeeGFS service configuration.

The options set in this file can also be defined on individual file nodes, for example if mixed hardware models are in use, or you have different passwords for each node. Configuration on individual file nodes will take precedence over the configuration in this file.

Steps

Create the file `group_vars/ha_cluster.yml` and populate it as follows:

1. Indicate how the Ansible Control node should authenticate with the remote hosts:

```
ansible_ssh_user: root
ansible_become_password: <PASSWORD>
```

Particularly for production environments, do not store passwords in plain text. Instead, use Ansible Vault (see Encrypting content with Ansible Vault) or the `--ask-become-pass` option when running the playbook. If the `ansible_ssh_user` is already root, then you can optionally omit the `ansible_become_password`.

2. If you are configuring static IPs on ethernet or InfiniBand interfaces (for example cluster IPs) and multiple interfaces are in the same IP subnet (for example if ib0 is using 192.168.1.10/24 and ib1 is using 192.168.1.11/24), additional IP routing tables and rules must be setup for multi-homed support to work properly. Simply enable the provided network interface configuration hook as follows:

```
eseries_ip_default_hook_templates:
  - 99-multihoming.j2
```

3. When deploying the cluster, depending on the storage protocol it may be necessary for nodes to be rebooted to facilitate discovering remote block devices (E-Series volumes) or apply other aspects of the configuration. By default nodes will prompt before rebooting, but you can allow nodes to restart automatically by specifying the following:

```
eseries_common_allow_host_reboot: true
```

   a. By default after a reboot, to ensure block devices and other services are ready Ansible will wait until the systemd `default.target` is reached before continuing with the deployment. In some scenarios when NVMe/IB is in use, this may not be long enough to initialize, discover, and connect to remote devices. This can result in the automated deployment continuing prematurely and failing. To avoid this when using NVMe/IB also define the following:

```
eseries_common_reboot_test_command: "! systemctl status
eseries_nvme_ib.service || systemctl --state=exited | grep
eseries_nvme_ib.service"
```

4. A number of firewall ports are required for BeeGFS and HA cluster services to communicate. Unless you wish to configure the firwewall manually (not recommended), specify the following to have required firewall zones created and ports opened automatically:

```
beegfs_ha_firewall_configure: True
```

5. At this time SELinux is not supported, and it is is recommended the state be set to disabled to avoid conflicts (especially when RDMA is in use). Set the following to ensure SELinux is disabled:

```
eseries_beegfs_ha_disable_selinux: True
eseries_selinux_state: disabled
```

6. Configure authentication so file nodes are able to communicate, adjusting the defaults as needed based on your organizations policies:

```
beegfs_ha_cluster_name: hacluster                    # BeeGFS HA cluster
name.
beegfs_ha_cluster_username: hacluster                # BeeGFS HA cluster
username.
beegfs_ha_cluster_password: hapassword               # BeeGFS HA cluster
username's password.
beegfs_ha_cluster_password_sha512_salt: randomSalt # BeeGFS HA cluster
username's password salt.
```

7. Based on the Plan the File System section specify the BeeGFS management IP for this file system:

```
beegfs_ha_mgmtd_floating_ip: <IP ADDRESS>
```

> While seemingly redundant, `beegfs_ha_mgmtd_floating_ip` is important when you scale the BeeGFS file system beyond a single HA cluster. Subsequent HA clusters are deployed without an additional BeeGFS management service and point at the management service provided by the first cluster.

8. Enable email alerts if desired:

```
beegfs_ha_enable_alerts: True
# E-mail recipient list for notifications when BeeGFS HA resources
change or fail.
beegfs_ha_alert_email_list: ["<EMAIL>"]
# This dictionary is used to configure postfix service
(/etc/postfix/main.cf) which is required to set email alerts.
beegfs_ha_alert_conf_ha_group_options:
      # This parameter specifies the local internet domain name. This is
optional when the cluster nodes have fully qualified hostnames (i.e.
host.example.com)
      mydomain: <MY_DOMAIN>
beegfs_ha_alert_verbosity: 3
#  1) high-level node activity
#  3) high-level node activity + fencing action information + resources
(filter on X-monitor)
#  5) high-level node activity + fencing action information + resources
```

9. Enabling fencing is strongly recommended, otherwise services can be blocked from starting on secondary nodes when the primary node fails.
   a. Enable fencing globally by specifying the following:

```
beegfs_ha_cluster_crm_config_options:
  stonith-enabled: True
```

i. Note any supported cluster property can also be specified here if needed. Adjusting these is not typically needed, as the BeeGFS HA role ships with a number of well tested defaults.

b. Next select and configure a fencing agent:

i. OPTION 1: To enable fencing using APC Power Distribution Units (PDUs):

```
beegfs_ha_fencing_agents:
  fence_apc:
    - ipaddr: <PDU_IP_ADDRESS>
      login: <PDU_USERNAME>
      passwd: <PDU_PASSWORD>
      pcmk_host_map:
"<HOSTNAME>:<PDU_PORT>,<PDU_PORT>;<HOSTNAME>:<PDU_PORT>,<PDU_PORT>
"
```

ii. OPTION 2: To enable fencing using the Redfish APIs provided by the Lenovo XCC (and other BMCs):

```
redfish: &redfish
  username: <BMC_USERNAME>
  password: <BMC_PASSWORD>
  ssl_insecure: 1 # If a valid SSL certificate is not available
specify "1".

beegfs_ha_fencing_agents:
  fence_redfish:
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
```

iii. For details on configuring other fencing agents refer to the RedHat Documentation.

10. The BeeGFS HA role can apply many different tuning parameters to help further optimize performance. These include optimizing kernel memory utilization and block device I/O, among other parameters. The role ships with a reasonable set of defaults based on testing with NetApp E-Series block nodes, but by default these aren't applied unless you specify:

```
beegfs_ha_enable_performance_tuning: True
```

a. If needed also specify any changes to the default performance tuning here. See the full performance tuning parameters documentation for additional details.

11. To ensure floating IP addresses (sometimes known as logical interfaces) used for BeeGFS services can fail over between file nodes, all network interfaces must be named consistently. By default network interface names are generated by the kernel, which is not guaranteed to generate consistent names, even across identical server models with network adapters installed in the same PCIe slots. This is also useful when creating inventories before the equipment is deployed and generated interface names are known. To ensure consistent device names, based on a block diagram of the server or `lshw -class network -businfo` output, specify the desired PCIe address-to-logical interface mapping as follows:

    a. For InfiniBand (IPoIB) network interfaces:

```
eseries_ipoib_udev_rules:
  "<PCIe ADDRESS>": <NAME> # Ex: 0000:41:00.0: i1a
```

    b. For Ethernet network interfaces:

```
eseries_ip_udev_rules:
  "<PCIe ADDRESS>": <NAME> # Ex: 0000:41:00.0: e1a
```

> To avoid conflicts when interfaces are renamed (preventing them from being renamed), you should not use any potential default names such as eth0, ens9f0, ib0, or ibs4f0. A common naming convention is to use 'e' or 'i' for Ethernet or InfiniBand, followed by the PCIe slot number, and a letter to indicate the the port. For example the second port of an InfiniBand adapter installed in slot 3 would be: i3b.

> If you are using a verified file node model, click here example PCIe address-to-logical port mappings.

12. Optionally specify configuration that should apply to all BeeGFS services in the cluster. Default configuration values can be found here, and per-service configuration is specified elsewhere:

    a. BeeGFS Management service:

```
beegfs_ha_beegfs_mgmtd_conf_ha_group_options:
  <OPTION>: <VALUE>
```

    b. BeeGFS Metadata services:

```
beegfs_ha_beegfs_meta_conf_ha_group_options:
  <OPTION>: <VALUE>
```

    c. BeeGFS Storage services:

```
beegfs_ha_beegfs_storage_conf_ha_group_options:
  <OPTION>: <VALUE>
```

13. As of BeeGFS 7.2.7 and 7.3.1 connection authentication must be configured or explicitly disabled. There are a few ways this can be configured using the Ansible based deployment:
    a. By default the deployment will automatically configure connection authentication, and generate a `connauthfile` that will be distributed to all file nodes and used with the BeeGFS services. This file will also be placed/maintained on the Ansible control node at `<INVENTORY>/files/beegfs/<sysMgmtdHost>_connAuthFile` where it should be maintained (securely) for reuse with clients that need to access this file system.
        i. To generate a new key specify `-e "beegfs_ha_conn_auth_force_new=True` when running the Ansible playbook. Note this is ignored if a `beegfs_ha_conn_auth_secret` is defined.
        ii. For advanced options refer to the full list of defaults included with the BeeGFS HA role.
    b. A custom secret can be used by defining the following in `ha_cluster.yml`:

```
beegfs_ha_conn_auth_secret: <SECRET>
```

    c. Connection authentication can be disabled entirely (NOT recommended):

```
beegfs_ha_conn_auth_enabled: false
```

Click here for an example of a complete inventory file representing common file node configuration.

Using HDR (200Gb) InfiniBand with NetApp EF600 block nodes:

To use HDR (200Gb) InfiniBand with the EF600 the subnet manager must support virtualization. If file and block nodes are connected using a switch, this will need to be enabled on the subnet manager manager for the overall fabric.

If block and file nodes are directly connected using InfiniBand, an instance of `opensm` must be configured on each file node for each interface directly connected to a block node. This is done by specifying `configure: true` when configuring file node storage interfaces.

