SDC

STORAGE DEVELOPER CONFERENCE

SNIA SANTA CLARA, 2017

Causally Ordering Distributed File System Events

Tanuj Khurana & Raeanne Marks
Dell EMC Isilon

Agenda

- Overview
- Solutions Considered
- Final implementation deep-dive

Overview: What Problem Did We Solve?

- We needed a causal list of events on the file system called an “edit log”
  - Events which modify file system metadata
  - Events must have monotonically increasing “transaction IDs” (TXIDs)

Overview: What Problem Did We Solve?

- Perfect causality must be preserved
  - Events must apply cleanly to one snapshot to perfectly match a later snapshot
- Only causal events must be ordered with respect to each other
  - Concurrent events can be in any order

Overview: Challenges

- POSIX systems are not causal, including Isilon
  - No path modification serialization
- Performance and scalability must be preserved
- Events can happen on any node in an Isilon cluster
- Distributed systems are hard

Method 1: Global File System Lock

- Serialize all operations with exclusive lock
- Distributed, cluster-wide lock
- All-else-fails approach

Method 1 Analysis

- Pros:
  - Can be guaranteed to log operations in order
- Cons:
  - Unacceptably slow, no concurrency benefit
  - Ping-pong lock between nodes

Method 2: Redirect Traffic to One Node

- Master/slave approach
- Serialize all operations with a mutex instead of a cluster-wide file system lock
- Another all-else-fails approach

Method 2 Analysis

- Pros:
  - Can be guaranteed to log operations in order
  - Faster than a global file system lock
- Cons:
  - Unacceptably slow, no concurrency benefit
  - Single point of failure

Method 3: “Path Locks” + TXID Server

- Shared lock on unmodified path components, exclusive lock on modified path components
- Artificial path modification serialization for causality guarantees
- One master node serves monotonically increasing “Transaction IDs” (TXIDs) which identify and guarantee order

Method 3 Analysis

- Pros
  - Only events on the same path are serialized
  - Causality and ordering are guaranteed
- Cons
  - Fault tolerance, single point of failure
  - Complexity of implementation
  - Lock starvation & performance hit

Method 4: Sorting with Object Versioning

- “Object versioning” on all events
- Collect version information about an operation as it is performed and record it
- Determine a causal ordering of the events later by sorting using the versions collected

Method 4 Analysis

- Pros:
  - No locking
  - Minimize performance hit in the “fast path”
  - All work is done in background
- Cons:
  - All events sorted on one node
  - Memory usage

Implementation: General Idea

- Isilon inodes have revision numbers
  - Monotonically increasing with each metadata revision
- Record “viewed” inodes on lookup, and “modified inode(s)” right after modification
- Shouldn’t it be possible to sort with this?

Implementation: General Idea (Example)

- Event 1: Chown /a/b
  - Viewed:(20,3)(33,4), Modified: (52,5)
- Event 2: Create /a/b/c
  - Viewed: (20, 3)(33, 4), Modified: (52, 6)
- Event 1 sees inode 52 at a lower revision, so it must have happened before event 2

Implementation: High Level

1. Load recorded out-of-order events into memory
2. Construct an in-memory data structure called the “inode map”
3. Draw a graph of causal dependencies between events using the inode map
4. Perform topological sort on the DAG to obtain causal order

Step 1: Load Events into Memory

- Load events into an indexed vector, where the event ID is its position in the vector
- Each recorded event contains:
  - A list of inode numbers identifying viewed inodes + revision numbers
  - A list of modified inode numbers + revision numbers
  - A blob containing the remaining event information

Step 2: Construct Inode Map

- The inode map is a map of inode numbers to maps of revision numbers to sets of event IDs which contained that inode at that revision

```
Map<InodeNumber, Map<RevisionNumber, Set<EventID>>>
```


Step 2: Construct Inode Map

- Iterate through each event ID
- For each event, iterate through both the viewed and modified (inode number, revision) pairs
- For each pair, insert the current event ID into the set of event IDs under the key equal to the inode number and then under the key equal to the inode revision number

