Ceph Snapshots: Diving into Deep Waters

Greg Farnum – Red hat

Vault – 2017.03.23

Hi, I'm Greg

- Greg Farnum
- Principal Software Engineer, Red Hat
- gfarnum@redhat.com

Outline

- RADOS, RBD, CephFS: (Lightning) overview and how writes happen
- The (self-managed) snapshots interface
- A diversion into pool snapshots
- Snapshots in RBD, CephFS
- RADOS/OSD Snapshot implementation, pain points

Ceph's Past & Present

- Then: UC Santa Cruz Storage Research Systems Center
- Long-term research project in petabyte-scale storage
- trying to develop a Lustre successor.

- Now: Red Hat, a commercial open-source software & support provider you might have heard of :)

  (Mirantis, SuSE, Canonical, 42on, Hastexo, ...)
- Building a business; customers in virtual block devices and object storage
- ...and reaching for filesystem users!

Ceph Projects

OBJECT	BLOCK	FILE
RGW S3 and Swift compatible object storage with object versioning, multi-site federation, and replication	**RBD** A virtual block device with snapshots, copy-on-write clones, and multi-site replication	**CEPHFS** A distributed POSIX file system with coherent caches and snapshots on any directory

LIBRADOS

A library allowing apps to direct access RADOS (C, C++, Java, Python, Ruby, PHP)

RADOS

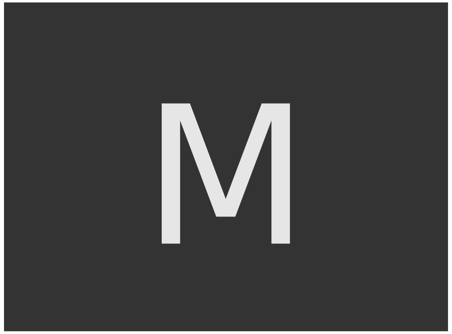
A software-based, reliable, autonomic, distributed object store comprised of self-healing, self-managing, intelligent storage nodes (OSDs) and lightweight monitors (Mons)

RADOS: Overview

RADOS Components

OSDs:

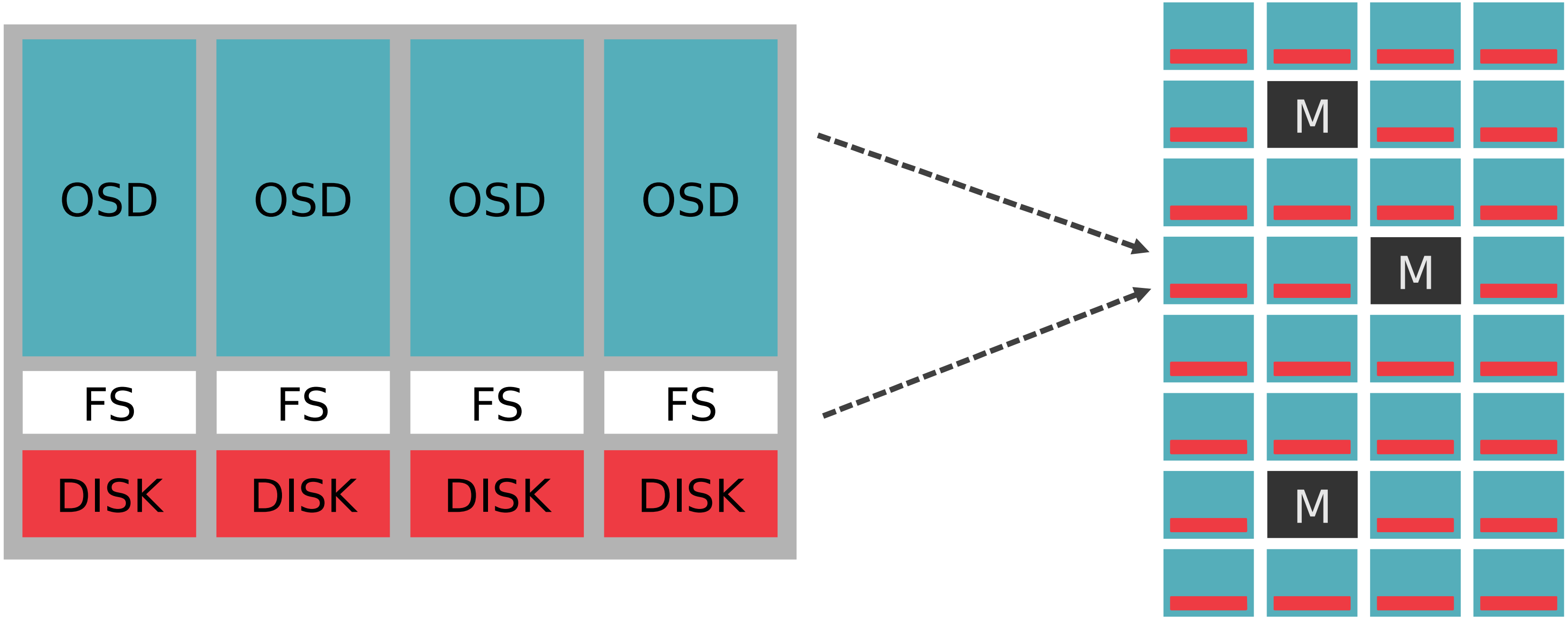
- 10s to 10000s in a cluster
- One per disk (or one per SSD, RAID group…)
- Serve stored objects to clients
- Intelligently peer for replication & recovery



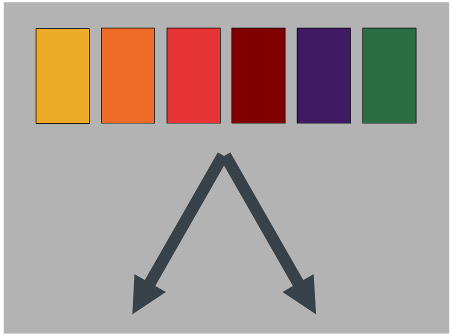
Monitors:

- Maintain cluster membership and state
- Provide consensus for distributed decision-making
- Small, odd number
- These do not serve stored objects to clients