Currently the inbox version of `opensm` shipped with supported Linux distributions does not support virtualization. Instead it is required you install and configure version of `opensm` from the Mellanox OpenFabrics Enterprise Distribution (OFED). Although deployment using Ansible is still supported, a few additional steps are required:

1. Using curl or your desired tool, download the packages for the version of OpenSM listed in the technology requirements section from Mellanox's website to the `<INVENTORY>/packages/` directory. For example:

```
curl -o packages/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
https://linux.mellanox.com/public/repo/mlnx_ofed/5.4-
1.0.3.0/rhel8.4/x86_64/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm

curl -o packages/opensm-5.9.0.MLNX20210617.c9f2ade-0.1.54103.x86_64.rpm
https://linux.mellanox.com/public/repo/mlnx_ofed/5.4-
1.0.3.0/rhel8.4/x86_64/opensm-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
```

2. Under `group_vars/ha_cluster.yml` define the following configuration:

```
### OpenSM package and configuration information
eseries_ib_opensm_allow_upgrades: true
eseries_ib_opensm_skip_package_validation: true
eseries_ib_opensm_rhel_packages: []
eseries_ib_opensm_custom_packages:
  install:
    - files:
        add:
          "packages/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm": "/tmp/"
          "packages/opensm-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm": "/tmp/"
    - packages:
        add:
          - /tmp/opensm-5.9.0.MLNX20210617.c9f2ade-0.1.54103.x86_64.rpm
          - /tmp/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm
  uninstall:
    - packages:
        remove:
          - opensm
          - opensm-libs
      files:
        remove:
          - /tmp/opensm-5.9.0.MLNX20210617.c9f2ade-0.1.54103.x86_64.rpm
          - /tmp/opensm-libs-5.9.0.MLNX20210617.c9f2ade-
0.1.54103.x86_64.rpm

eseries_ib_opensm_options:
  virt_enabled: "2"
```

Specify Common Block Node Configuration

Specify common block node configuration using group variables (group_vars).

Overview

Configuration that should apple to all block nodes is defined at `group_vars/eseries_storage_systems.yml`. It commonly includes:

- Details on how the Ansible control node should connect to E-Series storage systems used as block nodes.
- What firmware, NVSRAM, and Drive Firmware versions the nodes should run.
- Global configuration including cache settings, host configuration, and settings for how volumes should be provisioned.

> The options set in this file can also be defined on individual block nodes, for example if mixed hardware models are in use, or you have different passwords for each node. Configuration on individual block nodes will take precedence over the configuration in this file.

Steps

Create the file `group_vars/eseries_storage_systems.yml` and populate it as follows:

1. Ansible does not use SSH to connect to block nodes, and instead uses REST APIs. To achieve this we must set:

```
ansible_connection: local
```

2. Specify the username and password to manage each node. The username can be optionally omitted (and will default to admin), otherwise you can specify any account with admin privileges. Also specify if SSL certificates should be verified, or ignored:

```
eseries_system_username: admin
eseries_system_password: <PASSWORD>
eseries_validate_certs: false
```

> Listing any passwords in plaintext is not recommended. Use Ansible vault or provide the `eseries_system_password` when running Ansible using --extra-vars.

3. Optionally specify what controller firmware, NVSRAM, and drive firmware should be installed on the nodes. These will need to be downloaded to the `packages/` directory before running Ansible. E-Series controller firmware and NVSRAM can be downloaded here and drive firmware here:

```
eseries_firmware_firmware: "packages/<FILENAME>.dlp" # Ex.
"packages/RCB_11.70.2_6000_61b1131d.dlp"
eseries_firmware_nvsram: "packages/<FILENAME>.dlp" # Ex.
"packages/N6000-872834-D06.dlp"
eseries_drive_firmware_firmware_list:
  - "packages/<FILENAME>.dlp"
  # Additional firmware versions as needed.
eseries_drive_firmware_upgrade_drives_online: true # Recommended unless
BeeGFS hasn't been deployed yet, as it will disrupt host access if set
to "false".
```

If this configuration is specified, Ansible will automatically update all firmware including rebooting controllers (if necessary) with with no additional prompts. This is expected to be non-disruptive to BeeGFS/host I/O, but may cause a temporary decrease in performance.

4. Adjust global system configuration defaults. The options and values listed here are commonly recommended for BeeGFS on NetApp, but can be adjusted if needed:

```
eseries_system_cache_block_size: 32768
eseries_system_cache_flush_threshold: 80
eseries_system_default_host_type: linux dm-mp
eseries_system_autoload_balance: disabled
eseries_system_host_connectivity_reporting: disabled
eseries_system_controller_shelf_id: 99 # Required by default.
```

5. Configure global volume provisioning defaults. The options and values listed here are commonly recommended for BeeGFS on NetApp, but can be adjusted if needed:

```
eseries_volume_size_unit: pct # Required by default. This allows volume
capacities to be specified as a percentage, simplifying putting together
the inventory.
eseries_volume_read_cache_enable: true
eseries_volume_read_ahead_enable: false
eseries_volume_write_cache_enable: true
eseries_volume_write_cache_mirror_enable: true
eseries_volume_cache_without_batteries: false
```

6. If needed, adjust the order in which Ansible will select drives for storage pools and volume groups keeping in mind the following best practices:

   a. List any (potentially smaller) drives that should be used for management and/or metadata volumes first, and storage volumes last.

   b. Ensure to balance the drive selection order across available drive channels based on the disk shelf/drive enclosure model(s). For example with the EF600 and no expansions, drives 0-11 are on drive channel 1, and drives 12-23 are on drive channel. Thus a strategy to balance drive selection is to

select `disk shelf:drive` 99:0, 99:23, 99:1, 99:22, etc. In the event there is more than one enclosure, the first digit represents the drive shelf ID.

```
# Optimal/recommended order for the EF600 (no expansion):
eseries_storage_pool_usable_drives:
"99:0,99:23,99:1,99:22,99:2,99:21,99:3,99:20,99:4,99:19,99:5,99:18,99
:6,99:17,99:7,99:16,99:8,99:15,99:9,99:14,99:10,99:13,99:11,99:12"
```

Click here for an example of a complete inventory file representing common block node configuration.

Define BeeGFS services

Define the BeeGFS management service

BeeGFS services are configured using group variables (group_vars).

Overview

This section walks through defining the BeeGFS management service. Only one service of this type should exist in the HA cluster(s) for a particular file system. Configuring this service includes defining:

- The service type (management).
- Defining any configuration that should only apply to this BeeGFS service.
- Configuring one or more floating IPs (logical interfaces) where this service can be reached.
- Specifying where/how a volume should be to store data for this service (the BeeGFS management target).

Steps

Create a new file `group_vars/mgmt.yml` and referencing the Plan the File System section populate it as follows:

1. Indicate this file represents the configuration for a BeeGFS management service:

   ```
   beegfs_service: management
   ```

2. Define any configuration that should apply only to this BeeGFS service. This is not typically required for the management service unless you need to enable quotas, however any supported configuration parameter from `beegfs-mgmtd.conf` can be included. Note the following parameters are configured automatically/elsewhere and should not be specified here: `storeMgmtdDirectory`, `connAuthFile`, `connDisableAuthentication`, `connInterfacesFile`, and `connNetFilterFile`.

   ```
   beegfs_ha_beegfs_mgmtd_conf_resource_group_options:
     <beegfs-mgmt.conf:key>:<beegfs-mgmt.conf:value>
   ```

3. Configure one or more floating IPs that other services and clients will use to connect to this service (this will automatically set the BeeGFS `connInterfacesFile` option):

```
floating_ips:
  - <INTERFACE>:<IP/SUBNET> # Primary interface. Ex.
i1b:100.127.101.0/16
  - <INTERFACE>:<IP/SUBNET> # Secondary interface(s) as needed.
```

4. Optionally, specify one or more allowed IP subnets which may be used for outgoing communication (this will automatically set the BeeGFS `connNetFilterFile` option):

```
filter_ip_ranges:
  - <SUBNET>/<MASK> # Ex. 192.168.10.0/24
```

5. Specify the BeeGFS management target where this service will store data according to the following guidelines:
   a. The same storage pool or volume group name can be used for multiple BeeGFS services/targets, simply ensure to use the same `name`, `raid_level`, `criteria_*`, and `common_*` configuration for each (the volumes listed for each service should be different).
   b. Volume sizes should be specified as a percentage of the storage pool/volume group and the total should not exceed 100 across all services/volumes using a particular storage pool/volume group. Note when using SSDs it is recommended to leave some free space in the volume group to maximize SSD performance and wear life (click here for more details).
   c. Click here for a full list of configuration options available for the `eseries_storage_pool_configuration`. Note some options such as `state`, `host`, `host_type`, `workload_name`, and `workload_metadata` and volume names are generated automatically and should not be specified here.

```
beegfs_targets:
  <BLOCK_NODE>: # The name of the block node as found in the Ansible
inventory. Ex: ictad22a01
    eseries_storage_pool_configuration:
      - name: <NAME> # Ex: beegfs_m1_m2_m5_m6
        raid_level: <LEVEL> # One of: raid1, raid5, raid6, raidDiskPool
        criteria_drive_count: <DRIVE COUNT> # Ex. 4
        common_volume_configuration:
          segment_size_kb: <SEGMENT SIZE> # Ex. 128
        volumes:
          - size: <PERCENT> # Percent of the pool or volume group to
allocate to this volume. Ex. 1
            owning_controller: <CONTROLLER> # One of: A, B
```

Click here for an example of a complete inventory file representing a BeeGFS management service.

Define the BeeGFS metadata service

BeeGFS services are configured using group variables (group_vars).

Overview

This section walks through defining the BeeGFS metadata service. At least one service of this type should exist in the HA cluster(s) for a particular file system. Configuring this service includes defining:

- The service type (metadata).
- Defining any configuration that should only apply to this BeeGFS service.
- Configuring one or more floating IPs (logical interfaces) where this service can be reached.
- Specifying where/how a volume should be to store data for this service (the BeeGFS metadata target).

