Snapshotting Scale-out Storage

Pitfalls and Solutions

Alex Aizman, Nexenta

SNIA Legal Notice

SNIA Global Education

- The material contained in this tutorial is copyrighted by the SNIA unless otherwise noted.
- Member companies and individual members may use this material in presentations and literature under the following conditions:
  - Any slide or slides used must be reproduced in their entirety without modification
  - The SNIA must be acknowledged as the source of any material used in the body of any document containing material from these presentations.
- This presentation is a project of the SNIA Education Committee.
- Neither the author nor the presenter is an attorney and nothing in this presentation is intended to be, or should be construed as legal advice or an opinion of counsel. If you need legal advice or a legal opinion please contact your attorney.
- The information presented herein represents the author's personal opinion and current understanding of the relevant issues involved. The author, the presenter, and the SNIA do not assume any responsibility or liability for damages arising out of any reliance on or use of this information.

  NO WARRANTIES, EXPRESS OR IMPLIED. USE AT YOUR OWN RISK.

Abstract

SNIA Global Education

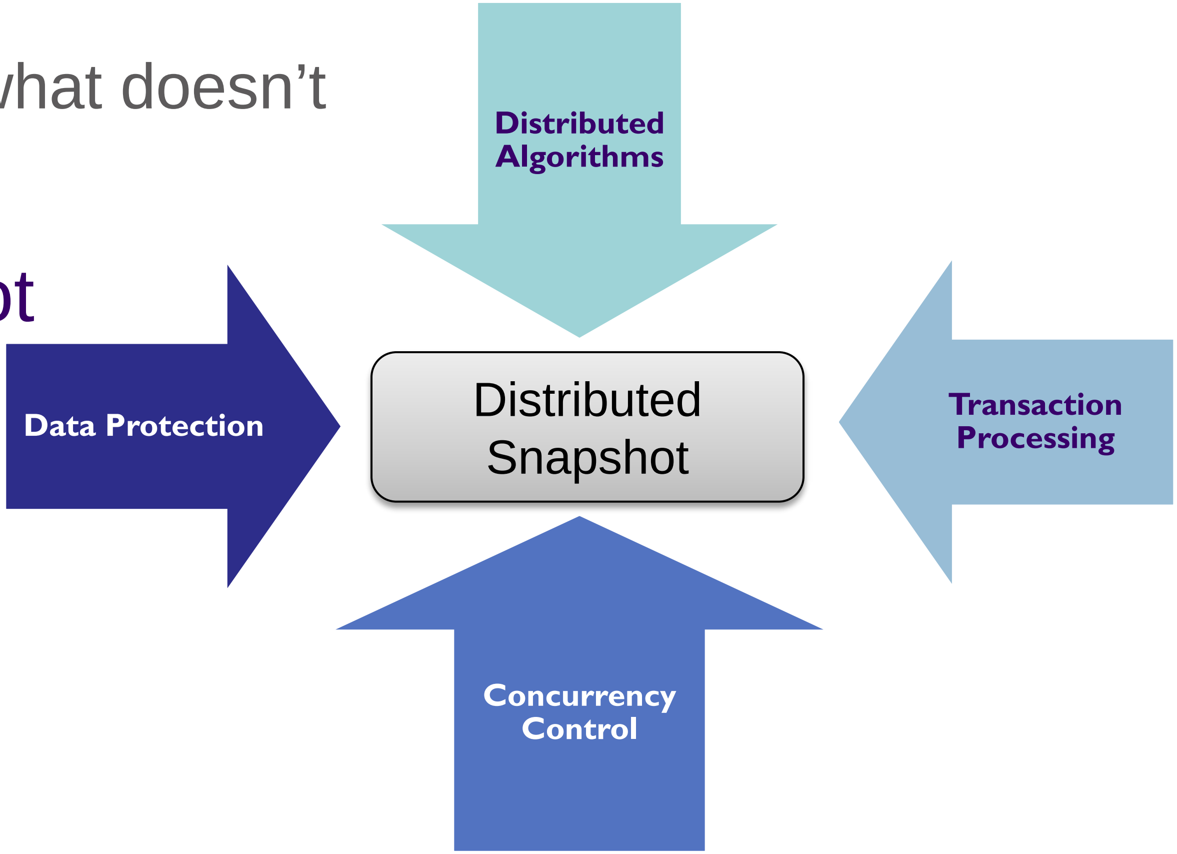
- **Distributed Snapshot: the challenge**
  - One of the toughest technical challenges for the implementors. The best existing distributed storage systems provide partial support or none whatsoever.
- **Distributed Snapshot: the operation**
  - Even for the distributed systems with relaxed consistency models, snapshotting must be a transaction that meets the familiar ACID requirements. Further, the operation must execute concurrently with updates and result in a persistent, immutable, consistent snapshot that can be read and cloned.
  - This presentation examines the topic from a variety of perspectives, and illustrates one possible way to snapshot eventually consistent distributed storage systems.

