Δfs

What’s Beyond IndexFS & BatchFS

Envisioning a Parallel File System without Dedicated Metadata Servers

Qing Zheng

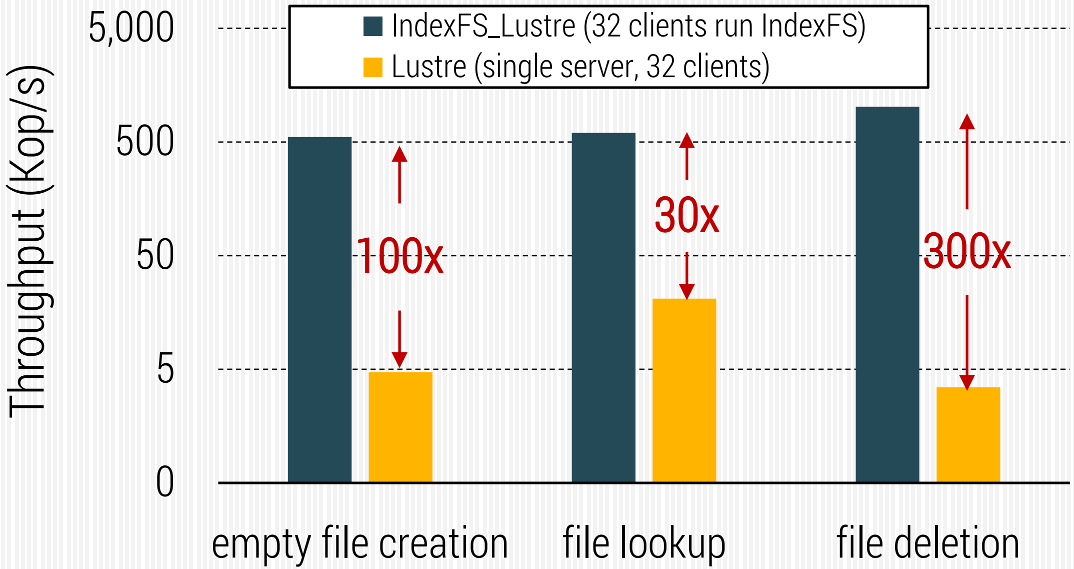
Kai Ren, Garth Gibson, Bradley W. Settlemyer, Gary Grider

Carnegie Mellon University

Los Alamos National Laboratory

Scaling needs decoupling

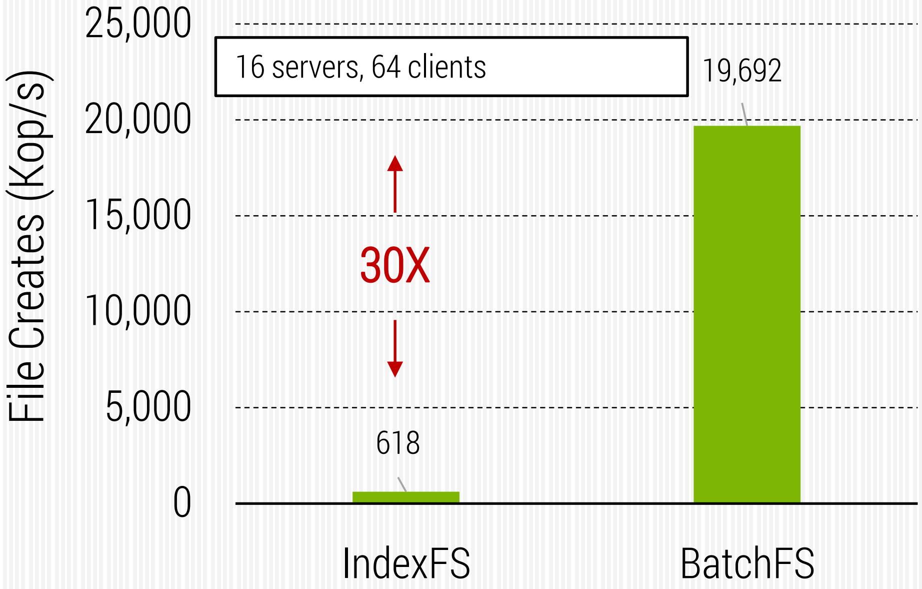
- NASD [asplos98]
  - decoupling data from metadata
  - Lustre, Google FS, etc
- IndexFS [sc14]
  - dynamically partitioned metadata middleware
  - orders of magnitude faster than Lustre in metadata



Exa- scaling demands ever more decoupling

Compute-side server code

- BatchFS [pdsw14]
  - decoupling clients from servers
  - temporarily scale beyond the total number of servers
  - very fast for a while and eventually clients communicate with servers to merge updates



How much further can we delay & decouple merging ?

ΔFS Goal

- Want the peak Tput BatchFS demonstrated
- Compel freedom from server synchronization
    - by eliminating all server machines
    - by dealing with issues rising from the absence of metadata servers
    - by not assuming an underlying PFS

