Ceph: A Scalable, High-Performance Distributed File System

Darrell Long
University of California, Santa Cruz

STORAGE SYSTEMS RESEARCH CENTER

Baskin Engineering
UC SANTA CRUZ

Project Goals

- *Reliable*, *high-performance* distributed file system with *excellent scalability*
- Massive scale
  - Petabytes to exabytes (10^{15}–10^{18})
  - Multi-terabyte files
  - Billions of files
  - Tens or hundreds of thousands of clients simultaneously accessing same files or directories
- Reliable
  - Failure of a component is the norm, not the exception
  - System must perform well despite failures
- POSIX interface (plus extended semantics)
  - Support security
  - Support quotas

Ceph: Key Design Decisions

1. Maximal separation of data and metadata
   - Object-based storage
   - Independent metadata management
   - CRUSH – data distribution *function*
2. Dynamic metadata management
   - Adaptive and scalable
3. Intelligent disks
   - Reliable Autonomic Distributed Object Store
4. Scalable security
5. Scalable quota management
6. High-performance local disk file system

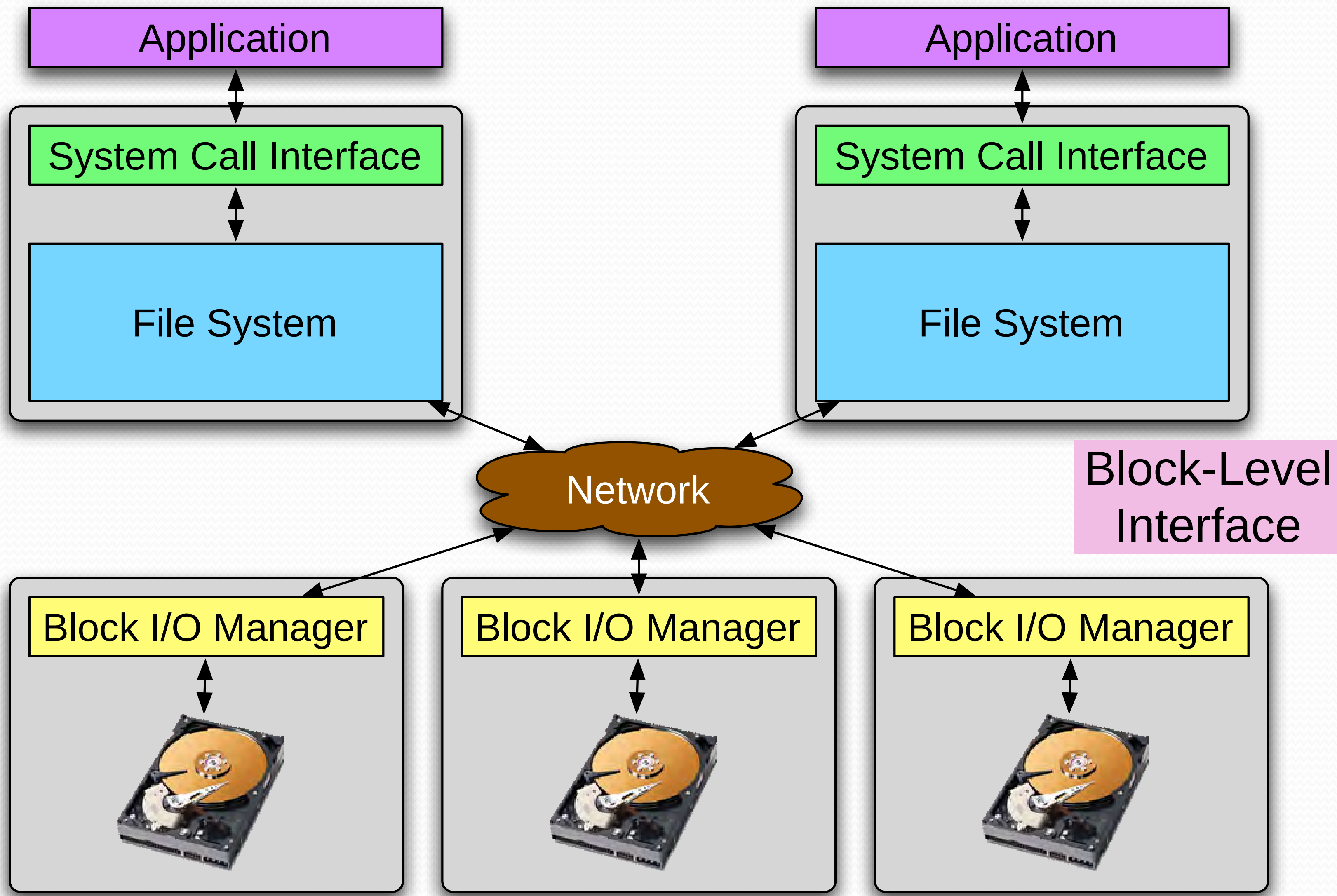
Outline

1. Maximal separation of data and metadata
   - Object-based storage
   - Independent metadata management
   - CRUSH – data distribution function
2. Dynamic metadata management
   - Adaptive and scalable
3. Intelligent disks
   - Reliable Autonomic Distributed Object Store
4. Scalable security
5. Scalable quota management
6. High-performance local disk file system

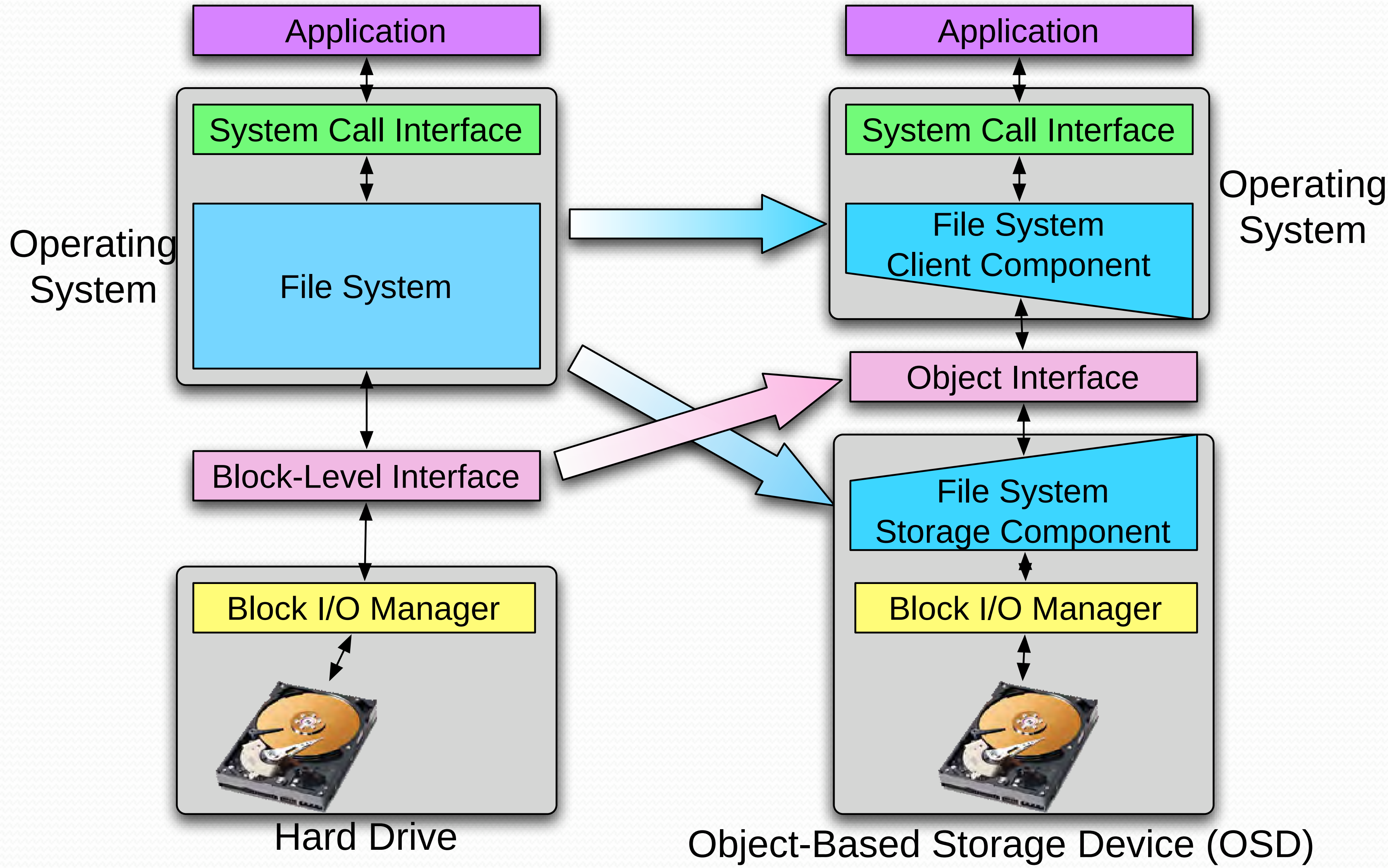
High Performance Workload Characteristics