The seminal 1985 paper by Chandy and Lamport: *Distributed Snapshots: Determining Global States of Distributed Systems*

- "The <snip> algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds, a scene so vast that it cannot be captured by a single photograph.
- The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene.
- The snapshots cannot all be taken at precisely the same instant <snip>. Furthermore, the photographers should not disturb the process that is being photographed…"

In this presentation

- Definitions
- Common patterns
  - What works and what doesn't
- Case Studies
- Eventual Snapshot



Definitions

Distributed

Snapshot

must be **Consistent**

Distributed Systems: the great diversity

SNIA Global Education

- Namespace federated vs striped (sharded)
- Block, Object, Key/Value, SQL, Partial POSIX, Full POSIX
- Specialized (e.g., HDFS) vs general purpose
- Crash-consistent vs not crash-consistent
- Single writer (SWMR) vs MWMR
- Single MDS (metadata server) vs federated metadata vs fully distributed metadata
- Single Initiator vs multiple storage initiators
- Eventually consistent vs stronger consistency levels
- Any combination of the above, and more

For example

- Ceph/RADOS is a
  - General purpose object storage system that is
    - **Fully distributed**
    - **Fault-tolerant**
    - **With distributed metadata**
    - **And concurrent access via multiple storage initiators (librados clients)**
- Distributed snapshotting will be, of course, as diverse as the underlying systems
  - Moreover, highly dependent on specific implementations/tradeoffs

Prerequisites

- [Dynamo: Amazon's Highly Available Key-value Store](#) 2007

Check out SNIA Tutorials:

- The Evolution of File Systems 2012
- Massively Scalable File Storage 2015

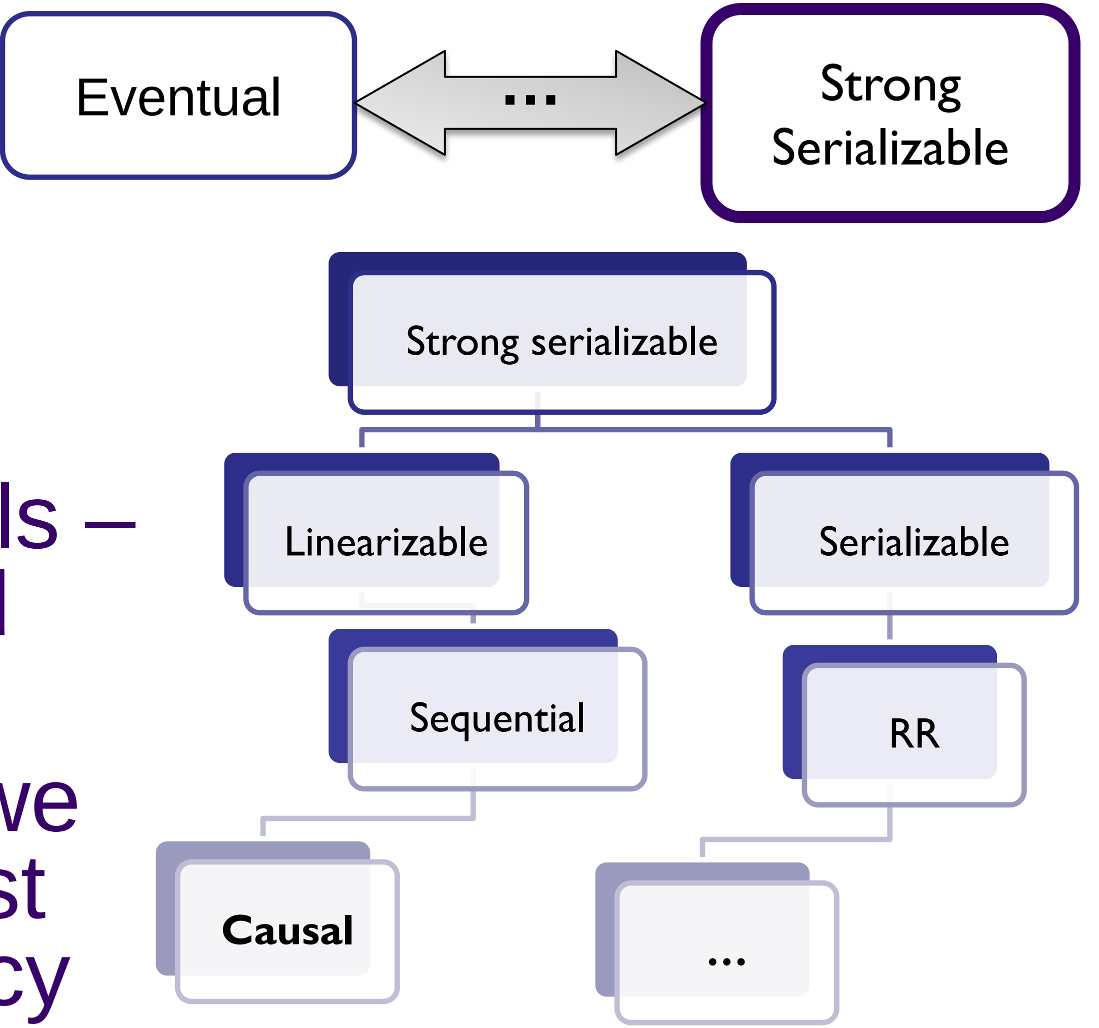
- An Overview of On-Premise File and Object Storage Access Protocols 2016

What is a "snapshot"

- **Snapshot is a read-only consistent dataset referencing a certain subset of persistent data at a given time**
  - Snapshotted data may not necessarily be persistent at the time of snapshotting
- **Local and distributed snapshots vary**
  - In terms of their supported scope, capabilities and internal consistency
- **Scope may be global or tenant, part of hierarchical namespace, a bucket, an object, etc.**
- **Capabilities in turn include:**
  - Copy-on-write (redirect-on-write)
  - Ability to incrementally replicate snapshot "delta"
  - Mount, rollback, clone, rebalance, and more
- **Generally, Data Protection today requires: just-in-time snapshotting and incremental replication**