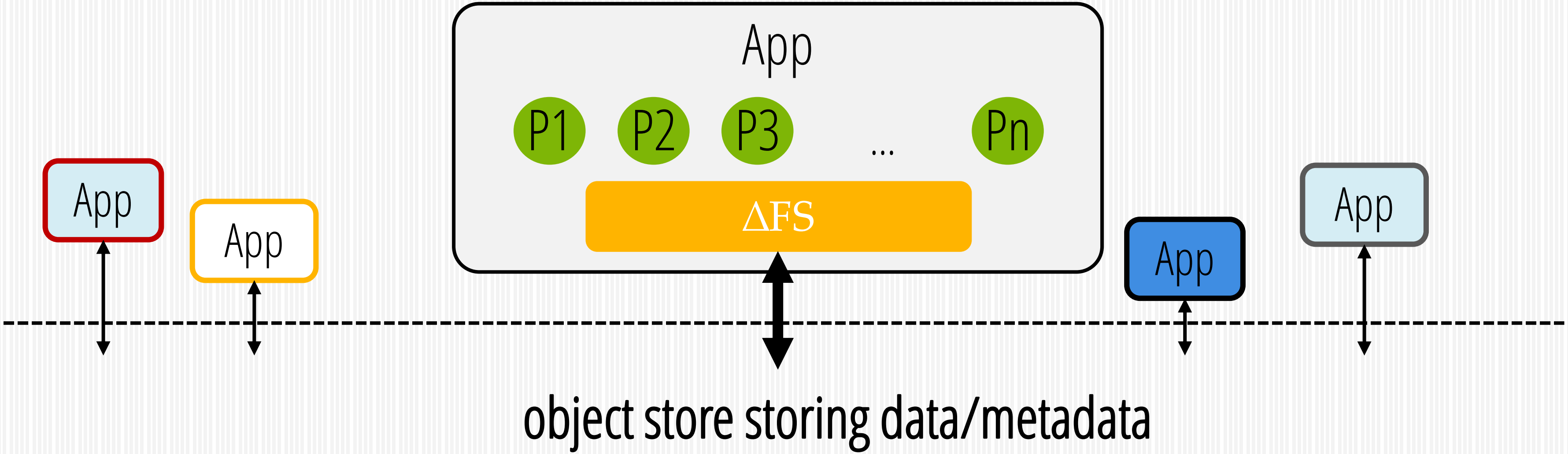
Scale beyond BatchFS

Agenda

- DeltaFS design
- Why no dedicated servers is not a problem

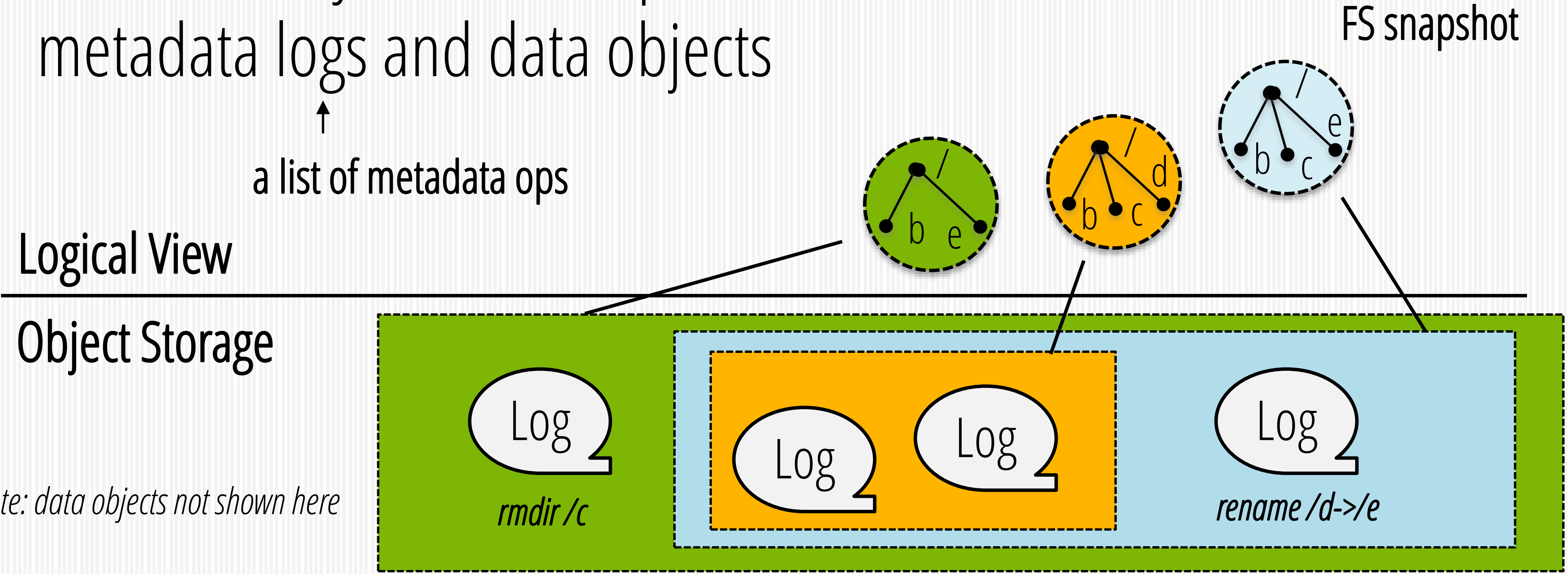
Middleware Design

ΔFS is middleware spawned by each parallel app



ΔFS Overview

FS defined by a set of snapshots stored as sets of metadata logs and data objects