- High performance, scientific applications
  - Very often have bursty access patterns
    - Write I/Os and opens most commonly come in bursts
  - A set of nodes tend use shared data files
    - I/Os tend to be done on a small set of files containing computation results
    - I/Os tend to result from a small set of nodes doing computations
  - Based on workload analysis from MSST 2004 paper
- Hot files and flash crowds are very common
- File accesses are not random
  - Very dependent on applications and users

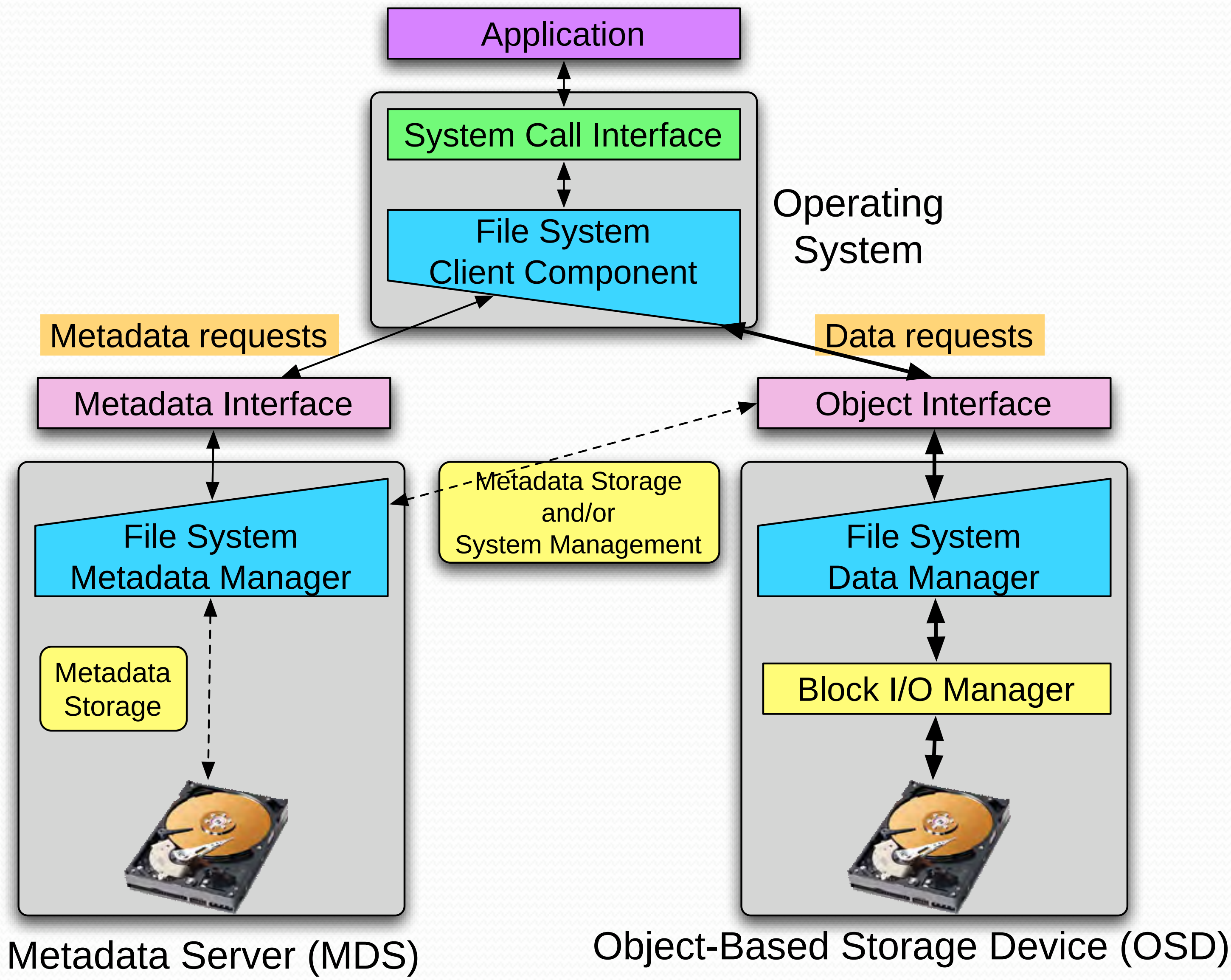
The Way Things Are



Object-Based Storage Paradigm

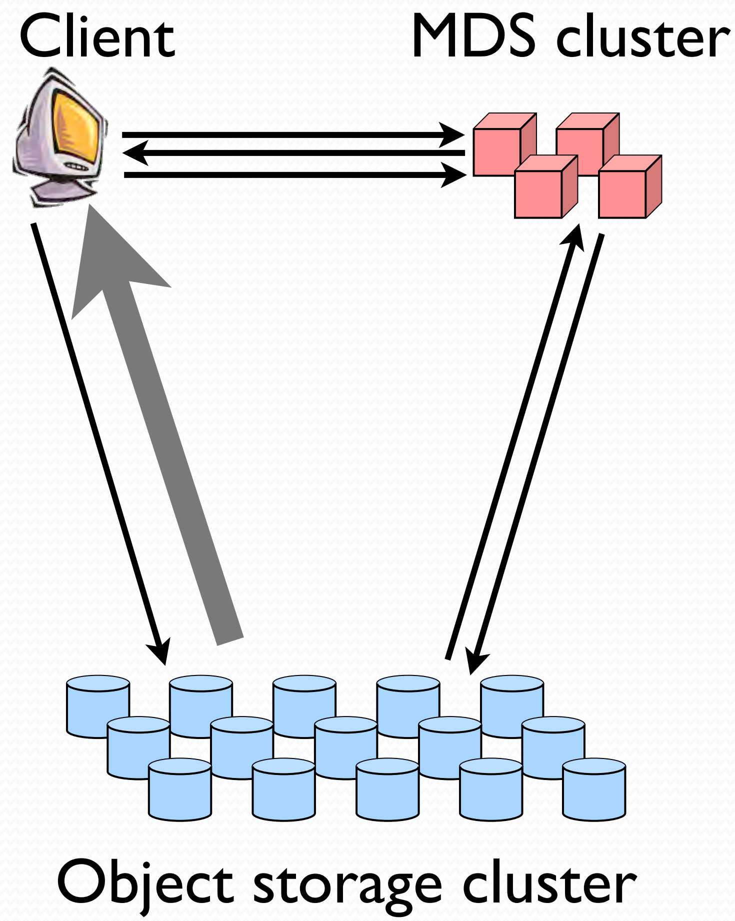


Separate Management of Data and Metadata



Ceph: A Simple Example

- fd=open("/foo/bar",O_RDONLY);
  - Client: requests open from MDS
  - MDS: reads directory "/foo" from OSDs
  - MDS: issues "capability" for "/foo/bar"
- read(fd,buf,10000);
  - Client: calculates name(s) and location(s) of data object(s)
  - Client: reads data from OSDs
- close(fd);
  - Client: relinquishes capability to MDS
- MDS stays out of I/O path
- Client doesn't need to look up the location of file data



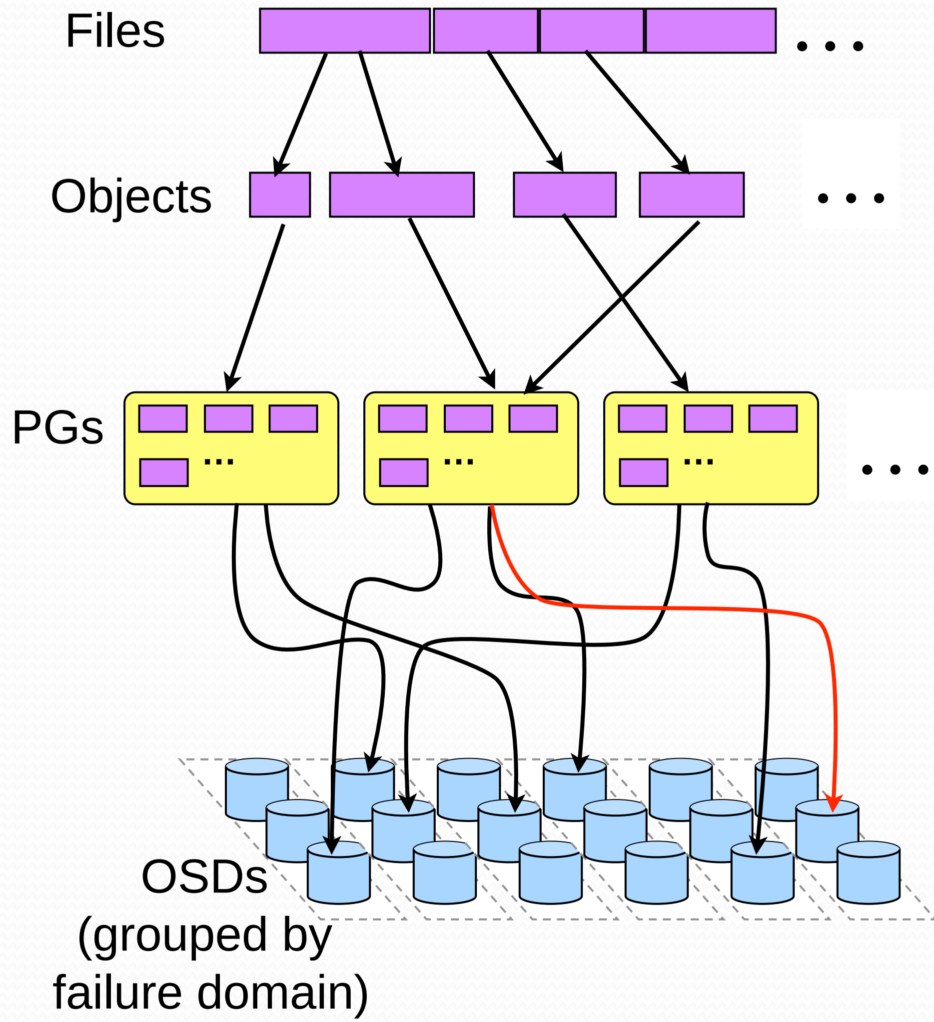
CRUSH—Simplifying Metadata

- Conventionally
  - Directory contents (file names)
  - File i-nodes
    - Ownership, permissions
    - File size
    - Block list

- CRUSH
  - Small "map" completely specifies data distribution
  - Eliminates allocation lists
  - I-nodes "collapse" back into small, almost fixed-sized structures