What is “consistency”

- Snapshot ***is a*** dataset
- Dataset must support a certain *consistency level*
- Consistency levels – the spectrum and the hierarchy:
- For a snapshot, we often want at least causal consistency

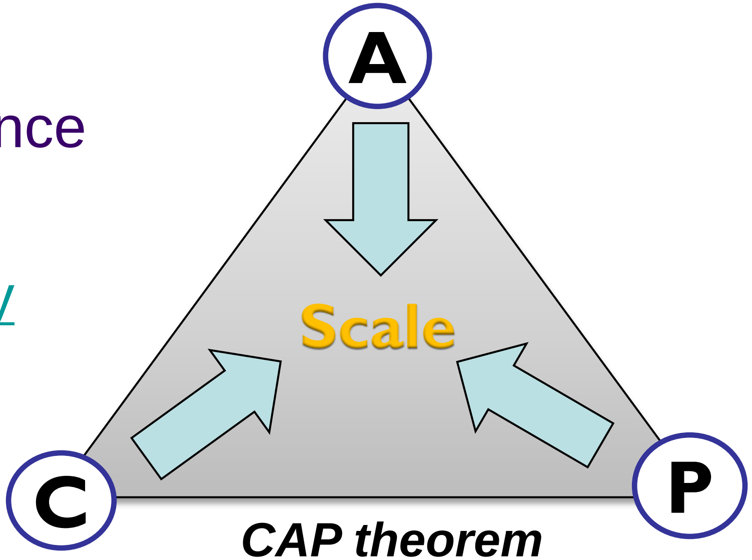


Snapshot Consistency vs Real-Life Scenarios

- **Just-in-time snapshot can be requested at any time, and in parallel with (for instance):**
  - Write operations that have been write-logged but not yet acknowledged back to user
  - Write operations – acknowledged but not yet majority-ACKed
  - Write operations – majority-ACKed but the associated metadata (updating) is still in progress
  - Asynchronous writes – persisted but the file is still open
  - Updates associated with file close and fsync operations – in progress but not finished yet
  - Destroy (or rename) operations – started but not finished yet
  - And more
- **"There is only one hard thing in Computer Science"**

Snapshot Consistency vs CAP theorem

- CAP theorem: distributed system vs **C**onsistency, **A**vailability, **P**artition tolerance
  - Brewer's **conjecture**, followed by:
  - Gilbert and Lynch proof for a narrowly scoped (*linearizable*) consistency
- The broadly accepted fact:
  - A distributed system **cannot** support simultaneously all 3 (three): **C**, **A**, and **P**



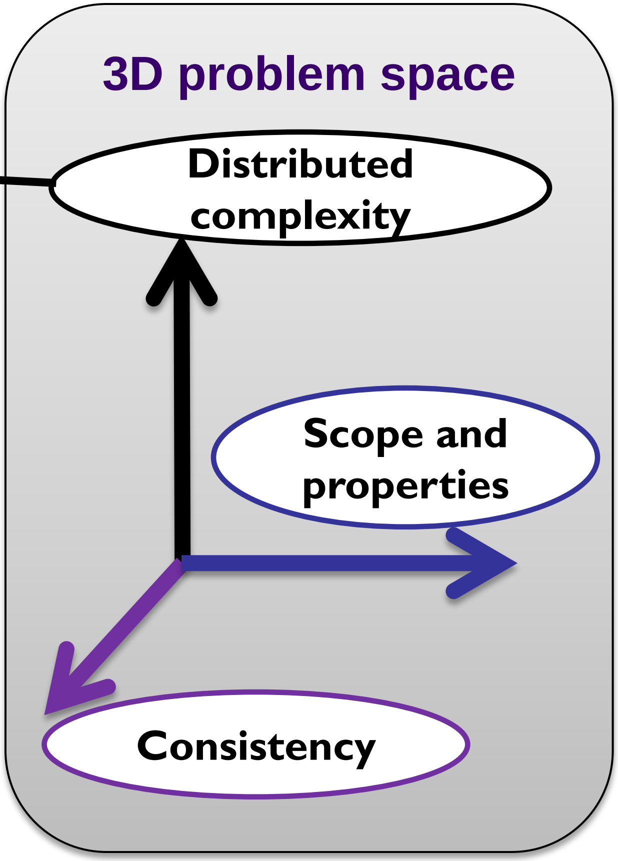
CAP theorem

- How do we snapshot a temporarily partitioned cluster?
  - Given that higher scales lead to increased chances of partitioning
- How do we produce a consistent snapshot in the AP system?

