Exo-clones: Better Container Runtime Image Management across the Clouds

Richard P. Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Christos Karamanolis, and Rawlinson Rivera

VMware

Abstract

Our key innovation is to allow volume snapshots in VDFS (our native hyper-converged distributed file system) to be exported to a stand-alone regular file that can be imported to another VDFS cluster efficiently (zero-copy when possible) called *exo-clones*. Our exo-clones carry provenance, policy, and similar to git commits, the fingerprints of the parent clones from which they were derived. They are analogous to commits in a distributed source control system, and can be stored outside of VDFS, rebased, and signed. Although they can be unpacked to any directory, when used with VDFS they can be mounted directly with zero-copying and are instantly available to all nodes mounting VDFS. VDFS with exo-clones provides the format and the tools necessary to both transfer, and run encapsulated applications in both public and private clouds, and in both test/dev and production environments.

1 Introduction

Applications that are deployed using a continuous deployment pipeline must be able to use storage in a variety of different environments, e.g., in development on a laptop or workstation, perhaps mounting a NAS, in a test deployment on a test cluster, or in an actual deployment on a cloud platform or software defined data center (SDDC).

In each of these environments, the application's storage must be able to provide some key basic functionality. When the application is running in a deployment environment the operator of the application must be able to monitor the application's IO usage, tune its performance, set storage and networking policies or service-level-agreements, migrate it to a different storage environment, and backup and recover the application if it crashes. When an application is being developed, the developer must be able to build and deploy the application in a test environment where it will behave as similarly as possible to how it executes in production, and can be inspected and debugged as it runs.

In this paper we explore what becomes possible in some cases, or more easily and more efficiently done in others, when an organization's storage infrastructure enables them to treat the storage consumed by applications in production and development, across multiple environments, in the exact same way, using a common set of tools that manipulate a single storage object: an *exo-clone*. An exo-clone is a volume clone that lives outside of the file system as a regular file. Exo-clones use a combination of block-level delta encoding between snapshots as well as a deduplication capability with volumes common to both sending and receiving parties to maintain a small network footprint, e.g., potentially much smaller than Docker file system layers.

To use exo-clones, an organization would need to use a file system that can efficiently do an exo-clone import, export, and support all other exo-clone operations. They would deploy such a file system on their storage infrastructure in each of their environments. Exo-clones can be most efficiently supported on more modern file systems that already support file and volume clones, like our virtual distributed file system (VDFS), for which we added support for exo-clones. Since our VDFS can support multiple backends including VSAN or RADOS [4], or even a POSIX directory, VDFS can be deployed as a portable file system layer to bring exo-clone support to a variety of environments, e.g., on-prem, in a public cloud, or on a developer workstation. Alternatively, native exo-clone support can be added to other existing file systems such as BtrFS or ZFS. While we are excited about bringing exo-clone support to other non-VDFS file systems, in this paper, we discuss specifically how *VDFS* supports exo-clones to serve as an example implementation and as a gold standard for expected performance behavior of different exo-clone operations.

Importantly, we intend that exo-clones be a standard

file format so that organizations do not need to install VDFS everywhere they use exo-clones. Just as with regular files, exo-clones can be easily transmitted, stored, backed up, shared, or distributed with other 3rd party (non-VDFS) systems. Since the file format is standard, we hope other file systems, and storage systems in general will adopt exo-clones to increase their usefulness.

For example, this would allow an organization to perform an incremental backup to S3 by merely uploading periodically taken exo-clones of that volume to S3 as regular files. This is easy to do as S3 has many existing tools and APIs for uploading regular files. Or exo-clones could also serve as a more efficient file system layer abstraction for Docker which could allow Docker to efficiently represent changes as exo-clones rather than sending a whole new file system layer when some few files or blocks are modified. A more efficient layer abstraction would make it feasible for Docker to extend its runtime versioning to application state: data that is read or written by the application while it is running, like media content, configuration data, or centralized databases, which currently Docker does not support (i.e., you cannot push/pull Docker instances, only images).

