GlusterFS 4.0 Features

NOTE: **this is not a spec**. This document is intended to describe the individual features that might together comprise GlusterFS 4.0, in a form that facilitates and aggregates discussion. That discussion should lead to a set of individual feature pages, which can then be voted on to determine the real 4.0 content. The items are grouped into three "themes" (similar to agile "epochs") that will distinguish 4.0 from its predecessors. *Italics* are used in the text as a lightweight form of cross-referencing to other features within this document.

In general terms, 4.0 is supposed to be about changes that don't fit well into the "evolutionary" progress of the 3.x series. That includes items which replace major components, require changing core code, or significantly alter the user experience. However, compatibility is still a goal. As much as possible, new code should be written to work with existing formats (e.g. for extended attributes) even if those are converted on the fly to a newer form. In some cases, a batch conversion might be necessary, but an upgrade process which requires extended downtime to scan and convert every file in a volume would be unacceptable.

THEME: node/performance scaling

FEATURE: glusterd scalability

Move membership, service registration/discovery, and configuration storage to a separate subsystem like Ceph's mon or etcd/consul. Glusterd retains a role as a higher-level coordinator, and possibly as an API server.

FEATURE: DHT scalability

In minimal form, this requires extensions along the same line as the "commit hash" approach in patch 7702, along with a grab bag of improvements in rebalance. This could probably be achieved in a 3.7 timeframe. For 4.0 and beyond, we need to start looking at a "DHT2" which handles directory and layout information more scalably, rebalances autonomously, and so on. Some of these features are linked to *data classification*, so we probably need to think hard about how to do something like this:

- Refactor DHT code to make some pieces more modular
- Re-implement some functionality as modules that can work either with (refactored) DHT1 in GlusterFS 3.x or with DHT2 in GlusterFS 4.0
- Write the core of DHT2 to use the new modules

Another necessary enhancement has to do with failover behavior. Currently, when a replicated brick fails, all of its load falls onto N-1 replica partners (just one partner for the most common two-way replication scenario). As failure becomes the norm in larger systems, this is insufficient. While both *data classification* and *sharding* offer some possibilities for allowing one-to-many failover, a multi-ring or virtual-node approach implemented within DHT itself allows proper failover even when those features aren't in use.

FEATURE: sharding

A new sharding translator - similar to current striping but implemented on top of DHT instead of below - will ease the restriction on adding bricks to a volume in multiples of the replication*stripe count. This will also help us to distribute load more effectively across other translators - e.g. NSR or EC - which might be implemented on servers using a leader/follower architecture.

FEATURE: new daemon/transport infrastructure

In the 4.0 timeframe we're likely to have more bricks per server than we do now - because servers will accommodate more disks, because we'll have bricks of different physical types (e.g. PCM vs. SSD vs. "normal" vs. shingled), because either users or we ourselves will "slice and dice" those disks more. With a
process per brick we're prone to port exhaustion, CPU/memory thrashing, and other problems. Therefore we need the higher level of coordination that comes with a multi-thread (instead of multi-process) model. This requires at least the following:

- Get own-thread and/or multi-epoll fully tested and enabled by default
- Fix the concurrency bugs that will inevitably surface elsewhere
- Implement at least minimal protection between the "tenants" (brick servers) now running within the same process

FEATURE: caching

Caching is normally thought of as a way to improve performance at clients, but it's also a way to reduce load at servers. Thus, 4.0 should include robust support for client-side caching (and prefetching) of data. The key infrastructure component to support this is lock management, including variable lock-compatibility matrices and server-initiated callbacks. Since these are also essential features for multi-protocol support (especially Samba) they have already been mostly implemented in that context. This work should be integrated into the 4.0 codebase, along with a data-caching translator that uses it to implement fully coherent client-side caching.

THEME: technology scaling

FEATURE: erasure coding

We need to get this up to production level, with respect to both behavior (e.g. proactive self-heal) and performance.

FEATURE: tiering, compliance, and data classification

At the very least, we need to have a very basic way of combining faster and slower disks/servers and encoding methods (e.g. replication vs. erasure coding) into a single volume, with some sort of migration upward/downward according to demand. The simplest method, suitable even for 3.7 based on the need being so urgent, would be to to have a tiered volume "absorb" two regular volumes that are left defined but inaccessible on their own while the tiered volume is running. For 4.0 this needs to be more integrated, and expanded to cover needs other than tiering. We have to support at least the following:

- A way to add tags to bricks, or to subdivide generic bricks into smaller ones with specific tags.
- A way to add tags to files, either explicitly (with inheritance from a parent directory) or automatically (e.g. by name or activity level).
- A generic way to sort or route files to bricks, based on a policy which reads the tags on each.
- Specific policies and tagging modules to support different use cases - tiering, compliance, topology-aware placement, multi-tenancy.

FEATURE: multiple network support

New network types, both commodity and custom, continue to proliferate. Some of these, especially board-level or enclosure-level embedded networks, will not be available to all nodes in a cluster. Others might be only partially deployed due to cost. Even for networks of the same type it might often be desirable to segregate different kinds of traffic - e.g. user I/O vs. replication vs. self-heal - onto different real or software-defined networks. The GlusterFS transport code and APIs need to be more aware of multiple networks and the need to route intelligently among them.

FEATURE: NSR

It might seem strange to include NSR as a technology-scaling feature, but along with its other advantages it's well able to take advantage of newer/faster storage and network types in a single brick. For example, a very fast PCIe PCM module might be even better used as a journal device than as a separate brick under data classification, or servers might have their own back-end network separate from that seen by clients (see *multiple network support*). A replication method that takes advantage of this heterogeneity is a natural complement to the other technology-scaling features here.

THEME: small-file performance

FEATURE: composite operations

Combining operations such as open+write+close into a single RPC can significantly reduce network round trips and improve performance. While there is no way to express these through the native POSIX/FUSE interface, we can certainly do so internally. Once that functionality is exposed through GFAPI, it also becomes available for protocols and access methods - e.g. NFS, SMB, Swift - which can benefit.

FEATURE: server stat/xattr cache or database

Caching of this information by the local file system (or VFS layer) is not always as complete or efficient as we need. Since we are - or at least should be - the only users of any files within a brick, it should be safe for us to cache this information ourselves. Diverting xattrs into a database might not only be more efficient, but would also eliminate the need for special local-filesystem configuration. Queries against this information, e.g. to make tiering decisions, could also be done in seconds instead of hours using the “traditional” method.

THEME: manageability

FEATURE: full REST API

Every operation possible through the CLI should also be possible through a REST API with full discoverability and introspection.

FEATURE: management plugins

As new translators or other components are added to the system, many of them require their own options or “hooks” e.g. into the volfile-generation infrastructure. An API should be provided to satisfy these needs, so that new features can be developed/tested and packaged/distributed as pure plugins without needing patches to the management code for them to be usable.

FEATURE: improved fault/performance metrics

Self-explanatory.