Despite all the diversity, here’s what is generally agreed upon today (1 of 2)

SNIA Global Education

In a distributed system, all components “conspire” to delay, drop and reorder messages



- **Must be consistent and isolated**
  - Can read snapshot in parallel with I/O (*snapshot isolation*)
  - Minimally expected consistency is either causal or the one defined for the (live) dataset that is being snapshotted
- **Must be CoW and immutable**

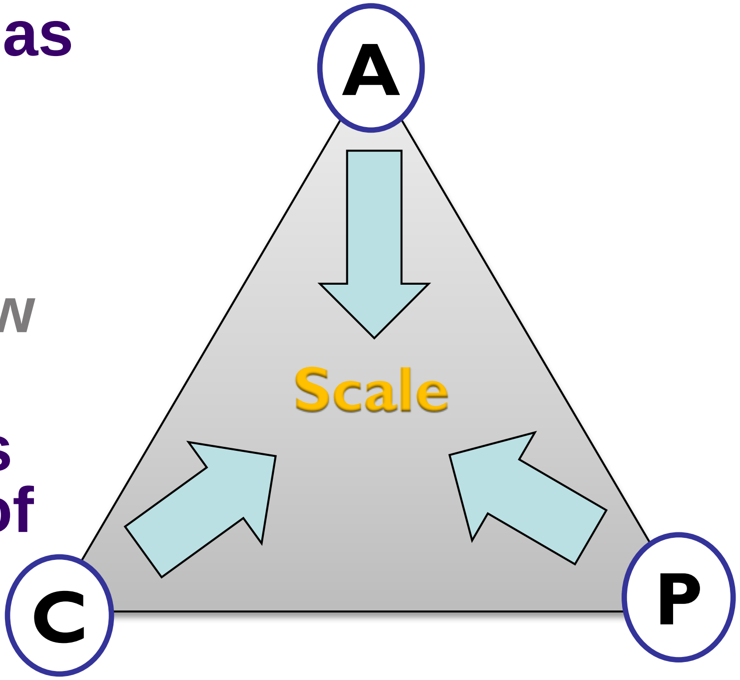
General Consensus (2 of 2)

- **Writeable snapshots are usually called *clones***
- **Distributed snapshot typically consists of local snapshots**
  - Assuming that storage metadata is distributed (central-metadata systems can simply snapshot or clone respective central metadata, case in point: HDFS)
- **Snapshot introduces additional metadata references**
  - Attempt to destroy snapshotted content fails unless the parent (snapshot and/or clone) is destroyed/expired first

Everything else is implementation dependent:
scope and content, capabilities and consistency of the snapshot…

CAP vs Snapshotting Transaction

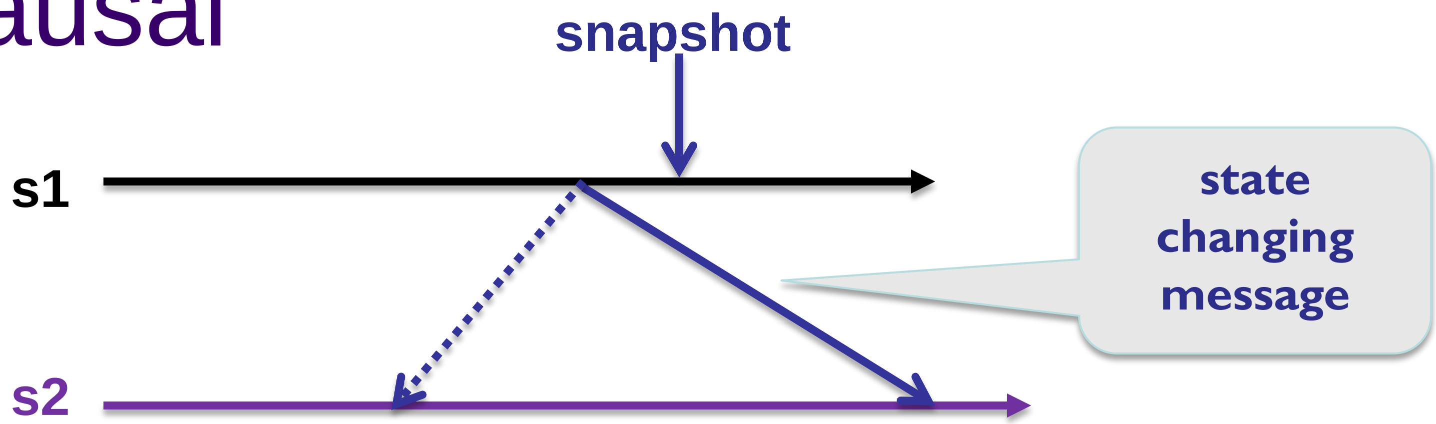
- **CAP theorem: all distributed systems can roughly be classified as CA, CP, and AP**
  - Example of an AP system: eventually consistent object
  - Chances of partitioning will grow with scale
- **Requirement of Availability pushes further down the achievable level of Consistency**
  - Highly Available Transactions: Virtues and Limitations
- **Distributed snapshotting must be transactional**



Common Patterns

The 3 Common Types (1 of 2)

- Point-in-time
- Loosely synchronized
- Causal



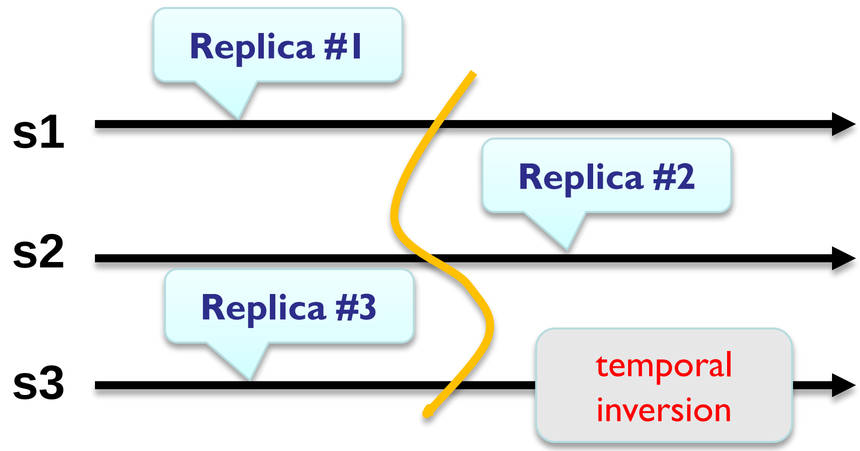
The 3 Common Types (2 of 2)