    - Embed i-nodes into directories that contain them
    - No more large, cumbersome i-node tables

CRUSH— Distributing data with a function

- Files striped across many objects
  - Striping strategy specified in inode
  - *object_id* = <*inode_num*, *object_num*>
- Objects mapped to placement groups (PGs): *pg_id* = hash(*object_id*) MOD *number_of_pgs*
- CRUSH maps PGs to OSDs
  - Pseudo-random distribution
  - Statistically uniform
  - Replicated on multiple OSDs
- CRUSH is…
  - Calculable everywhere (no explicit tables)
  - Stable: adding/removing OSDs moves few objects
  - Reliable: replicas span failure domains
  - ...everything you'd normally want to do using conventional allocation tables!



Cluster Map: Placement Rules

- Cluster map
  - CRUSH views OSDs as hierarchy of devices and buckets
    - OSDs are leaves in the hierarchy: weights set by an administrator
    - Buckets contain OSDs or other buckets: bucket weights are the sum of their contents
  - Hierarchy reflects underlying storage organization in terms of physical placement or infrastructure
    - rooms in a data center ➟ rows of cabinets ➟ cabinets of shelves ➟ shelves of disks
- Placement rules
  - Minimal command set to define object replica placement behavior
    - take: choose an initial bucket
    - select: select one or more elements from all "current" buckets
    - emit: use the "current elements" as locations for data
  - Different rules for each redundancy policy
    - Mirror-2, Mirror-3, RAID10, etc.