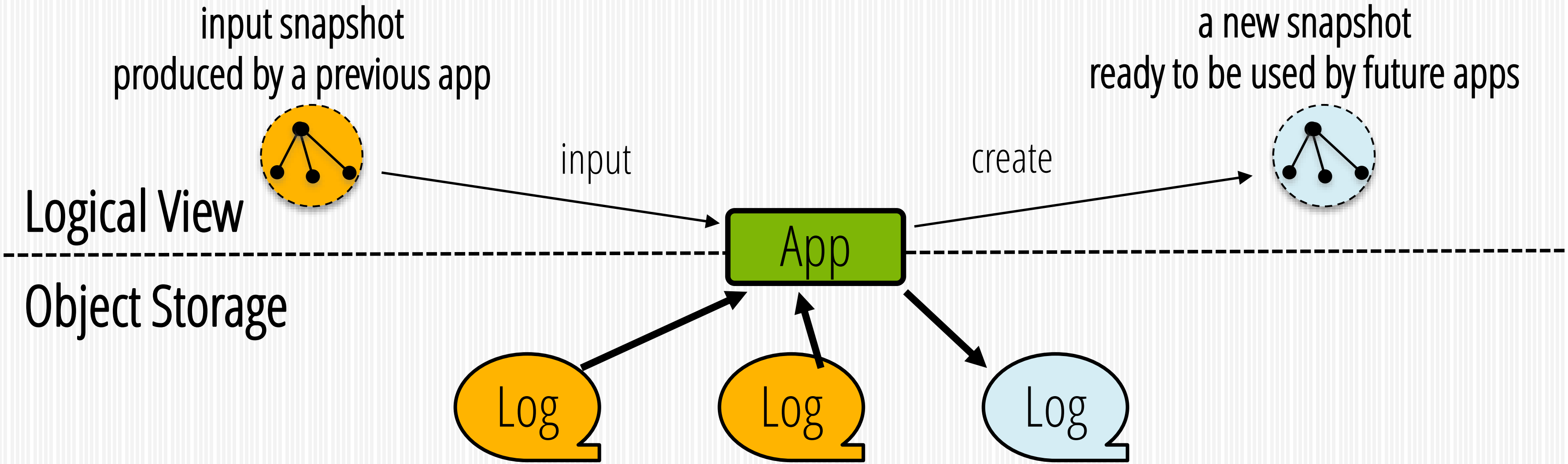


Note: data objects not shown here

System Model

Reads input dataset from an existing FS snapshot

Creates a new snapshot with output data inserted



Key take-away

- NO global namespace

Each namespace is defined by the app and the logs loaded by it

- NO false sharing

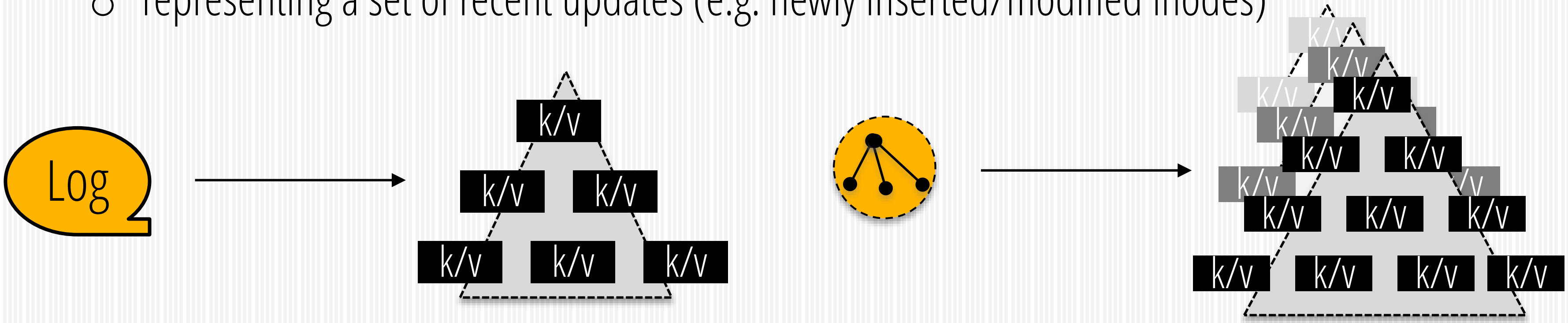
Apps don’t access logs not needed by them

- NO dedicated metadata servers

App directly communicates with the storage to load/dump metadata logs

How logs are implemented ?

- TableFS [atc13]
  - namespace = a large dir entry table + embedded inodes
    - implemented as **LSM-Tree** (a collection of ordered B-Trees)
- Each log object is a differential B-Tree (diff)
  - representing a set of recent updates (e.g. newly inserted/modified inodes)



Why LSM-Tree is a good idea ?

- Logs are 1st–class data

No need to replay logs to recover namespaces

Near-zero cost of merging namespaces

- Each log is self-indexed

Scanning/reading within a single log is fast: O(logN)

Scanning/reading a series of non-overlapping logs is as fast as a single log

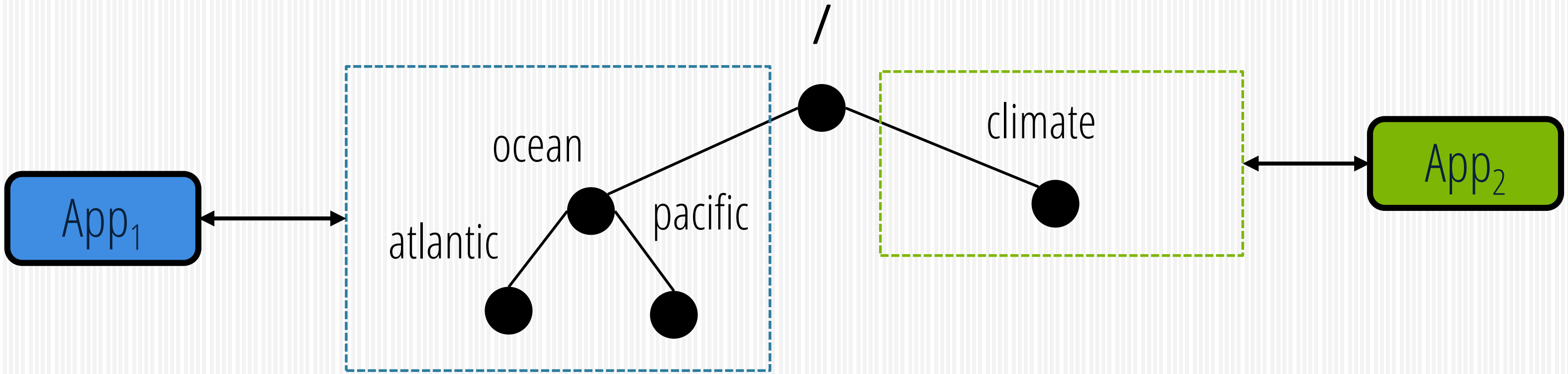
Agenda

- DeltaFS design
- Why no dedicated servers is not a problem

P1: Do my apps need the FS to communicate/synchronize ?