SNIA Global Education

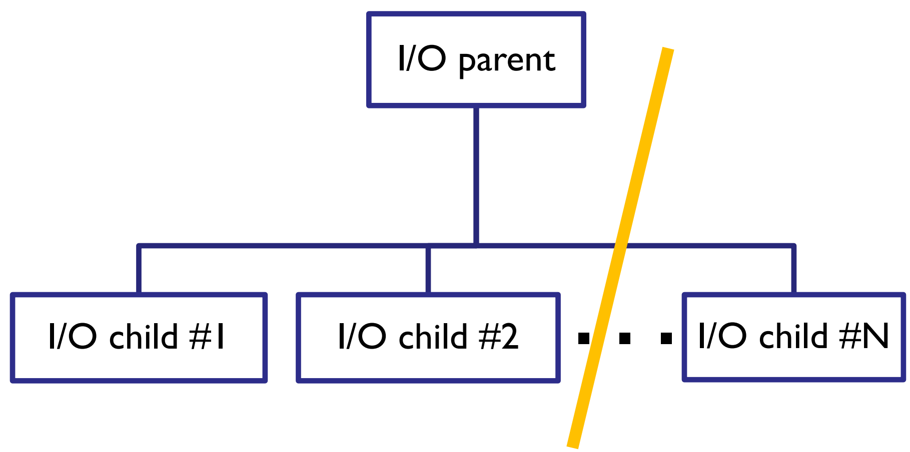
- **Point-in-time**
  - Requires global system time via perfectly synchronized clocks (which is impossible)
  - Or, a single serialization context, not necessarily global or permanent/static (which will hurt performance)
- **Loosely synchronized**
  - Assumes upper bound on clock drift
  - Ti and Tj are considered equal if ABS(Ti – Tj) <= drift
  - Metadata updates delayed until (previous-update + drift)
- **Causal**
  - Snapshot(server-2) must *reflect* Snapshot(server-1)
  - Requires more inter-server messages, more synchronization

The Cut, or when two anomalies meet

- Anomaly type 1: clock drift
  - Solution: global logical time
    - **Time T does not increment as long as there are state-changing events that must be handled by T**
  - Solution: vector clock and variations



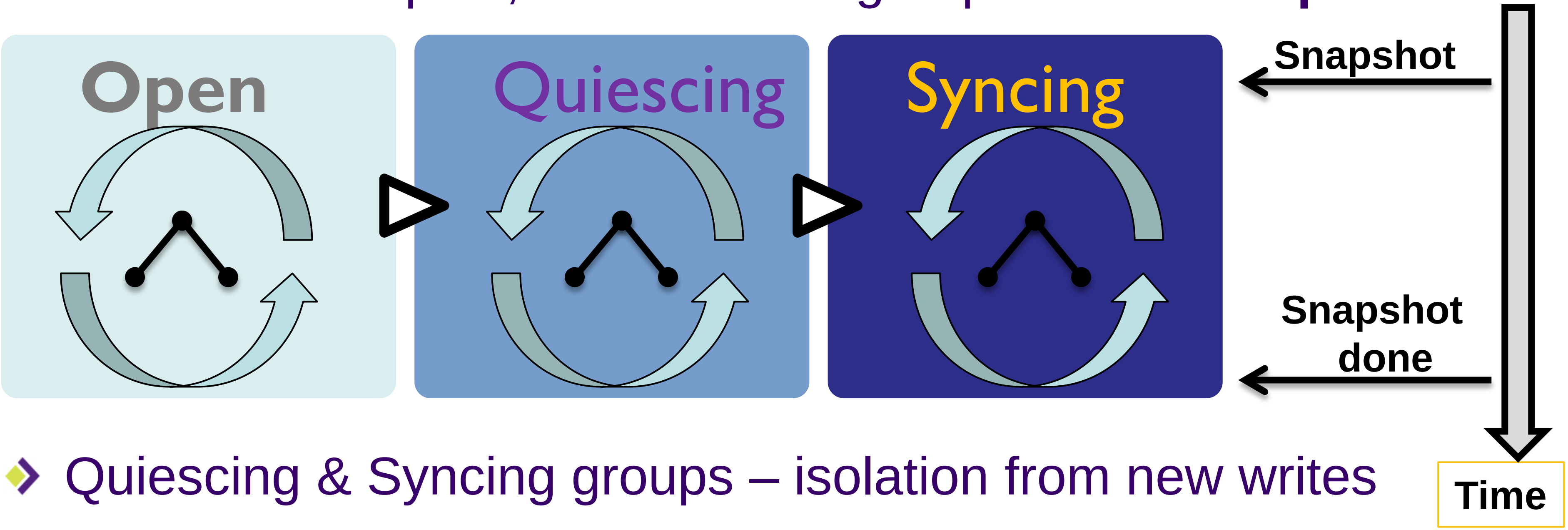
- Anomaly type 2: caused by I/O propagation through pipeline
  - I/Os can multiply and fork, split and join, mutate and even self-cancel
  - Common pattern: I/O parent forking children to execute in parallel