Object Storage Daemons



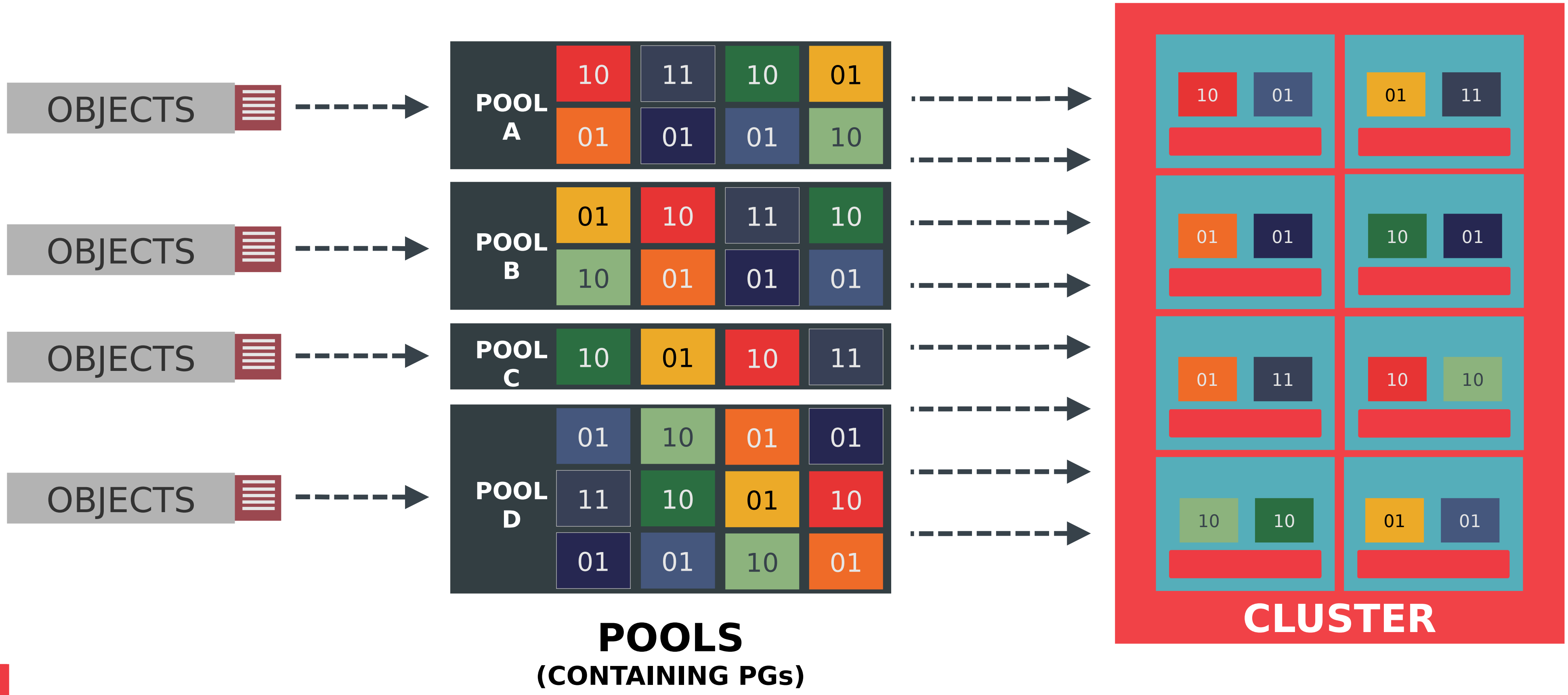
CRUSH: Dynamic Data Placement



CRUSH:

- Pseudo-random placement algorithm
  - Fast calculation, no lookup
  - Repeatable, deterministic
- Statistically uniform distribution
- Stable mapping
  - Limited data migration on change
- Rule-based configuration
  - Infrastructure topology aware
  - Adjustable replication
  - Weighting

DATA IS ORGANIZED INTO POOLS



librados: RADOS Access for Apps

LIBRADOS:

- Direct access to RADOS for applications
- C, C++, Python, PHP, Java, Erlang
- Direct access to storage nodes
- No HTTP overhead

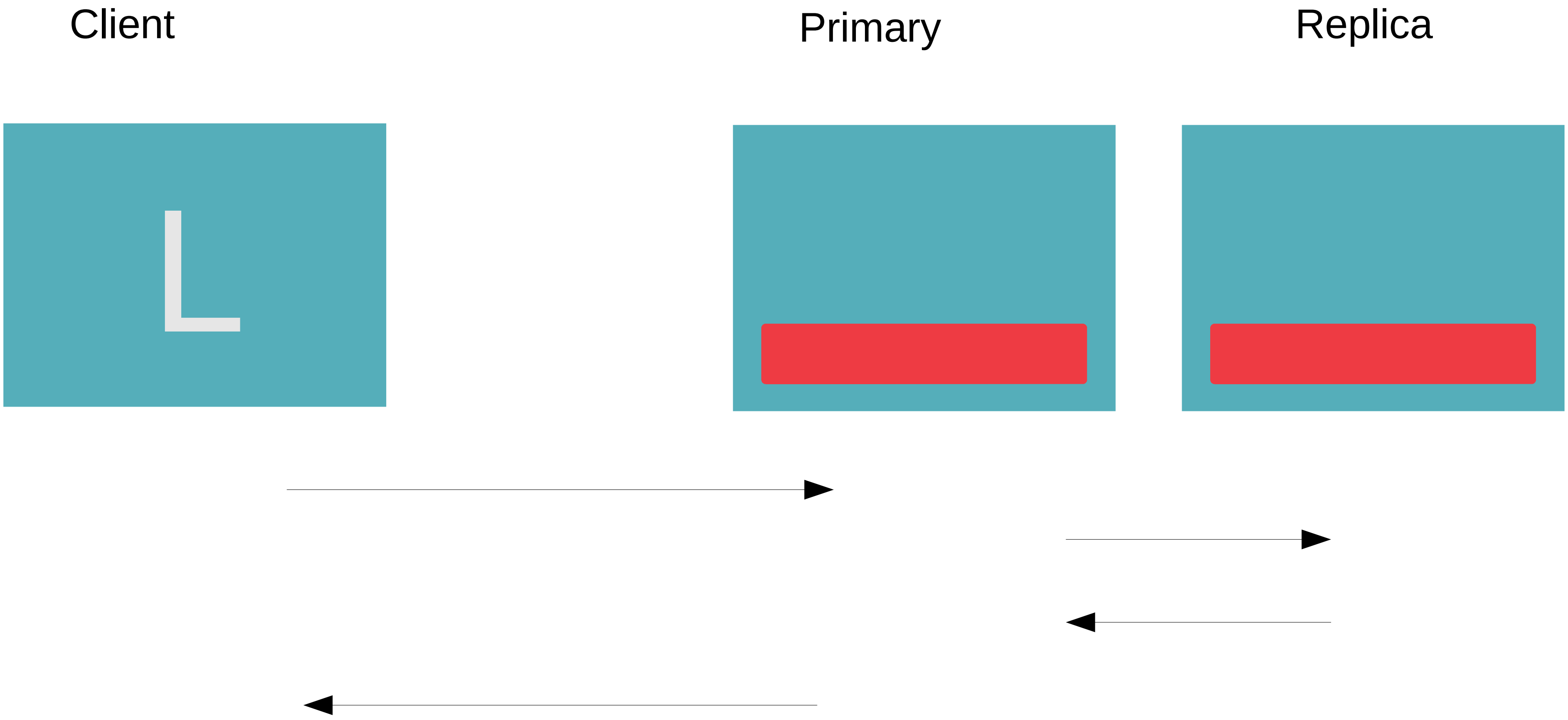
- Rich object API
- Bytes, attributes, key/value data
- Partial overwrite of existing data
- Single-object compound atomic operations
- RADOS classes (stored procedures)

RADOS: The Write Path (user)

```
aio_write(const object_t &oid, AioCompletionImpl *c,
          const bufferlist& bl, size_t len, uint64_t off);
c->wait_for_safe();
```

```
write(const std::string& oid, bufferlist& bl, size_t len, uint64_t off)
```

RADOS: The Write Path (Network)

