Building a Cloud File System

JEFF DARCY

Jeff Darcy has been working on network and distributed storage since that meant DECnet and NFS version 2 in the early '90s. Since then he has been a key developer on the MPFS project at EMC, product architect at Revivio, and well-known blogger on various related topics. He is currently the founder and technical leader of the CloudFS project at Red Hat.
jeff@pl.atyp.us

Cloud file systems must address challenges that are not addressed by traditional network or distributed file systems. These challenges mostly revolve around isolation, identity, and privacy, but also include features such as adaptability to frequent changes in demand for capacity or capability. In this article, I'll elaborate on some of these challenges and describe how one project, CloudFS, attempts to address them.

Cloud Problems

To understand the requirements for a cloud file system, one must consider the special properties of cloud computing. While everyone who writes about the subject has their own list of properties that define "the cloud" or make it special, one property of particular relevance is that a cloud is shared. When you use resources in a cloud, especially a public cloud, you share the underlying physical resources with others whose intentions and resource needs are totally unknown to you. Typically, this sharing is between accounts rather than actual users; each *tenant* holding an account with a cloud provider might actually be a complex enterprise representing their own large and ever-changing set of end users. In a public cloud, such as those at Amazon or Rackspace, tenants are likely to be companies. In a private cloud, tenants might be departments or projects. In all of these cases, though, the important thing is that tenants don't trust each other. They might not even trust the cloud provider very much. In many situations—especially those involving medical or financial information—they're not even legally allowed to extend such trust to others. This extreme lack of trust implies that a cloud file system must ensure that tenants are isolated from one another to a far greater degree than would be the case using a traditional distributed file system.

Processors and disks aren't the only resources that are shared in the cloud. Another often-overlooked resource that can also be the subject of conflict is identity. Every online identity lives in a certain space within which it has some specific meaning. Once, each physical machine had its own identity space because each ID conferred only access to local resources. Many have returned to that model within their virtual machines today, because machine virtualization provides a similar level of isolation. In fact, it's entirely possible to run a traditional distributed file system across multiple nodes allocated within a cloud, relying on the cloud's existing machine- and network-level isolation to protect storage as well. Unfortunately, this approach is insufficient when the file system must be deployed as a shared service. A cloud provider might favor such a deployment to capitalize

on the resource-utilization efficiency that comes from sharing resources between tenants with non-correlated peak demands (which is a core value proposition of cloud computing generally) and/or to provide an "added attraction" in a public cloud. The problem with identity in such a shared deployment is that it introduces a possibility of conflict involving reuse of the same ID by different tenants (Figure 1). If tenants are responsible for assigning IDs themselves, then two tenants can assign the same ID to different users—either intentionally or maliciously, whether IDs are numbers or strings—and present that ID to the shared service. As a result, client-provided IDs are insufficient as a basis for authentication or authorization in such a service.

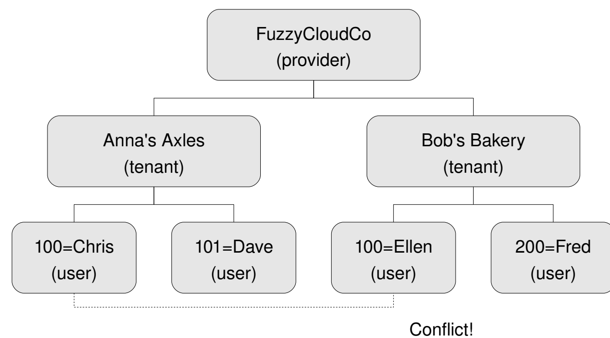

Figure 1: Illustration of a UID conflict between tenants. Anna's Axles and Bob's Bakery are unrelated tenants, but their users Chris and Ellen share the same UID.

One common approach to this problem in traditional environments is to say that machines in a network cannot simply assign their own user IDs. Instead, identity management is "outsourced" to something like Kerberos, establishing a new, centrally administered ID space which can be shared between clients and servers. How might this be applied to the cloud? Consider the scale of such a service at a large public cloud provider. There might be thousands of tenants, each with thousands or even millions of users. How many new user registrations per day would that be? How many ID-mapping operations per minute? Even if such a system could be implemented cost-effectively, tenants would resist any requirement to register their own users with the provider's identity service. They would object for privacy reasons, and also for operational-complexity reasons. This is especially true if they're already running their own identity management systems internally and would thus be forced to support two such systems side by side. Centralized identity management is as inapplicable in the cloud as client-side identity management.

