Placement Layouts in NFS

Adam C. Emerson <aemerson@cohortfs.com>
Marcus Watts <mdw@cohortfs.com>

August 29, 2014

CohortFS

As you know. . .

pNFS layouts were introduced in RFC 5661 as part of NFSv4.1 to provide access to scale-out storage. That is, clusters can

- Store data on multiple machines
- Provide direct access to data
- Synchronize updates to metadata

Provides faster throughput, decreased load on a single server.

CohortFS

As you know. . .

The layout is a recallable resource associated with a file that gives you permnission to access the data store directly.

- Recall allows the clients to be informed and clean up gracefully before access becomes available
- Refusing to grant a layout or recalling to manage conflicting access
- Client uses `LAYOUTCOMMIT` to update file metadata after modifying data
- `LAYOUTRETURN` returns all or part of layout to the server.
- Server side fencing cuts off errant clients from accessing data they shouldn't.

CohortFS

As you know. . .

In addition to arranging permission to access data, pNFS layouts also specify the protocol to access it, with NFS itself, T10 OSD protocol, and SCSI all being supported by current standards.
It also has to tell the client wanting data where to find it.

Where to find it. . .

- The Block/Volume layout of RFC 5663 associates file data with lists of extents in a virtual volume assembled of block devices.
- The Object-based layout of RFC 5664 associates file data with objects in nested striping or mirrored configurations over multiple storage devices.

CohortFS

Where to find it?

- The NFSv4.1 Files Layout of RFC 5661 only allows simple (non-nested) striping across mirrored data servers.
- The Flexible Files draft by Halevy and Haynes has essentiall the same placement as NFSv4.1 files, though disambiguates between multiple interfaces and mirroring.

Each of these designs presumes a *locator service* that centrally records and maintains information on where to find the data for various objects.

CohortFS

Where to find it!

These are a poor match for many clustering strategies:

- Most famously, Ceph uses the CRUSH, a derivative of Honicky and Miller's RUSH family of algorithms, for object placement.
- The OpenStack Swift storage system uses a distributed hash table.
- Some commercial clustered storage systems use similar algorithmic strategies.

Several modern systems do away with the locator service and thereby eliminate a point of contention, replacing it with a globally known function.

Layout for each strategy?

We could take the obvious approach:

- Write and deploy a CRUSH layout module for every platform
- Once we have pathless objects, write and deploy a Swift layout module on every platform
- And one for every other clustered storage system...

Layout for each strategy!

And now our generic, commodity network file access protocol has a specialized module for every single clustered backing store.

***** You have missed the point entirely *****

Another reason not to do that

There is no ideal placement strategy for all situations

- CRUSH works well on uniform, random access patterns, badly on sequential reads or mixed sequential/random workloads.
- SCADDAR works well on continuous media.
- Pseudo-random traversal strategies like CRUSH and RUSH_T have higher time complexity than distributed hash tables but allow one to account for the structure of the cluster.

Some filesystems might use more than one algorithm, and research on placement algorithms is still ongoing.

CohortFS

Concept

Consider the web. How do we describe handle runtime customization of a client?
We send it code.

- The cluster uses one or more globally known functions?
- Send them to the client
- Probably just once, give paramenters to supply to function in layout
- All the other parts of pNFS work as previously described

Prototype

We are developing a proof of concept implementation:

- Back-end is a heavily modified Ceph cluster
- Server interface is a Ganesha FSAL
- Device contains vector of functions, indexed list of all OSDs
- Layout contains index of function to use, parameters
- Functions are sent as C code
- Compiled by a helper in user space
- Linked as a kernel module, called by Kernel layout driver

Wait, what?

C has obvious disadvantages, of course, it’s untrusted, unsafe code linked right into the kernel, but:

- You can actually link it into the kernel, no upcalls or interpreter
- Idea of what performance can be like for testing
- It’s easy to put together and run around with

There are alternatives.

CohortFS

Alternatives, you say?

Domain specific language

- Portable
- Easy to restrict
- Could have primitives for hashes, number theoretic operations to take advantage of CPU features
- Opportunities for high level optimization
- Year long research project to design one
- Still have to compile it
- High level optimizations are expensive, better to do them centrally, but that degenerates to

Alternatives, I say

A virtual machine/bytecode

- Re-use Linux kernel virtual machine? (NFTables/BPF)
  - Not portable
  - Domain specific for the wrong domain
- Supervised runtime
  - Well understood, commonly used
  - Can use existing projects
  - Can be AOT or JIT compiled for better speed

Burn onto FPGA

- Likely not effective if functions change often
- For relatively stable functions on commonly used filesystems, it could be lots of fun

Security

Metadata servers can be put in various trust categories.
Trusted servers (those under the same administrative control as the client) could have their placement functions run essentially unsupervised or even linked into the kernel for maximum performance.
Functions from untrusted sources could be run under some combination of sandboxing and static analysis at some possible performance penalty.

Layout to Function

Rather than shoving everything in the world into on `da_addr_body` opaque, we expect a returned layout to give reference a single `deviceid4`, as well as providing information like block size, placement seed, and the like. The `deviceid4` would return a single function.

CohortFS

Function to Devices

The prototype method of including the entire OSD map does not scale to large clusters. Instead the function should generate `deviceid4`s. If the cluster natively identifies OSDs with integers (Ceph uses 31-bit integers, for example) this can easily be accomplished using whatever function the cluster would employ internally and prepending a prefix to fill out the rest of the `deviceid4`.

Functions should generate sets of `deviceid4`s, marked as suitable for read, write, erasure coding, and so forth. These should be OSDs that mirror each other.

Deviceids to Devices

Each `deviceid4` should return a multipath_list, giving relevant interfaces of the device to aid in trunking if supported by the chosen protocol.

Protocol? What protocol?

The layout concept was flawed by conflating placement and data access protocol. Ideally, the means for specifying striping patterns or the like would be specified, and data-access protocols could use whatever placement method made sense. It is too late to change that in general, but placement layouts could benefit from a `LAYOUTGETPLUS` call which would allow the client to send a list of supported transport protocols so that the server could match the layout to its capabilities.

Example

For Ceph, we could have a system where each layout contain:

- `deviceid4` specifying the function to be executed
- Inode number
- Block size
- Opaque (in this case number of placement groups in the system)

CohortFS

Example

The placement function in the layout would take the opaque, the inode number, and a block number as parameters and be a composition of:

- Wrapper, form object name of the form "`inode.block`"
- Using number of placement groups, find placement group for object
- Using placement group, execute CRUSH rules compiled to formalism
- Yield list of `deviceid4`s

Each yielded `deviceid4` would then specify an OSD.

CohortFS

Placement Layouts

In addition to locating data blocks, a placement layout like mechanism would also be able to expand the Metadata Layouts we have been developing, allowing the native mechanism of a cluster for scattering filenames between directories to be exported.

CohortFS

Other Possibilities

Extending the layout mechanism with dynamically propagated code has other applications besides placement. Systems supporting client-side erasure coding (as the Object layout does) would be more robust and capable if new erasure coding methods could be added, and clients without native support for them could be sent code for an implementation.

In Summary

- Propagating functions seems the best way for NFS to support the clustered filesystems being developed well.
- They might be used to support other things
- They can be made fast
- They can be made secure
- They *shall* be made the future