Placement Rules – Collisions

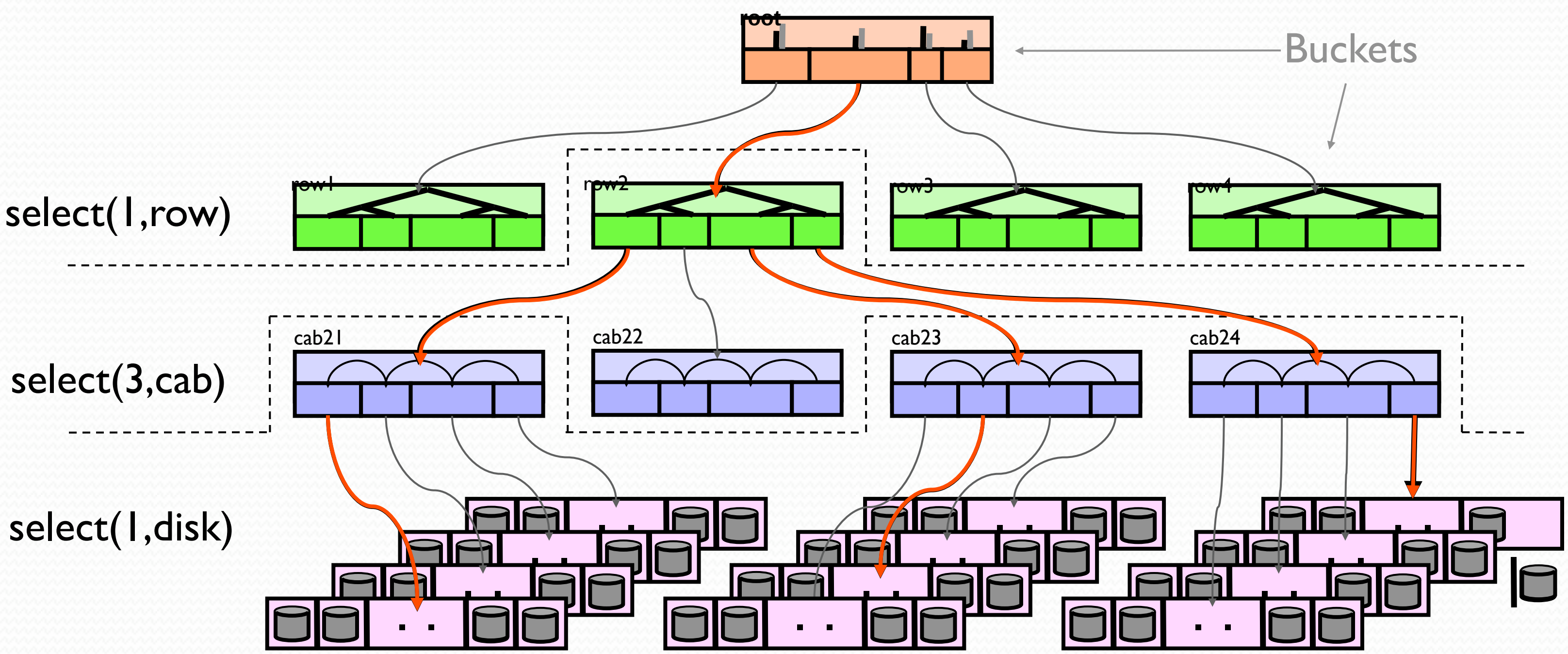
- **select**(*n*,*t*) may reject items for three reasons
  - A disk is on the “failed” list
    - We leave individually failed disks “in place” but selectively relocate their contents
  - A disk is overloaded
    - We can selectively offload any fraction of items off a disk that is overloaded
  - An item has already been chosen for the current set
    - We want distinct targets; duplicates aren’t allowed
- In all cases, CRUSH backtracks up its recursive tree descent to re-select a different item

Placement Rules – Data Safety

- CRUSH can separate redundancy information across different failure domains
  - If cabinets have their own power, each replica is placed under a different power circuit
  - Physical separation reduces vulnerability to heat or other physical disturbances
- Especially important for systems that *decluster* replication
  - (*i.e.* replicas on one OSD are scattered across many other OSDs)
  - Consider 2-way replication – if one OSD fails, the “backup” replicas may be spread across many other devices
    - Faster rebuild: replicas copied in parallel to lots of other disks
    - Larger set of disks whose subsequent failure will cause data loss
  - Separation across failure domains reduces probability of coincident failures causing data loss

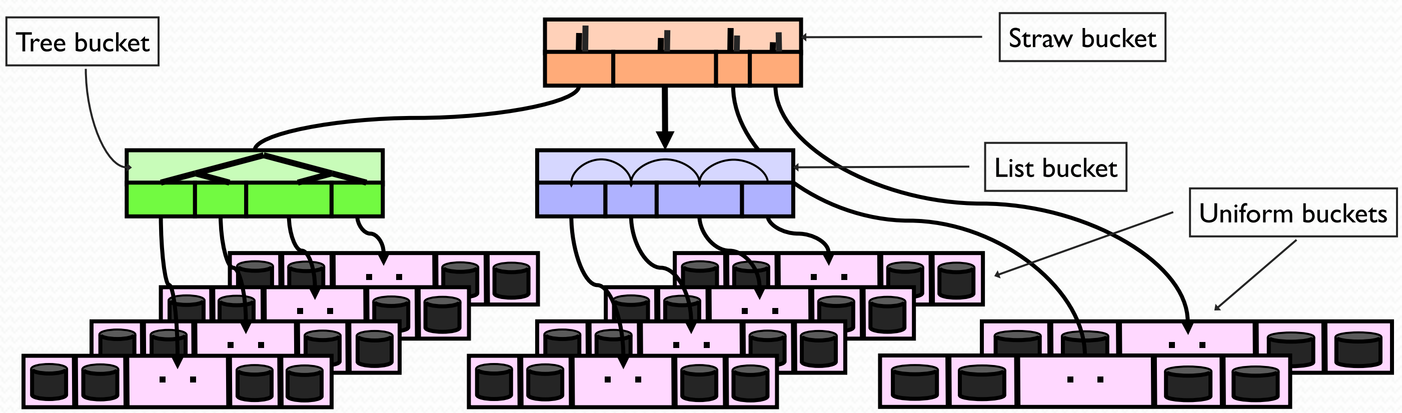
Sample Rule: Mirror-3

Command	Working Value
take(root)	root
select(1,row)	row2
select(3,cabinet)	cab21 cab23 cab24
select(1,disk)	disk2107 disk2313 disk2437
emit	

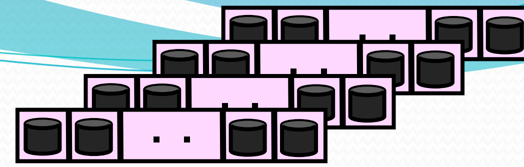


Cluster Map – Buckets

- CRUSH defines four kinds of buckets
  - Uniform, List, Tree, Straw
- Each type represents different tradeoff between
  - Computational complexity: difficulty in calculating object location
  - Data reorganization efficiency: I/O required to reshuffle when something changes