If both of these options are precluded, what's left? The answer is the same as it has been for domain names, or for email addresses which depend on them—delegation. If a flat ID space won't work, use a hierarchical one and delegate smaller pieces to lower-level authorities—in this case tenants. Identity in the cloud has to be contextual, not just "user X" but "user X within tenant Y" as distinct from "user X within tenant Z" or any similar combination. Tenants would have complete freedom to manage their own identity space on their own machines, just as they do when using only their own services and just as they do with subdomains even in a shared higher-level domain. The only burden on the provider would be to maintain any stored end-user identities, such as the UID and GID associated with a file, so that it can be sent back later. It is not responsible for allocating these identities or for discovering them beyond the "which tenant" level.

Resource sharing might be the most important problem with which a cloud file system must contend, but it's far from the only one. Adaptability is another important feature for any cloud service, and providing that adaptability usually involves some kind of virtualization. For example, hypervisors allow many virtual machines to exist within one physical machine. In a similar way, a tenant's virtual file system might be carved out of a much larger physical file system spread across many machines and shared among many tenants. Just as the virtual machine's processor count or memory size can be changed at a moment's notice, so can the virtual file system's size and performance profile and redundancy characteristics. Besides being convenient, this preserves the basic cloud value proposition of "pay as you go" access to efficiently allocated hardware.

There, in a nutshell, are the problems a cloud file system must address: shared resources introducing issues of privacy and isolation, hierarchical identity, and adaptability to changing user needs. CloudFS, which will be discussed in the following section, represents one attempt to address these issues.

File System Virtualization

As mentioned earlier, it could be argued that current distributed file systems adequately solve the problems they were designed to solve. Examples such as GlusterFS [2], PVFS [3], and Ceph [4] allow the capacity and performance of multiple servers to be harnessed within a single file system. They handle things like distributing files across servers, striping data within files, replicating files, adding and removing servers, rebalancing and repairing, etc. Most importantly, they provide a POSIX file system API to all of this functionality, or "POSIX enough" to keep operating systems and most users happy. What some of them also provide is a convenient way to "mix and match" or even add functionality without having to rewrite what's already there. In GlusterFS, for example, most functionality—e.g., replication, striping, client and server protocols—exists in the form of "translators" which convert a higher-level file system request into one or more lower-level requests using the same API. These translators allow many local file systems on many servers to be combined and then recombined, providing layers of increasing functionality until they all combine into one translator providing one unified view on a client machine. This structure is illustrated in Figure 2.

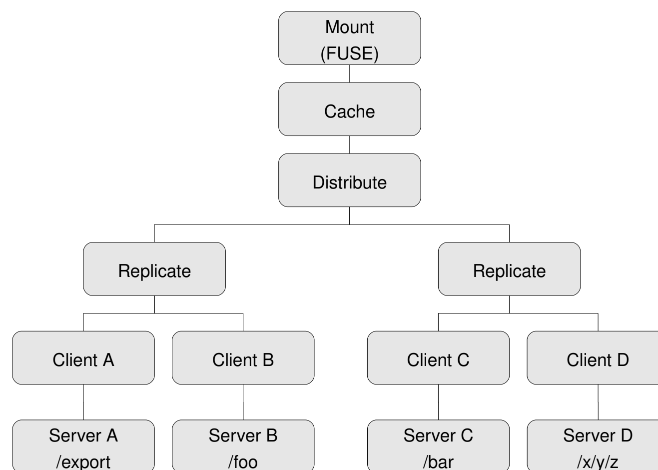

Figure 2: Translator structure in a typical GlusterFS deployment. Translators convert I/O requests from "above" into one or more "below" using the same API.

What we see in Figure 2 is several local file systems on several servers (“bricks” in GlusterFS terminology) being exported to clients, which first combine them into replica groups and then distribute files across those groups. Next, a cache translator is added; this is an example of a translator with a 1:1 mapping between input and output requests. Finally, the combined result is presented as a single mountable file system. Because they use a common interface, translators can readily be stacked in different orders and even be moved from one side of the client/server divide to the other. For example, the caching in the structure in Figure 2 could easily be done first instead of last, on the server side instead of the client.