Organizations must simplify and standardize how applications consume and manage storage. The current complexity of application version management is so paralyzing that organizations cannot upgrade from insecure application versions even a year after the vulnerability is reported [1]. Exo-clones could become a standard that allows organizations to overcome protocol and storage format incompatibilities that prevent them from rapidly deploying, backing up, migrating, and revision controlling applications in different storage environments.

2 Background

We introduce exo-clones as a potential standard file format for sharing file system changes across different storage systems across different environments. We also introduce an efficient, example implementation supporting exo-clones on one particular file system, our VDFS [2]. Throughout the remainder of this paper, we will refer to exo-clones as a feature of VDFS as that is currently the only implementation.

VDFS is being actively developed in VMware as a next generation file system service. VDFS is a hyper-converged distributed POSIX file system with advanced features such as file clones, volume clones, and volume snapshots. Part of the design of VDFS is similar to Btrfs. VDFS uses copy-on-write B-tree to map file name space to blocks and maintains reference counts of blocks to support features such as snapshots and clones. VDFS is built on top of a shared block storage platform, such as VSAN. VDFS will support parallel data path and distributed locking in a way similar to GPFS. A few more features are added to the VDFS product plan to better support exo-clone. The basics of these features and how they are used by exo-clones are discussed below.

File Clone. A file can be cloned instantly to another file on the same volume or different volume. Future writes to any of the cloned files will use Copy-On-Write (COW) to preserve contents of unmodified files.

Volume Snapshot and Clone. A VDFS file system contains a set of volumes. Volumes can be instantly cloned by doing copy-on-write when the clone or original is written to. Volumes contain a set of snapshots that can also be instantly cloned.

Snapshot Diff. The difference between two snapshots of a volume can be efficiently calculated, which has a cost proportional to the amount of metadata changed between them. The diff contains both metadata changes, e.g., file creation, and file data changes.

3 Design

Exo-clones exist in two states: (1) as a special kind of volume clone in the file system (e.g., VDFS), where they can be mounted, modified, cloned, and otherwise operated on in ways similar to how `git` manipulates commits, and (2) as a regular file that can be transferred to other storage systems.

The operations supported by VDFS on exo-clones are the basic operations supported by `git` on commits:

checkout: Create a writable clone of an existing exo-clone in order to make changes, discussed in Section 3.1.

commit: Commit changes made to a checked out exo-clone as a new exo-clone, discussed in Section 3.1.

import/export: Export an exo-clone volume to a file representation that can be copied to another environment, discussed in Section 3.1. Conversely, import can import an exo-clone file so it appears as a volume that can be mounted or checked out, discussed in Section 3.3.

rebase/merge: Apply or merge a set of changes made based on one exo-clone to a different exo-clone, while resolving any potential conflicts, discussed in Section 3.4.

We now discuss how VDFS implements the above exo-clone operations efficiently for whole file system volumes that can be multiple terabytes large.

3.1 Creating an Exo-Clone

When an exo-clone exists as a volume in VDFS, the whole exo-clone has a reference count (to detect if it is orphaned), and is read-only. Initially, a VDFS cluster will have no exo-clones, and may have one or more volume clones and snapshots. An exo-clone is created by first making a clone of an existing volume snapshot or

exo-clone. This operation is similar to a `git` checkout of a previous commit, except it applies to an entire file system volume and so we call it *checkout*.

When we create an exo-clone we must assign a UUID and afterward, the exo-clone cannot be modified (i.e., exo-clones are immutable objects). The alternative to identifying exo-clones with a UUID would be a content hash using, e.g., SHA1. Exo-clones are identified with a UUID to avoid hashing the contents of updated content, which would be excessively expensive at the scale of a file system volume. Since exo-clones are implemented by the file system it is easier to enforce the immutability of the exo-clone by not allowing any writes to the exo-clone. At the same time, checksumming by the underlying storage layer (e.g., VSAN) can still provide reasonable safe-guards against unintended corruption.