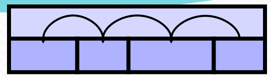


Uniform Buckets

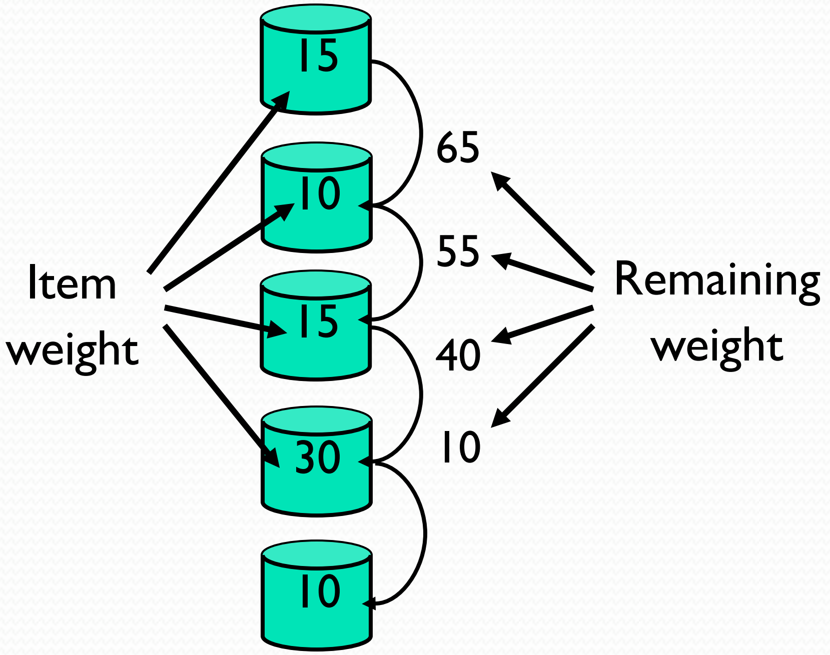


- Arrays of *equally weighted* devices
  - In a large system, storage is typically added and decommissioned as sets of disks
- CRUSH selects storage targets using modular arithmetic
  - $f(x,r) = (hash(x,r)+rp) \bmod n$
    - p is a pseudo-random prime number greater than n
  - For $r \leq n$ this describes a random permutation of $[0, n\text{-}1]$
- Computation: fast, O(1)
  - Chooses an item in constant time, regardless of bucket size
- Reorganization: very poor
  - Complete reshuffle if bucket changes: unnecessary I/O

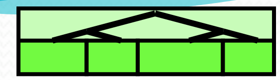
List Buckets



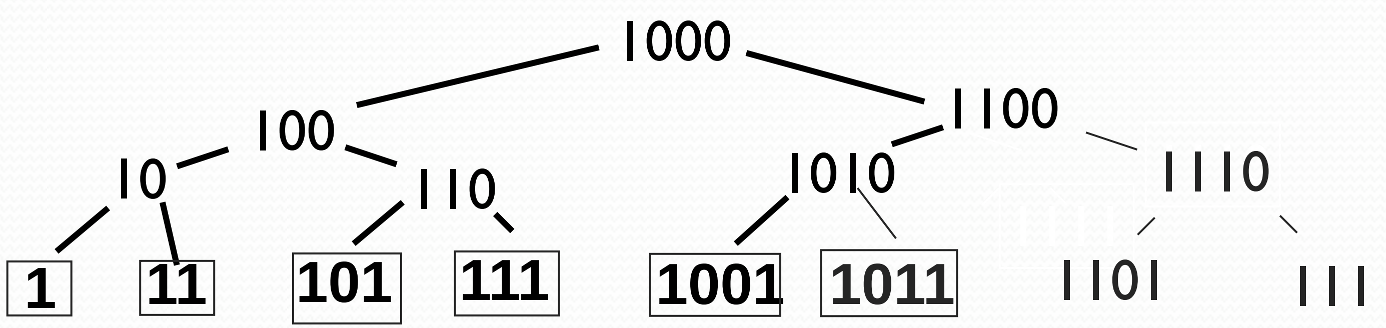
- How is relocation done?
  - Start at head of list
  - Select current item with appropriate probability, or continue down the list
    - Current item's weight vs. remaining weight
- Items are arbitrarily weighted
- We can add new items to the front
  - Subset of existing data shifts to new device
- Computation: Slow, O(*n*)
  - On average we traverse half the list to choose an item
- Reorganization: Optimal additions, poor removals



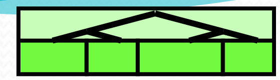
Tree Buckets



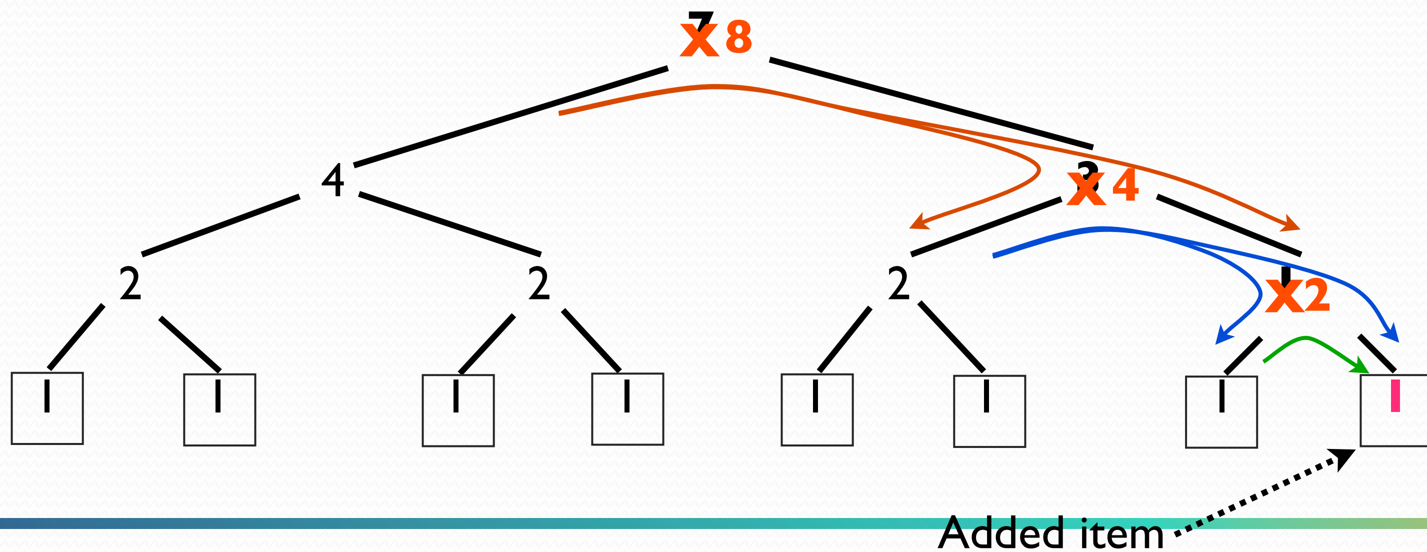
- Weighted binary decision tree
  - Bucket items at the leaves
  - Select left or right branch proportional to subtree weights
    - based on hash(x, label)
- Tree nodes are labeled using a specific strategy
  - Only add new nodes at the root
- Computation: Very fast, O(log *n*)
- Reorganization: Good
  - Worst case proportional to tree depth



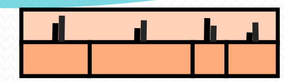
Data Movement in a Tree Bucket



- Data is rebalanced as we move down the tree
  - Data moved into a subtree is uniformly distributed
    - may not get placed where the weight increased
    - ➡ Weight change may result in some unnecessary data movement
- We see this effect
  - Within the binary tree internal to a tree bucket node
  - In the overall cluster map hierarchy