When one seeks to add new functionality which is itself quite complex, as is the case with CloudFS, this kind of modularity and flexibility can speed development considerably. Because CloudFS is based on GlusterFS, this makes it relatively easy to provide a “virtual” file system for each tenant across a common set of “bricks” without having to build a whole new file system from scratch or modify the internals of a working and widely accepted file system.

In GlusterFS, a client’s view is effectively the union of its component bricks. CloudFS takes this same set of bricks and turns each tenant’s view into the union of per-tenant subdirectories on those bricks. If the whole file system is considered as a matrix of bricks along the X axis and tenants along the Y axis, then a server is managing a vertical slice and a tenant sees a horizontal one. This provides a complete and reasonably straightforward separation of each tenant’s namespace (directories and files) from any other tenant’s. This requires the addition of extra translators, as shown in Figure 3.

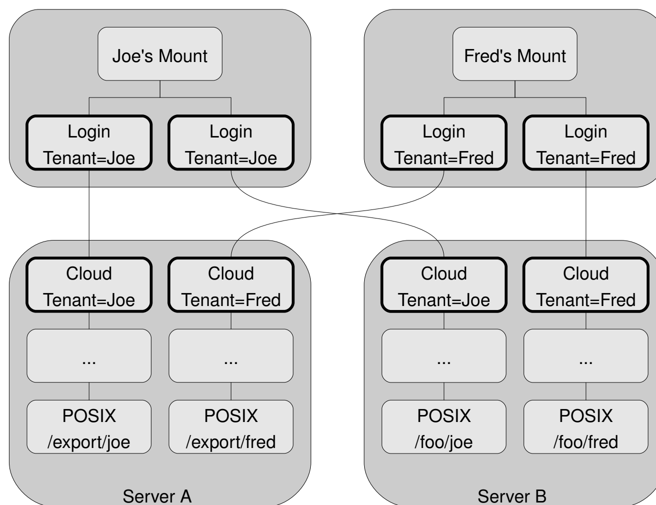

Figure 3: GlusterFS translator structure when using CloudFS. The translators with the bold outlines are provided by CloudFS.

The extra translators are added to the system by scripts which automatically rewrite both server-side and client-side configuration files according to the contents of a tenant list, including credentials for each tenant. (Actually, the credentials are identifiers for tenant-specific keys or certificates, to avoid having that sensitive information show up directly in the server or client configurations.) On the server side, the single translator stack for a brick is replaced by several per-

tenant stacks. Each per-tenant stack leads down from a "cloud" translator, which provides authentication services down to a "posix" (local file system) translator, which is configured to use a per-tenant subdirectory of the original brick. On the client side, a "login" translator is added to do the client's side of the authentication handshake. The net effect of all this is to construct a tenant's horizontal slice of the aforementioned matrix for each client mounting with that tenant's configuration and credentials. In fact, each tenant might see a different horizontal slice across a different set of servers and replicate or distribute differently within its slice, providing some of the adaptability mentioned earlier.

Identity Mapping

So far we've succeeded in virtualizing each tenant's namespace. What about their identity space? A user's effective identity on a CloudFS server is actually the combination of their tenant identity (established when the tenant connected) and the tenant-provided UID. This identity can be stored directly in an inode or embedded in an extended attribute (e.g., for access control lists), or it can be associated with a request—and likewise for GIDs. To the maximum extent possible, the actual use of these values to enforce access control should be done by the kernel rather than by duplicated code in user space, but the kernel only understands a single ID space shared by all tenants. We therefore need to map a tenant ID plus a tenant-specific UID into a unique server UID for storage, and then map in the opposite direction upon retrieval. Fortunately, this mapping does not need to be coordinated across servers. Each server can safely use its own separate mapping table, populated automatically whenever it sees a new tenant/UID pair, so long as the mapping process is reversible locally. This is done by the "cloud" translator which sits at the top of each per-tenant translator stack on the server. All instances of this translator on a server share the same mapping tables, to avoid creating duplicate mappings, but otherwise (e.g., with respect to authentication) they're separate.