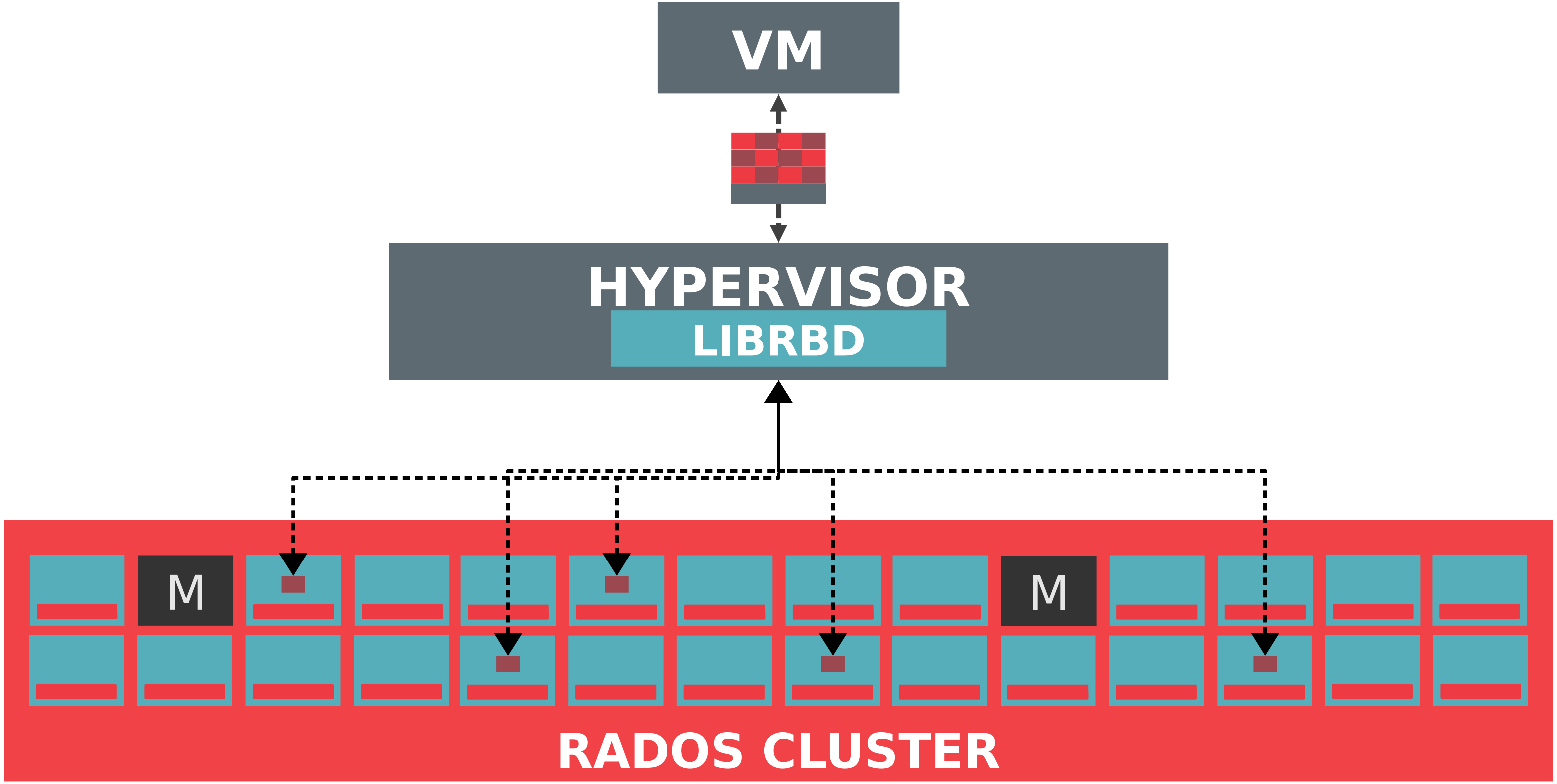


RADOS: The Write Path (OSD)

- Queue write for PG
- Lock PG
- Assign order to write op
- Package it for persistent storage
  - Find current object state, etc
- Send to replica op
- Send to local persistent storage
- Unlock PG
- Wait for commits from persistent storage and replicas
- Send commit back to client

RBD: Overview

STORING VIRTUAL DISKS



RBD STORES VIRTUAL DISKS

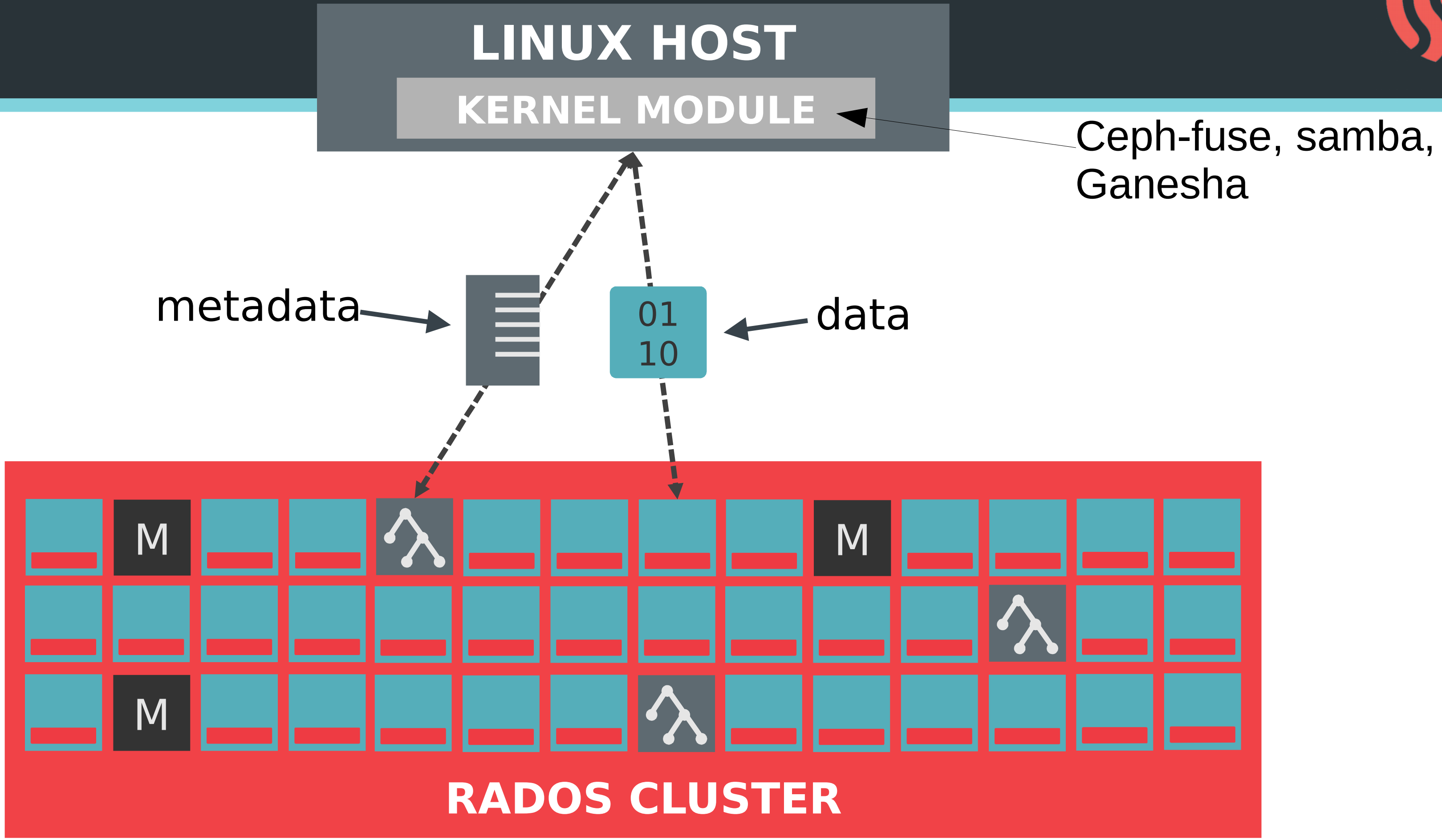
RADOS BLOCK DEVICE:

- Storage of disk images in RADOS
- Decouples VMs from host
- Images are striped across the cluster (pool)
- Snapshots
- Copy-on-write clones
- Support in:
    - Mainline Linux Kernel (2.6.39+)
    - Qemu/KVM, native Xen coming soon
    - OpenStack, CloudStack, Nebula, Proxmox

RBD: The Write Path

```
ssize_t Image::write(uint64_t ofs, size_t len, bufferlist& bl)



int Image::aio_write(uint64_t off, size_t len, bufferlist& bl,
                     RBD::AioCompletion *c)
```

CephFS: Overview


LINUX HOST
KERNEL MODULE
Ceph-fuse, samba,
Ganesha
metadata
01
10
data
M
M
M
RADOS CLUSTER

CephFS: The Write Path (User)

```
extern "C" int ceph_write(struct ceph_mount_info *cmount, int fd, const char *buf, int64_t size, int64_t offset)
```

CephFS: The Write Path (Network)

CephFS: The Write Path

- Request write capability from MDS if not already present
- Get “cap” from MDS
- Write new data to “ObjectCacher”
- (Inline or later when flushing)
  - Send write to OSD
  - Receive commit from OSD
- Return to caller

The Origin of Snapshots

```
[john@schist backups]$ touch history
[john@schist backups]$ cd .snap
[john@schist .snap]$ mkdir snap1
[john@schist .snap]$ cd ..
[john@schist backups]$ rm -f history
[john@schist backups]$ ls
[john@schist backups]$ ls .snap/snap1
history
# Deleted file still there in the snapshot!
```

Snapshot Design: Goals & Limits

- For CephFS
  - Arbitrary subtrees: lots of seemingly-unrelated objects snapshotting together
- Must be cheap to create
- We have external storage for any desired snapshot metadata

Snapshot Design: Outcome

- Snapshots are per-object
- Driven on object write
  - So snaps which logically apply to any object don’t touch it if it’s not written
- Very skinny data
  - per-object list of existing snaps
  - Global list of deleted snaps