Unrelated Apps

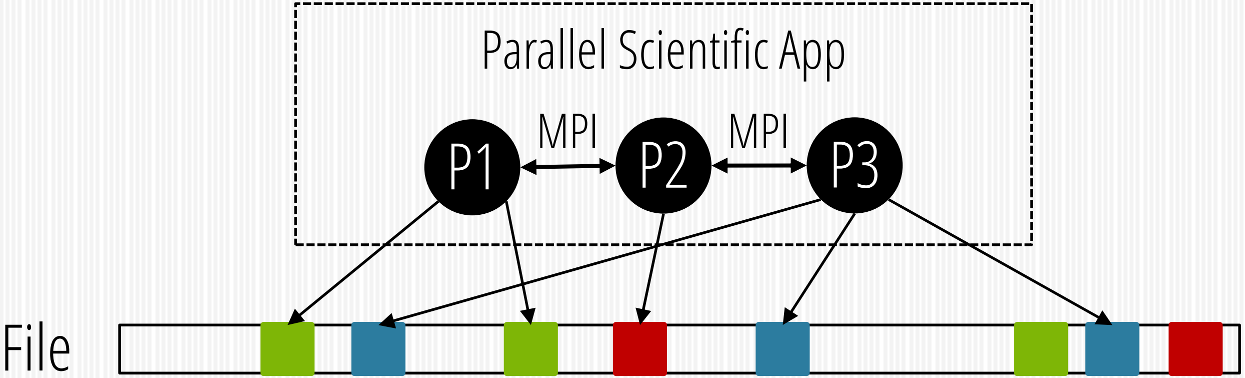
Work on different datasets and don't communicate.



Don't need the FS to communicate

Self-Coordinating Apps

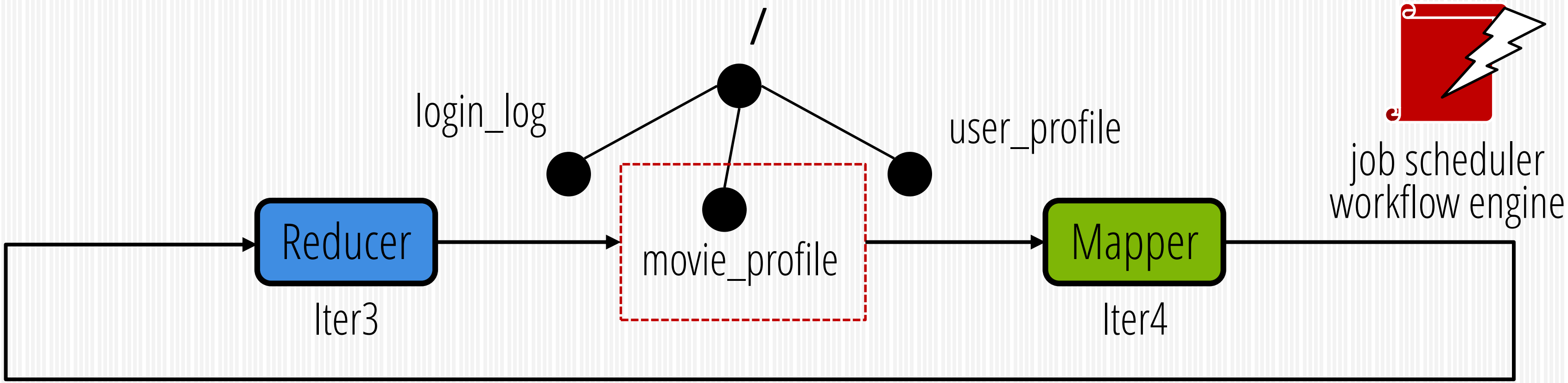
Use middleware to share faster & more efficiently



Don’t need the FS to communicate

Workflow Apps

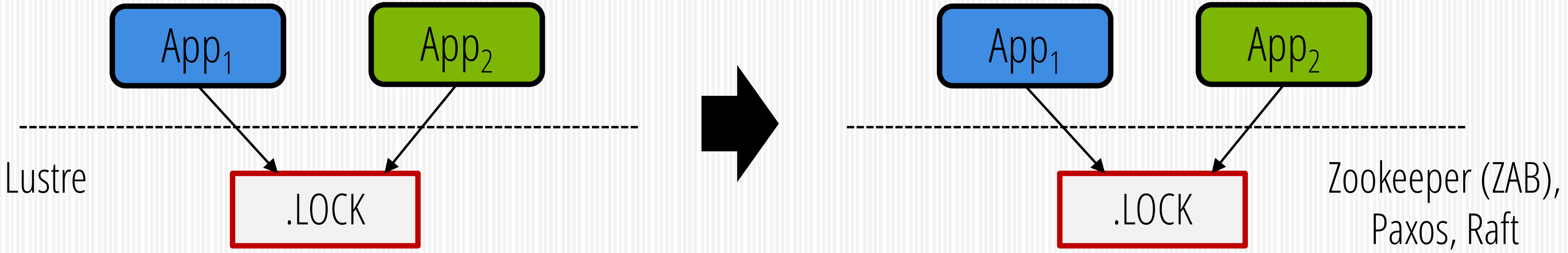
Externally coordinated by job schedulers