Steps

Referencing the Plan the File System section, create a file at `group_vars/meta_<ID>.yml` for each metadata service in the cluster, and populate them as follows:

1. Indicate this file represents the configuration for a BeeGFS metadata service:

```
beegfs_service: metadata
```

2. Define any configuration that should apply only to this BeeGFS service. At minimum you must specify the desired TCP and UDP port, however any supported configuration parameter from `beegfs-meta.conf` can also be included. Note the following parameters are configured automatically/elsewhere and should not be specified here: `sysMgmtdHost`, `storeMetaDirectory`, `connAuthFile`, `connDisableAuthentication`, `connInterfacesFile`, and `connNetFilterFile`.

```
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: <TCP PORT>
  connMetaPortUDP: <UDP PORT>
  tuneBindToNumaZone: <NUMA ZONE> # Recommended if using file nodes with
multiple CPU sockets.
```

3. Configure one or more floating IPs that other services and clients will use to connect to this service (this will automatically set the BeeGFS `connInterfacesFile` option):

```
floating_ips:
  - <INTERFACE>:<IP/SUBNET> # Primary interface. Ex.
i1b:100.127.101.1/16
  - <INTERFACE>:<IP/SUBNET> # Secondary interface(s) as needed.
```

4. Optionally, specify one or more allowed IP subnets which may be used for outgoing communication (this will automatically set the BeeGFS `connNetFilterFile` option):

```
filter_ip_ranges:
  - <SUBNET>/<MASK> # Ex. 192.168.10.0/24
```

5. Specify the BeeGFS metadata target where this service will store data according to the following guidelines (this will also automatically configure the `storeMetaDirectory` option):
   a. The same storage pool or volume group name can be used for multiple BeeGFS services/targets, simply ensure to use the same `name`, `raid_level`, `criteria_*`, and `common_*` configuration for each (the volumes listed for each service should be different).
   b. Volume sizes should be specified as a percentage of the storage pool/volume group and the total should not exceed 100 across all services/volumes using a particular storage pool/volume group. Note when using SSDs it is recommended to leave some free space in the volume group to maximize SSD performance and wear life (click here for more details).
   c. Click here for a full list of configuration options available for the `eseries_storage_pool_configuration`. Note some options such as `state`, `host`, `host_type`, `workload_name`, and `workload_metadata` and volume names are generated automatically and should not be specified here.

```
beegfs_targets:
  <BLOCK_NODE>: # The name of the block node as found in the Ansible
inventory. Ex: ictad22a01
    eseries_storage_pool_configuration:
      - name: <NAME> # Ex: beegfs_m1_m2_m5_m6
        raid_level: <LEVEL> # One of: raid1, raid5, raid6, raidDiskPool
        criteria_drive_count: <DRIVE COUNT> # Ex. 4
        common_volume_configuration:
          segment_size_kb: <SEGMENT SIZE> # Ex. 128
        volumes:
          - size: <PERCENT> # Percent of the pool or volume group to
allocate to this volume. Ex. 1
            owning_controller: <CONTROLLER> # One of: A, B
```

Click here for an example of a complete inventory file representing a BeeGFS metadata service.

Define the BeeGFS storage service

BeeGFS services are configured using group variables (group_vars).

Overview

This section walks through defining the BeeGFS storage service. At least one service of this type should exist in the HA cluster(s) for a particular file system. Configuring this service includes defining:

- The service type (storage).
- Defining any configuration that should only apply to this BeeGFS service.
- Configuring one or more floating IPs (logical interfaces) where this service can be reached.
- Specifying where/how volume(s) should be to store data for this service (the BeeGFS storage targets).

Steps

Referencing the Plan the File System section, create a file at `group_vars/stor_<ID>.yml` for each storage

service in the cluster, and populate them as follows:

1. Indicate this file represents the configuration for a BeeGFS storage service:

```
beegfs_service: storage
```

2. Define any configuration that should apply only to this BeeGFS service. At minimum you must specify the desired TCP and UDP port, however any supported configuration parameter from `beegfs-storage.conf` can also be included. Note the following parameters are configured automatically/elsewhere and should not be specified here: `sysMgmtdHost`, `storeStorageDirectory`, `connAuthFile`, `connDisableAuthentication`, `connInterfacesFile`, and `connNetFilterFile`.

```
beegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <TCP PORT>
  connStoragePortUDP: <UDP PORT>
  tuneBindToNumaZone: <NUMA ZONE> # Recommended if using file nodes with
multiple CPU sockets.
```

3. Configure one or more floating IPs that other services and clients will use to connect to this service (this will automatically set the BeeGFS `connInterfacesFile` option):

```
floating_ips:
  - <INTERFACE>:<IP/SUBNET> # Primary interface. Ex.
i1b:100.127.101.1/16
  - <INTERFACE>:<IP/SUBNET> # Secondary interface(s) as needed.
```

4. Optionally, specify one or more allowed IP subnets which may be used for outgoing communication (this will automatically set the BeeGFS `connNetFilterFile` option):

```
filter_ip_ranges:
  - <SUBNET>/<MASK> # Ex. 192.168.10.0/24
```

5. Specify the BeeGFS storage target(s) where this service will store data according to the following guidelines (this will also automatically configure the `storeStorageDirectory` option):

   a. The same storage pool or volume group name can be used for multiple BeeGFS services/targets, simply ensure to use the same `name`, `raid_level`, `criteria_*`, and `common_*` configuration for each (the volumes listed for each service should be different).

   b. Volume sizes should be specified as a percentage of the storage pool/volume group and the total should not exceed 100 across all services/volumes using a particular storage pool/volume group. Note when using SSDs it is recommended to leave some free space in the volume group to maximize SSD performance and wear life (click here for more details).

   c. Click here for a full list of configuration options available for the `eseries_storage_pool_configuration`. Note some options such as `state`, `host`, `host_type`, `workload_name`, and `workload_metadata` and volume names are generated automatically and

should not be specified here.

```
beegfs_targets:
  <BLOCK_NODE>: # The name of the block node as found in the Ansible
inventory. Ex: ictad22a01
    eseries_storage_pool_configuration:
      - name: <NAME> # Ex: beegfs_s1_s2
        raid_level: <LEVEL> # One of: raid1, raid5, raid6,
raidDiskPool
        criteria_drive_count: <DRIVE COUNT> # Ex. 4
        common_volume_configuration:
          segment_size_kb: <SEGMENT SIZE> # Ex. 128
        volumes:
          - size: <PERCENT> # Percent of the pool or volume group to
allocate to this volume. Ex. 1
            owning_controller: <CONTROLLER> # One of: A, B
        # Multiple storage targets are supported / typical:
          - size: <PERCENT> # Percent of the pool or volume group to
allocate to this volume. Ex. 1
            owning_controller: <CONTROLLER> # One of: A, B
```

Click here for an example of a complete inventory file representing a BeeGFS storage service.

Map BeeGFS services to file nodes

Specify what file nodes can run each BeeGFS service using the `inventory.yml` file.

Overview

This section walks through how to create the `inventory.yml` file. This includes listing all block nodes and specifying what file nodes can run each BeeGFS service.

Steps

Create the file `inventory.yml` and populate it as follows:

1. From the top of the file, create the standard Ansible inventory structure:

```
# BeeGFS HA (High_Availability) cluster inventory.
all:
  children:
```

2. Create a group containing all block nodes participating in this HA cluster:

```
# Ansible group representing all block nodes:
eseries_storage_systems:
  hosts:
    <BLOCK NODE HOSTNAME>:
    <BLOCK NODE HOSTNAME>:
    # Additional block nodes as needed.
```

3. Create a group that will contain all BeeGFS services in the cluster, and the file nodes that will run them:

```
# Ansible group representing all file nodes:
ha_cluster:
  children:
```

4. For each BeeGFS service in the cluster, define the preferred and any secondary file node(s) that should run that service:

```
    <SERVICE>: # Ex. "mgmt", "meta_01", or "stor_01".
      hosts:
        <FILE NODE HOSTNAME>:
        <FILE NODE HOSTNAME>:
        # Additional file nodes as needed.
```

Click here for an example of a complete inventory file.

Deploy the BeeGFS file system

Ansible Playbook Overview

Deploying and managing BeeGFS HA clusters using Ansible.

Overview

The previous sections walked through the steps needed to build an Ansible inventory representing a BeeGFS HA cluster. This section introduces the Ansible automation developed by NetApp to deploy and manage the cluster.

Ansible: Key Concepts

Before proceeding, it is helpful to be familiar with a few key Ansible concepts:

- Tasks to execute against an Ansible inventory are defined in what is known as a **playbook**.
  - Most tasks in Ansible are designed to be **idempotent**, meaning they can be run multiple times to verify the desired configuration/state is still applied without breaking things or making unnecessary updates.
- The smallest unit of execution in Ansible is a **module**.

    - Typical playbooks use multiple modules.
        - Examples: Download a package, update a config file, start/enable a service.
    - NetApp distributes modules to automate NetApp E-Series systems.
- Complex automation is better packaged as a role.
    - Essentially a standard format to distribute a reusable playbook.
    - NetApp distributes roles for Linux hosts and BeeGFS file systems.

BeeGFS HA role for Ansible: Key Concepts

All the automation needed to deploy and manage each version of BeeGFS on NetApp is packaged as an Ansible role and distributed as part of the NetApp E-Series Ansible Collection for BeeGFS:

- This role can be thought of as somewhere between an **installer** and modern **deployment/management** engine for BeeGFS.
    - Applies modern infrastructure as code practices and philosophies to simplify managing storage infrastructure at any scale.
    - Similar to how the Kubespray project allows users to deploy/maintain an entire Kubernetes distribution for scale out compute infrastructure.
- This role is the **software-defined** format NetApp uses to package, distribute, and maintain BeeGFS on NetApp solutions.
    - Strive to create an “appliance-like” experience without needing to distribute an entire Linux distribution or large image.
    - Includes NetApp authored Open Cluster Framework (OCF) compliant cluster resource agents for custom BeeGFS targets, IP addresses, and monitoring that provide intelligent Pacemaker/BeeGFS integration.
- This role is not simply deployment "automation" and is intended to manage the entire file system lifecycle including:
    - Applying per-service or cluster-wide configuration changes and updates.
    - Automating cluster healing and recovery after hardware issues are resolved.
    - Simplifying performance tuning with default values set based on extensive testing with BeeGFS and NetApp volumes.
    - Verifying and correcting configuration drift.

NetApp also provides an Ansible role for BeeGFS clients, that can optionally be used to install BeeGFS and mount file systems to compute/GPU/login nodes.

Deploy the BeeGFS HA cluster

Specify what tasks should run to deploy the BeeGFS HA cluster using a playbook.

Overview

This section covers how to assemble the standard playbook used to deploy/manage BeeGFS on NetApp.