Case Studies

ZFS: transaction group pipeline

- Local FS (but sets user expectations as far as snapshots)
- For each ZFS pool, 3 transaction groups execute **in parallel**:



- Quiescing & Syncing groups – isolation from new writes
- Snapshot can only be created at the end of the Syncing
  - Resulting in a new uberblock (“superblock”) referencing a new consistent set of metadata branches

Ceph snapshots

- http://ceph.com
- Unified distributed object storage system with object (RADOS), block (via RBD), and file (via CephFS)
- Distributed snapshots = work in progress
  1) RADOS http://ceph.com/dev-notes/rados-snapshots/
  2) RBD http://docs.ceph.com/docs/hammer/rbd/rbd-snapshot/
  3) CephFS (jewel) http://docs.ceph.com/docs/jewel/cephfs/early-adopters/
- RBD: recommends to stop I/O before taking a snapshot
- CephFS (jewel): recommends to use a single active MDS and not to use snapshots
- Ceph RADOS introduces "*Snapshot Context*" (librados)
  - Serving as a reference (root) of the snapshotted metadata

HDFS snapshots

- https://hadoop.apache.org/docs/stable
- Distributed storage used by **Hadoop** applications
- Cluster consists of a NameNode (MD) and multiple DataNodes
- Snapshot creation: entry under **.snapshot/** of the snapshotted dir:
  - **<snapshotted-directory>/.snapshot/<snapshot-name>**
- CoW (redirect-on-write):
  - "Modifications are recorded in reverse chronological order"
  - "Snapshot is computed by subtracting the modifications from the current data"
- Notes:
  - Keeping "backward" diffs of the metadata is a fairly unique approach
  - Although easy to implement (and serialize) given a single NameNode
  - Retrieving an old snapshot is going to be (more) time-consuming over time
  - The same applies to destroying older snapshots

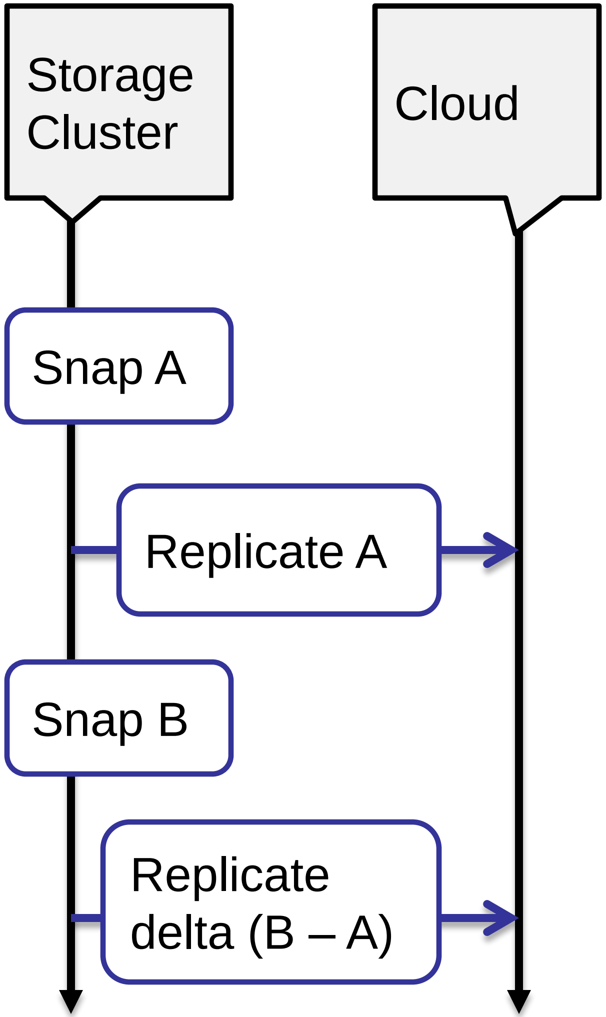
XtreemFS snapshots

- http://www.xtreemfs.org
- Fault-tolerant, distributed, POSIX compliant, relies on local FS
- Excellent white paper on distributed snapshotting:
  - Loosely Time-Synchronized Snapshots in Object-Based File Systems
- XtreemFS metadata: BabuDB (LSM-tree-like local database)
  - https://github.com/xtreemfs/babudb
- Distributed loosely-synchronized snapshot:
  - Consists of local snapshots
  - Serialization via (centralized & replicated) BabuDB metadata transaction
  - Configurable limits on the “fuzziness” of local timestamps
  - Lacking causal consistency
  - New file content (data) snapshot on every close, and never deleted

Eventually Consistent Scale-Out Object Storage

Object Storage vs Snapshots

- **Unlike Ceph RADOS, Amazon S3 (API) and OpenStack Swift do not support self-snapshotting**
  - Both offer block storage snapshots, however (EBS and Cinder volume respectively, the latter – via vendor driver)
- **Object storage is often used to store someone else's snapshots (often, in the Cloud)**
  - DR use case, (incremental) backup and restore, advanced capabilities
- **Cluster-wide snapshots of the object storage itself – not yet a commonly requested feature**
  - Technically, can be done, and will scale (next)