Don’t need the FS to communicate

Anonymous Synchronization

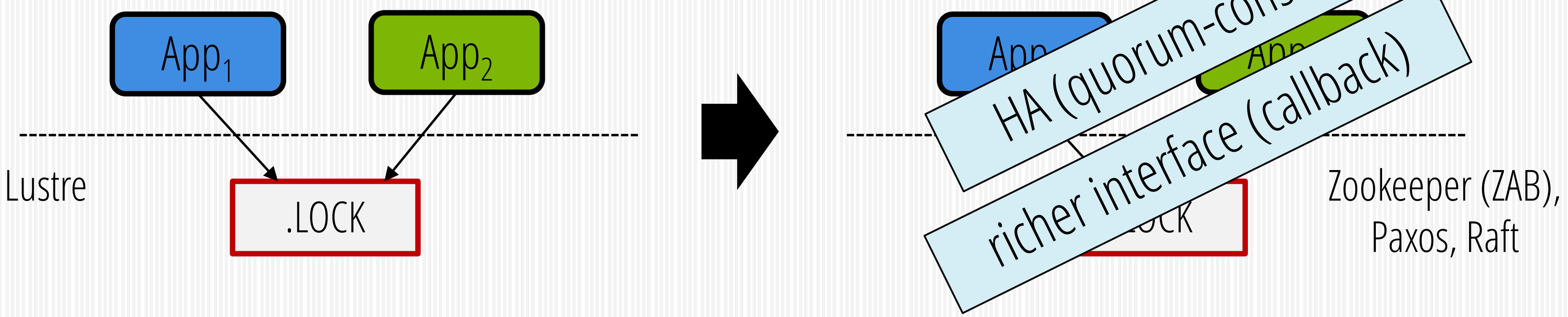
e.g. Two app instances competing for mastership



Turn to a mechanism outside the FS to coordinate

Anonymous Synchronization

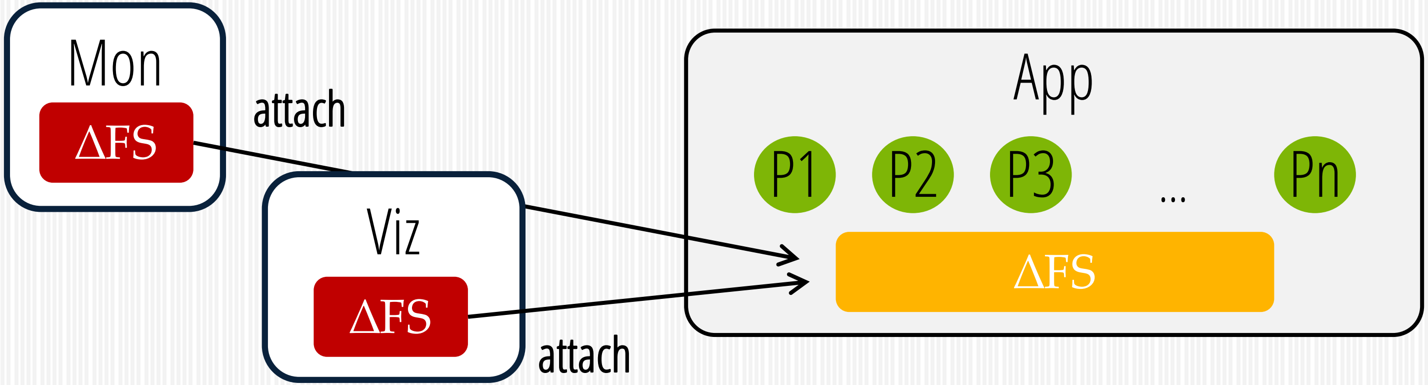
e.g. Two app instances competing for mastership



Turn to a mechanism outside the FS to coordinate

P2: But I often use different programs to access data concurrently !

User requested concurrent sharing

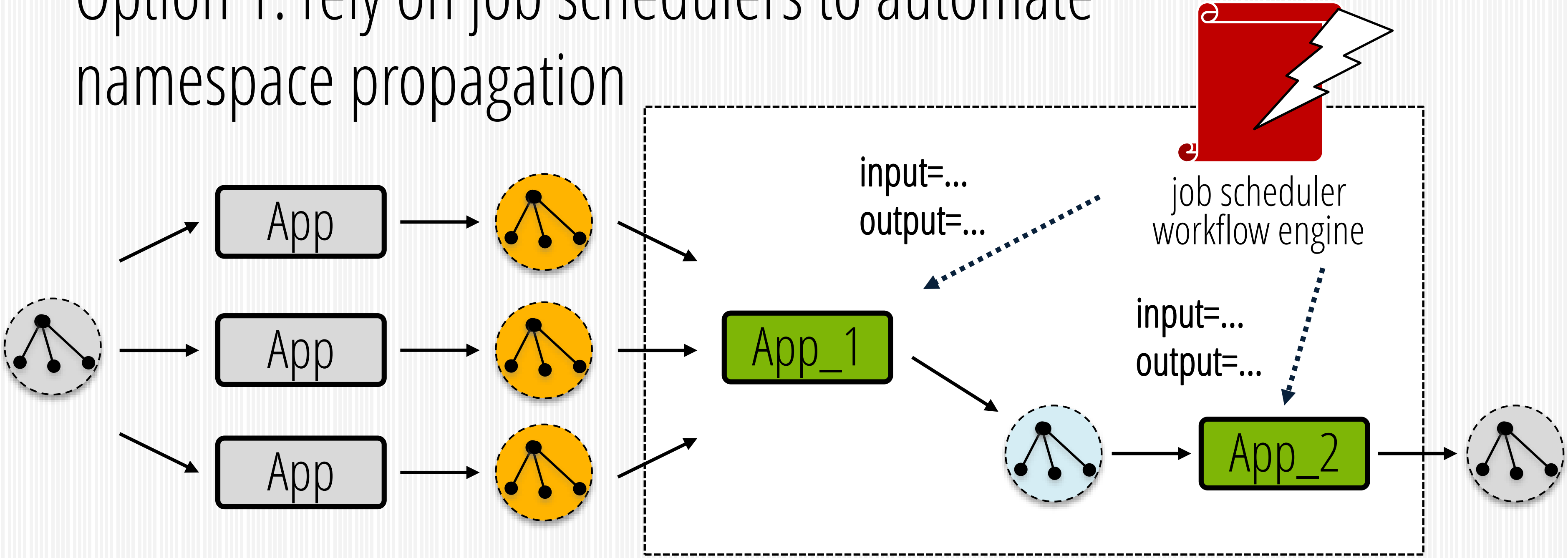


Link to ΔFS middleware and attach to the primary parallel app

P3: Which snapshots to use ?