Steps

Create the Ansible Playbook

Create the file `playbook.yml` and populate it as follows:

1. First define a set of tasks (commonly referred to as a play) that should only run on NetApp E-Series block nodes. We use a pause task to prompt before running the installation (to avoid accidental playbook runs), then import the `nar_santricity_management` role. This role handles applying any general system configuration defined in `group_vars/eseries_storage_systems.yml` or individual `host_vars/<BLOCK NODE>.yml` files.

```
- hosts: eseries_storage_systems
  gather_facts: false
  collections:
    - netapp_eseries.santricity
  tasks:
    - name: Verify before proceeding.
      pause:
        prompt: "Are you ready to proceed with running the BeeGFS HA
role? Depending on the size of the deployment and network performance
between the Ansible control node and BeeGFS file and block nodes this
can take awhile (10+ minutes) to complete."
    - name: Configure NetApp E-Series block nodes.
      import_role:
        name: nar_santricity_management
```

2. Define the play that will run against all file and block nodes:

```
- hosts: all
  any_errors_fatal: true
  gather_facts: false
  collections:
    - netapp_eseries.beegfs
```

3. Within this play we can optionally define a set of "pre-tasks" that should run before deploying the HA cluster. This can be useful to verify/install any prerequisites like Python. We can also inject any pre-flight checks, for example verifying the provided Ansible tags are supported:

```
  pre_tasks:
    - name: Ensure a supported version of Python is available on all
file nodes.
      block:
        - name: Check if python is installed.
          failed_when: false
          changed_when: false
          raw: python --version
          register: python_version
```

```
        - name: Check if python3 is installed.
          raw: python3 --version
          failed_when: false
          changed_when: false
          register: python3_version
          when: 'python_version["rc"] != 0 or (python_version["stdout"]
| regex_replace("Python ", "")) is not version("3.0", ">=")'

        - name: Install python3 if needed.
          raw: |
            id=$(grep "^ID=" /etc/*release* | cut -d= -f 2 | tr -d '"')
            case $id in
              ubuntu) sudo apt install python3 ;;
              rhel|centos) sudo yum -y install python3 ;;
              sles) sudo zypper install python3 ;;
            esac
          args:
            executable: /bin/bash
          register: python3_install
          when: python_version['rc'] != 0 and python3_version['rc'] != 0
          become: true

        - name: Create a symbolic link to python from python3.
          raw: ln -s /usr/bin/python3 /usr/bin/python
          become: true
          when: python_version['rc'] != 0
      when: inventory_hostname not in
groups[beegfs_ha_ansible_storage_group]

    - name: Verify any provided tags are supported.
      fail:
        msg: "{{ item }} tag is not a supported BeeGFS HA tag. Rerun
your playbook command with --list-tags to see all valid playbook tags."
      when: 'item not in ["all", "storage", "beegfs_ha",
"beegfs_ha_package", "beegfs_ha_configure",
"beegfs_ha_configure_resource", "beegfs_ha_performance_tuning",
"beegfs_ha_backup", "beegfs_ha_client"]'
      loop: "{{ ansible_run_tags }}"
```

4. Lastly, this play imports the BeeGFS HA role for the version of BeeGFS you want to deploy:

```
tasks:
  - name: Verify the BeeGFS HA cluster is properly deployed.
    import_role:
      name: beegfs_ha_7_3 # Alternatively specify: beegfs_ha_7_2.
```

> A BeeGFS HA role is maintained for each supported major.minor version of BeeGFS. This allows users to choose when they want to upgrade major/minor versions. Currently either BeeGFS 7.3.x (`beegfs_7_3`) or BeeGFS 7.2.x (`beegfs_7_2`) are supported. By default both roles will deploy the latest BeeGFS patch version at the time of release, though users can opt to override this and deploy the latest patch if desired. Refer to the latest upgrade guide for more details.

5. Optional: If you wish to define additional tasks, keep in mind if the tasks should be directed to `all` hosts (including the E-Series storage systems) or only the file nodes. If needed define a new play specifically targeting file nodes using `- hosts: ha_cluster`.

Click here for an example of a complete playbook file.

Install the NetApp Ansible Collections

The BeeGFS collection for Ansible and all dependencies are maintained on Ansible Galaxy. On your Ansible control node run the following command to install the latest version:

```
ansible-galaxy collection install netapp_eseries.beegfs
```

Though not typically recommended, it is also possible to install a specific version of the collection:

```
ansible-galaxy collection install netapp_eseries.beegfs:
==<MAJOR>.<MINOR>.<PATCH>
```

Run the Playbook

From the directory on your Ansible control node containing the `inventory.yml` and `playbook.yml` files, run the playbook as follows:

```
ansible-playbook -i inventory.yml playbook.yml
```

Based on the size of the cluster the initial deployment can take 20+ minutes. If the deployment fails for any reason, simply correct any issues (e.g., miscabling, node wasn't started, etc.), and restart the Ansible playbook.

When specifying common file node configuration, if you choose the default option to have Ansible automatically manage connection based authentication, a `connAuthFile` used as a shared secret can now be found at `<playbook_dir>/files/beegfs/<sysMgmtdHost>_connAuthFile` (by default). Any clients needing to access the file system will need to use this shared secret. This is handled automatically if clients are configured using the BeeGFS client role.

Deploy BeeGFS clients

Optionally, Ansible can be used to configure BeeGFS clients and mount the file system.

Overview

Accessing BeeGFS file systems requires installing and configuring the BeeGFS client on each node that needs to mount the file system. This section documents how to perform these tasks using the available Ansible role.

Steps

Create the Client Inventory File

1. If needed, set up passwordless SSH from the Ansible control node to each of the hosts you want to configure as BeeGFS clients:

```
ssh-copy-id <user>@<HOSTNAME_OR_IP>
```

2. Under `host_vars/`, create a file for each BeeGFS client named `<HOSTNAME>.yml` with the following content, filling in the placeholder text with the correct information for your environment:

```
# BeeGFS Client
ansible_host: <MANAGEMENT_IP>
```

3. Optionally include one of the following if you want to use the NetApp E-Series Host Collection's roles to configure InfiniBand or Ethernet interfaces for clients to connect to BeeGFS file nodes:

   a. If the network type is InfiniBand (using IPoIB):

```
eseries_ipoib_interfaces:
- name: <INTERFACE>  # Example: ib0 or i1b
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

   b. If the network type is RDMA over Converged Ethernet (RoCE):

```
eseries_roce_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

   c. If the network type is Ethernet (TCP only, no RDMA):

```
eseries_ip_interfaces:
- name: <INTERFACE>  # Example: eth0.
  address: <IP/SUBNET> # Example: 100.127.100.1/16
- name: <INTERFACE>  # Additional interfaces as needed.
  address: <IP/SUBNET>
```

4. Create a new file `client_inventory.yml` and specify the user Ansible should use to connect to each client, and the password Ansible should use for privilege escalation (this requires `ansible_ssh_user` be root, or have sudo privileges):

```
# BeeGFS client inventory.
all:
  vars:
    ansible_ssh_user: <USER>
    ansible_become_password: <PASSWORD>
```

Do not store passwords in plain text. Instead, use the Ansible Vault (see the Ansible documentation for Encrypting content with Ansible Vault) or use the `--ask-become-pass` option when running the playbook.

5. In the `client_inventory.yml` file, list all hosts that should be configured as BeeGFS clients under the `beegfs_clients` group, and then refer to the inline comments and uncomment any additional configuration required to build the BeeGFS client kernel module on your system:

```
children:
    # Ansible group representing all BeeGFS clients:
    beegfs_clients:
      hosts:
        <CLIENT HOSTNAME>:
        # Additional clients as needed.

      vars:
        # OPTION 1: If you're using the Mellanox OFED drivers and they
are already installed:
        #eseries_ib_skip: True # Skip installing inbox drivers when
using the IPoIB role.
        #beegfs_client_ofed_enable: True
        #beegfs_client_ofed_include_path:
"/usr/src/ofa_kernel/default/include"

        # OPTION 2: If you're using inbox IB/RDMA drivers and they are
already installed:
        #eseries_ib_skip: True # Skip installing inbox drivers when
using the IPoIB role.

        # OPTION 3: If you want to use inbox IB/RDMA drivers and need
them installed/configured.
        #eseries_ib_skip: False # Default value.
        #beegfs_client_ofed_enable: False # Default value.
```

When using the Mellanox OFED drivers, make sure that beegfs_client_ofed_include_path points to the correct "header include path" for your Linux installation. For more information, see the BeeGFS documentation for RDMA support.

6. In the `client_inventory.yml` file, list the BeeGFS file systems you want mounted under any previously defined `vars`:

```
        beegfs_client_mounts:
          - sysMgmtdHost: <IP ADDRESS>  # Primary IP of the BeeGFS
management service.
            mount_point: /mnt/beegfs    # Path to mount BeeGFS on the
client.
            connInterfaces:
              - <INTERFACE> # Example: ibs4f1
              - <INTERFACE>
            beegfs_client_config:
              # Maximum number of simultaneous connections to the same
node.
              connMaxInternodeNum: 128 # BeeGFS Client Default: 12
              # Allocates the number of buffers for transferring IO.
              connRDMABufNum: 36 # BeeGFS Client Default: 70
              # Size of each allocated RDMA buffer
              connRDMABufSize: 65536 # BeeGFS Client Default: 8192
              # Required when using the BeeGFS client with the shared-
disk HA solution.
              # This does require BeeGFS targets be mounted in the
default "sync" mode.
              # See the documentation included with the BeeGFS client
role for full details.
              sysSessionChecksEnabled: false
        # Specify additional file system mounts for this or other file
systems.
```

7. As of BeeGFS 7.2.7 and 7.3.1 connection authentication must be configured or explicitly disabled. Depending how you choose to configure connection based authentication when specifying common file node configuration, you may need to adjust your client configuration:
   a. By default the HA cluster deployment will automatically configure connection authentication, and generate a `connauthfile` that will be placed/maintained on the Ansible control node at `<INVENTORY>/files/beegfs/<sysMgmtdHost>_connAuthFile`. By default the BeeGFS client role is setup to read/distribute this file to the clients defined in `client_inventory.yml`, and no additional action is needed.
      i. For advanced options refer to the full list of defaults included with the BeeGFS client role.
   b. If you choose to specify a custom secret with `beegfs_ha_conn_auth_secret` specify it in the `client_inventory.yml` file as well:

      ```
      beegfs_ha_conn_auth_secret: <SECRET>
      ```

   c. If you choose to disable connection based authentication entirely with `beegfs_ha_conn_auth_enabled`, specify that in the `client_inventory.yml` file as well:

```
beegfs_ha_conn_auth_enabled: false
```

For a full list of supported parameters and additional details refer to the full BeeGFS client documentation. For a complete example of a client inventory click here.

