

# Changing Requirements for Distributed File Systems in Cloud Storage

Wesley Leggette  
Cleversafe

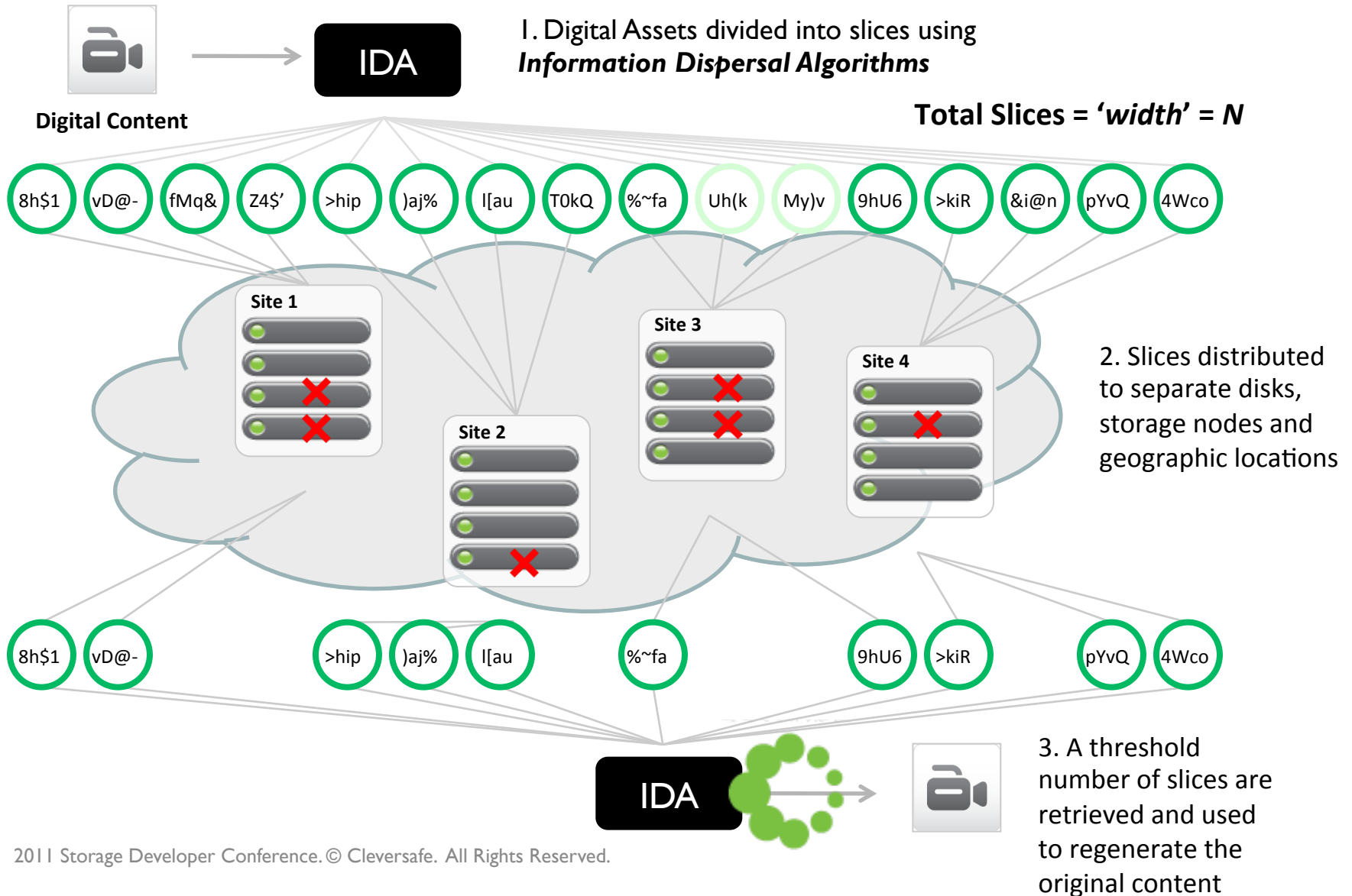
# Presentation Agenda

- ❑ About Cleversafe
- ❑ Scalability, our core driver
- ❑ Object storage as basis for filesystem technology
  - ❑ Namespace-based routing
  - ❑ Distributed transactions
  - ❑ Optimistic concurrency
- ❑ Designing an ultra-scalable filesystem
  - ❑ Filesystem operations on object layer
- ❑ Conclusions

# About Cleversafe

- ❑ We offer scalable storage solutions
  - ❑ Target market is massive storage (> 10 PiB)
  - ❑ Information Dispersal Algorithms (Erasure Codes)
    - ❑ Reduce cost by avoiding replication overhead
    - ❑ Maximize reliability by tolerating many failures
- ❑ Object storage core product offering
- ❑ How to translate this technology to filesystem space
  - ❑ Evolution from object storage concepts
  - ❑ Also influenced by distributed databases and P2P
  - ❑ Techniques we investigate not unique to IDA