Privacy and Encryption

The remaining focus of CloudFS is ensuring tenant privacy, and the main tool we use for this is encryption. In fact, CloudFS needs to do two kinds of encryption: for data "in flight" on the network and for data "at rest" on disk. Techniques and tools such as TLS for doing in-flight encryption are fairly well understood, and CloudFS applies them in fairly straightforward ways, so here we'll focus on at-rest encryption.

For cloud storage of any kind, it's important for any tenant to consider whether they trust the cloud provider with their data (or with which data). "Trust" is a funny word, though, since it can apply to intentions, skills, or diligence. It's entirely possible to trust a cloud provider's intentions and skills which allow them to secure disks that are physically under their control, but not trust their diligence when those disks leave their control—e.g., when those disks are removed from service and sold. Stories about disks being sold online while still holding medical, financial, or even defense-related data are legion, and cloud providers go through a lot of disks. If you trust your cloud provider in all three of these ways, or simply don't care whether that data becomes public, then there's no need for at-rest encryption. In all other cases, though, at-rest encryption is necessary. CloudFS takes an even harder line: if the provider has keys for your data (and a surprising

number of cloud-storage solutions require this), then they might as well have the data itself, so all at-rest encryption has to be done entirely by tenants using keys that only they possess.

Unfortunately, tenant- or client-side encryption presents its share of problems. Chief among these is the problem of partial-block writes. For encryption methods with the property that every byte within a (cipher) block affects the output, a read-modify-write sequence is necessary to incorporate the unwritten bytes. For other methods, the need to avoid re-using initialization vectors would require the same sort of sequence to re-encrypt the unwritten bytes (plus some mechanism to manage the ever-changing vectors). Adding authentication codes to prevent tampering involves many of the same problems. In all of these cases, some form of concurrency control is necessary to make the read-modify-write atomic even in the face of concurrent updates. Truly conflicting writes without higher-level locking can still clobber each other's data, just as they always have; this mechanism only prevents corruption that could otherwise be caused by the encryption process itself.

Currently, CloudFS handles this need with a combination of queuing and leasing. When a client needs to do a write that involves partial blocks, it first sends a request to the server to obtain a lease and retrieve the unaligned head/tail pieces. This lease is relinquished when the matching write occurs. Any conflicting writes received while the lease is valid will be queued behind the lease holder and resumed when the lease is either relinquished normally or revoked due to passage of time. Because there's no synchronous locking involved, the overhead for this approach is only the extra round trip for the initial lease acquisition—and even then, only when a write is unaligned.

Results

To validate the approaches described above, some simple tests were performed on machines available to the author through his employer. These consisted of nine machines which represent the low end of the node-count spectrum for CloudFS but also the high end of the per-node-capability range (each 24 cores, 48 GB of memory, 63 disks with hardware RAID, and 10 Gb/s Ethernet). Tests were done on Linux kernel 2.6.32-125.el6, including NFSv4 on a single server and with GlusterFS 3.1.2 on three servers. It should be noted that these are preliminary results for code still under active development. Results for higher node counts and more stable software versions will be published on the CloudFS project site [1] as they become available.

To test the effect of CloudFS's at-rest encryption on aggregate bandwidth, 1 MB writes were tested using iozone, with 12 threads per client machine. The results are shown in Chart 1. The first conclusion that might be reached from this result is that the underlying GlusterFS infrastructure is sound, outperforming NFS even on a per-server basis and easily using additional servers to scale beyond what the single NFS server could ever achieve. The second conclusion is that the encryption does exact a heavy toll on performance. However, the linear slope indicates that this is almost certainly a pure client-side bottleneck. To the extent that this is likely to be the result of insufficient parallelism within the encryption translator, this should be easily fixable. In the meantime, while performance is relatively poor, it is still adequate for many users in the cloud and scales well as clients are added.