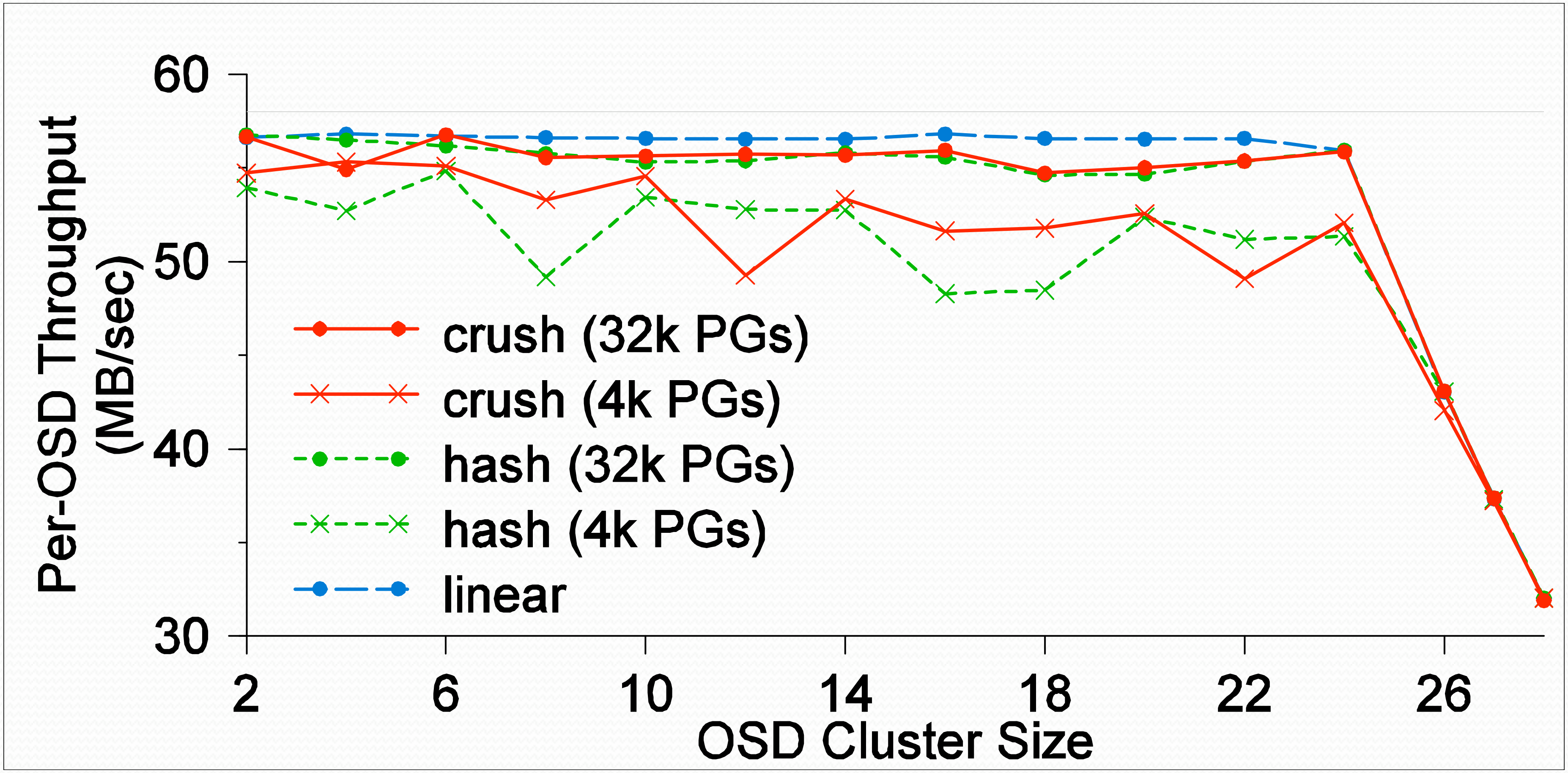


Straw Buckets



- All items "compete" against each other for objects
  - Draw a straw of random length for each item
    - Based on hash(*x*, *item*)
    - Length is adjusted based on item weight
  - Bucket item with longest straw "wins" object
- Computation: slower, O(*n*)
  - We calculate a hash value for every bucket item, every time
- Reorganization: optimal
  - Adjusting any item weight results only in data moving to or from that item

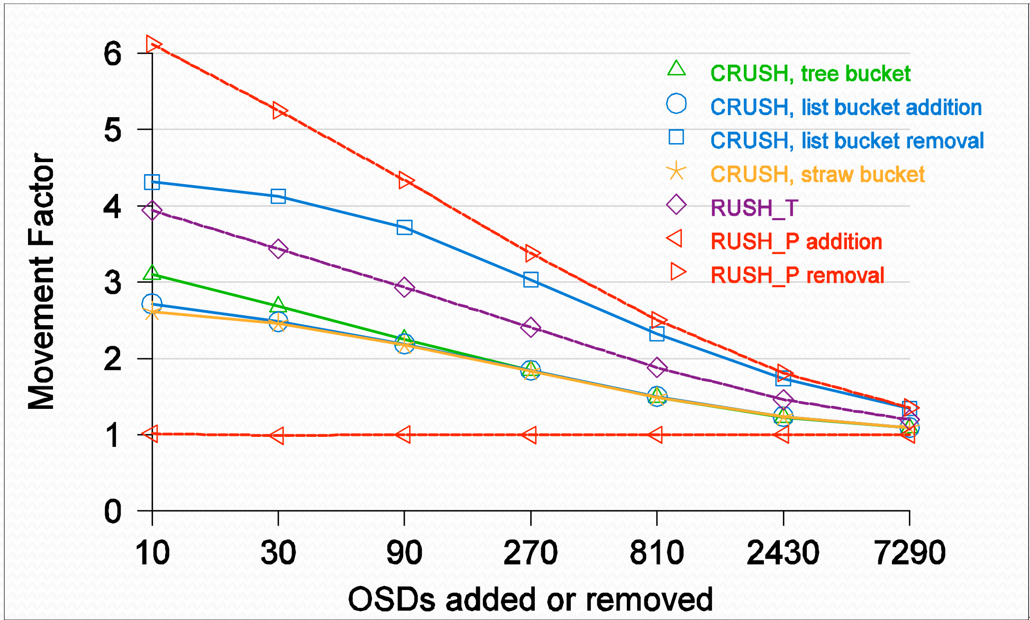
OSD Cluster Scaling—CRUSH vs. careful striping

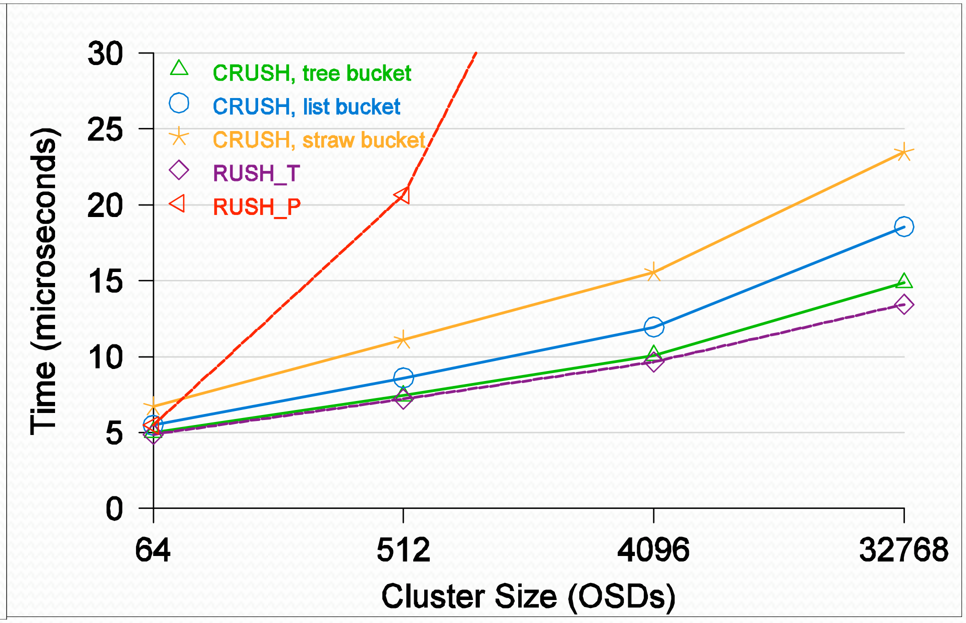


- Higher placement group count reduces statistical variance, divergence from optimal (write throughput shown)