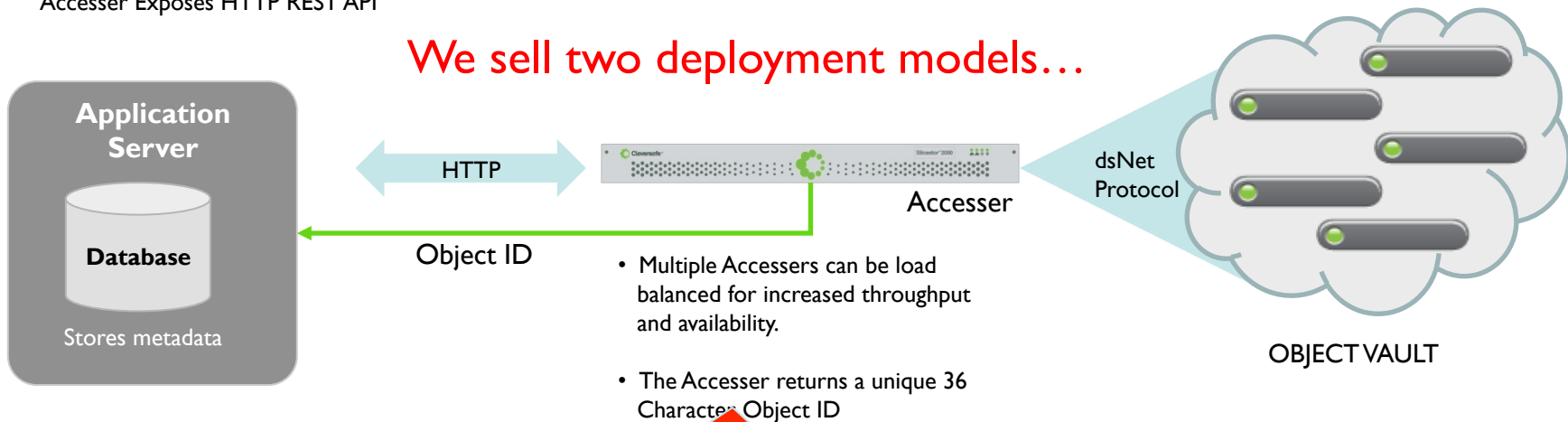
# How Dispersed Storage Works



# Access Methods

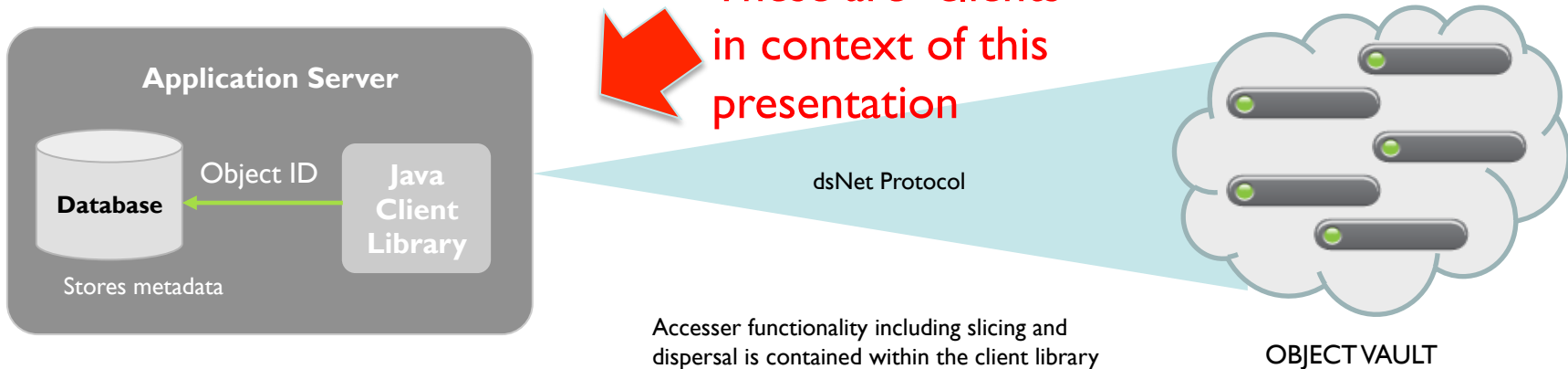
## Simple Object HTTP

Accesser Exposes HTTP REST API



## Simple Object Client Library

Accesser Function Embedded into the Client



# Scalability – A Primary Requirement

- ❑ **Big Data** customers are petabyte to exabyte scale
- ❑ Scale out architecture
  - ❑ Add storage capacity with commodity machines
  - ❑ Reduce costs: commodity hard drives
- ❑ Invariants
  - ❑ Reliability – keep data even as cheap disks fail
  - ❑ Availability – access data during node failures
  - ❑ Performance – linear performance growth

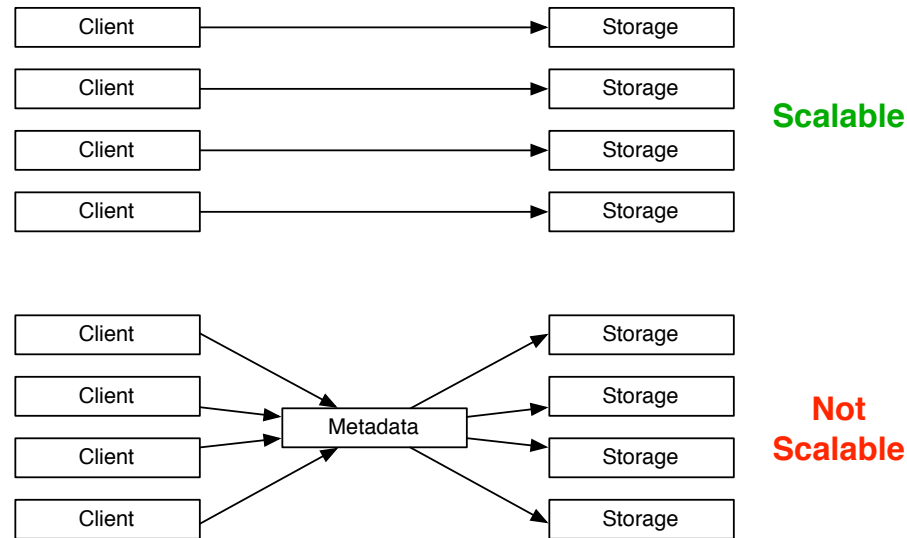
# Scale Example

- ❑ Shutterfly
  - ❑ 10 PB Cleversafe dsNet storage system
  - ❑ All commodity hard drives
  - ❑ Single storage container for all photos
  - ❑ 10's thousands of large photos stored per minute
    - ❑ Max capacity many times this level
  - ❑ 14 access nodes for load balanced read/write
    - ❑ No single point of failure
    - ❑ Linear performance growth with each new node
- ❑ This uses object storage product

# Investigating Filesystem Space

- ❑ We have scalable object storage
  - ❑ Limitless capacity and performance growth
  - ❑ Fully concurrent read/write
- ❑ Some customers want the same with a filesystem
  
- ❑ Is this technically possible?
- ❑ What tradeoffs would have to be made?