Create the BeeGFS Client Playbook File

1. Create a new file `client_playbook.yml`

```
# BeeGFS client playbook.
- hosts: beegfs_clients
  any_errors_fatal: true
  gather_facts: true
  collections:
    - netapp_eseries.beegfs
    - netapp_eseries.host
  tasks:
```

2. Optional: If you want to use the NetApp E-Series Host Collection's roles to configure interfaces for clients to connect to BeeGFS file systems, import the role corresponding with the interface type you are configuring:
   a. If you are using are using InfiniBand (IPoIB):

```
    - name: Ensure IPoIB is configured
      import_role:
        name: ipoib
```

   b. If you are using are using RDMA over Converged Ethernet (RoCE):

```
    - name: Ensure IPoIB is configured
      import_role:
        name: roce
```

   c. If you are using are using Ethernet (TCP only, no RDMA):

```
    - name: Ensure IPoIB is configured
      import_role:
        name: ip
```

3. Lastly import the BeeGFS client role to install the client software and setup the file system mounts:

```
    # REQUIRED: Install the BeeGFS client and mount the BeeGFS file
system.
    - name: Verify the BeeGFS clients are configured.
      import_role:
        name: beegfs_client
```

For a complete example of a client playbook click here.

Run the BeeGFS Client Playbook

To install/build the client and mount BeeGFS, run the following command:

```
ansible-playbook -i client_inventory.yml client_playbook.yml
```

Verify the BeeGFS deployment

Verify the file system deployment before placing the system in production.

Overview

Before you place the BeeGFS file system in production, perform a few verification checks.

Steps

1. Login to any client and run the following to ensure all expected nodes are present/reachable, and there are no inconsistencies or other issues reported:

```
beegfs-fsck --checkfs
```

2. Shutdown the entire cluster, then restart it. From any file node run the following:

```
pcs cluster stop --all # Stop the cluster on all file nodes.
pcs cluster start --all  # Start the cluster on all file nodes.
pcs status # Verify all nodes and services are started and no failures
are reported (the command may need to be reran a few times to allow time
for all services to start).
```

3. Place each node in standby and verify BeeGFS services are able to failover to secondary node(s). To do this login to any of the file nodes and run the following:

```
pcs status # Verify the cluster is healthy at the start.
pcs node standby <FILE NODE HOSTNAME> # Place the node under test in
standby.
pcs status # Verify services are started on a secondary node and no
failures are reported.
pcs node unstandby <FILE NODE HOSTNAME> # Take the node under test out
of standby.
pcs status # Verify the file node is back online and no failures are
reported.
pcs resource relocate run # Move all services back to their preferred
nodes.
pcs status # Verify services have moved back to the preferred node.
```

4. Use performance benchmarking tools such as IOR and MDTest to verify the file system performance meets expectations. Examples of common tests and parameters used with BeeGFS can be found in the Design Verification section of the BeeGFS on NetApp Verified Architecture.

Additional tests should be performed based on the acceptance criteria defined for a particular site/installation.

Administer BeeGFS Clusters

Overview, key concepts, and terminology

Learn how to administer BeeGFS HA clusters after they have been deployed.

Overview

This section is intended for cluster administrators that need to manage BeeGFS HA clusters after they are deployed. Even those familiar with Linux HA clusters should thoroughly read this guide as there are a number of differences in how to manage the cluster, especially around reconfiguration due to the use of Ansible.

Key Concepts

While some of these concepts are introduced on the main terms and concepts page, it is helpful to reintroduce them in the context of a BeeGFS HA cluster:

Cluster Node: A server running Pacemaker and Corosync services and participating in the HA cluster.

File Node: A cluster node used to run one or more BeeGFS management, metadata, or storage services.

Block Node: A NetApp E-Series storage system that provides block storage to file nodes. These nodes do not participate in the BeeGFS HA cluster as they provide their own standalone HA capabilities. Each node consists of two storage controllers that provide high availability at the block layer.

BeeGFS service: A BeeGFS management, metadata or storage service. Each file node will run one or more services that will use volumes on the block node to store their data.

Building Block: A standardized deployment of BeeGFS file nodes, E-Series block nodes, and the BeeGFS services running on them that simplifies scaling out a BeeGFS HA cluster / file system following a NetApp Verified Architecture. Custom HA clusters are also supported, but often follow a similar building block approach to simplify scaling.

BeeGFS HA Cluster: A scalable number of file nodes used to run BeeGFS services backed by block nodes to store BeeGFS data in a highly available fashion. Built on industry-proven open-source components Pacemaker and Corosync using Ansible for packaging and deployment.

Cluster services: Refers to Pacemaker and Corosync services running on each node participating in the cluster. Note it is possible for a node to not run any BeeGFS services and just participate in the cluster as a "tiebreaker" node in the event there is only a need for two file nodes.

Cluster resources: For each BeeGFS service running in the cluster you will see a BeeGFS monitor resource and a resource group containing resources for BeeGFS target(s), IP address(es) (floating IPs), and the BeeGFS service itself.

Ansible: A tool for software provisioning, configuration management, and application-deployment, enabling infrastructure as code. It is how BeeGFS HA clusters are packaged to simplify the process of deploying, reconfiguring and updating BeeGFS on NetApp.

pcs: A command line interface available from any of the file nodes in the cluster used to query and control the state of nodes and resources in the cluster.

Common Terminology

Failover: Each BeeGFS service has a preferred file node it will run on unless that node fails. When a BeeGFS service is running on non-preferred/secondary file node it is said to be in failover.

Failback: The act of moving BeeGFS services from a non-preferred file node back to their preferred node.

HA pair: Two file nodes that can access the same set of block nodes are sometimes referred to as an HA pair. This is a common term used throughout NetApp to refer to two storage controllers or nodes that can "take over" for each other.

Maintenance Mode: Disables all resource monitoring and prevents Pacemaker from moving or otherwise managing resources in the cluster (also see the section on maintenance mode).

HA cluster: One or more file nodes running BeeGFS services that can failover between multiple nodes in the cluster to create a highly available BeeGFS file system. Often file nodes are configured into HA pairs that are able to run a subset of the BeeGFS services in the cluster.

When to use Ansible versus the pcs tool

When should you use Ansible versus the pcs command line tool to manage the HA cluster?

All cluster deployment and reconfiguration tasks should be completed using Ansible from an external Ansible control node. Temporary changes in the cluster state (e.g., placing nodes in and out of standby) will typically be performed by logging into one node of the cluster (preferably one that is not degraded or about to undergo maintenance) and using the pcs command line tool.

Modifying any of the cluster configuration including resources, constraints, properties, and the BeeGFS services themselves should always be done using Ansible. Maintaining an up-to-date copy of the Ansible inventory and playbook (ideally in source control to track changes) is part of maintaining the cluster. When you need to make changes to the configuration, update the inventory and rerun the Ansible playbook that imports the BeeGFS HA role.

The HA role will handle placing the cluster into maintenance mode then making any necessary changes before restarting BeeGFS or cluster services to apply the new configuration. As full node reboots aren’t typically needed outside the initial deployment, rerunning Ansible is generally considered a "safe" procedure, but always recommended during maintenance windows or off-hours in case any BeeGFS services need to restart. These restarts shouldn’t typically cause application errors, but may hurt performance (which some applications may handle better than others).

Rerunning Ansible is also an option when you want to return the entire cluster back to a fully optimal state, and may in some cases be able to recover the state of the cluster more easily than using pcs. Especially during an emergency where the cluster is down for some reason, once all nodes are back up rerunning Ansible may more quickly and reliably recover the cluster than attempting to use pcs.

Examine the state of the cluster

Use pcs to view the state of the cluster.

Overview

Running `pcs status` from any of the cluster nodes is the easiest way to see the overall state of the cluster and the status of each resource (such as BeeGFS services and their dependencies). This section walks through what you will find in the output of the `pcs status` command.

Understanding the output from `pcs status`

Run `pcs status` on any cluster node where the cluster services (Pacemaker and Corosync) are started. The top of the output will show you a summary of the cluster:

```
[root@ictad22h01 ~]# pcs status
Cluster name: hacluster
Cluster Summary:
  * Stack: corosync
  * Current DC: ictad22h01 (version 2.0.5-9.el8_4.3-ba59be7122) -
partition with quorum
  * Last updated: Fri Jul  1 13:37:18 2022
  * Last change:  Fri Jul  1 13:23:34 2022 by root via cibadmin on
ictad22h01
  * 6 nodes configured
  * 235 resource instances configured
```

The section below lists nodes in the cluster:

```
Node List:
  * Node ictad22h06: standby
  * Online: [ ictad22h01 ictad22h02 ictad22h04 ictad22h05 ]
  * OFFLINE: [ ictad22h03 ]
```

This notably indicates any nodes that are in standby or offline. Nodes in standby are still participating in the cluster but marked as ineligible to run resources. Nodes that are offline indicate cluster services are not running on that node, either due to being manually stopped or because the node was rebooted/shutdown.

When nodes first start up, cluster services will be stopped and need to be manually started to avoid accidentally failing back resources to an unhealthy node.

If nodes are in standby or offline due to a non-administrative reason (for example a failure) additional text will be displayed next to the node's state in parenthesis. For example if fencing is disabled and a resource encounters a failure you will see `Node <HOSTNAME>: standby (on-fail)`. Another possible state is `Node <HOSTNAME>: UNCLEAN (offline)`, which will briefly be seen as a node is being fenced, but will persist if fencing failed indicating the cluster cannot confirm the state of the node (this can block the resources from starting on other nodes).

The next section shows a list of all resources in the cluster and their states:

```
Full List of Resources:
  * mgmt-monitor    (ocf::eseries:beegfs-monitor):   Started ictad22h01
  * Resource Group: mgmt-group:
    * mgmt-FS1  (ocf::eseries:beegfs-target):    Started ictad22h01
    * mgmt-IP1  (ocf::eseries:beegfs-ipaddr2):   Started ictad22h01
    * mgmt-IP2  (ocf::eseries:beegfs-ipaddr2):   Started ictad22h01
    * mgmt-service  (systemd:beegfs-mgmtd):  Started ictad22h01
[...]
```

Similar to nodes, additional text will be displayed next to the resource state in parenthesis if there are any issues with the resource. For example if Pacemaker requests a resource stop and it fails to complete within the time allocated, then Pacemaker will attempt to fence the node. If fencing is disabled or the fencing operation fails, the resource state will be `FAILED <HOSTNAME> (blocked)` and Pacemaker will be unable to start it on a different node.