RADOS: “Self-managed” snapshots

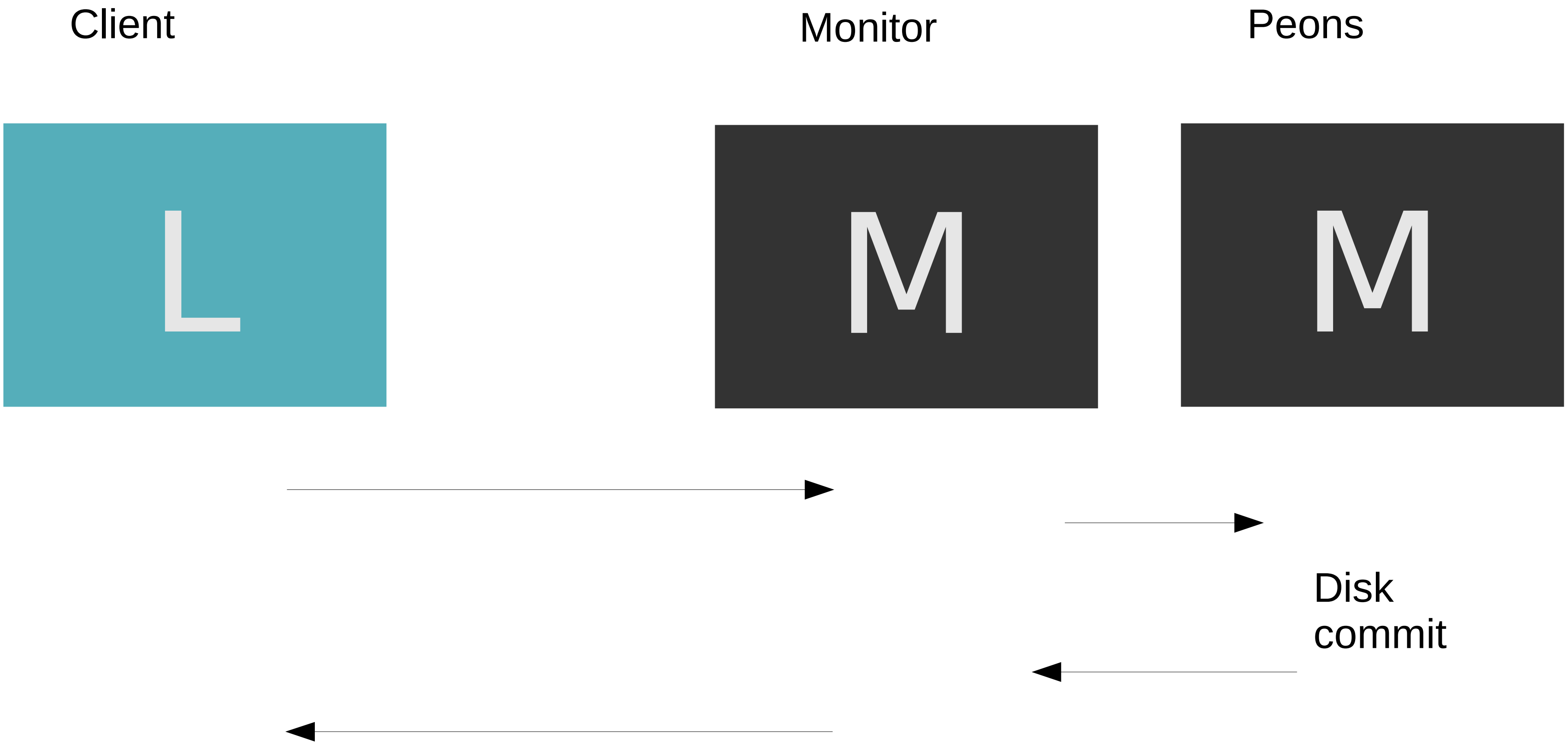
Librados snaps interface

```
int set_snap_write_context(snapid_t seq, vector<snapid_t>& snaps);

int selfmanaged_snap_create(uint64_t *snapid);

void aio_selfmanaged_snap_create(uint64_t *snapid, AioCompletionImpl
*c);

int selfmanaged_snap_remove(uint64_t snapid);

void aio_selfmanaged_snap_remove(uint64_t snapid, AioCompletionImpl
*c);

int selfmanaged_snap_rollback_object(const object_t& oid,
::SnapContext& snapc, uint64_t snapid);
```

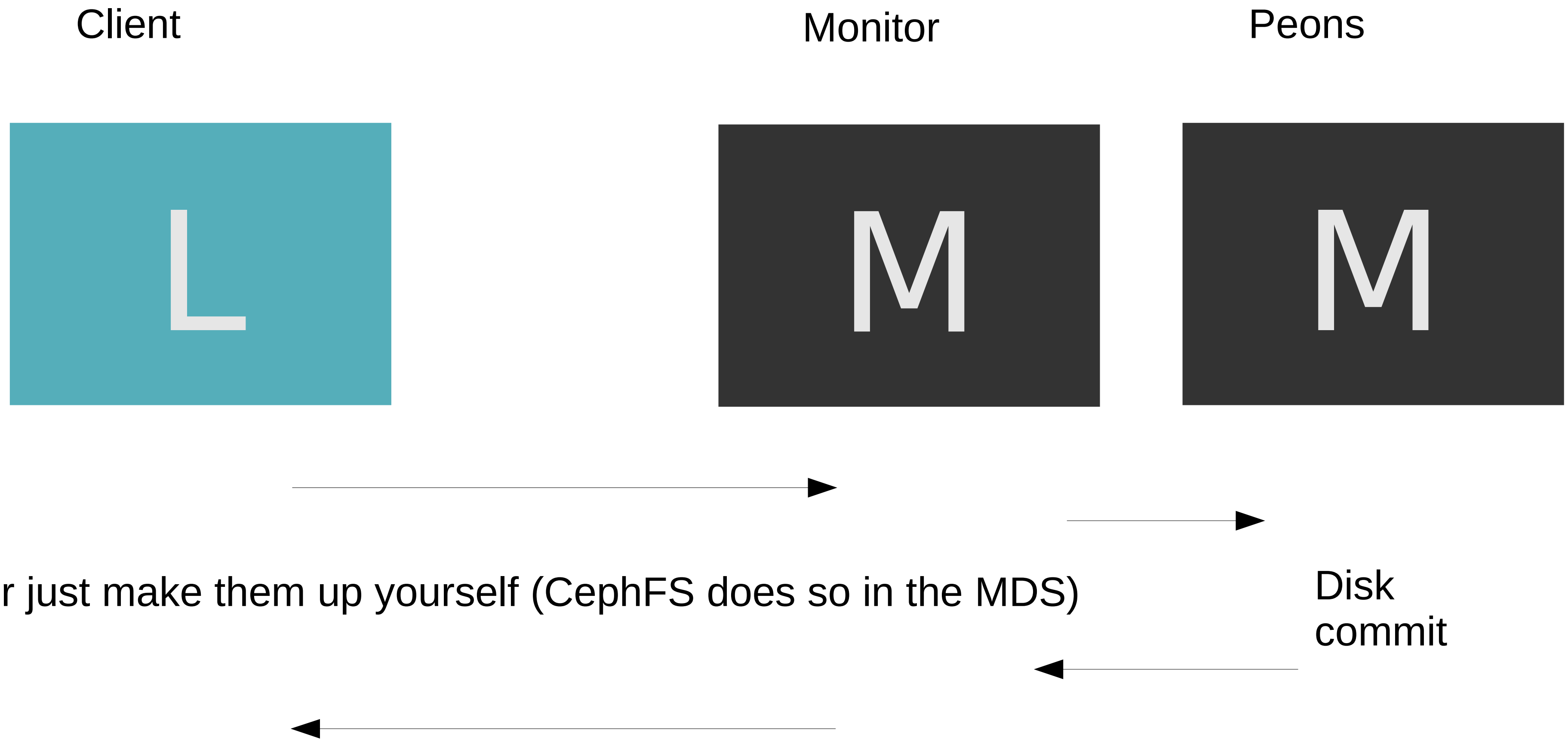
Allocating Self-managed Snapshots

“snapids” are allocated by incrementing the “snapid” and “snap_seq” members of the per-pool “pg_pool_t” OSDMap struct