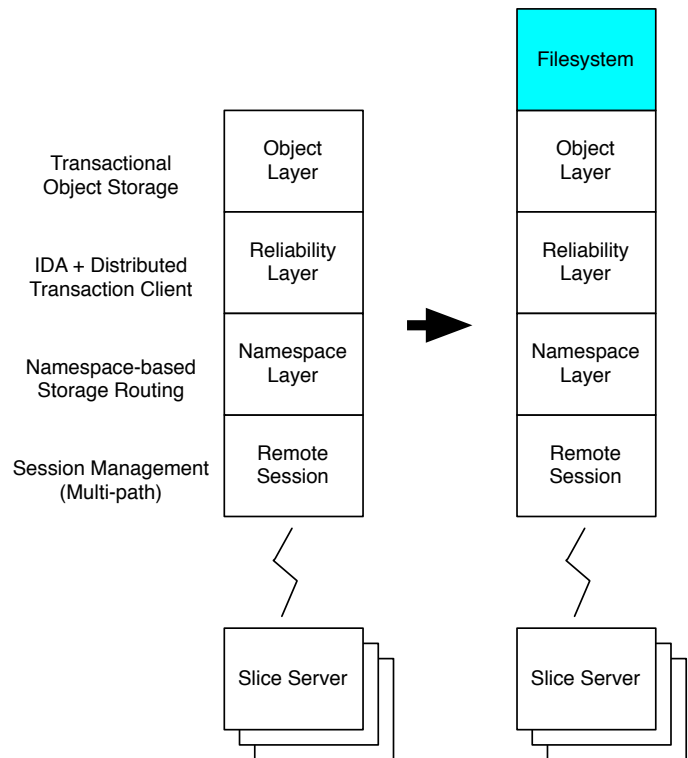
# Scale comes from homogeneity



- ❑ To scale out, we need to do so at each layer
  - ❑ Eliminate central chokepoint for data operations
    - ❑ Central point of failure, central point of...
  - ❑ We accomplish this today with object storage
    - ❑ Consider same concept in a filesystem

# What approach can we take?

- ❑ Start with scalable transactional object storage
- ❑ Add filesystem implementation on top