It is worth noting BeeGFS HA clusters make use of a number of BeeGFS optimized custom OCF resource agents. In particular the BeeGFS monitor is responsible for triggering a failover when BeeGFS resources on a particular node are not available.

Reconfigure and update

Reconfigure the HA cluster and BeeGFS

Use Ansible to reconfigure the cluster.

Overview

Generally reconfiguring any aspect of the BeeGFS HA cluster should be done by updating your Ansible inventory and re-running the `ansible-playbook` command. This includes updating alerts, changing the permanent fencing configuration, or adjusting BeeGFS service configuration. These are adjusted using the `group_vars/ha_cluster.yml` file and a full list of options can be found in the Specify Common File Node Configuration section.

See below for additional details on select configuration options that administrators should be aware of when performing maintenance or servicing the cluster.

How to Disable and Enable Fencing

Fencing is enabled/required by default when setting up the cluster. In some instances it may be desirable to temporarily disable fencing to ensure nodes aren't accidentally shutdown when performing certain maintenance operations (such as upgrading the operating system). While this can be disabled manually, there are tradeoffs administrators should be aware off.

OPTION 1: Disable fencing using Ansible (recommended).

When fencing is disabled using Ansible, the on-fail action of the BeeGFS monitor is changed from "fence" to "standby". This means if the BeeGFS monitor detects a failure it will attempt to place the node in standby and failover all BeeGFS services. Outside active troubleshooting/testing this is typically more desirable than option 2. The disadvantage is if a resource fails to stop on the original node it will be blocked from starting elsewhere (which is why fencing is typically required for production clusters).

1. In your Ansible inventory at `groups_vars/ha_cluster.yml` add the following configuration:

```
beegfs_ha_cluster_crm_config_options:
    stonith-enabled: False
```

2. Rerun the Ansible playbook to apply the changes to the cluster.

OPTION 2: Disable fencing manually.

In some instances you may want to temporarily disable fencing without rerunning Ansible, perhaps to facilitate troubleshooting or testing of the cluster.

In this configuration if the BeeGFS monitor detects a failure, the cluster will attempt to stop the corresponding resource group. It will NOT trigger a full failover or attempt to restart or move the impacted resource group to another host. To recover, address any issues then run `pcs resource cleanup` or manually place the node in standby.

Steps:

1. To determine if fencing (stonith) is globally enabled or disabled run: `pcs property show stonith-enabled`
2. To disable fencing run: `pcs property set stonith-enabled=false`
3. To enable fencing run: `pcs property set stonith-enabled=true`

Note: This setting will be overridden the next time you run the Ansible playbook.

Update the HA cluster and BeeGFS

Use Ansible to update BeeGFS and the HA cluster.

Overview

BeeGFS is versioned following a `major.minor.patch` versioning scheme and BeeGFS HA Ansible roles are provided for each supported BeeGFS `major.minor` version (for example `beegfs_ha_7_2` and `beegfs_ha_7_3`). Each of the HA roles is pinned to the latest BeeGFS patch version at the time the Ansible collection was released.

Ansible should be used for all BeeGFS upgrades, including moving between major, minor, and patch versions of BeeGFS. To update BeeGFS you will first need to update the BeeGFS Ansible collection, which will also pull in the latest fixes and enhancements to the deployment/management automation and underlying HA cluster. Even after updating to the latest version of the collection, BeeGFS will not be upgraded until `ansible-playbook` is ran with the `-e "beegfs_ha_force_upgrade=true"` set.

For more information on BeeGFS versions see the BeeGFS Upgrade documentation.

Tested Upgrade Paths

Each version of the BeeGFS collection is tested with specific versions of BeeGFS to ensure interoperability between all components. Testing is also performed to ensure upgrades can be performed from the BeeGFS

version(s) supported by the last version of the collection, to those supported in the latest release.

Original Version	Upgrade Version	Multirail	Details
7.2.6	7.3.2	Yes	Upgrading beegfs collection from v3.0.1 to v3.1.0, multirail added
7.2.6	7.2.8	No	Upgrading beegfs collection from v3.0.1 to v3.1.0
7.2.8	7.3.1	Yes	Upgrade using beegfs collection v3.1.0, multirail added
7.3.1	7.3.2	Yes	Upgrade using beegfs collection v3.1.0

BeeGFS Upgrade Steps

The follow sections provide the steps to update the BeeGFS Ansible collection and BeeGFS itself. Pay special attention to any extra step(s) for updating BeeGFS major or minor versions.

Step 1: Upgrade BeeGFS Collection

For collection upgrades with access to Ansible Galaxy, run the following command:

```
ansible-galaxy collection install netapp_eseries.beegfs --upgrade
```

For offline collection upgrades, download the collection from Ansible Galaxy by clicking on the desired `Install Version` and then `Download tarball`. Transfer the tarball to your Ansible control node and run the following command.

```
ansible-galaxy collection install netapp_eseries-beegfs-<VERSION>.tar.gz
--upgrade
```

See Installing Collections for more information.

Step 2: Update Ansible Inventory

Make any required or desired updates to your cluster's Ansible inventory files. See the Version Upgrade Notes section below for details about your specific upgrade requirements. See the Use Custom Architectures section for general information on configuring your BeeGFS HA inventory.

Step 3: Update Ansible Playbook (when updating major or minor versions only)

If you are moving between major or minor versions, in the `playbook.yml` file used to deploy and maintain the cluster, update the name of the `beegfs_ha_<VERSION>` role to reflect the desired version. For example, if you wanted to deploy BeeGFS 7.3 this would be `beegfs_ha_7_3`:

```
- hosts: all
  gather_facts: false
  any_errors_fatal: true
  collections:
    - netapp_eseries.beegfs
  tasks:
    - name: Ensure BeeGFS HA cluster is setup.
      ansible.builtin.import_role:  # import_role is required for tag
availability.
        name: beegfs_ha_7_3
```

For more details on the contents of this playbook file see the Deploy the BeeGFS HA cluster section.

Step 4: Run the BeeGFS Upgrade

To apply the BeeGFS update:

```
ansible-playbook -i inventory.yml beegfs_ha_playbook.yml -e
"beegfs_ha_force_upgrade=true" --tags beegfs_ha
```

Behind the scenes the BeeGFS HA role will handle:

- Ensure the cluster is in an optimal state with each BeeGFS service located on its preferred node.
- Put the cluster in maintenance mode.
- Update the HA cluster components (if needed).
- Upgrade each file node one at a time as follows:
  - Place it into standby and failover its services to the secondary node.
  - Upgrade BeeGFS packages.
  - Fallback back services.
- Move the cluster out of maintenance mode.

Version Upgrade Notes

Upgrading from BeeGFS version 7.2.6 or 7.3.0

Changes to Connection Based Authentication

BeeGFS versions released after 7.3.1 will no longer allow services to start without either specifying a `connAuthFile` or setting `connDisableAuthentication=true` in the service's configuration file. It is highly recommended to enable connection based authentication security. See BeeGFS Connection Based Authentication for more information.

By default the `beegfs_ha*` roles will generate and distribute this file, also adding it to the Ansible control node at `<playbook_directory>/files/beegfs/<beegfs_mgmt_ip_address>_connAuthFile`. The `beegfs_client` role will also check for the presence of this file and supply it to the clients if available.

If the `beegfs_client` role was not used to configure clients, this file will need to be manually distributed to each client and the `connAuthFile` configuration in the `beegfs-client.conf` file set to use it. When upgrading from a previous version of BeeGFS where connection based authentication was not enabled, clients will loose access unless connection based authentication is disabled as part of the upgrade by setting `beegfs_ha_conn_auth_enabled: false` in `group_vars/ha_cluster.yml` (not recommended).

For additional details and alternate configuration options see the step to configure connection authentication in the Specify Common File Node Configuration section.

Service and maintain

Failover and failback services

Moving BeeGFS services between cluster nodes.

Overview

BeeGFS services can failover between nodes in the cluster to ensure clients are able to continue accessing the file system if a node experiences a fault, or you need to perform planned maintenance. This section describes various ways administrators can heal the cluster after recovering from a failure, or manually move services between nodes.

Steps

Failover and Failback

Failover (Planned)

Generally when you need to bring a single file node offline for maintenance you'll want to move (or drain) all BeeGFS services from that node. This can be accomplished by first putting the node in standby:

```
pcs node standby <HOSTNAME>
```

After verifying using `pcs status` all resources have been restarted on the alternate file node, you can shutdown or make other changes to the node as needed.

Failback (after a planned failover)

When you are ready to restore BeeGFS services to the preferred node first run `pcs status` and verify in the "Node List"the status is standby. If the node was rebooted it will show offline until you bring the cluster services online:

```
pcs cluster start <HOSTNAME>
```

Once the node is online bring it out of standby with:

```
pcs cluster node unstandby <HOSTNAME>
```

Lastly relocate all BeeGFS services back to their preferred nodes with:

```
pcs resource relocate run
```

Failback (after an unplanned failover)

If a node experience a hardware or other fault, the HA cluster should automatically react and move its services to a healthy node, providing time for administrators take corrective action. Before proceeding reference the troubleshooting section to determine the cause of the failover and resolve any outstanding issues. Once the node is powered back on and healthy you can proceed with failback.

When a node boots following an unplanned (or planned) reboot, cluster services are not set to start automatically, so you will first need to bring the node online with:

```
pcs cluster start <HOSTNAME>
```

Next cleanup any resource failures and reset the node's fencing history:

```
pcs resource cleanup node=<HOSTNAME>
pcs stonith history cleanup <HOSTNAME>
```

Verify in `pcs status` the node is online and healthy. By default BeeGFS services will not automatically failback to avoid accidentally moving resources back to an unhealthy node. When you are ready return all resources in the cluster back to their preferred nodes with:

```
pcs resource relocate run
```

Moving individual BeeGFS services to alternate file nodes

Permanently move a BeeGFS service to a new file node

If you want to permanently change the preferred file node for an individual BeeGFS service, adjust the Ansible inventory so the preferred node is listed first and rerun the Ansible playbook.