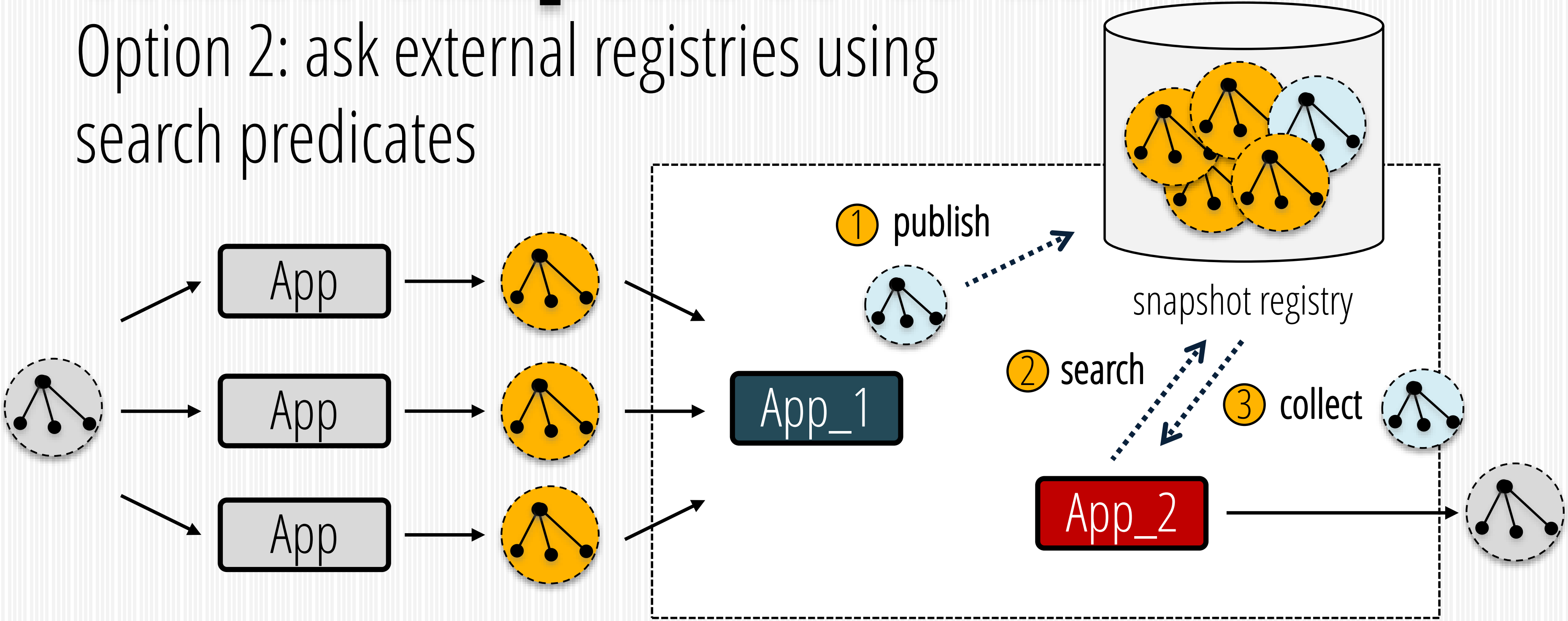
Which snapshots to use ?

Option 1: rely on job schedulers to automate namespace propagation



Which snapshots to use ?

Option 2: ask external registries using search predicates



Finding snapshots is like searching a page using Google

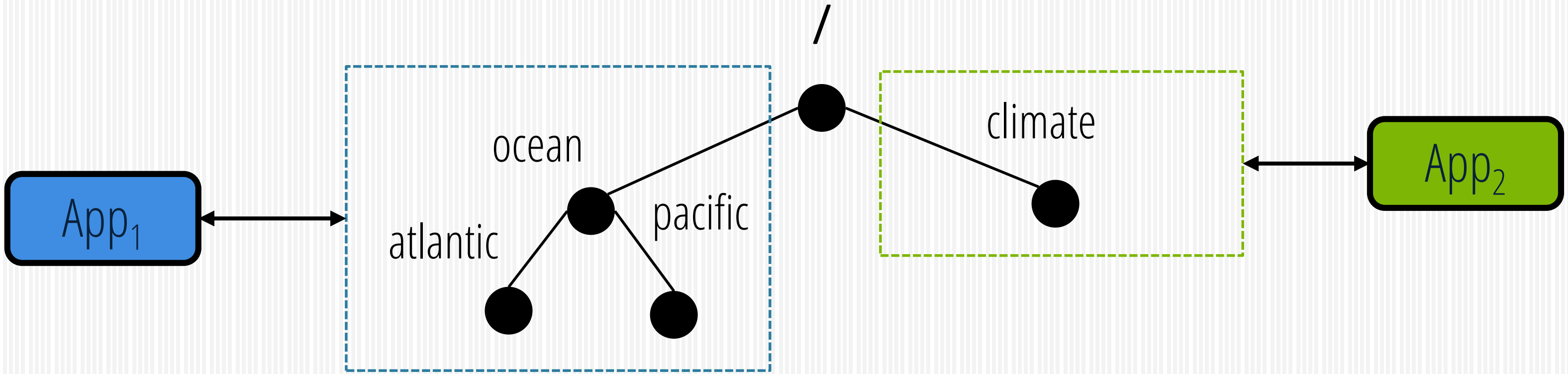
- Possible search predicates
  - find latest stable science code for my science
  - find latest recommended mesh model and cleaned input data
  - find latest vendor recommended HW libraries
- Also, there can be multiple snapshot registries

Allows programmable namespace composition

P4: What about potential conflicts among different snapshots ?