## ❑ Object Layer

- ❑ Check-and-write transactions

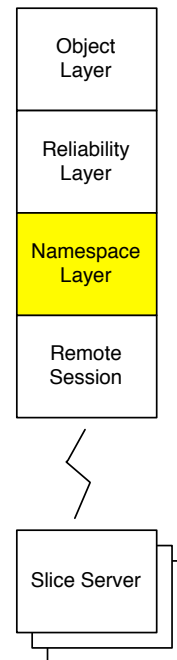
## ❑ Reliability Layer

- ❑ Ensures committed objects reliable and consistent

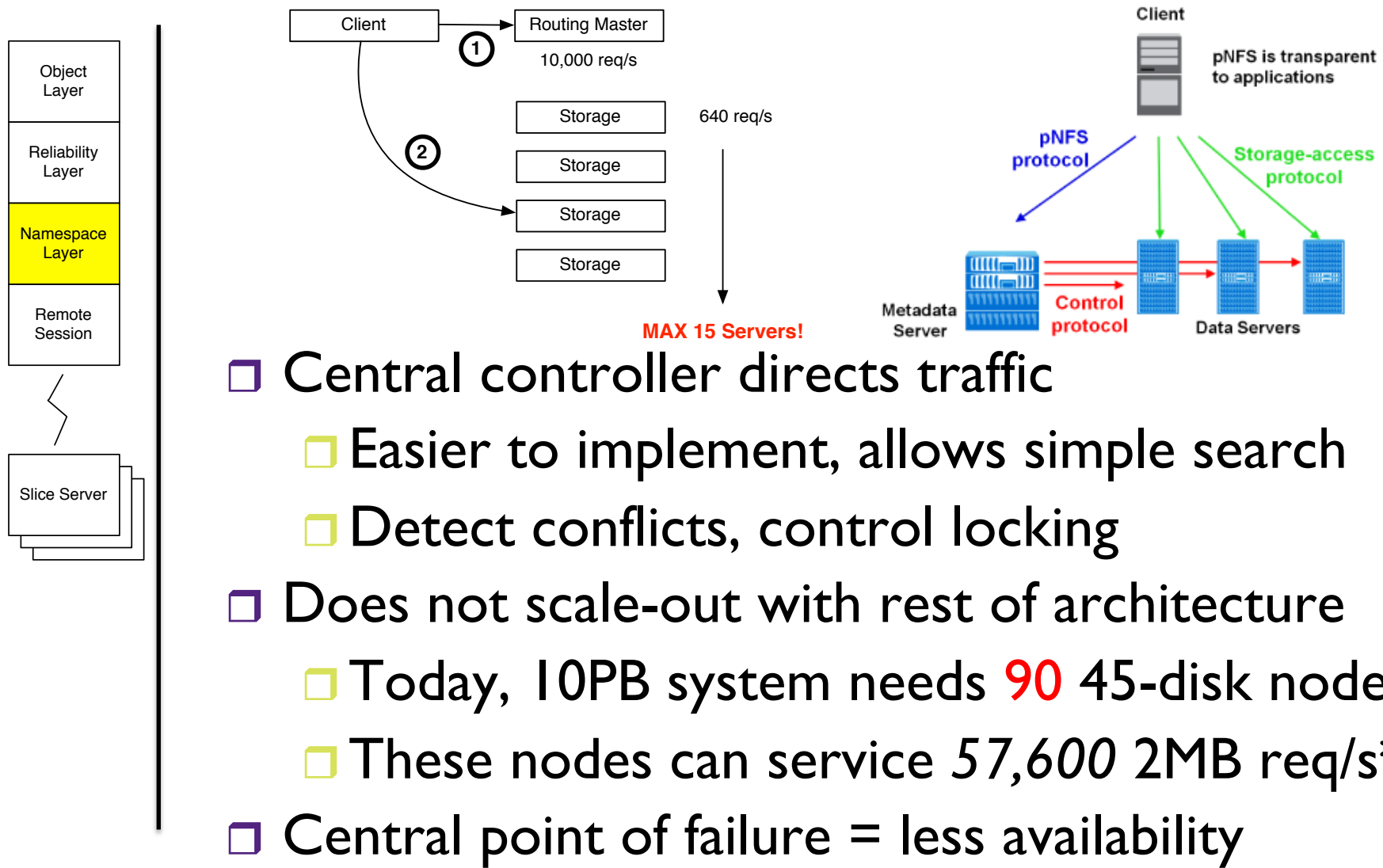
## ❑ Namespace Layer

- ❑ Routes actual data storage
- ❑ No central I/O manager

# Namespace Layer

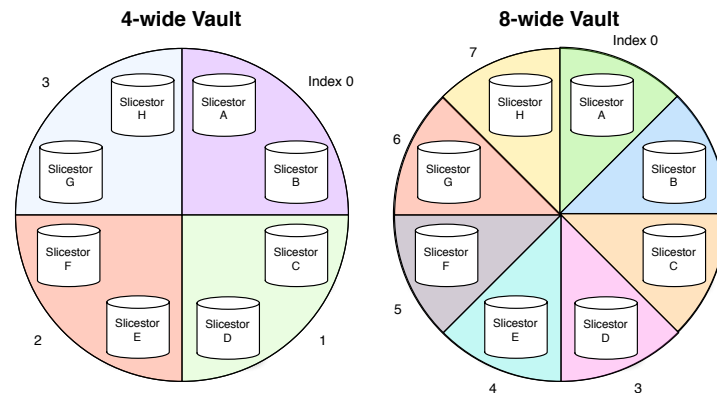
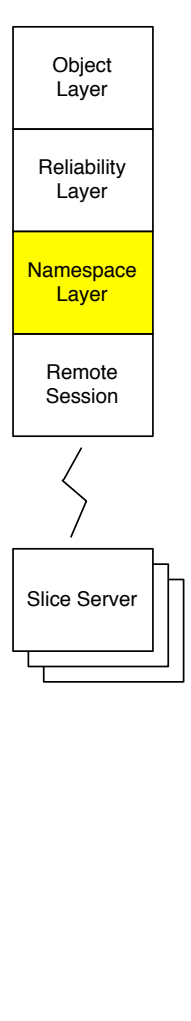


# Traditional Centralized Routing



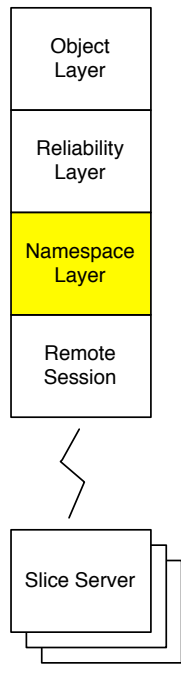
- ❑ Central controller directs traffic
  - ❑ Easier to implement, allows simple search
  - ❑ Detect conflicts, control locking
- ❑ Does not scale-out with rest of architecture
  - ❑ Today, 10PB system needs **90** 45-disk nodes\*
  - ❑ These nodes can service **57,600** 2MB req/s\*\*
- ❑ Central point of failure = less availability

# Namespace-based Routing

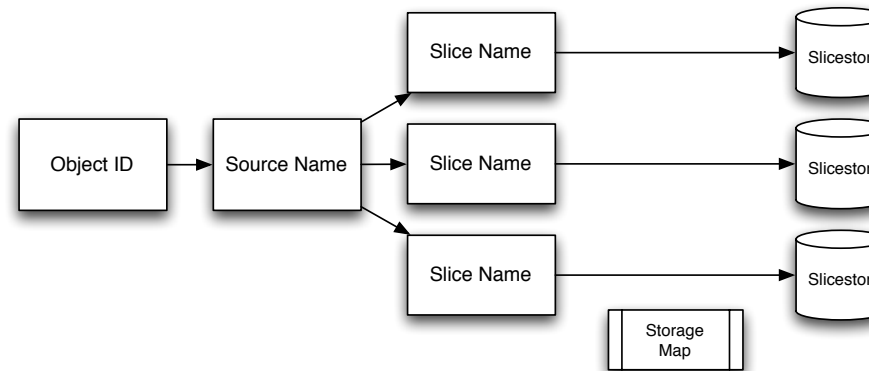


- Namespace concept from P2P systems
  - Chord, CAN, Kademila
  - MongoDB, CouchDB production examples
- Physical mapping determined by **storage map**
  - Small data (<10KiB) loaded at start-up
    - P2P systems use dynamic overlay protocol
    - We'll have 10's thousands of nodes, not millions

