PolyFS - The Easiest to Use Distributed Filesystem

Katharos Technology

ABSTRACT: In the software community today, there is a shocking lack of easy-to-use, Open Source, distributed filesystems. Shared storage is crucial to any business beond the scale of single server, and a distributed filesystem is the most universal way to enable shared storage. PolyFS seeks to close the gap in shared filesystems and provide a solution that is easy to use, manage, and deploy while at the same time providing top-of-the-line performance. PolyFS, while not being limited to any particular application or usage, will be designed to include a Docker deployment and volume plugin and will provide an end-to-end solution for distributed volumes hosted by and for Docker Swarm.

NOTE: This is a work-in-progress design document outlining the plan for the design of PolyFS. This document will be updated as we discover more and change the design. The architecture in outlined in this paper has not been tested as of yet.

Motivation

As an organization with experience running stateful containers in the Cloud with Docker Swarm, we are well acquainted with the need for a distributed Docker storage solution. When running applications on a Docker cluster, you have to have a way to make sure that each application can get to its persisted data. Unlike running an app on a single server, you cannot use the local disc for storage because you never know which server a particular container may start up on. You could constrain your app to run only on a specific server, but that eliminates much of the advantages of having the Docker cluster in the first place. You want to be able to view your cluster as a pool of resources that you can run your applications in, without having to worry about where those resources are comming from.

The need for shared storage is not limited to Docker. Some applications *require* that you provide some form of a shared filesystem in order to cluster the application. The only way that you can run these applications in a fault tolerant way for high availability is to provide a highly available distributed filesystem.

Existing Solutions

There are a number of existing distributed filesystems, but so far we have not found one that statisfies our requirements:

Project	Problems
Ceph	Large and complicated to deploy and manage
Gluster	They say not to run databases on it and we have heard that it can be slow
LizardFS	Unresponsive dev team, and support is low while they rewrite it
MooseFS	Proprietary high availability system that is not Open Source
Portworx	Not Open Source
SeaweedFS	Afraid of the lack of data integrity assureance

Ceph

Ceph is a big name in distributed filesystems and was our first choice for attempting a distributed filesystem, but after trying to install and run it on Docker Swarm, we realized that it was very complicated setup. Running and managing Ceph on Swarm seemed to be difficult to impossible. Additionally other people also have stories to tell of how horrible Ceph was to try and stand up before they gave up and sought out a different solution.

Gluster

We have heard some report of Gluster's performance being low which made us unsure about trying it. Eventually we decided we would try it ourselves until we read the following in their documentation:

> Gluster does not support so called "structured data", meaning live, SQL databases. Of course, using Gluster to backup and restore the database would be fine.

LizardFS and MooseFS

LizardFS is a fork of MooseFS, an easy to deploy distributed filesystem, but they have been developed separately for the last 6 years. LizardFS is developed more openly than MooseFS with the biggest difference being that MooseFS's HA deployment software is proprietary. LizardFS appeared for a long time to be the filesystem that we were looking for. It was super easy to deploy and manage, and we successfully ran it on a Swarm cluster and wrote a Docker volume driver for it. Everything seemed about right until we discovered that development was stopping on the current version in favor of a rewrite. This ocurred after about 7 months of silence on the part of the dev team. This served to remove our confidence in LizardFS, and, due to the closed source HA solution for MooseFS, left us without a filesystem.

Portworx

Portworx has an impressive offering that achieves most of what we are looking for, but the installation requires more setup on the Docker host machine that we would like and thier software is not Open Source.

SeaweedFS

SeaweedFS is an extemely interesting project that has gotten a lot of attention on GitHub. SeaweedFS seems to be the most promising Open Source project for distributed filesystems out right now. The issues that we have with SeaweedFS is the lack of assurance when it comes to data integrity, and its unstability as a quickly developing tool. There are still concerns that have yet to be addressed and SeaweedFS is just too immature and untested for us to trust our production data with it yet.

Summary Of Existing Solutions

None of these options have satisfied our requirement for an Open Source distributed filesystem that can run in Docker containers on Docker Swarm and provide shared Docker volumes for the Swarm. It is worth noting that there are promising projects for providing storage for Docker containers on Kubernetes, such as OpenEBS and Rook, but we are looking for a solution that can run in Docker Swarm, or anywhere else.

After months of searching for a satisfactory existing solution in vain, we have decided to atempt to build our own filesystem tailored to our specific goals.

PolyFS Goals

Ease of Use and Simplicity

One of the highest concerns for PolyFS is ease of use and, by extension, simplicity. Building a distributed filesystem is a difficult job, but running one should not have to be. We want PolyFS to be as easy as absolutely possible to think about, deploy, and manage. Our design seeks to eliminate as many unnecessary components and services as possible. There is no dependency on any form of external database and the system requirements are limited to having FUSE installed on the machines that mount the filesystem.