At any point, a volume clone can be marked as an exo-clone, at which point it is either the first exo-clone in its lineage (analogous to an initial `git` init) or it was created by cloning an existing exo-clone, or merging multiple exo-clones (analogous to a `git` commit). If it is the first exo-clone it has no parent, but if it was created from one or more exo-clones then its parent UUIDs are the UUIDs of those parent exo-clones. In either case, it is similar to a `git` commit, but again for an entire file system volume, and we call this operation *commit*. Each parent exo-clone's reference count is increased when referred to by the new child exo-clone.

Now we describe how an exo-clone volume is converted into an exo-clone file. This is called the *export* operation. First we describe the exo-clone file format, which is illustrated in Figure 1:

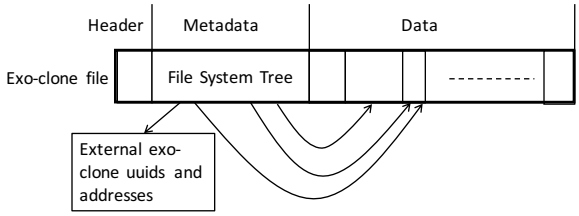

Figure 1: Exo-clone On-disk Format

Parent UUIDs: lists all the other exo-clones that are immediate parents to this exo-clone.

Dedup UUIDs: lists other exo-clones that are additional dependencies of this exo-clone. Additional dependencies contain file content referred to by the exo-clone but need not be ancestors or parents of this exo-clone. Only exo-clones can be dependencies, but any volume can be converted to an exo-clone instantly.

Metadata: the metadata part contains a serialized file system tree which represents added and deleted file system metadata, such as created files, deleted directories, etc... Our current prototype implements this by recording journal log entries that when replayed by our recovery code, reconstruct the exo-clone.

Data: all new file data are stored in this section, and are pointed by the file system tree in the metadata. The figure also shows that when data are deduplicated, multiple metadata can point to the same data extent.

Commit Metadata: this section contains some descriptive information easily parsed by user-level tools without having to mount the exo-clone as a volume.

Content UUID: the UUID that represents the state of the content of this exo-clone.

When an exo-clone file is first created, its file system meta-data is encoded (e.g., using FlatBuffers) into the payload section of the file format. Next only the block ranges that were modified via this exo-clone are cloned into the payload section, if the underlying file system supports file cloning. By cloning the blocks rather than copying them, we avoid writing file data and only need to write metadata (directory entries, pointers to data, etc), so creating these files in VDFS is very efficient and consumes only space to hold meta-data.

It is important to not scan the entire exo-clone volume to produce the file. VDFS uses a COW B-tree to represent a volume, and clones the B-tree when creating a new snapshot. COW B-tree file systems like VDFS and BtrFS use a copy-on-write B-tree algorithm, and so when a volume is cloned and then modified, only B-tree nodes that were modified are duplicated, and unmodified B-tree nodes are shared as in Figure 2 where FOO is removed and FOO' is added. Therefore we can determine what data and meta-data has been changed relative to the parent UUID by comparing the parent snapshot from which the current volume was created. Furthermore, we can limit our comparison of these two B-trees to only the B-tree nodes which were actually modified.

This is how VDFS performs a zero-copy *export* of an exo-clone volume to a regular file representation without scanning unchanged meta-data, data, and without performing any hashing of content.

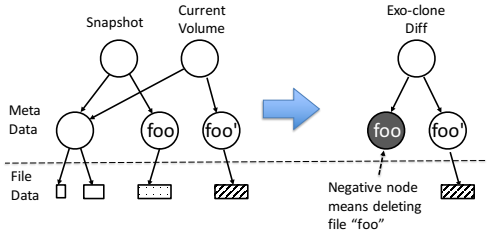

Figure 2: How to get exo-clone diff from snapshots

3.2 Exo-clones Outside of VDFS

An exo-clone can be copied outside of the original VDFS instance and used by another VDFS cluster, or in a 3rd-party storage system. As discussed in Section 3.1, the file format provides a section for *Dedup UUIDs*. This section is optional, but if used, it can dramatically reduce the size of exo-clone files so they can be efficiently transmitted over the network.