Allocating Self-managed Snapshots



Allocating Self-managed Snapshots



...or just make them up yourself (CephFS does so in the MDS)

Librados snaps interface

int set_snap_write_context(snapid_t seq, vector<snapid_t>& snaps);

int selfmanaged_snap_create(uint64_t *snapid);

void aio_selfmanaged_snap_create(uint64_t *snapid, AioCompletionImpl *c);

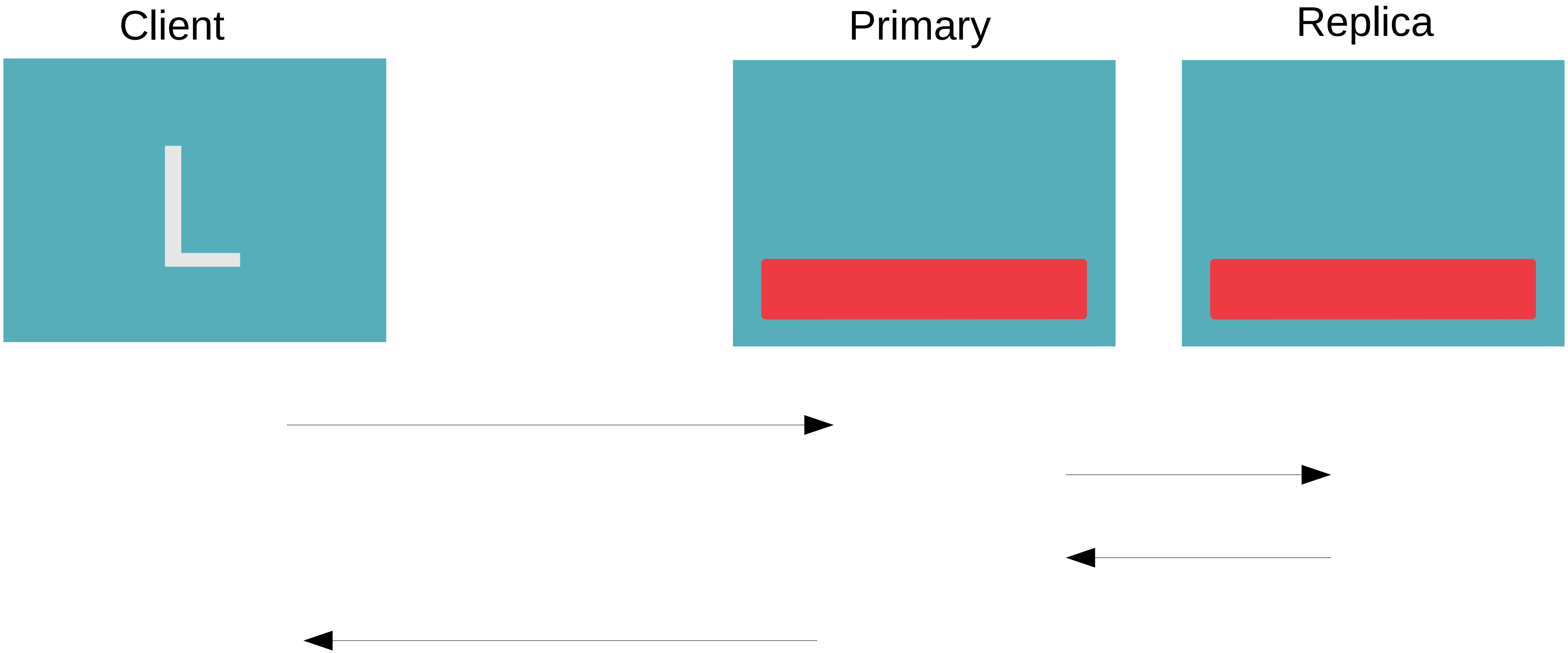
int selfmanaged_snap_remove(uint64_t snapid);

void aio_selfmanaged_snap_remove(uint64_t snapid, AioCompletionImpl *c);

int selfmanaged_snap_rollback_object(const object_t& oid, ::SnapContext& snapc, uint64_t snapid);

Writing With Snapshots

write(const std::string& oid, bufferlist& bl, size_t len, uint64_t off)



Snapshots: The OSD Path

- Queue write for PG
- Lock PG
- Assign order to write op
- Package it for persistent storage
  - Find current object state, etc
  - **make_writeable()**
- Send to replica op
- Send to local persistent storage
- Wait for commits from persistent storage and replicas
- Send commit back to client

Snapshots: The OSD Path

- The PrimaryLogPG::make_writeable() function
  - Is the “SnapContext” newer than the object already has on disk?
  - (Create a transaction to) clone the existing object
  - Update the stats and clone range overlap information
- PG::append_log() calls update_snap_map()
  - Updates the “SnapMapper”, which maintains LevelDB entries from:
    - snapid → object
    - And Object → snapid

Snapshots: OSD Data Structures

```
struct SnapSet {
  snapid_t seq;
  bool head_exists;
  vector<snapid_t> snaps;    // descending
  vector<snapid_t> clones;   // ascending
  map<snapid_t, interval_set<uint64_t> > clone_overlap;
  map<snapid_t, uint64_t> clone_size;
}
```

- This is attached to the “HEAD” object in an xattr

RADOS: Pool Snapshots :(

Pool Snaps: Desire

- Make snapshots “easy” for admins
- Leverage the existing per-object implementation
  - Overlay the correct SnapContext automatically on writes
  - Spread that SnapContext via the OSDMap

Librados pool snaps interface

```
int snap_list(vector<uint64_t> *snaps);
int snap_lookup(const char *name, uint64_t *snapid);
int snap_get_name(uint64_t snapid, std::string *s);
int snap_get_stamp(uint64_t snapid, time_t *t);
int snap_create(const char* snapname);
int snap_remove(const char* snapname);
int rollback(const object_t& oid, const char *snapName);
```

- Note how that’s still per-object!

Pool Snaps: Reality

- “Spread that SnapContext via the OSDMap”
  - It’s *not* a point-in-time snapshot
  - SnapContext spread virally as OSDMaps get pushed out
  - *No* guaranteed temporal order between two different RBD volumes in the pool – even when attached to the same VM :(
- Inflates the OSDMap size:

  per-pool map<snapid_t, pool_snap_info_t> snaps;

  struct pool_snap_info_t { snapid_t snapid; utime_t stamp; string name; }

- They are unlikely to solve a problem you want

Pool Snaps: Reality

- “Overlay the correct SnapContext automatically on writes”
  - No sensible way to merge that with a self-managed SnapContext
  - ...so we don’t: pick one or the other for a pool

All in all, pool snapshots are unlikely to usefully solve any problems.

RBD: Snapshot Structures

RBD Snapshots: Data Structures

```
struct cls_rbd_snap {
  snapid_t id;
  string name;
  uint64_t image_size;
  uint64_t features;
  uint8_t protection_status;
  cls_rbd_parent parent;
  uint64_t flags;
  utime_t timestamp;
  cls::rbd::SnapshotNamespaceOnDisk snapshot_namespace;
}
```

RBD Snapshots: Data Structures

- cls_rbd_snap for every snapshot
- Stored in “omap” (read: LevelDB) key-value space on the RBD volume’s header object
- RBD object class exposes get_snapcontext() function, called on mount
- RBD clients “watch” on the header, get “notify” when a new snap is created to update themselves

CephFS: Snapshot Structures

CephFS Snapshots: Goals & Limits

- For CephFS
  - Arbitrary subtrees: lots of seemingly-unrelated objects snapshotting together
- Must be cheap to create
- We have external storage for any desired snapshot metadata

CephFS Snapshots: Memory

- Directory “Cinodes” have “SnapRealms”
- Important elements:

snapid_t seq; // a version/seq # for changes to _this_ realm.

snapid_t created; // when this realm was created.

snapid_t last_created; // last snap created in _this_ realm.