# Storing Data in a Namespace

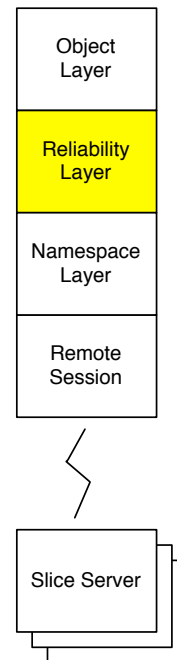


- ❑ No central lookup for data I/O
  1. Generate “object id”
  2. Map to storage

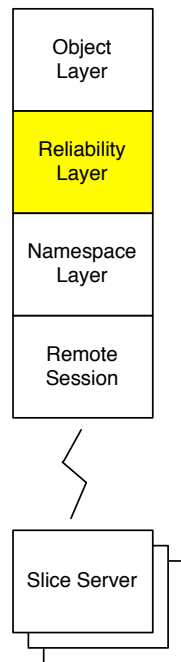


- ❑ With object storage, object id → database
- ❑ **How do we map file name to object id?**

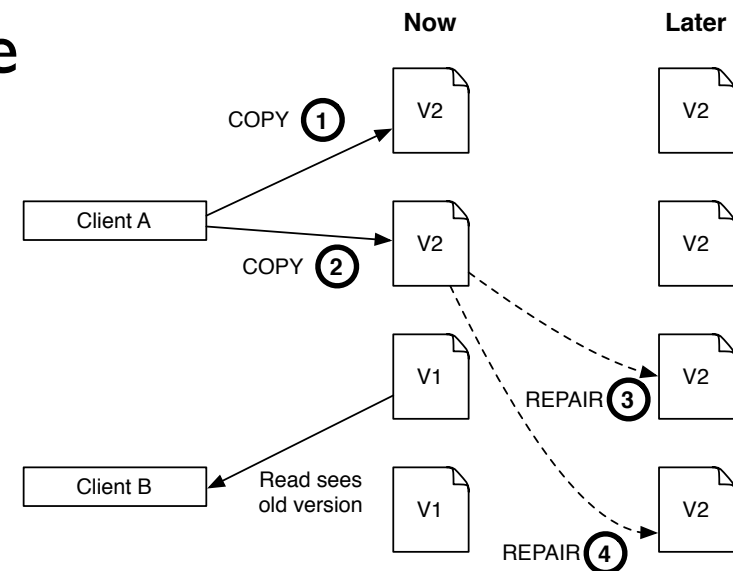
# Reliability Layer



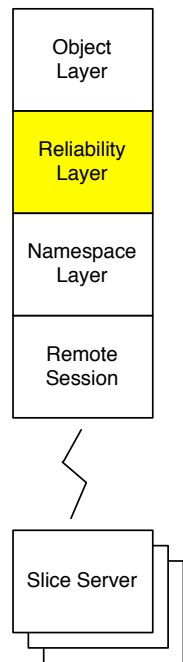
# Replication and Eventual Consistency



- Eventual consistency often used with replication
  - Client writes new versions to available nodes
  - Versions sync to other replicas lazily
- Application responsible for consistency
  - Already true in filesystems
- Allows partition tolerant systems



# Dispersal Requires Consistency



## ❑ Dispersal doesn't store replicas

- ❑ Threshold of slices required to recover data
- ❑ Crash during “unsafe” periods can cause loss

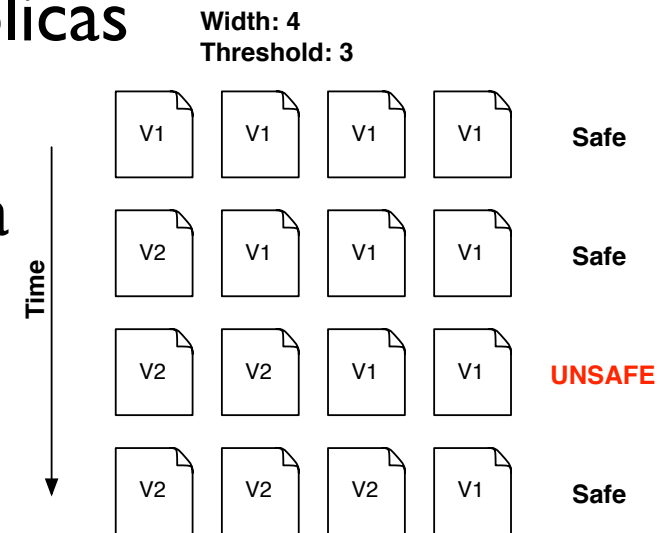
## ❑ Methods to prevent loss

### ❑ Three-phase distributed transaction

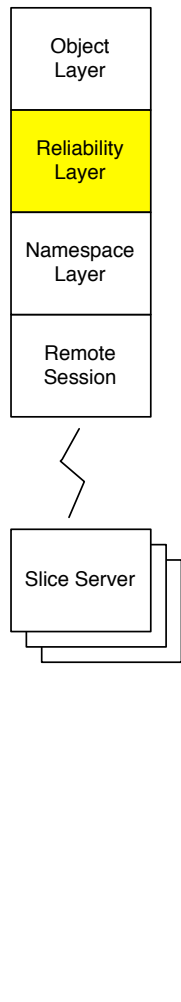
- ❑ Commit: All revisions visible during unsafe period
- ❑ Finalize: Cleanup when new version commit safe