The filesystem should do everything that it can to manage itself so that there is very little operational demand. You should not have to struggle to understand how to run PolyFS wherever you need it.

Performance and Reliability

Obviously no amount of ease of use is actually useful if the performance and reliability of the system is not adequate. PolyFS should perform as well or better than the other existing filesystems and you should have no fear that you data will be any less safe with it than with your local hard drive. Our design decisions should put no bottelneck on how fast the filesystem can perform or on how much storage you can add to the cluster.

Lightweight

PolyFS should be able to run on commodity hardware. You should not need an exhorbitant amount of CPU, memory, iops, or storage to successfully deploy PolyFS. PolyFS will use whatever hardware you have and run the filesystem on top of it without complaining. The higher performance your servers, the faster the filesystem will run, but it should not take high performance servers just to get the filesystem functioning properly.

PolyFS will do everything it can to use as little system resources as possible with a goal of being able to run PoylFS on the same servers that are running your application workloads. You should not need a dedicated cluster of servers to host PolyFS. You should be able to run it on the servers that you already have and make use of whatever storage is available on those servers. The servers in the cluster can be similar or vastly different in available resources. PolyFS will balance the load of the filesystem across the cluster to maximize performance and availability.

Scalability

There should be no limit to the number of servers that PolyFS can scale to. PolyFS will distribute the load evenly across the width of the cluster to balance resource usage and maximize network throughput.

Docker Swarm Integration

With PolyFS we will provide the documentation and configuration necessary to stand up PolyFS on Docker Swarm, along with a Docker volume plugin that can be used to mount Docker volumes on top of the fileystem. Kubernetes support may be added later. PolyFS will not be designed specific to any orchestrator, though, and should be able to be integrated into any system. Docker will not be required to use PolyFS.

PolyFS Design

PolyFS's overall design can be seen in **Figure 1**.

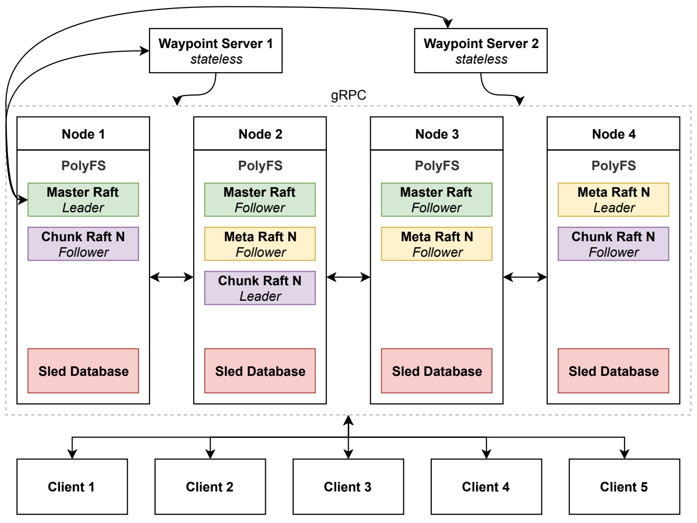

Figure 1: PolyFS Architecture

PolyFS is distributed as either a Docker image or a single binary that is deployed to all the servers in the cluster. There are only two different kinds of services in PolyFS: the **Waypoint Servers** and the **Storage Servers**.

Waypoint Servers

The waypoint servers are stateless services that do one thing: they tell any node attempting to join the cluster who the active cluster leader is. The cluster leader will ping the waypoint servers periodically to make sure that they stay up-to-date.

Waypoint servers are incredibly simple and they have only two RPCs:

- `get_master() -> Ip`
- `set_master(Option<Ip>)`

The waypoint server will store the master IP address set by `set_master()` in memory and return it to any server that calls `get_master()`. The waypoint servers will not incur any practical load on the server that they run on and there is no reason that you cannot run them on the same servers that your storage servers are running on.

The Waypoint servers only play a role when new storage servers join the cluster. The cluster join procedure can be seen in **Figure 2**.

Server Join Procedure

The list of waypoint servers are provided on the commandline to each PolyFS storage server. When a storage server starts up it selects a random waypoint server to try to get the acting master IP from. If it can't connect to one waypoint server it will try the others in the list. If the storage server connects to the waypoint server and recieves a null address, it will wait a couple heartbeats to make sure that, if there is another master, it will have time to register itself on the waypoint server. Once the server gets the IP of the acting master, it will be able to join to the cluster by contacting the master. If there is no other acting master to be found, the new node will assume it is the only node and start a new cluster.

Any number of waypoint servers can be run to make sure that storage servers can join the cluster even if one waypoint server is down.