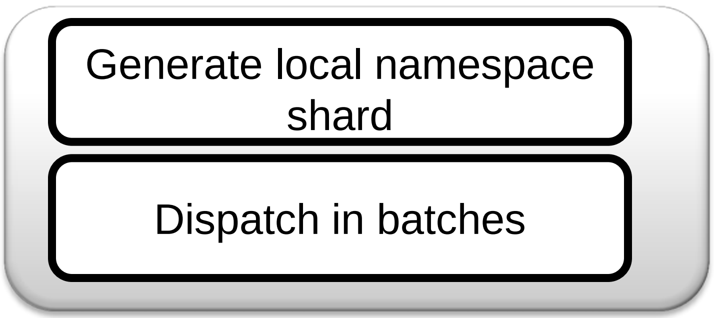
Snapshotting Distributed Object Store: Requirements

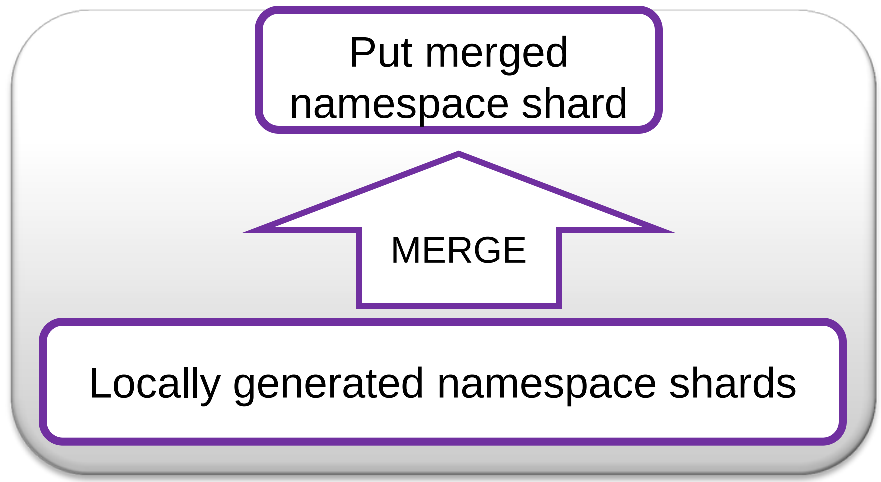
- Must be point-in-time
- Must be a fully distributed transaction, with each server contributing a fair share
- Must be consistent
- Must tolerate network partitioning
- May be **unavailable** for a while
  - In the photographic analogy, it takes time to develop the "film"
- The snapshot itself must be a namespace (metadata) object containing Key/Value list of object-versions, sharded as required

Eventual Snapshotting at a glance

SNIA Global Education

- **Two required primitives**
  - Given Snapshot **Scope**, find whether an object is in the Scope
  - Given Snapshot **Time**, find the *right* object version
- **Map**:



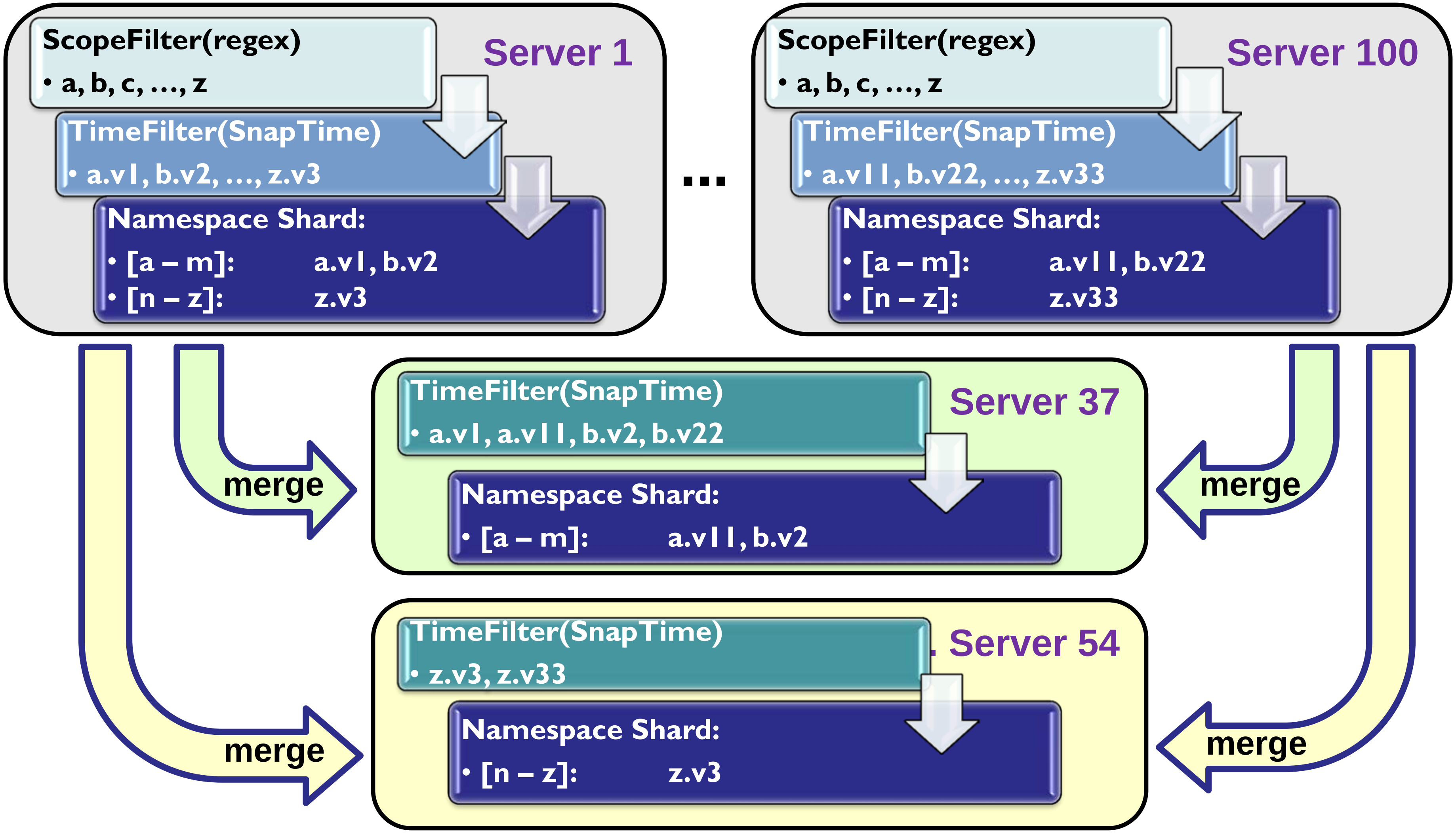


- **Reduce**:
  - **Reconcile multiple versions of the same object**
  - **Put the resulting shards as payloads of the new snapshot object**

Multi-versioned Eventual Consistency: Two Basic Primitives

- Snapshotting an object store boils down to carving out a part of its versioned namespace, and storing it as an object
- Let's illustrate the process with the help of two abstractions:
  - **ScopeFilter(Name, Scope)**
    - **Finds out whether a named object is in the specified scope**
    - Usage #1: **ScopeFilter("/a/b/c", "a/*")** - *will return TRUE*
    - Usage #2: **ScopeFilter(/tenant/bucket/some-name, "*.pdf")**
  - **TimeFilter(Name, Version1, Version2, SnapTime)**
    - **Finds out version(s) of the named object that correspond to the time of the snapshot**
    - **Returns: Version1 | Version2 | Both | None**

Snapshotting 100-node Object Cluster (1 of 3)



Snapshotting 100-node Object Cluster (2 of 3)

SNIA Global Education

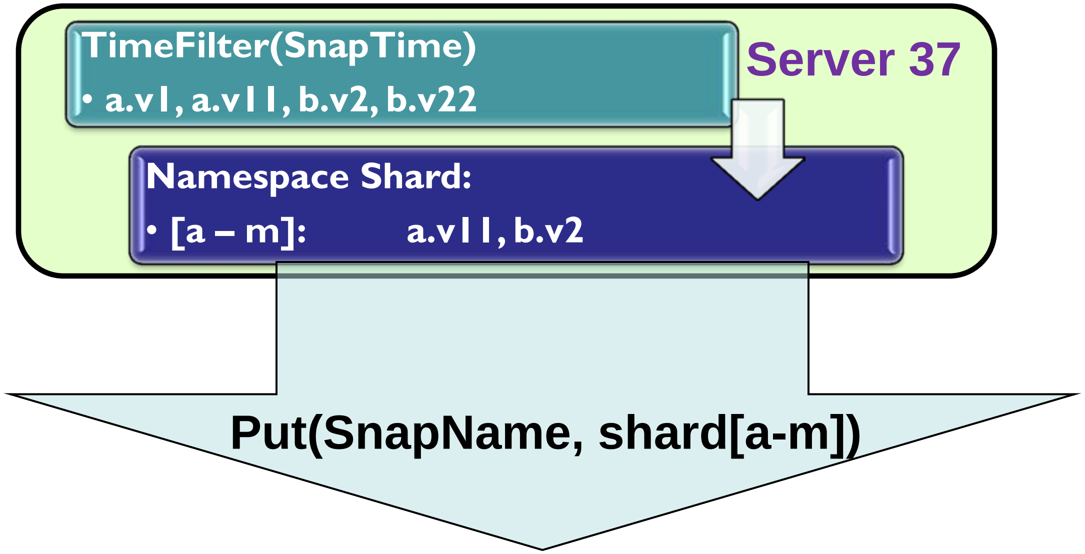
Mapping step:

- Snapshotting job starts processing locally stored namespace shards and local write logs
- Each of the 100 servers (in this example) applies the two filters: ScopeFilter() and TimeFilter()

Reducing step:

- Locally generated namespace shards get distributed internally – for instance, by hashes of their respective names
  - In this example, the targets are: server 37 and server 54
- Each shard includes a sorted KV list of object names and versions
- In the example, the resulting snapshot will contain two namespace shards

Snapshotting 100-node Object Cluster (3 of 3)



- Finally, each of the targets executes an optimized variant of internal Put
- The created snapshot object's payload, in this example, will contain two namespace shards
- Each shard will reference only those versioned objects that are, effectively, snapshotted

Key Takeaways

1. **Distributed snapshotting will *eventually* become a standard checklist item**
2. **Storage systems must be designed from ground up to support snapshotting**
   - **Late addition of the capability may prove to be difficult and costly**
3. **HDFS, Ceph, and other storage systems contribute to usable case studies and learning experience**
4. **Eventually consistent object storage can be snapshotted as illustrated**
   - **To satisfy the requirements stated above as well**

Attribution & Feedback

The SNIA Education Committee thanks the following Individuals for their contributions to this Tutorial.

Authorship History

Alex Aizman, 08/2016

Updates: 09/2016

Additional Contributors

Caitlin Bestler

*Please send any questions or comments regarding this SNIA Tutorial to **tracktutorials@snia.org***