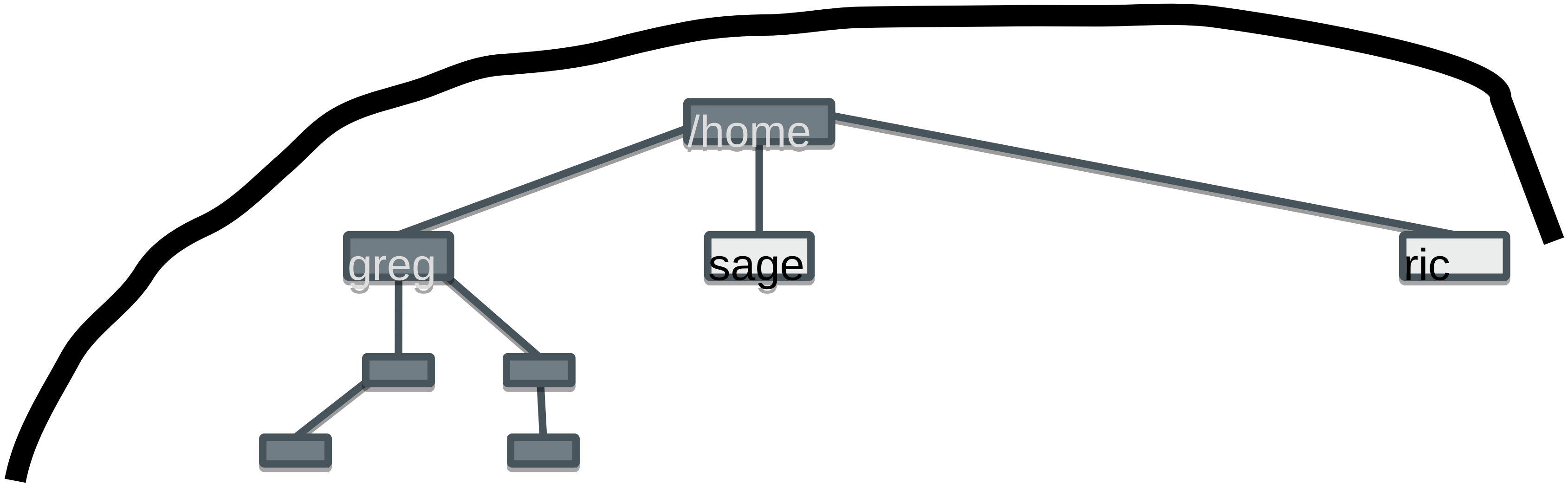
snapid_t last_destroyed; // seq for last removal

snapid_t current_parent_since;

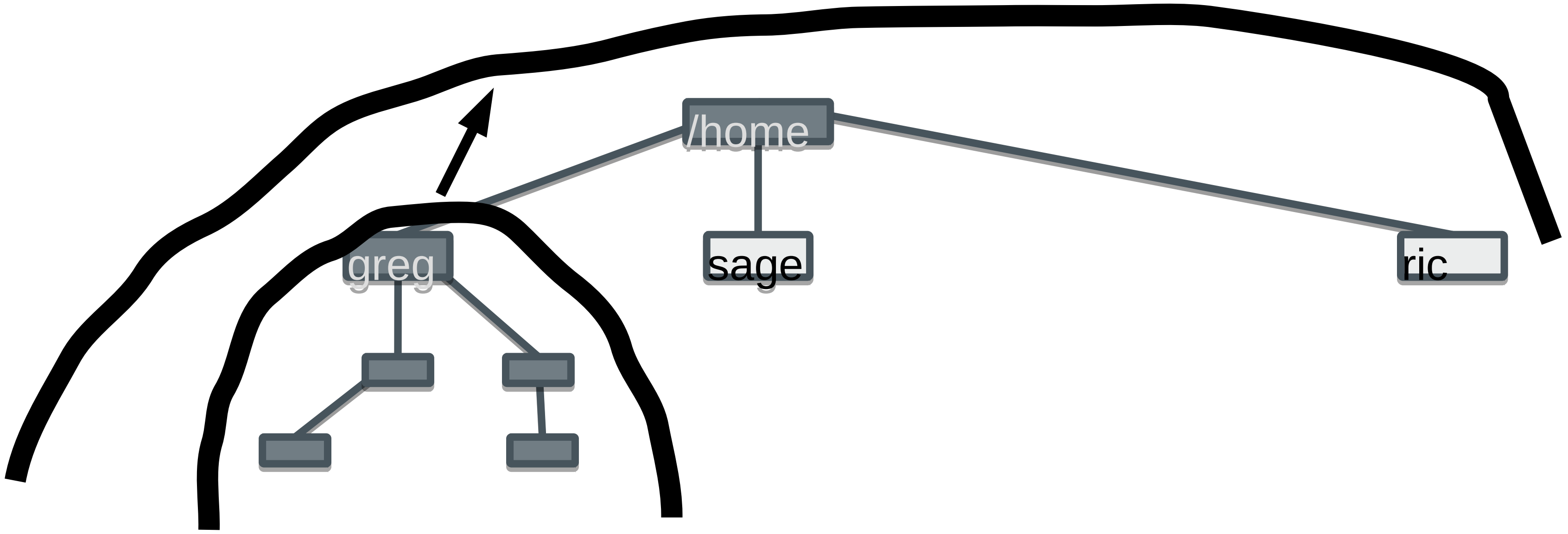
map<snapid_t, SnapInfo> snaps;

map<snapid_t, snaplink_t> past_parents; // key is "last" (or NOSNAP)