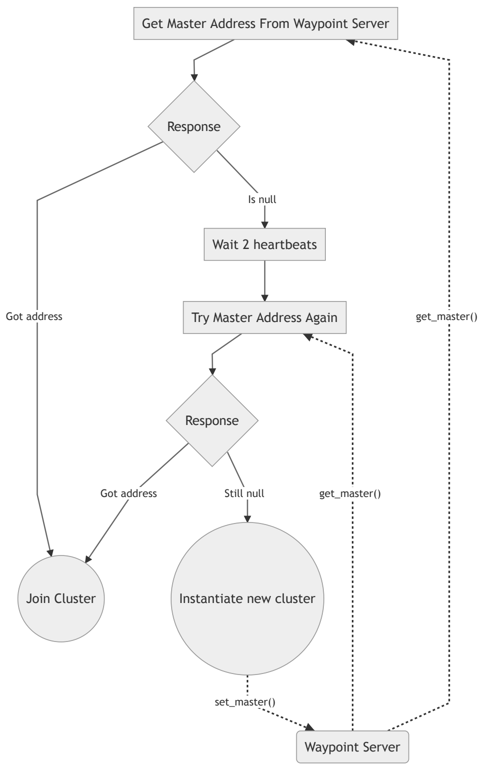

Figure 2: Server Join Procedure

Storage Servers

The storage servers take on all of the the actual work in the cluster and are designed to support every need of the filesystem. All of the elections of cluster masters and other cluster management jobs will be completely automated. An operator will not have to make any distinction between different servers in the cluster; he can view the entire cluster as a cluster of identical servers. This will make it particularly easy to deploy on orchestraters like Docker Swarm and Kubernetes, and it grants the ability to scale the filesystem with a push of a button.

Cluster Roles

There are three different roles that servers in the cluster may play: **master**, **metadata server**, and **chunk server**. Most likely, all servers in the cluster will serve as metadata and chunk servers, while a few of them will additionally play the master role as well.

> **Implementation Detail:** The different roles that run on the same server will probably be run in separate threads to parallelize their processing. The roles are relatively independent of eachother and should not have to share much, if any, data with each other, which will make the threading easy to implement. Each role thread will use asyncronous processing to handle its communication and storage operations.

The Master Role

Masters are the "brain" of the cluster. They will check on the availability of nodes in the cluster, how much storage each node has, and they serve as an index to point clients to the servers that have the metadata or chunk that they are looking for. The leader of the Master raft group is responsible for carying out all cluster management tasks such as rebalancing and promoting/demoting nodes.

Master Group Membership

The master role of PolyFS is made up of the Master raft group, which is indicated by the blue boxes in **Figure 1**. The very first PolyFS node that is created will set itself up as the leader and only member of the master raft group. As more nodes are added to the cluster, and the node count reaches three, the leader will automatically register the two new nodes as followers in the master raft group. This grants a fault tolerance of one node. If the node count reaches five or more, the leader will make sure that five servers are a part of the master group to allow for a fault tolerance of two nodes. While the behaviour is configurable, by default, the leader will not add more than five nodes to to the master group because that will reduce the performance of writes to the master group state.

If one of the nodes in the master group go down permanently (by default that means it is unresponsive for 30 minutes), the active leader of the master group will select another node in the cluster, if available, and add it to the master group to bring the master member count back up to its goal (either three or five, as described above).

Cluster Status

The status of the cluster is monitored by the master leader through heartbeats that are sent to it from every other node in the cluster. The Leader keeps track of each node's IP and information about the node such as available storage space and what metadata and file chunks it stores.

The Index And Data Access/Placement

Another job of the master cluster is to store the cluster index. The index keeps track of the ranges of metadata and file chunks that reside on the metadata and chunk servers in the cluster. In order to understand the index we have to think about how clients will access file data.

When a client needs to request file data from the cluster, it will have only a few different queries that will need to make (function names and types are approximate and there may be other non-listed queries):

- `get_inode_id(parent_inode_id: int, filename: string) -> int`
- `get_file_metadata(inode_id: int) -> FileMetadata`
- `get_chunk(chunk_id: int) -> ChunkData`

Each of these queries are made with a set of parameters. Before the actual query is made, the parameters are hashed and then serialize into a byte array. This byte array becomes the key that the data is indexed by in the cluster.

The goal of the index is to to tell clients which server in the cluster holds the data for a particular key. This brings us to how the data is distributed throughout the cluster. The the data asociated to each key is stored in regions of contiguous byte-sorted key-value pairs. Because the keys are hashed before they are stored and storted, the data is guaranteed to be randomly distributed across the cluster, removing the risk of hotspots. Each region of keys are turned into its own raft group that is used to replicate the data and provide fault tolerance. It is the master's job to decide where to put each region in the cluster and to decide when to split a region into two regions so that the data can be rebalanced.

Metadata And Chunk Servers

Metadata servers serve file metadata including file attributes and where to find the file contents on the chunk servers. In a similar way, chunk servers serve the actual data in the files. For both types of servers, the data is split up into chunks and distributed evenly to random locations across the server. The random distribution make sure that there are no hotspots in file access, and the even distribution across the cluster makes sure that the load is spread across all of your servers.

Each data chunk is made up of a Raft group that handles replicating the data to provide fault tolerance.