For example in this sample `inventory.yml` file, ictad22h01 is the preferred file node to run the BeeGFS management service:

```
        mgmt:
          hosts:
            ictad22h01:
            ictad22h02:
```

Reversing the order would cause the management services to be preferred on ictad22h02:

```
        mgmt:
          hosts:
            ictad22h02:
            ictad22h01:
```

Temporarily move a BeeGFS service to an alternate file node

Generally if a node is undergoing maintenance you will want to use the [failover and failback steps](#failover-and-failback) to move all services away from that node.

If for some reason you do need to move an individual service to a different file node run:

```
pcs resource move <SERVICE>-monitor <HOSTNAME>
```

Do not specify individual resources or the resource group. Always specify the name of the monitor for the BeeGFS service you wish to relocate. For example to move the BeeGFS management service to ictad22h02 run: `pcs resource move mgmt-monitor ictad22h02`. This process can be repeated to move one or more services away from their preferred nodes. Verify using `pcs status` the services were relocated/started on the new node.

To move a BeeGFS service back to its preferred node first clear the temporary resource constraints (repeating this step
as needed for multiple services):

```
pcs resource clear <SERVICE>-monitor
```

Then when ready to actually move service(s) back to their preferred node(s) run:

```
pcs resource relocate run
```

Note this command will relocate any services that no longer have temporary resource constraints not located on their preferred nodes.

Place the cluster in maintenance mode

Prevent the HA cluster from accidentally reacting to intended changes in the environment.

Overview

Putting the cluster in maintenance mode disables all resource monitoring and prevents Pacemaker from moving or otherwise managing resources in the cluster. All resources will remain running on their original nodes, regardless if there is a temporary failure condition that would prevent them from being accessible. Scenarios where this is recommended/useful include:

- Network maintenance that may temporarily disrupt connections between file nodes and BeeGFS services.
- Block Node upgrades.
- File Node operating system, kernel, or other package updates.

Generally the only reason to manually put the cluster in maintenance mode is to prevent it from reacting to external changes in the environment. If an individual node in the cluster requires physical repair do not use maintenance mode and simply place that node in standby following the procedure above. Note that rerunning Ansible will automatically put the cluster in maintenance mode facilitating most software maintenance including upgrades and configuration changes.

Steps

To check if the cluster is in maintenance mode run:

```
pcs property show maintenance-mode
```

This will return false when the cluster is operating normally. To enable maintenance mode run:

```
pcs property set maintenance-mode=true
```

You can verify by running pcs status and ensuring all resources show "(unmanaged)". To take the cluster out of maintenance mode run:

```
pcs property set maintenance-mode=false
```

Stop and start the cluster

Gracefully stopping and starting the HA cluster.

Overview

This section describes how to gracefully shutdown and restart the BeeGFS cluster. Example scenarios where this may be required include electrical maintenance or migrating between datacenters or racks.

Steps

If for any reason you need to stop the entire BeeGFS cluster and shutdown all services run:

```
pcs cluster stop --all
```

It is also possible to stop the cluster on individual nodes (which will automatically failover services to another node), though it is recommended to first put the node in standby (see the failover section):

```
pcs cluster stop <HOSTNAME>
```

To start cluster services and resources on all nodes run:

```
pcs cluster start --all
```

Or start services on a specific node with:

```
pcs cluster start <HOSTNAME>
```

At this point run `pcs status` and verify the cluster and BeeGFS services start on all nodes, and services are running on the nodes you expect.

Depending on the size of the cluster it can take sometime (seconds to minutes) for the entire cluster to stop, or show started in `pcs status`. If `pcs cluster <COMMAND>` hangs for more than five minutes, before running "Ctrl+C" to cancel the command, login to each node of the cluster and use `pcs status` to see if cluster services (Corosync/Pacemaker) are still running on that node. From any node where the cluster is still active you can check what resources are blocking the cluster. Manually address the issue and the command should either complete or can be rerun to stop any remaining services.

Replace file nodes

Replacing a file node if the original server is faulty.

Overview

This is an overview of the steps needed to replace a file node in the cluster. These steps presume the file node failed due to a hardware issue, and was replaced with a new identical file node.

Steps:

1. Physically replace the file node and restore all cabling to the block node and storage network.
2. Reinstall the operating system on the file node including adding Red Hat subscriptions.
3. Configure management and BMC networking on the file node.
4. Update the Ansible inventory if the hostname, IP, PCIe-to-logical interface mappings, or anything else changed about the new file node. Generally this is not needed if the node was replaced with identical server hardware and you are using the original network configuration.
   a. For example if the hostname changed, create (or rename) the node's inventory file (`host_vars/<NEW_NODE>.yml`) then in the Ansible inventory file (`inventory.yml`), replace the old node's name with the new node name:

```
all:
    ...
    children:
    ha_cluster:
        children:
        mgmt:
            hosts:
            node_h1_new:    # Replaced "node_h1" with "node_h1_new"
            node_h2:
```

5. From one of the other nodes in the cluster, remove the old node: `pcs cluster node remove <HOSTNAME>`.

   DO NOT PROCEED BEFORE RUNNING THIS STEP.

6. On the Ansible control node:
   a. Remove the old SSH key with:

   ```
   `ssh-keygen -R <HOSTNAME_OR_IP>`
   ```

   b. Configure passwordless SSH to the replace node with:

   ```
   ssh-copy-id <USER>@<HOSTNAME_OR_IP>
   ```

7. Rerun the Ansible playbook to configure the node and add it to the cluster:

   ```
   ansible-playbook -i <inventory>.yml <playbook>.yml
   ```

8. At this point, run `pcs status` and verify the replaced node is now listed and running services.

Expand or shrink the cluster

Add or remove building blocks from the cluster.

Overview

This section documents various considerations and options to adjust the size of your BeeGFS HA cluster. Typically cluster size is adjusted by adding or removing building blocks, which are typically two file nodes setup as an HA pair. It is also possible to add or remove individual file nodes (or other types of cluster nodes) if needed.

Adding a Building Block to the Cluster

Considerations

Growing the cluster by adding additional building blocks is a straightforward process. Before you begin keep in mind restrictions around the minimum and maximum number of cluster nodes in each individual HA cluster, and determine if you should add nodes to the existing HA cluster, or create a new HA cluster. Typically each building block consists of two file nodes, but three nodes is the minimum number of nodes per cluster (to establish quorum), and ten is the recommended (tested) maximum. For advanced scenarios it is possible to add a single "tiebreaker" node that does not run any BeeGFS services when deploying a two node cluster. Please contact NetApp support if you are considering such a deployment.

Keep in mind these restrictions and any anticipated future cluster growth when deciding how to expand the cluster. For example if you have a six node cluster and need to add four more nodes, it would be recommended to just start a new HA cluster.

Remember, a single BeeGFS file system can consist of multiple independent HA clusters. This allows file systems to continue scaling far past the recommended/hard limits of the underlying HA cluster components.

Steps

When adding a building block to your cluster, you will need to create the `host_vars` files for each of the new file nodes and block nodes (E-Series arrays). The names of these hosts need to be added to the inventory, along with the new resources that are to be created. The corresponding `group_vars` files will need to be created for each new resource. See the Use custom architectures section for details.

After creating the correct files, all that is needed is to rerun the automation using the command:

```
ansible-playbook -i <inventory>.yml <playbook>.yml
```

Removing a Building Block from the Cluster

There are a number of considerations to keep in mind when you need to retire a building block, for example:

- What BeeGFS services are running in this building block?
- Are just the file nodes retiring and the block nodes should be attached to new file nodes?
- If the entire building block is being retired, should the data be moved to a new building block, dispersed into existing nodes in the cluster, or moved to a new BeeGFS file system or other storage system?
- Can this happen during an outage or should it be done non-disruptively?
- Is the building block actively in use, or does it primarily contain data that is no-longer active?

Because of the diverse possible starting points and desired end states, please contact NetApp support so we can identify and help implement the best strategy based on your environment and requirements.

Troubleshoot

Troubleshooting a BeeGFS HA cluster.

Overview

This section walks through how to investigate and troubleshoot various failures and other scenarios that may arise when operating a BeeGFS HA cluster.

Troubleshooting Guides

Investigating Unexpected Failovers

When a node is unexpectedly fenced and its services moved to another node, the first step should be to see if the cluster indicates any resource failures at the bottom of `pcs status`. Typically nothing will be present if fencing completed successfully and the resources were restarted on another node.

Generally the next step will be to search through the systemd logs using `journalctl` on any one of the remaining file nodes (Pacemaker logs are synchronized on all nodes). If you know the time when the failure occurred you can start the search just before the failure occurred (generally at least ten minutes prior is recommended):

```
journalctl --since "<YYYY-MM-DD HH:MM:SS>"
```

The following sections show common text you can grep for in the logs to further narrow down the investigation.

Steps to Investigate/Resolve

Step 1: Check if the BeeGFS monitor detected a failure:

If the failover was triggered by the BeeGFS monitor you should see an error (if not proceed to the next step).

```
journalctl --since "<YYYY-MM-DD HH:MM:SS>" | grep -i unexpected
[...]
Jul 01 15:51:03 ictad22h01 pacemaker-schedulerd[9246]:  warning:
Unexpected result (error: BeeGFS service is not active!) was recorded for
monitor of meta_08-monitor on ictad22h02 at Jul  1 15:51:03 2022
```

In this instance BeeGFS service meta_08 stopped for some reason. To continue troubleshooting we should boot ictad22h02 and review logs for the service at `/var/log/beegfs-meta-meta_08_tgt_0801.log`. For example, the BeeGFS service could have encountered an application error due to an internal issue or problem with the node.

Unlike the logs from Pacemaker, logs for BeeGFS services are not distributed to all nodes in the cluster. To investigate those types of failures, the logs from the original node where the failure occurred are required.