Computation vs. Reorganization

- Trade off mapping computation and reorganization I/O
- Cluster map hierarchy
- Bucket types
- 4 level CRUSH hierarchy vs. 2-level RUSH mapping





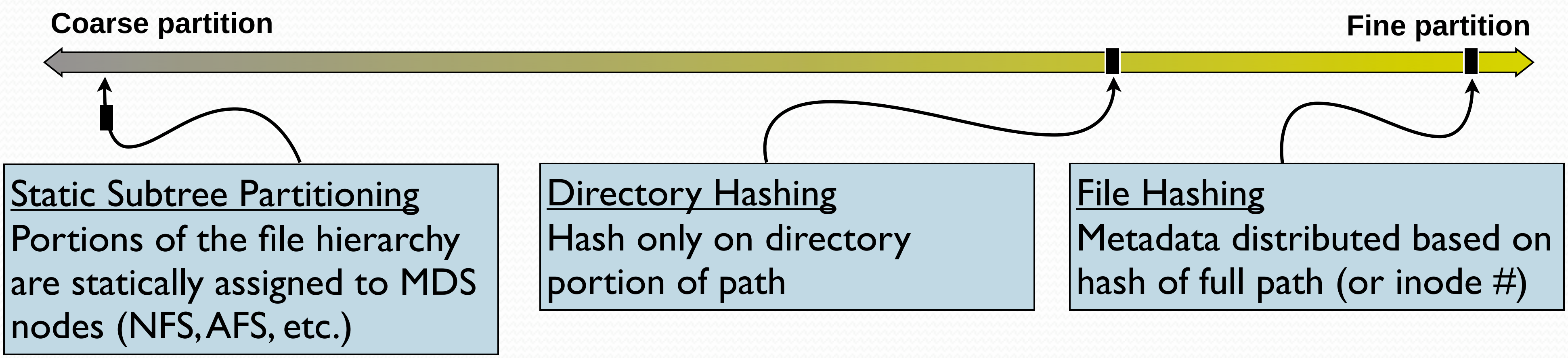
CRUSH – Summary

- CRUSH distributes data randomly with respect to device weights
  - Includes overload mechanism to cope with statistical variance inherent in any "random" process
- Flexible placement rules – variety of redundancy schemes
- Improved reliability by separating replicas across failure domains
- Fast and distributed
  - O(log *n*) performance, tens of microseconds
  - Minimal metadata – any party can calculate data locations
- Minimizes data movement when storage cluster changes
  - Balance reorganization I/O vs. mapping computation

Outline

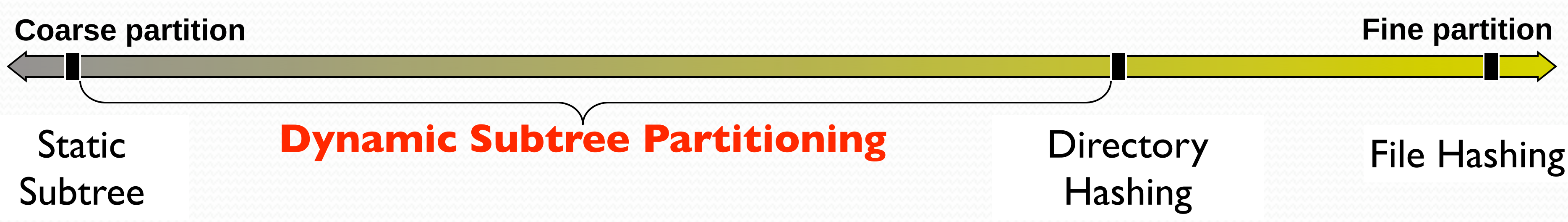
1. Maximal separation of data and metadata
   - Object-based storage
   - Independent metadata management
   - CRUSH – data distribution function
2. Dynamic metadata management
   - Adaptive and scalable
3. Intelligent disks
   - Reliable Autonomic Distributed Object Store
4. Scalable security
5. Scalable quota management
6. High-performance local disk file system

Metadata—Traditional Partitioning



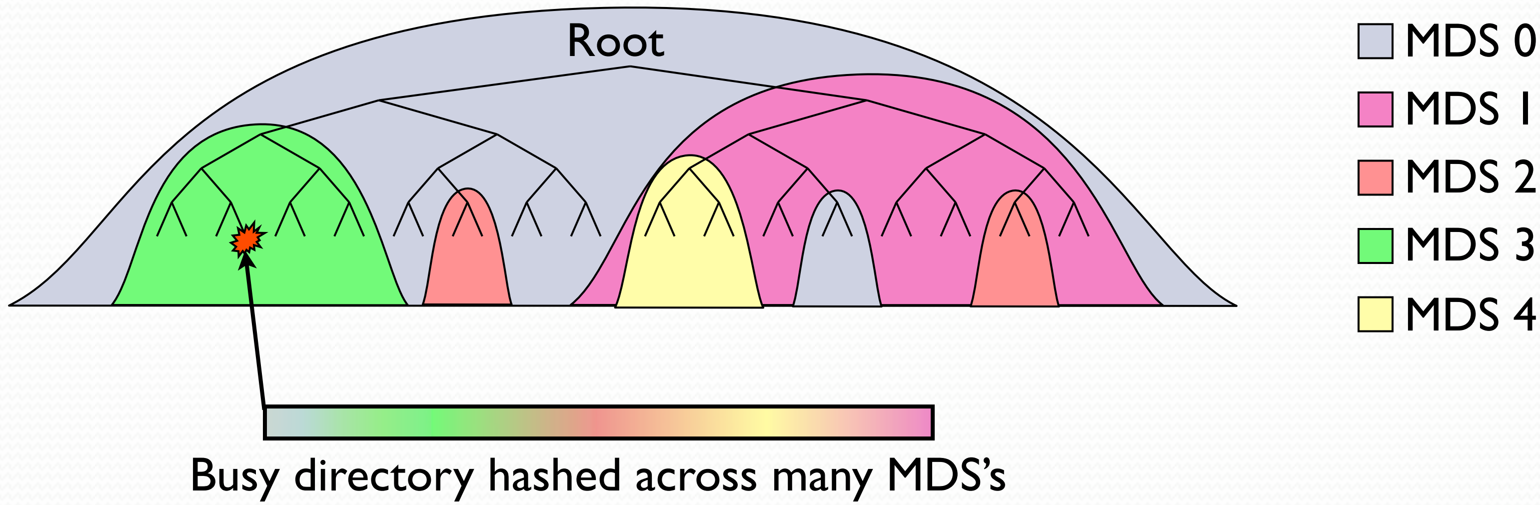
- Coarse distribution (static subtree partitioning)
  - Hierarchical partition preserves locality
  - High management overhead: distribution becomes imbalanced as file system, workload change
- Finer distribution (hash-based partitioning)
  - Probabilistically less vulnerable to “hot spots,” workload change
  - Destroys locality (ignores underlying hierarchical structure)