### ❑ Quorum-based voting

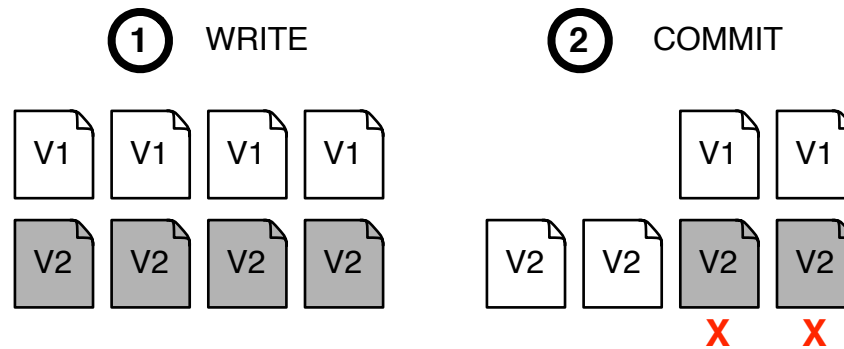
- ❑ Writes fail if  $<T$  successful



# Three-Phase Commit Protocol



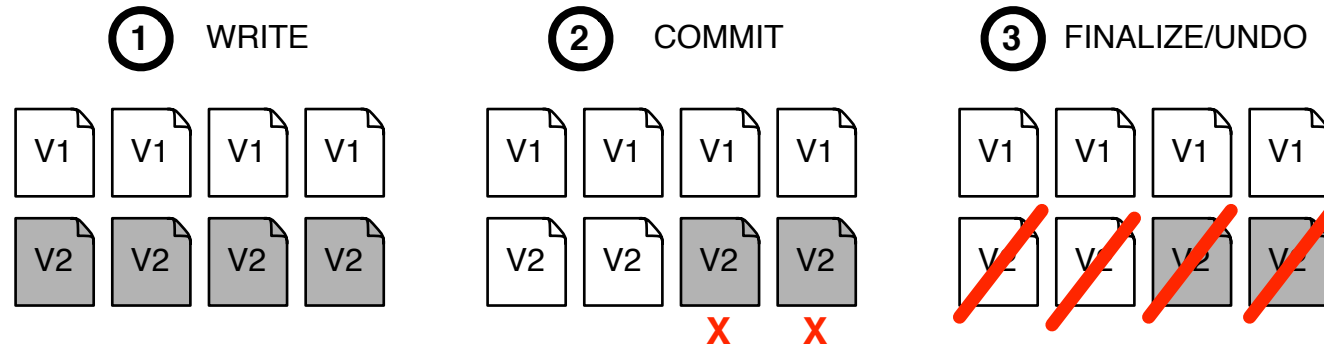
## 2-Phase Commit Protocol



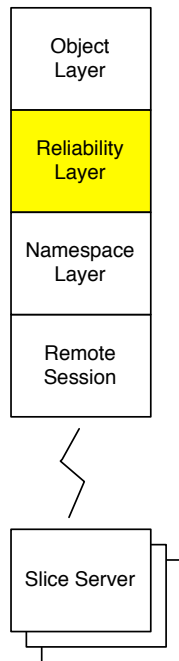
Width: 4  
Threshold: 3

**Commit Failure Causes Loss!**

## 3-Phase Commit Protocol

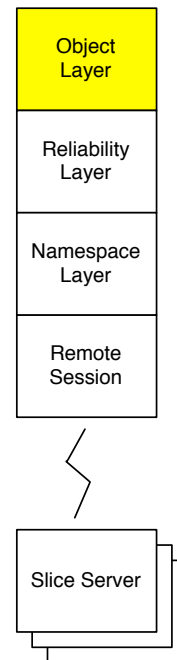


# Consistent Transactional Interface

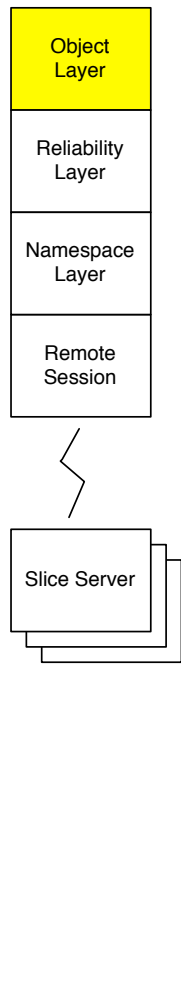


- ❑ Distributed transaction makes dispersal safe
  - ❑ All happens in client, no server coordination
- ❑ Write consistency
  - ❑ Side-effect of distributed transactions
  - ❑ Writes either succeed or fail “atomically”
- ❑ Limitation: Consistency = less partition tolerance
  - ❑ CAP Theorem (we also choose availability)
  - ❑ Either read or write fails during partition
    - ❑ Still “shardable”: affects availability, not scalability
- ❑ **Is consistency useful for filesystem directories?**