Example: Unsorted Events + Inode Map

Key: / = 1, a = 2, b = 3, c,d = 4

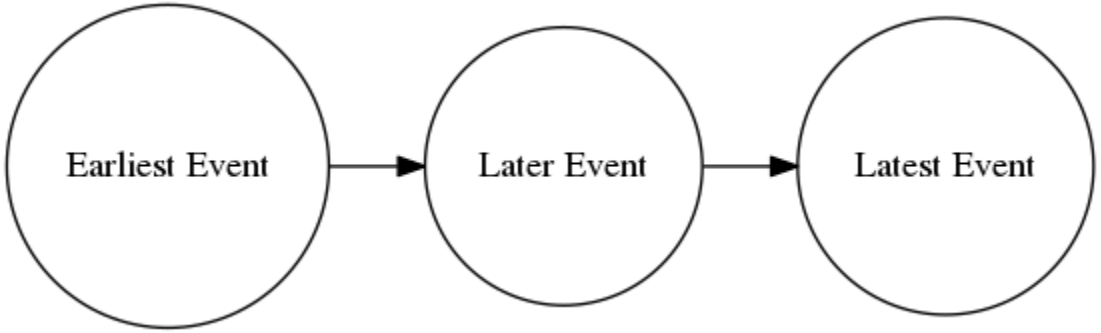
- Rename ‘/a/c’ -> ‘/b/d’
  - Viewed: (1,2) Modified: (2,3) (3,3)
- Delete ‘/b/d’
  - Viewed: (1,2) Modified: (3,4)
- Chmod ‘/’
  - Viewed: None Modified: (1,2)
- Append ‘/a/c’
  - Viewed: (1,2) (2,2) Modified: (4,3)
- Create ‘/a/c’
  - Viewed: (1,1) Modified: (2,2)

Inode Number	Revision Number	Event IDs
1	1	5
	2	1, 2, 3, 4
2	2	4, 5
	3	1
3	3	1
	4	2
4	3	4

Step 3: Draw Directed Acyclic Graph

- Each vertex in the DAG represents an event ID
- Each edge in the graph represents a “causal dependency”
- An edge from vertex 3 to vertex 10 means event 3 must have happened before event 10
- In our example, we have vertices 1-5 for events 1-5

Step 3: Draw Directed Acyclic Graph



Format of DAG

Step 3: Draw Directed Acyclic Graph

- Iterate through each event ID
- For each event, iterate through only the modified (inode number, revision) pairs
- Populate the “before” and “after” causal dependency edges
- Color the dependency edges “strong” or “weak”

“Strong” vs “Weak” Edges

- We have “strong” and “weak” dependencies
- “Strong” means causality is guaranteed
- “Weak” means causality is possible, not guaranteed
- Edge coloring helps cycle mitigation

Strong Dependencies

- Fetching modified inodes is atomic to modification
- Ordering with earlier events based on modified only is guaranteed to be causal
- Ordering later events based on modified and viewed is guaranteed to be causal

Strong Dependencies

- Before dependency:
  - Event 1: Rename ‘/a/c’ -> ‘/b/d’
    - Viewed: (1,2) Modified: (2,2) (3,3)
  - Event 2: Delete ‘/b/d’
    - Viewed: (1,2) Modified: (3,4)

Strong Dependencies

- After dependency:
  - Event 3: Chmod ‘/’
    - Viewed: None Modified: (1,2)
  - Event 4: Append ‘/a/c’
    - Viewed: (1,2) (2,1) Modified: (4,3)

Weak Dependencies

- Viewing inodes on lookup is *not* atomic to modification
- Viewed inode revision numbers could be stale by time of modification
- Ordering earlier events based on viewed inodes is not guaranteed to be causal

Weak Dependencies

- Before dependency:
  - Event 5: Create ‘/a/c’
    - Viewed: (1,1) Modified: (2,2)
  - Event 3: Chmod ‘/’
    - Viewed: None Modified: (1,2)

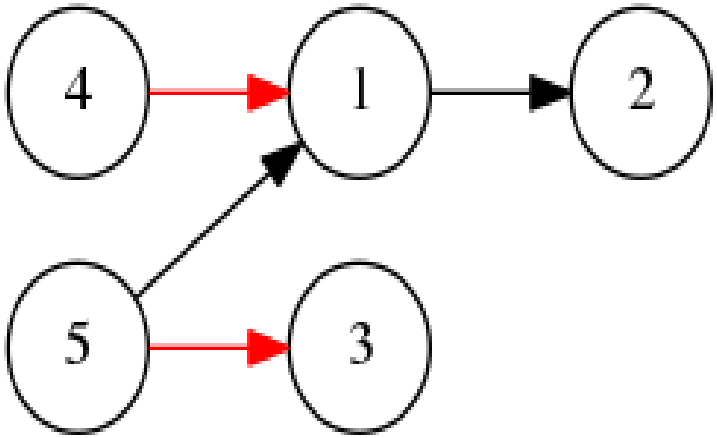
Step 3.1: Draw “Before” Dependencies

- For each (inode number, revision) pair in the current event’s modified inodes, locate the set of event IDs with that inode number and the next lowest revision number in the inode map
- For each vertex in the set, draw an edge from it to the current vertex

Step 3.1: Draw “Before” Dependencies

- These might be “weak” edges - causality is NOT guaranteed
- If another event has this inode at a lower revision number, it may have happened before this event