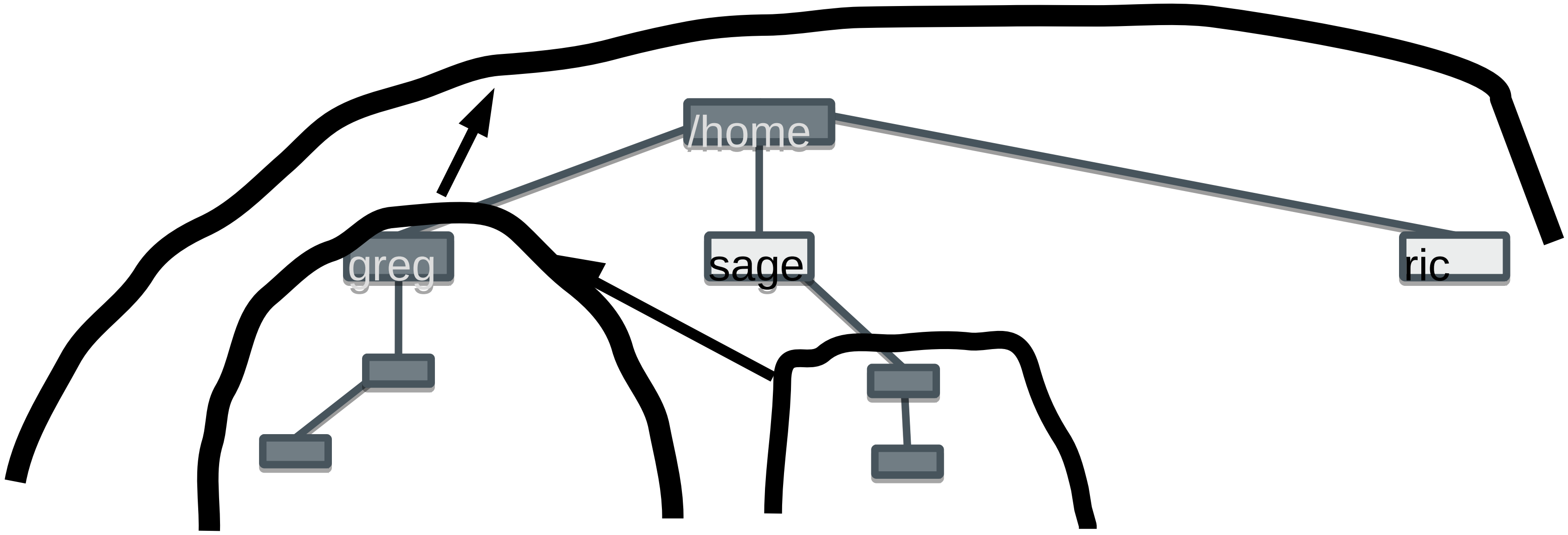
CephFS Snapshots: SnapRealms



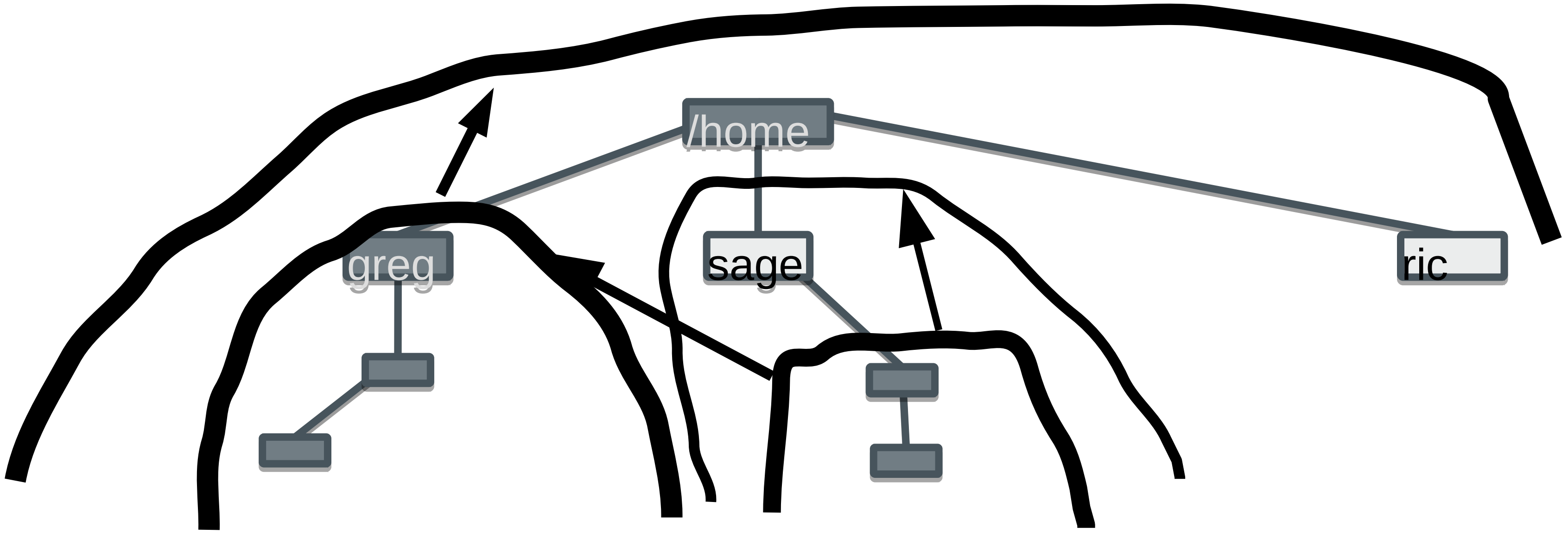
CephFS Snapshots: SnapRealms



CephFS Snapshots: SnapRealms



CephFS Snapshots: SnapRealms



CephFS Snapshots: SnapRealms

- Directory “Cinodes” have “SnapRealms”
- Important elements:

```
snapid_t seq;                      // a version/seq # for changes to _this_ realm.
snapid_t created;                  // when this realm was created.
snapid_t last_created;             // last snap created in _this_ realm.
snapid_t last_destroyed;           // seq for last removal
snapid_t current_parent_since;
map<snapid_t, SnapInfo> snaps;
map<snapid_t, snaplink_t> past_parents;  // key is "last" (or NOSNAP)
```

Construct the SnapContext!

CephFS Snapshots: Memory

- All “CInodes” have “old_inode_t” map representing its past states for snapshots

```
struct old_inode_t {
  snapid_t first;
  inode_t inode;
  std::map<string,bufferptr> xattrs;
}
```

CephFS Snapshots: Disk

- SnapRealms are encoded as part of inode
- Snapshotted metadata stored as old_inode_t map in memory/disk
- Snapshotted data stored in RADOS object self-managed snapshots

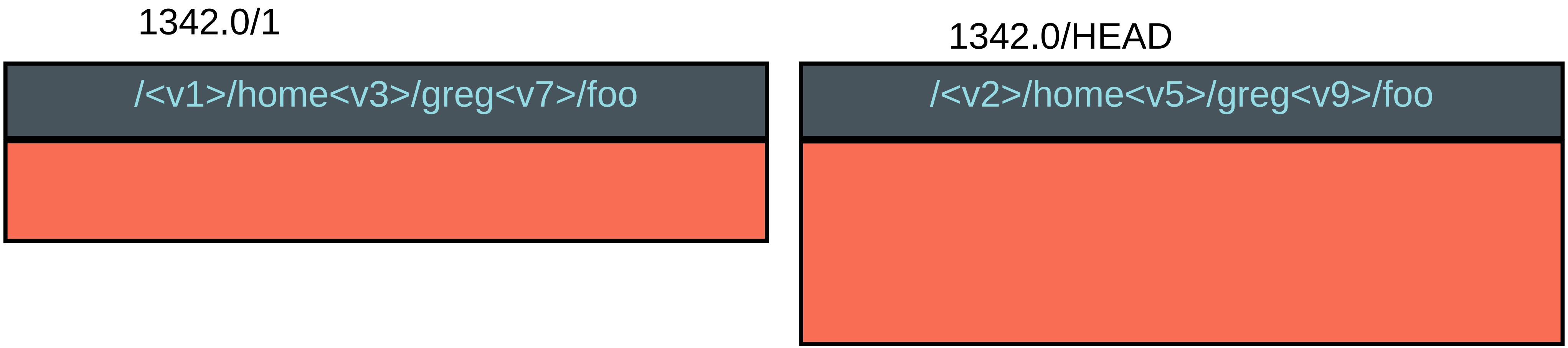
/<ino 0,v2>/home<ino 1,v5>/greg<ino 5,v9>/
Mydir[01], total size 7MB
foo -> ino 1342, 4 MB, [<1>,<3>,<10>]
bar -> ino 1001, 1024 KBytes
baz -> ino 1242, 2 MB

1342.0/HEAD

/<v2>/home<v5>/greg<v9>/foo

CephFS Snapshots

- Arbitrary sub-tree snapshots of the hierarchy
- Metadata stored as old_inode_t map in memory/disk
- Data stored in RADOS object snapshots



CephFS: Snapshot Pain

CephFS Pain: Opening past parents

- Directory "Cinodes" have "SnapRealms"
- Important elements:

snapid_t seq; // a version/seq # for changes to _this_ realm.

snapid_t created; // when this realm was created.

snapid_t last_created; // last snap created in _this_ realm.

snapid_t last_destroyed; // seq for last removal

snapid_t current_parent_since;

map<snapid_t, SnapInfo> snaps;

map<snapid_t, snaplink_t> past_parents;

CephFS Pain: Opening past parents

- To construct the SnapContext for a write, we need the all the SnapRealms it has **ever** participated in
  - Because it could have been logically snapshotted in an old location but not written to since, and a new write must reflect that old location's snapid
- So we must open all the directories the file has been a member of!
  - With a single MDS, this isn't too hard
  - With multi-MDS, this can be very difficult in some scenarios
    - We may not know who is "authoritative" for a directory under all failure and recovery scenarios
    - If there's been a disaster metadata may be inaccessible, but we don't have mechanisms for holding operations and retrying when "unrelated" metadata is inaccessible

CephFS Pain: Opening past parents

- Directory “Cinodes” have “SnapRealms”
- Important elements:

```
snapid_t seq;                    // a version/seq # for changes to _this_ realm.
snapid_t created;                // when this realm was created.
snapid_t last_created;           // last snap created in _this_ realm.
snapid_t last_destroyed;         // seq for last removal
snapid_t current_parent_since;
map<snapid_t, SnapInfo> snaps;
map<snapid_t, snaplink_t> past_parents;
```

Why not store snaps in all descendants
Instead of maintaining ancestor links?

CephFS Pain: Eliminating past parents

- The MDS opens an inode for any operation performed on it
  - This includes its SnapRealm
- So we can merge snapid lists down whenever we open an inode that has a new SnapRealm
- So if we rename a directory/file into a new location; its SnapRealm already contains all the right snapids and then we don't need a link to the past!