Possible issues that could be reported by the monitor include:

- Target(s) are not accessible!
  - Description: Indicates the block volumes were not accessible.
  - Troubleshooting:

        - If the service also failed to start on the alternate file node, confirm the block node is healthy.
        - Check for any physical issues that would prevent access to the block nodes from this file node, for example faulty InfiniBand adapters or cables.
- Network is not reachable!
    - Description: None of the adapters used by clients to connect to this BeeGFS service were online.
    - Troubleshooting:
        - If multiple/all file nodes were impacted, check if there was a fault on the network used to connect the BeeGFS clients and file system.
        - Check for any physical issues that would prevent access to the clients from this file node, for example faulty
          InfiniBand adapters or cables.
- BeeGFS service is not active!
    - Description: A BeeGFS service stopped unexpectedly.
    - Troubleshooting:
        - On the file node that reported the error, check the logs for the impacted BeeGFS service to see if it reported a crash. If this happened, open a case with NetApp support so the crash can be investigated.
        - If there are no errors reported in the BeeGFS log, check the journal logs to see if systemd logged a reason the service was stopped. In some scenarios the BeeGFS service may not have have been given a chance to log any messages before the process was terminated (for example if someone ran `kill -9 <PID>`).

Step 2: Check if the node left the cluster unexpectedly

In the event the node suffered some catastrophic hardware failure (e.g., the system board died) or there was a kernel panic or similar software issue, the BeeGFS monitor will not report an error. Instead look for the hostname and you should see messages from Pacemaker indicating the node was lost unexpectedly:

```
journalctl --since "<YYYY-MM-DD HH:MM:SS>" | grep -i <HOSTNAME>
[...]
Jul 01 16:18:01 ictad22h01 pacemaker-attrd[9245]:  notice: Node ictad22h02
state is now lost
Jul 01 16:18:01 ictad22h01 pacemaker-controld[9247]:  warning:
Stonith/shutdown of node ictad22h02 was not expected
```

Step 3: Verify Pacemaker was able to fence the node

In all scenarios you should see Pacemaker attempt to fence the node to verify it is actually offline (exact messages may vary by cause of the fencing):

```
Jul 01 16:18:02 ictad22h01 pacemaker-schedulerd[9246]:  warning: Cluster
node ictad22h02 will be fenced: peer is no longer part of the cluster
Jul 01 16:18:02 ictad22h01 pacemaker-schedulerd[9246]:  warning: Node
ictad22h02 is unclean
Jul 01 16:18:02 ictad22h01 pacemaker-schedulerd[9246]:  warning:
Scheduling Node ictad22h02 for STONITH
```

If the fencing action completes successfully you will see messages like:

```
Jul 01 16:18:14 ictad22h01 pacemaker-fenced[9243]:  notice: Operation
'off' [2214070] (call 27 from pacemaker-controld.9247) for host
'ictad22h02' with device 'fence_redfish_2' returned: 0 (OK)
Jul 01 16:18:14 ictad22h01 pacemaker-fenced[9243]:  notice: Operation
'off' targeting ictad22h02 on ictad22h01 for pacemaker-
controld.9247@ictad22h01.786df3a1: OK
Jul 01 16:18:14 ictad22h01 pacemaker-controld[9247]:  notice: Peer
ictad22h02 was terminated (off) by ictad22h01 on behalf of pacemaker-
controld.9247: OK
```

If the fencing action failed for some reason, then the BeeGFS services will be unable to restart on another node to avoid risking data corruption. That would be an issue to investigate separately, if for example the fencing device (PDU or BMC) was inaccessible or misconfigured.

Address Failed Resource Actions (found at the bottom of pcs status)

If a resource required to run a BeeGFS service fails, a failover will be triggered by the BeeGFS monitor. If this occurs there will likely be no "Failed Resource Actions" listed at the bottom of `pcs status` and you should refer to the steps on how to failback after an unplanned failover.

Otherwise there should generally only be two scenarios where you will see "Failed Resource Actions".

Steps to Investigate/Resolve

Scenario 1: A temporary or permanent issue was detected with a fencing agent and it was restarted or moved to another node.

Some fencing agents are more reliable than others, and each will implement their own method of monitoring to ensure the fencing device is ready. In particular the Redfish fencing agent has been seen to report failed resource actions like the following even though it will still show started:

```
  * fence_redfish_2_monitor_60000 on ictad22h01 'not running' (7):
call=2248, status='complete', exitreason='', last-rc-change='2022-07-26
08:12:59 -05:00', queued=0ms, exec=0ms
```

A fencing agent reporting failed resource actions on a particular node is not expected to trigger a failover of the BeeGFS services running on that node. It should simply be automatically restarted on the same or a different node.

Steps to resolve:

1. If the fencing agent consistently refuses to run on all or a subset of nodes, check if those nodes are able to connect to the fencing agent, and verify the fencing agent is configured correctly in the Ansible inventory.
    a. For example if a Redfish (BMC) fencing agent is running on the same node as it is responsible for fencing, and the OS management and BMC IPs are on the same physical interface, some network switch configurations will not allow communication between the two interfaces (to prevent network loops). By default the HA cluster will attempt to avoid placing fencing agents on the node they are responsible for fencing, but this can happen in some scenarios/configurations.
2. Once all issues are resolved (or if the issue appeared to be ephemeral), run `pcs resource cleanup` to reset the failed resource actions.

Scenario 2: The BeeGFS monitor detected an issue and triggered a failover, but for some reason resources could not start on a secondary node.

Provided fencing is enabled and the resource wasn't blocked from stopping on the original node (see the troubleshooting section for "standby (on-fail)"), the most likely reasons include problems starting the resource on a secondary node because:

- The secondary node was already offline.
- A physical or logical configuration issue prevented the secondary from accessing the block volumes used as BeeGFS targets.

Steps to resolve:

1. For each entry in the failed resource actions:
    a. Confirm the failed resource action was a start operation.
    b. Based on the resource indicated and the node specified in the failed resource actions:
        i. Look for and correct any external issues that would prevent the node from starting the specified resource. For example if BeeGFS IP address (floating IP) failed to start, verify at least one of the required interfaces is connected/online and cabled to the right network switch. If a BeeGFS target (block device / E-Series volume) is failed, verify the physical connections to the backend block node(s) are connected as expected, and verify the block nodes are healthy.
    c. If there are no obvious external issues and you desire a root cause for this incident, it is suggested you open a case with NetApp support to investigate before proceeding as the following steps may make root cause analysis (RCA) challenging/impossible.
2. After resolving any external issues:
    a. Comment out any non-functional nodes from the Ansible inventory.yml file and rerun the full Ansible playbook to ensure all logical configuration is setup correctly on the secondary node(s).
        i. Note: Don't forget to uncomment these nodes and rerun the playbook once the nodes are healthy and you are ready to failback.
    b. Alternatively you can attempt to manually recover the cluster:
        i. Place any offline nodes back online using: `pcs cluster start <HOSTNAME>`
        ii. Clear all failed resource actions using: `pcs resource cleanup`
        iii. Run pcs status and verify all services start as expected.
        iv. If needed run `pcs resource relocate run` to move resources back to their preferred node (if it is available).

Common Issues

BeeGFS services don't failover or failback when requested

Likely issue: The `pcs resource relocate` run command was executed, but never finished successfully.

How to check: Run `pcs constraint --full` and check for any location constraints with an ID of `pcs-relocate-<RESOURCE>`.

How to resolve: Run `pcs resource relocate clear` then rerun `pcs constraint --full` to verify the extra constraints are removed.

One node in pcs status shows "standby (on-fail)" when fencing is disabled

Likely issue: Pacemaker was unable to successfully confirm all resources were stopped on the node that failed.

How to resolve:

1. Run `pcs status` and check for any resources that aren't "started" or show errors at the bottom of the output and resolve any issues.
2. To bring the node back online run `pcs resource cleanup --node=<HOSTNAME>`.

After an unexpected failover, resources show "started (on-fail)" in pcs status when fencing is enabled

Likely issue: A problem occurred that triggered a failover, but Pacemaker was unable to verify the node was fenced. This could happen because fencing was misconfigured or there was an issue with the fencing agent (example: the PDU was disconnected from the network).

How to resolve:

1. Verify the node is actually powered off.

> If the node you specify is not actually off, but running cluster services or resources, data corruption/cluster failure WILL occur.

2. Manually confirm fencing with: `pcs stonith confirm <NODE>`

At this point services should finish failing over and be restarted on another healthy node.

Common Troubleshooting Tasks

Restart individual BeeGFS services

Normally if a BeeGFS service needs to be restarted (say to facilitate a configuration change) this should be done by updating the Ansible inventory and rerunning the playbook. In some scenarios it may be desirable to restart individual services to facilitate faster troubleshooting, for example to change the logging level without needing to wait for the entire playbook to run.

> Unless any manual changes are also added to the Ansible inventory, they will be reverted the next time the Ansible playbook runs.

Option 1: Systemd controlled restart

If there is a risk the BeeGFS service won't properly restart with the new configuration, first place the cluster in maintenance mode to prevent the BeeGFS monitor from detecting the service is stopped and triggering an unwanted failover:

```
pcs property set maintenance-mode=true
```

If needed make any changes to the services configuration at `/mnt/<SERVICE_ID>/_config/beegfs-.conf` (example: `/mnt/meta_01_tgt_0101/metadata_config/beegfs-meta.conf`) then use systemd to restart it:

```
systemctl restart beegfs-*@<SERVICE_ID>.service
```

Example: `systemctl restart beegfs-meta@meta_01_tgt_0101.service`

Option 2: Pacemaker controlled restart

If you aren't concerned the new configuration could cause the service to stop unexpectedly (for example simply changing the logging level), or you're in a maintenance window and not concerned about downtime you can simply restart the BeeGFS monitor for the service you want to restart:

```
pcs resource restart <SERVICE>-monitor
```

For example to restart the BeeGFS management service: `pcs resource restart mgmt-monitor`

Legal notices

Legal notices provide access to copyright statements, trademarks, patents, and more.

Copyright

http://www.netapp.com/us/legal/copyright.aspx

Trademarks

NETAPP, the NETAPP logo, and the marks listed on the NetApp Trademarks page are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.

http://www.netapp.com/us/legal/netapptmlist.aspx

Patents

A current list of NetApp owned patents can be found at:

https://www.netapp.com/us/media/patents-page.pdf

Privacy policy

https://www.netapp.com/us/legal/privacypolicy/index.aspx

Open source

Notice files provide information about third-party copyright and licenses used in NetApp software.

Notice for E-Series/EF-Series SANtricity OS