Step 3.1: Draw “Before” Dependencies



Red edges are weak

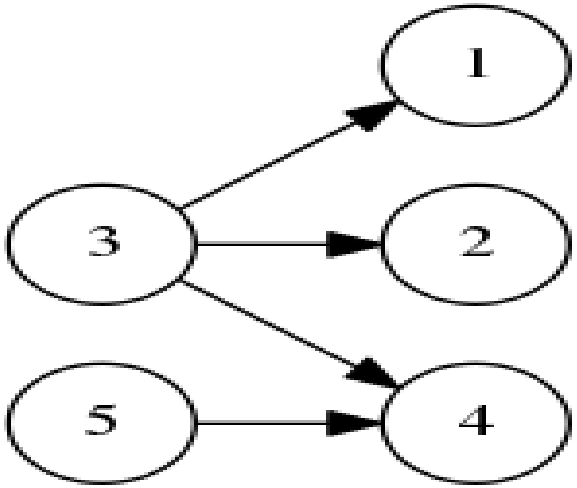
Step 3.2: Draw “After” Dependencies

- For each (inode number, revision) pair in the current event’s modified inodes, locate the set of event IDs with that inode number and same revision number in the inode map
- Draw an edge from the current event’s vertex to the vertices of each event in the set

Step 3: Draw “After” Dependencies

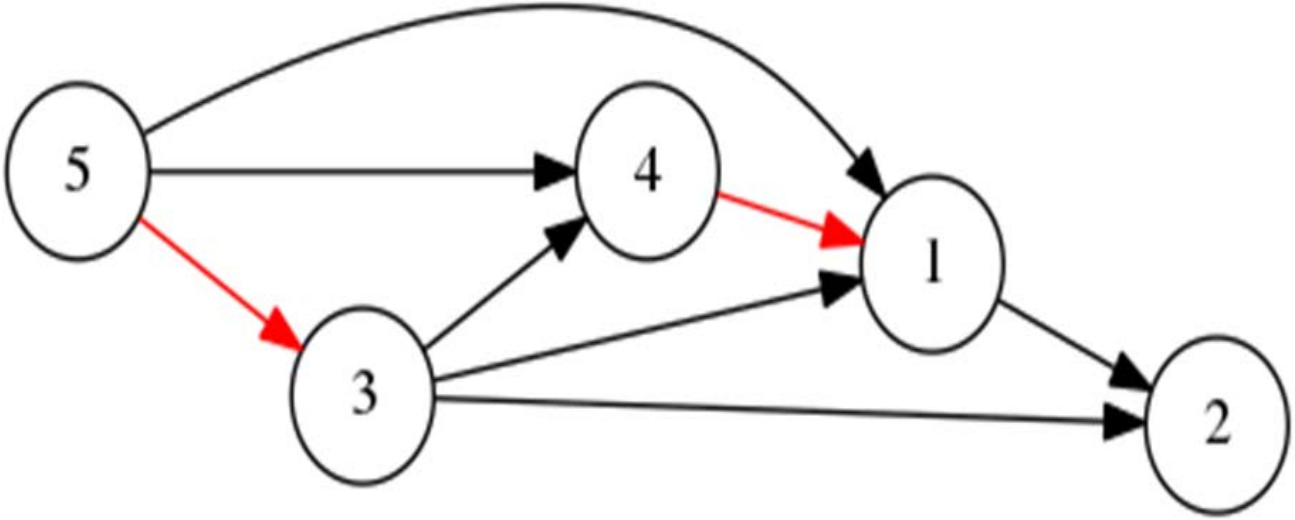
- These are “strong” edges - causality is guaranteed
- If the current event modified this inode, it was the first to see it at this revision number
- Any other event with the same inode and revision number must have occurred after the current event

Step 3: Draw “After” Dependencies



All Edges are “Strong”

Example: Complete Graph



Example: Resulting Causal Order

- Create ‘/a/c’
- Chmod ‘/’
- Append ‘/a/c’
- Rename ‘/a/c’ -> ‘/b/d’
- Delete ‘/b/d’

* Recall: Previous order was Rename, Delete, Chmod, Append, Create

Step 4: Perform Topological Sort

- Topological sort on the DAG produces causal order
- Linear ordering of the graph such that for all directed edges uv, u is before v in the ordering
- Shuffle the events vector to fit the ordering dictated by topological sort
- Write to log, done!

Analysis

- Populating the inode map is O(n)
- Drawing the edges is $O(n^2)$, observed to be less
- Topological sort is O(n+e)
- Re-ordering the vector is O(n)
- No observed issues with performance or scalability with our use case

Problems Encountered

- Cycles
- Race conditions with “stale” viewed inodes
- Other processes modify inodes, revisions were missing
- Mitigation required based on “weak edges”

Cycle Mitigation

- Draw fewer edges
  - Don’t draw a weak edge if there is already a strong edge in any direction, etc
- If a cycle is encountered with one weak edge, remove it and sort again

Outcome

- Edits can be applied cleanly to a snapshot to match a later snapshot
- Perfect causal ordering
- No cycles after cycle mitigation efforts
- Sorting is fast enough
- No performance hit in the fast path

Outcome

- Limitations
  - Chronology is not preserved for non-causal events
  - “Concurrent” events will be logged in arbitrary order
- Overall, it fits exactly what we needed

Questions?

- Thank you!