- I got this almost completely finished
  - Reduced code line count
  - Much simpler snapshot tracking code
  - But….

CephFS Pain: Hard links

- Hard links and snapshots **do not** interact :(
- They should!
- That means we need to merge SnapRealms from all the linked parents of an inode
  - And this is the exact same problem we have with past_parents
  - Since we need to open “remote” inodes correctly, avoiding it in the common case doesn’t help us
- So, back to debugging and more edge cases

RADOS: Deleting Snapshots

Librados snaps interface

int set_snap_write_context(snapid_t seq, vector<snapid_t>& snaps);

int selfmanaged_snap_create(uint64_t *snapid);

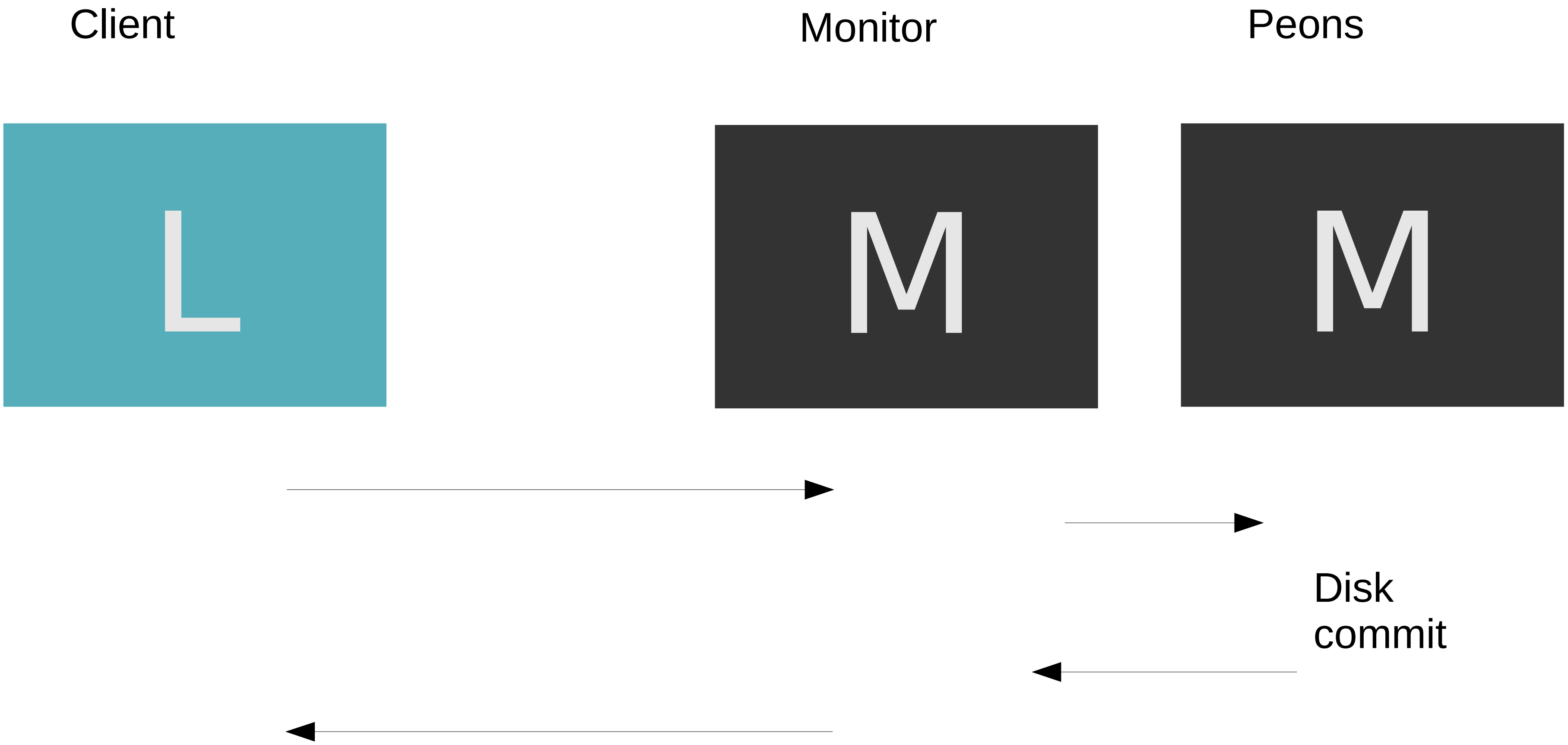
void aio_selfmanaged_snap_create(uint64_t *snapid, AioCompletionImpl *c);

int selfmanaged_snap_remove(uint64_t snapid);

void aio_selfmanaged_snap_remove(uint64_t snapid, AioCompletionImpl *c);

int selfmanaged_snap_rollback_object(const object_t& oid, ::SnapContext& snapc, uint64_t snapid);

“Deleting” Snapshots (Client)



Deleting Snapshots (Monitor)

- Generate new OSDMap updating pg_pool_t

```
interval_set<snapid_t> removed_snaps;
```

- This is really space-efficient if you consistently delete your oldest snapshots!
  - Rather less so if you keep every other one forever
    - ...and this looks sort of like some sensible RBD snapshot strategies (daily for a week, weekly for a month, monthly for a year)

Deleting Snapshots (OSD)

- OSD advances its OSDMap
- Asynchronously:
  - List objects with that snapshot via “SnapMapper”
    - int get_next_objects_to_trim(snapid_t snap, unsigned max, vector<hobject_t> *out);
  - For each object:
    - “unlink” object clone for that snapshot – 1 coalescable IO
      - Sometimes clones belong to multiple snaps so we might not delete right away
    - Update object HEAD’s “SnapSet” xattr – 1+ unique IO
    - Remove SnapMapper’s LevelDB entries for that object/snap pair – 1 coalescable IO
    - Write down “PGLog” entry recording clone removal – 1 coalescable IO
  - Note that if you trim a bunch of snaps, you do this for each one – no coalescing it down to one pass on each object :(

Deleting Snapshots (OSD)

- So that's at least 1 IO per object in a snap
  - potentially a lot more if we needed to fetch KV data off disk, didn't have directories cached, etc
  - This will be a *lot* better in BlueStore! It's just coalescable metadata ops
- Ouch!
- Even worse: throttling is hard
  - Why is a whole talk on its own
  - It's very difficult to not overwhelm clusters if you do a lot of trimming at once

RADOS: Alternate Approaches

Past: Deleting Snapshots

- Maintain a per snapid directory with hard links!
  - Every clone is linked into (all) its snapid directory(s)
  - Just list the directory to identify them, then
    - update the object’s SnapSet
    - Unlink from all relevant directories

- Turns out this destroys locality, in addition to being icky code

Present: Why per-object?

- For instance, LVM snapshots?
- We don't want to snapshot everything on an OSD at once
  - No implicit "consistency groups" across RBD volumes, for instance
- So we ultimately need a snap→object mapping, since each snap touches so few objects

Future: Available enhancements

- Update internal interfaces for more coalescing
  - There’s no architectural reason we need to scan each object per-snapshot
  - Instead, maintain iterators for each snapshot we are still purging and advance them through the keyspace in step so we can do all snapshots of a particular object in one go

- Change deleted snapshots representation so it doesn’t inflate ODSMaps
  - “deleting_snapshots” instead, which can be trimmed once all OSDs report they’ve been removed
  - Store the full list of deleted snapshots in config-keys or similar, handle it with ceph-mgr

Future: Available enhancements

- BlueStore: It makes everything better
  - Stop having to map our semantics onto a filesystem
  - Snapshot deletes still require the snapid→object mapping, but the actual delete is a few key updates rolled into RocksDB – easy to coalesce
- Better throttling for users
  - Short-term: hacks to enable sleeps so we don’t overwhelm the local FS
  - Long-term: proper cost estimates for BlueStore that we can budget correctly (not really possible in Fses since they don’t expose number of IOs needed to flush current dirty state)

THANK YOU!

Greg Farnum

Principal Engineer, Ceph

gfarnum@redhat.com

@gregsfortytwo

ceph