Unrelated Apps

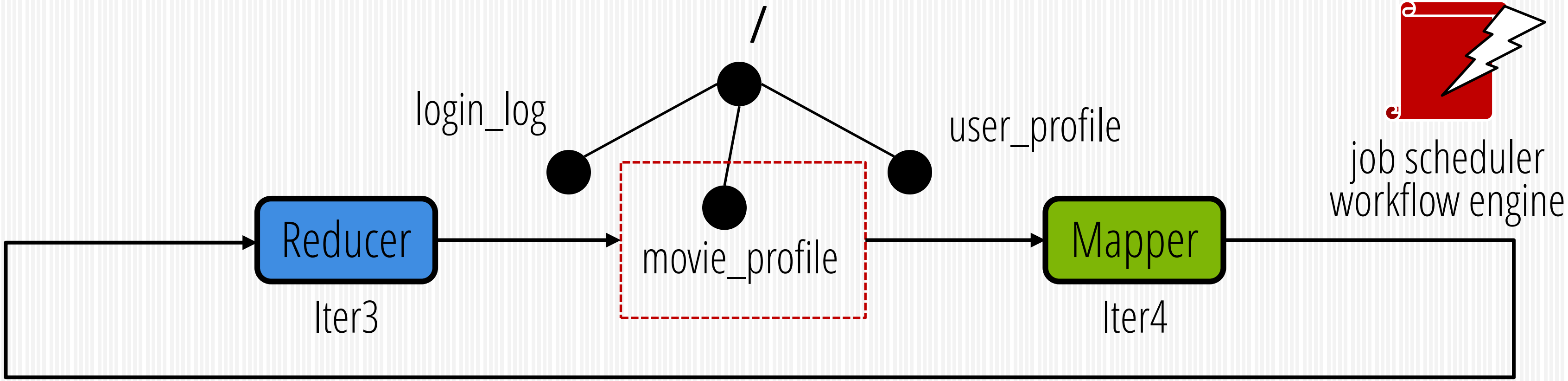
Work on different portions of the namespace



Won’t generate any conflicts

Workflow Apps

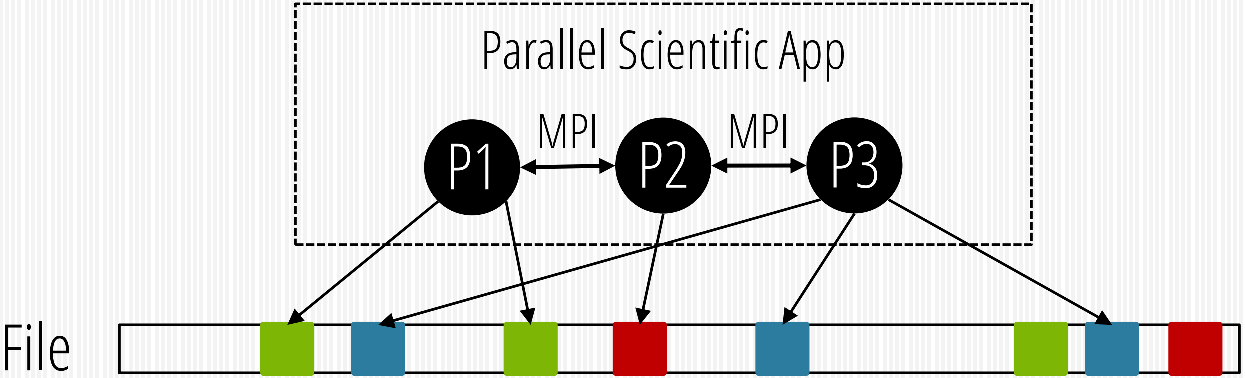
Access the same dataset at different time