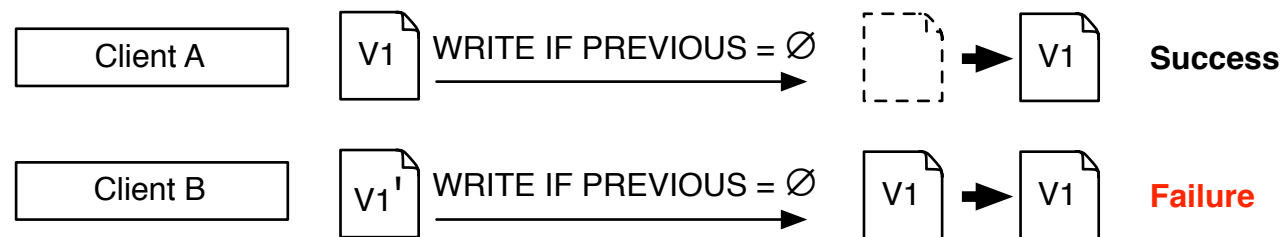
# Object Layer



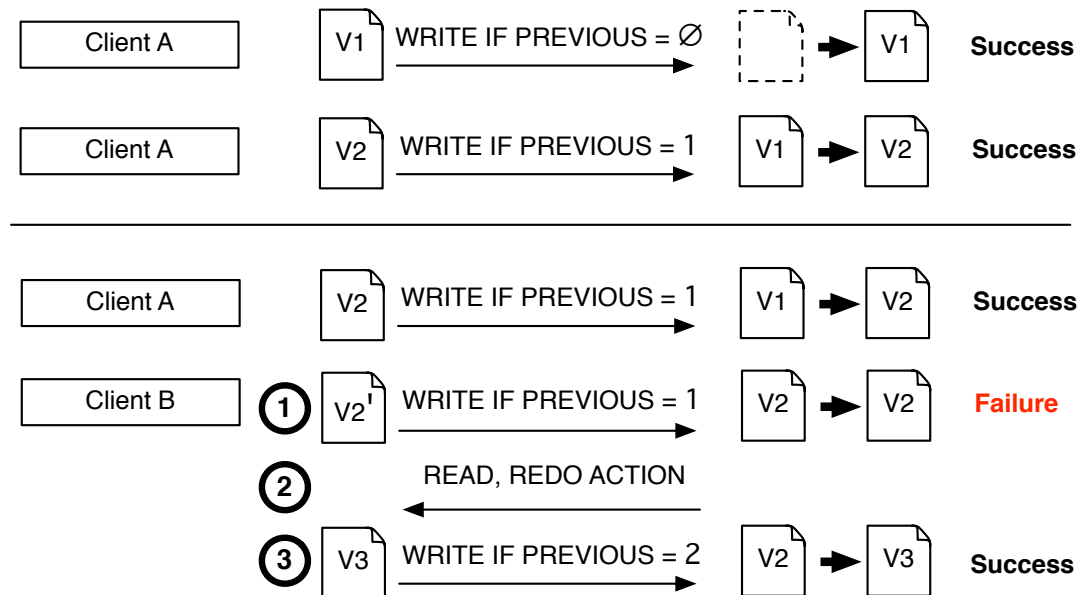
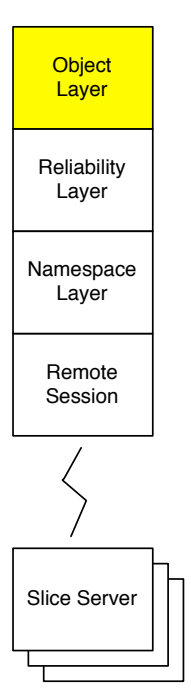
# Write-if-absent for WORM



- ❑ Object storage is WORM
  - ❑ Enforced by underlying storage
  - ❑ Write-if-absent model built on transactions
    - ❑ Distributed transactions emulate atomicity
    - ❑ “Checked write” fails if previous revision exists

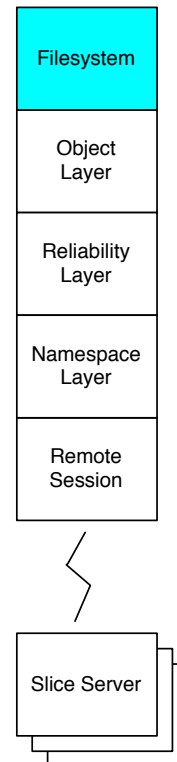


# Optimistic Concurrency Control

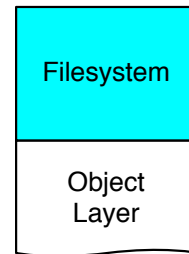


- Easy to extend this model to multiple revisions
  - Write succeeds IFF last revision matches given
  - Basis for “optimistic concurrency”
- How do concurrent writers update a directory?

# Filesystem Layer



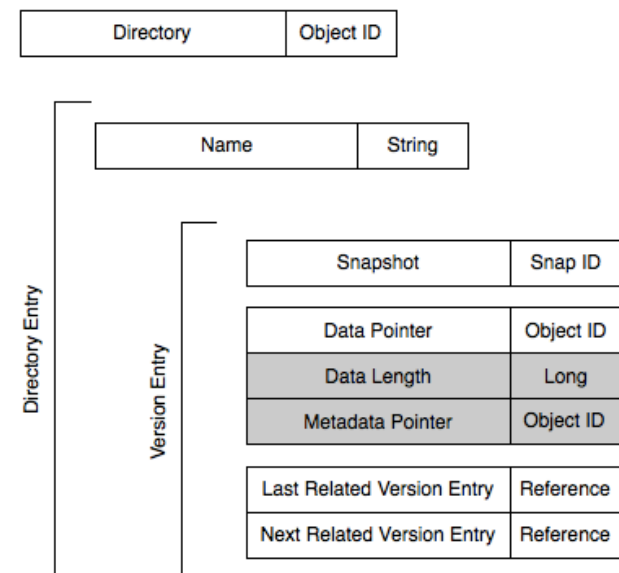
# Ultra-Scalable Filesystem Technology



- ❑ Filesystem layer on top of object storage
  - ❑ Scalable no-master storage
  - ❑ Inherits reliability, security, and performance
- ❑ How do we map file name to object id?
- ❑ Is consistency useful for filesystem directories?
- ❑ How do concurrent writers update a directory?