When creating an exo-clone file the user can choose to deduplicate the exo-clone if they want to make the resulting file as small as possible for transmission. When deduplicating an exo-clone, VDFS selects some number of other exo-clones that it believes the receiving party will also have, and VDFS rewrites the exo-clone such that it refers to the data in the other exo-clones. All exo-clones that become dependencies in this way are listed in the Dedup UUIDs section of the file format.

Even though exo-clones in VDFS exist as volumes with uncompressed block data consuming storage capacity, they can be compressed with a tool like `gzip`, just like `tar` archives, before they are stored in a 3rd-party storage system, e.g., for incremental backup. In this use-case, the information in the Parent, Dedup, and Content UUID sections will allow customers and operators to write simple programs to ensure all dependencies are present for an exo-clone, while simultaneously reaping the benefits of deduplication, without having to mount the exo-clone as a volume.

3.3 Mounting Exo-Clones

The goal with mounting exo-clones in VDFS is to avoid decoding and deserialization of the vast majority of the exo-clone file data. We call this zero-copy *import*. This operation is similar to a fast-forward merge operation in `git`. Import follows two steps: (1) verify mounting is possible, (2) deserialize the file system tree serialized in the meta-data section, and finally (3) increment the reference counts of all file data blocks referred to by the exo-clone.

For step one, we simply ensure all exo-clones listed in the Parent UUIDs and Dedup UUIDs sections are locally available and can be mounted by VDFS. For step two, we deserialize the file system tree updates. This is done by making a writable clone of the parent exo-clone and replaying the file system operations in the meta-data section on top of it. For step three, we walk the file system tree of the exo-clone and increase the reference count of all file extents referred to by the tree. Once all steps are complete we mark the writable clone as an exo-clone (which is read-only) and grant it the content UUID listed in the exo-clone header.

3.4 Resolving Conflicting Exo-clones

Merging exo-clones is analalgous to how a concurrent source control system would merge commits. When merging two conflicting exo-clones we have a parent exo-clone, and two child exo-clones (e.g., A and B) that each list the parent as their parent. We must construct a fourth exo-clone called a merge exo-clone that has A and B as its two parents and within which all conflicts have been resolved.

We first create the merge exo-clone by making a writable clone of one of the children, e.g., A. Second we make clones of all the files that were added or modified in B into the merge exo-clone. If no two files were modified (or added) in both A and B then there are no conflicts and we can commit the merge exo-clone by marking it read-only and assigning it a UUID. If there were conflicts, then we do a third step: for each file that conflicts (was modified/added in both A and B) we create clones of both A and B's version in the merge exo-clone. This way all the conflicting files can be directly modified by the user until the user has manually resolved conflicts as they see fit. Once they are done, they can manually conclude the merge, at which point it is marked read-only and assigned a UUID.

4 Related Work

Docker is today's de facto standard for container runtime image management. It uses a file system layer format to represent layers of an application, but changes to any file in that layer are represented as an entirely new file system layer, even though other files in the layer have not been modified. To overcome this, Docker expects developers to place files that are frequently updated (e.g., media assets, model files, sound and image files, etc...), either in development or by the application at run-time, in a separate volume. This decision forces the user to find another solution to manage changes to these files and then to manually map them in when running the container. Exo-clones efficiently track changes to any file and do not require bifurcating the developer and management interface based on the size of files and the rate at which files are revised. With exo-clones Docker can efficiently represent the entire application, not just the portions that are infrequently revised.

Recent local file systems like ZFS and BtrFS have the ability to send and receive volume snapshots using custom stream format. They stop short at both efficiency (zero-copy) and change tracking features (merge and rebase). ZFS send iterates all metadata of a snapshot at block level and form a logical log like zdump file. The zdump file includes the create/delete/write op-

erations which can be replayed at the target file system. Btrfs send compares two file system trees (skipping common subtrees) and generates a logical log (dump file). Files cloned at the source remain cloned when the volume is recreated at the target.