Won't generate any conflicts

Self-Coordinating Apps

Coded to be conflict-free



Won’t generate any conflicts

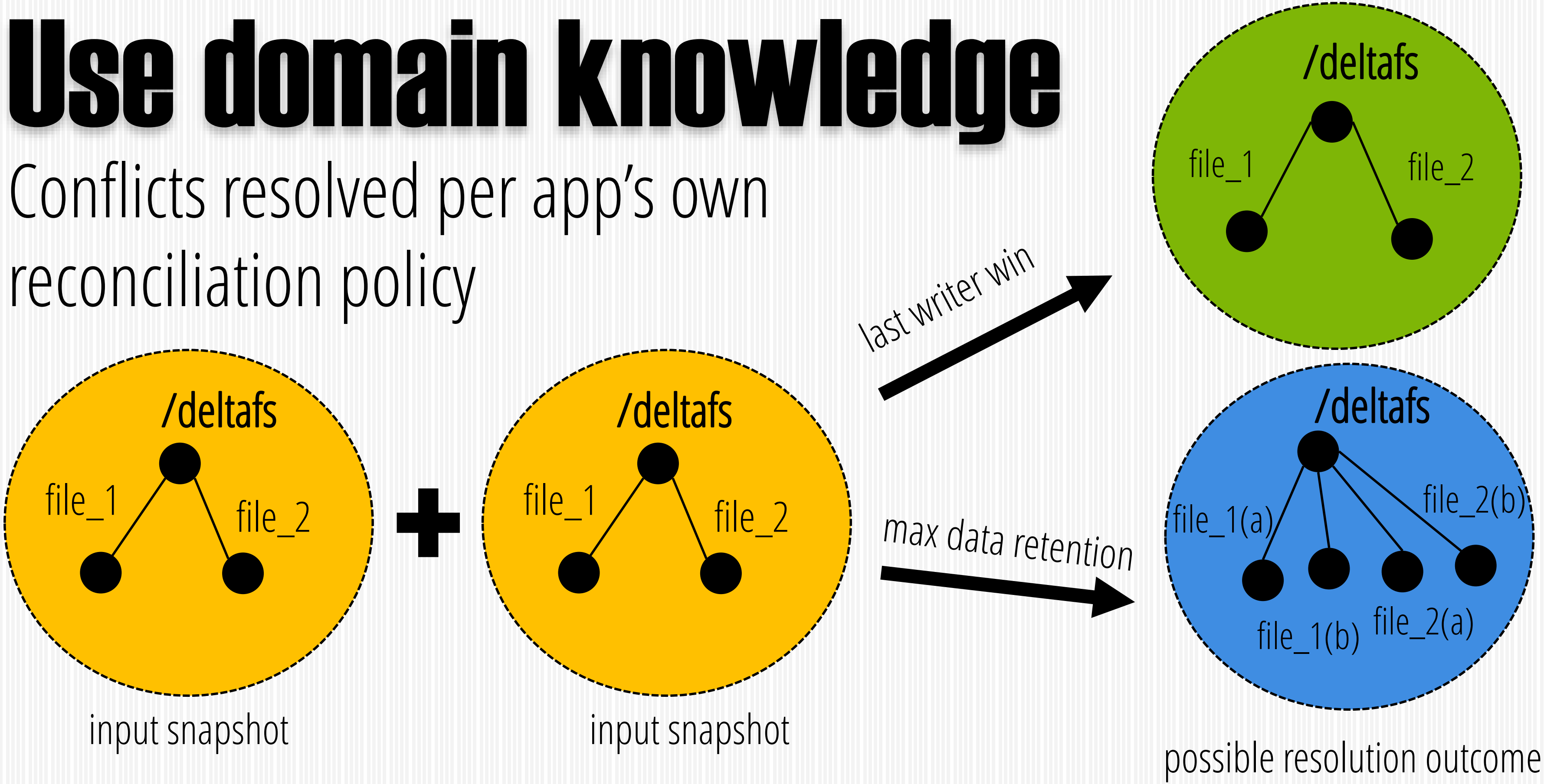
Namespace composition is fast if there is no conflict

- Recall: near-zero cost of merging logs
  - better if those logs do not overlap with each other

What if there are conflicts ?

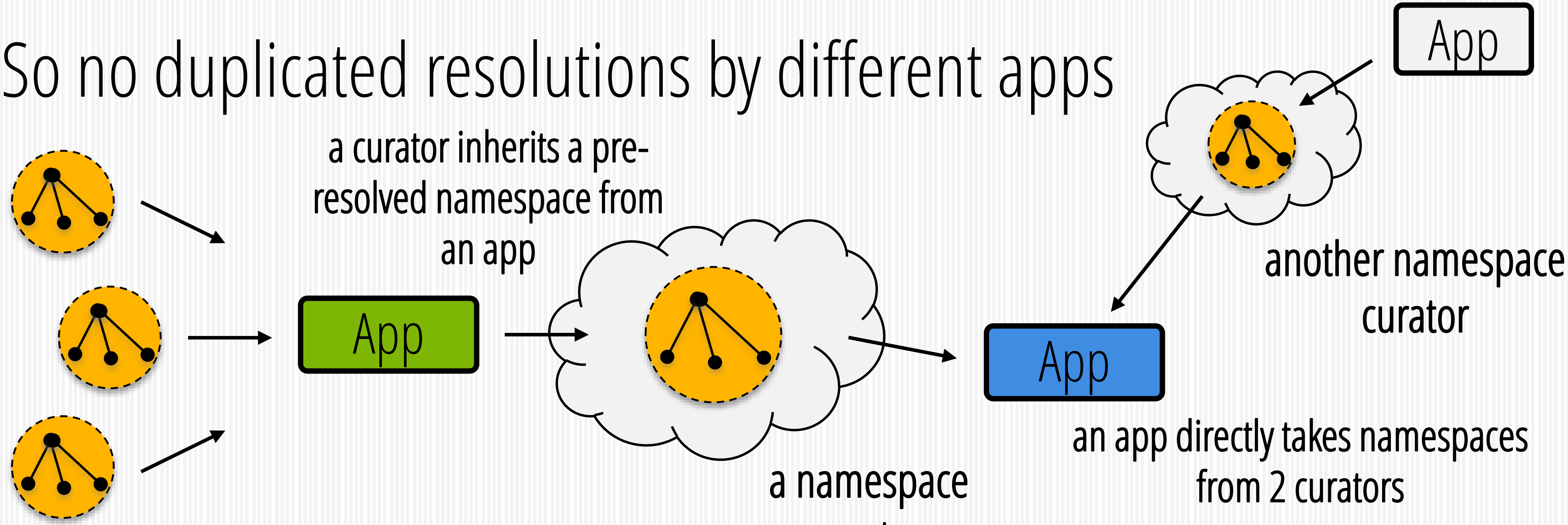
Use domain knowledge

Conflicts resolved per app's own reconciliation policy



Use curators to remember conflict resolution results

So no duplicated resolutions by different apps


App
a curator inherits a pre-resolved namespace from an app
App
App
another namespace curator
an app directly takes namespaces from 2 curators
a namespace curator

Conclusion

- Strong scalability needs strong decoupling
  - exiting clients synch too often with servers
  - removing servers force us to rethink on what is necessary
  - need to try radically different model for shared storage