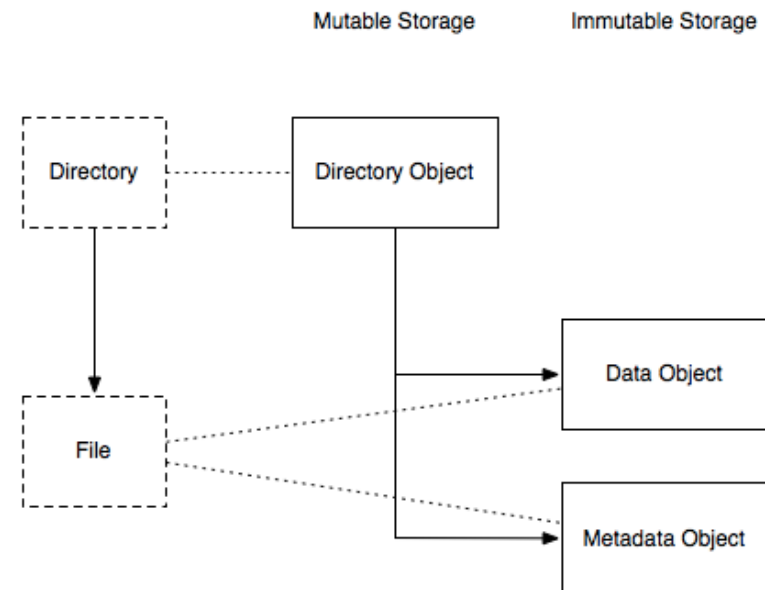
# Object-based directory tree

- ❑ How do we map file name to object id?
- ❑ Directories stored as objects
  - ❑ *Filesystem structure as reliable as data*
- ❑ Directory content data is map of file name to object id
  - ❑ Object id points to another object on system
  - ❑ Id for content data
  - ❑ Id for metadata (xattr, etc)
  - ❑ Data objects WORM
    - ❑ Zero-copy snapshot support
      - ❑ Reference counting
- ❑ Well known object id for “root”



# Directory Internal Consistency

- ❑ Is consistency useful for filesystem directories?
- ❑ Object layer allows “atomic” directory updates
  - ❑ This mimics model used by traditional filesystems
- ❑ Content data stored in separate “immutable” storage
  - ❑ Safe snapshot support
- ❑ Eventual consistency
  - ❑ Temporary effects
  - ❑ Writes: Orphaned data
  - ❑ Deletes: Read error
- ❑ Absolute requirement? No.



# Concurrency Requires Serialization

- ❑ **How do concurrent writers update a directory?**
- ❑ Updates to directory entries are atomic (definition)
  - ❑ More precisely, filesystem operations are serialized
    - ❑ Client A adds file, Client B adds file, Client C deletes file
    - ❑ First to call wins, application must have sane order
- ❑ Kernels use mutexes (locks) for serialization
  - ❑ Master controller (pNFS, GoogleFS) does this
  - ❑ We want to use “multiple/no master” model
- ❑ Distributed locking protocols exist (e.g., PAXOS)
  - ❑ It's hard: Protocols complex and have drawbacks
  - ❑ It's slow: Overhead for every operation

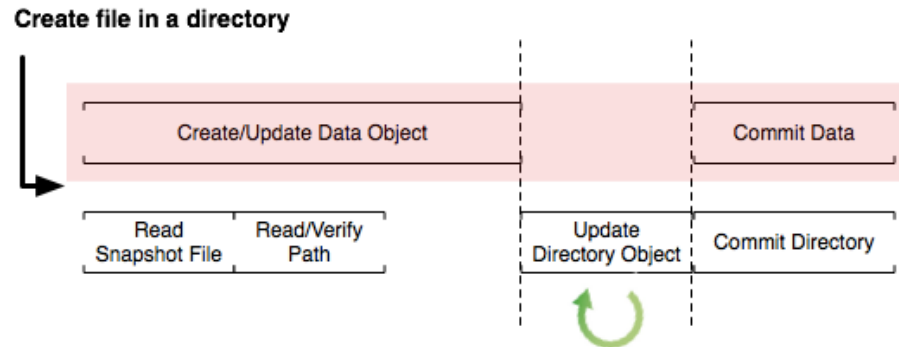
# Optimistic Concurrency

- ❑ ***We want to serialize without locking***
- ❑ Observation: File writes have two steps
  - ❑ Write the data (long, no contention)\*
  - ❑ Modify the directory (short, serialized)\*\*
- ❑ Use checked writes for directory
  - ❑ Always read directory before writes
  - ❑ Write new revision “if-not-modified-since”
  - ❑ On “write conflict”, re-read, replay, repeat

\* Consider workload where files > 1 MiB, we write content data in WORM storage

\*\* Because directories stored as objects themselves, modifying directory is re-writing object

# Lockless Directory Update



- ❑ Optimistic concurrency guarantees serialization
  - ❑ Operation is simple (“add file”), so replay trivial
  - ❑ On conflict, operation replay semantics are clear
    - ❑ Content data (large) is **not** rewritten on conflict
  - ❑ Highly parallelizable
- ❑ Potentially unbounded contention latency
  - ❑ Back-off protocol can help
  - ❑ Not good for high directory contention use cases

# Conclusions

---

- ❑ Advantages
- ❑ Limitations
- ❑ Final Thoughts

# Advantages

- ❑ Scalability and Performance
  - ❑ Content data I/O quick and contention free
  - ❑ No-master concurrent read and write
  - ❑ Linearly scalable performance
- ❑ Availability
  - ❑ Load balancing without complicated HA setups
- ❑ Reliability
  - ❑ Information dispersal
  - ❑ Both data and metadata have same reliability
  - ❑ No separate “backup” required for index server

# Limitations

- ❑ Optimistic concurrency sensitive to high contention
- ❑ Cache requirements limit directory size
  - ❑ No intrinsic limit, but a 100MiB directory object?
- ❑ No central master makes explicit file locking hard
  - ❑ SMB, NFS protocols support these
- ❑ Not suitable for random-write workloads
- ❑ Not suitable for majority small file workloads
  - ❑ Directory write times eclipse file write times
- ❑ Requires separate index service for search

# Final Thoughts

- ❑ Significant advances come from P2P and NOSQL space
  - ❑ Three key techniques allow for ultra-scalable FS
    - ❑ Namespace-based routing
    - ❑ Distributed transactions using quorum/3-phase commit
    - ❑ Optimistic concurrency using checked write
  - ❑ Techniques useable with IDA or replicated systems
- ❑ Filesystem would not be general purpose
  - ❑ Techniques have some trade-offs
  - ❑ Excellent for specific big data use cases

Questions?