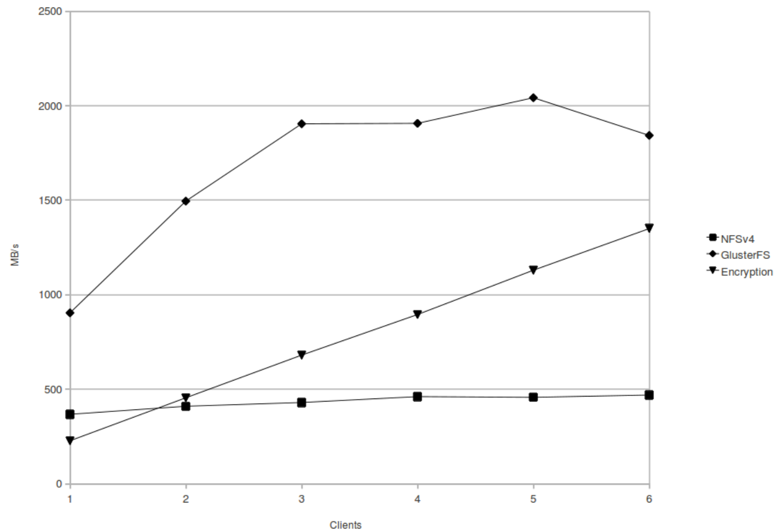

Chart 1: CloudFS encryption performance compared to NFSv4 and GlusterFS

To test the overhead of CloudFS's multi-tenant features, a different, more synchronous and metadata-intensive test was called for. In this case we used fs_mark to create many thousands of small files, again using 12 threads per client. The results are shown in Chart 2. This time, the performance is very nearly the same as for plain GlusterFS, and even better for much of the tested range. This is thought to be the result of lower contention on the servers, but bears further investigation.

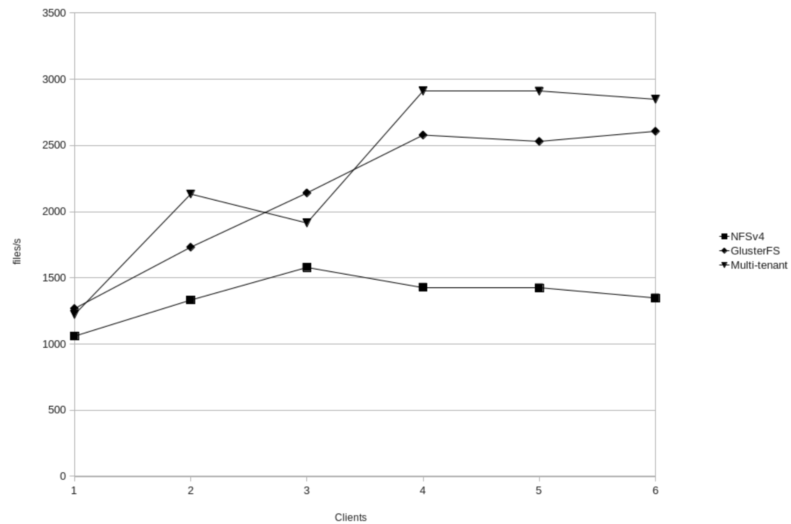

Chart 2: CloudFS multi-tenancy performance compared to NFSv4 and GlusterFS

Conclusions and Future Directions

The premises of CloudFS are that existing distributed-file system solutions already provide performance and scalability for cloud storage use and that

additional features needed to make such use safe can be added in a modular fashion without excessive sacrifice in speed. Preliminary testing seems to bear out the first point quite well. With regard to the second point, the picture is less clear. A 25% performance degradation at n=6 is a matter for serious concern even if the starting point is good; a 75% degradation at n=1 is probably unacceptable to many users. On the other hand, the modular approach taken by CloudFS means that users who do not require this level of protection need not pay the price, and the data also suggests that the price can be lowered significantly with little effort.

In addition to ongoing work on the features mentioned here and the implicit goal of making configuration/maintenance ever easier, work has already begun on several other features also of value in a cloud environment. Chief among these are in-flight encryption, easier addition/removal of servers, and asynchronous multi-site replication. This last feature is likely to be a major focus for CloudFS going forward, both to address disaster-recovery needs and to facilitate migration between clouds. Location transparency and cost efficiency are often cited as advantages of the cloud model, but are lost if moving computational resources from one cloud to another requires waiting for an entire large data set to be transferred as well.

References

[1] CloudFS: http://cloudfs.org/cloudfs-overview.

[2] GlusterFS: http://www.gluster.org/.

[3] PVFS: http://pvfs.org/documentation/index.php.

[4] Ceph: http://ceph.newdream.net/publications/.