Source version control systems like Perforce and various git extensions for handling large files, have their own ways to support binary files by only versioning metadata and store the binary data as a whole. Since these tools are not in the IO path, users must manually track changes made to a separate working directory, and cannot utilize copy-on-write making them impractical for tracking whole volumes or providing consistent point-in-time snapshots.

Solutions exist for other use-cases that can be addressed by exo-clones. Users of VDFS can backup volume snapshots as exo-clones that are copied off-site. This provides the same kind of incremental backup provided by SnapVault except that SnapVault provides a way of searching through backed up content. On the other hand, VDFS exo-clones are regular files that can be stored in any 3rd party storage system, like S3 or a NAS. This gives users control of where to store their backup.

5 Comparison to Other Approaches

Developers build applications on top of a base layer, in Docker, this is often the UBUNTU base layer, which is just the Ubuntu LTS distribution. When a change happens to a layer, the new layer must be retransmitted to any developers that want to rebuild their own application layer on top of the new base update, or to deployment nodes that want to run the newer container. With exo-clones, only the chunks that are modified must be retransmitted, not the whole layer, and we examine the effects this has on network overhead in this experiment.

Our current prototype is very early in development and does not yet support block or variable length chunking so we simulate our results using the deduplication tool published by Meister [3]. We look at how much data must be sent to the application environment, both for Docker and VDFS exo-clones, when a base layer is updated.

Since the UBUNTU base layer is updated less frequently (six month intervals for security) it also makes sense to look at another base layer that may be transmitted more frequently, but also would be specific to an organization. For this analysis we looked at two versions of ESXi built one week apart as a PXE-bootable image.

Figure 3 shows our results. LTS UPGRADE shows the size of the container image file that must be transmitted over network to perform an LTS Upgrade. Docker must transmit the entire FS layer for the new version of LTS 14.04.3 which is 199MB large. If the upgrade is encoded as an exo-clone relative to the older LTS 14.04.2, either with variable or block-aligned chunking, then only a 24MB exo-clone need be transmitted, a file which is 88% smaller than the necessary Docker FS layer. If Docker were modified to support whole-file deduplication it would generate layers that are the size of WHOLE FILE, i.e., 58MB for LTS. For a PXE upgrade, which consists primarily of zipped image files, Figure 3 shows no improvement until a variable-length chunking scheme is used, in which case the variable-length chunking exo-clone is 25% smaller than either the whole-file exo-clone or the plain Docker FS layer.

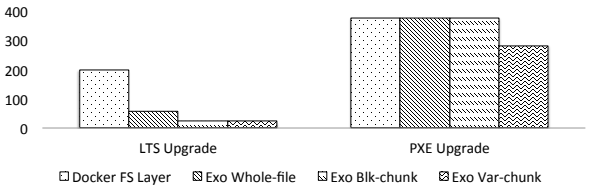

Figure 3: Projected Container Image Sizes in MBs

6 Conclusion

Docker and other application revision tracking technologies, or incremental backup solutions, or other use cases for syncing file system state across multiple storage environments assume the file system is incapable of efficiently making and managing writable snapshots, or of deduplicating common content. Consequently each of these solutions is designed around a false trade-off: that storage solutions can exploit block-granularity copy-on-write for efficiency, but only if they are built into a specific storage stack that sacrifices portability. In this paper we have shown how these technologies and tools, specifically Docker file system layers, can be dramatically improved or simplified if modern file systems that support snapshots provide a common primitive for efficiently representing updates between different file systems.

References

[1] 2015 data breach investigations report. 15–17.

[2] LU, L., WANG, W., AUSTRUY, M., LI, Z., HUANG, G., PAI, A., AND KARAMANOLIS, C. VDFS: A cloud-centric virtual distributed file system. In *Proceedings of VMware RADIO 2015* (May 2015).

[3] MEISTER, D. *Advanced data deduplication techniques and their application*. PhD thesis, Universitätsbibliothek Mainz, 2013.

[4] WEIL, S., LEUNG, A., BRANDT, S., AND MALTZAHN, C. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. In *Proceedings of the ACM Petascale Data Storage Workshop 2007 (PDSW 07)* (Nov. 2007).