Ceph's Dynamic Partitioning



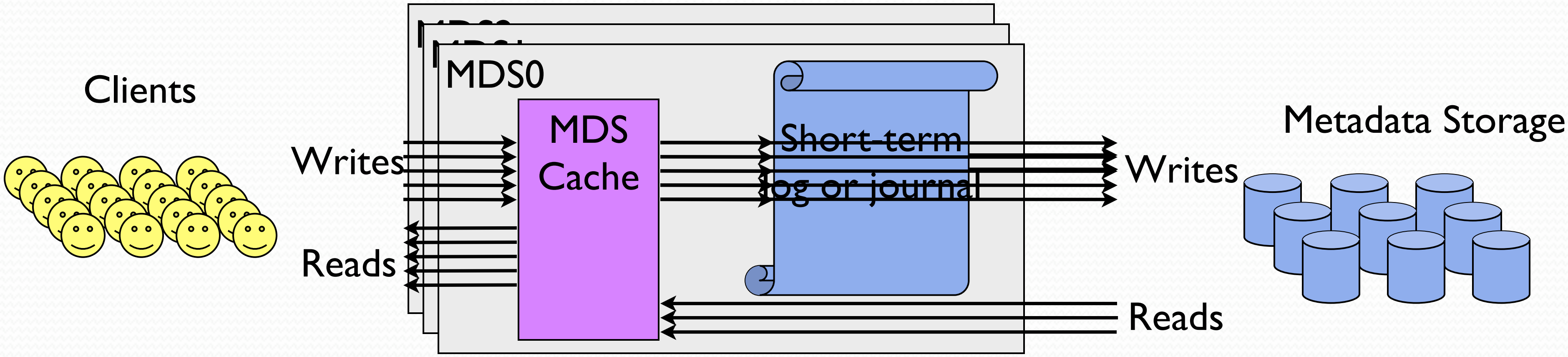
- Ceph dynamically distributes arbitrary subtrees of the hierarchy
  - Coarse partition preserves locality
  - Adapt distribution to keep workload balanced
    - Migrate subtrees between MDSs as workload changes
- Adapt distribution to cope with hot spots
  - Heavily read directories replicated on multiple MDSs
  - Heavily written directories individually hashed across multiple nodes

Metadata Partition



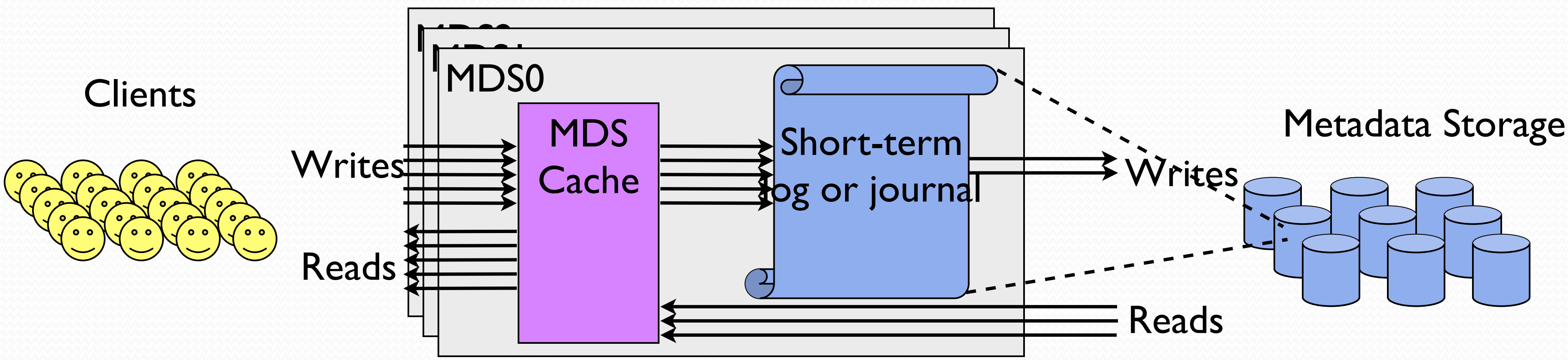
- Scalability
  - Arbitrarily partitioned metadata
- Adaptability
  - Cope with workload changes over time, and hot spots

Metadata Storage



- Consider MDS cluster as an intelligent metadata cache
- MDS cluster must serve both read and update transactions
- MDS cache absorbs some fraction of read requests
- All updates immediately committed to stable storage for safety
  - …but most metadata is updated multiple times in a short period!
- Short-term log absorbs multiple updates: flushed (very) lazily
  - Obsolete updates are discarded
  - Valid updates are applied to regular on-disk metadata structures

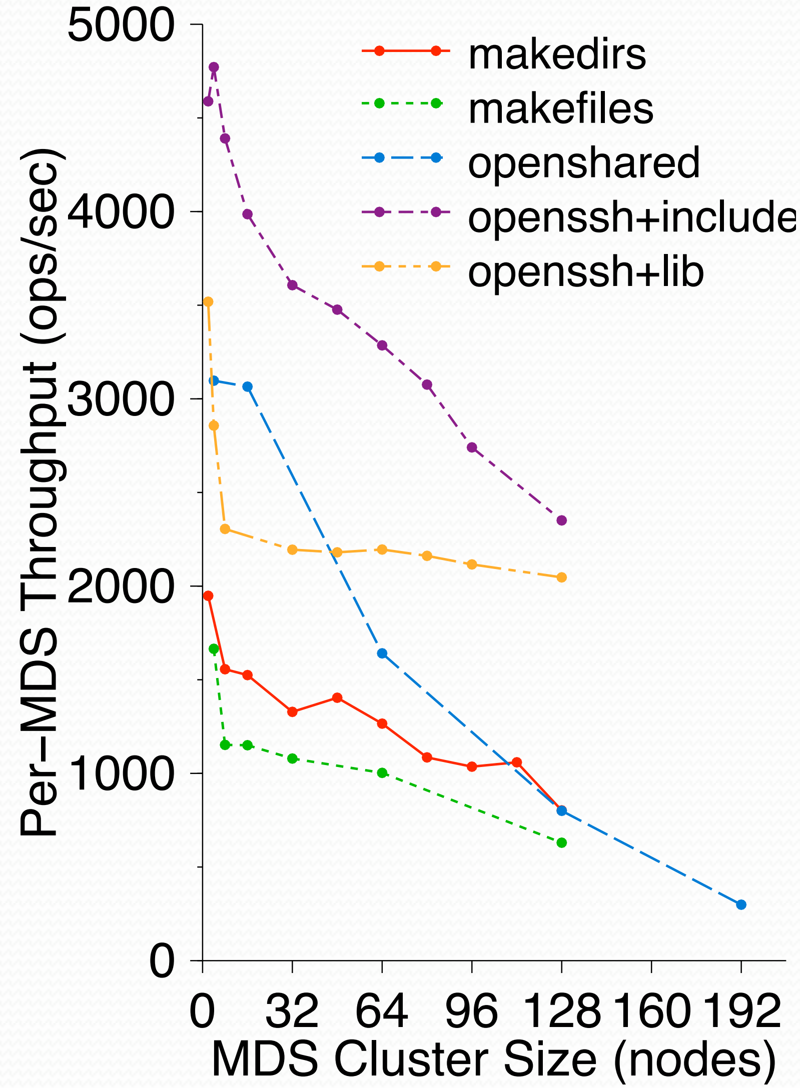
Metadata Storage—Two Tiers



- Short-term storage in metadata journal
  - Updates take advantage of **high sequential write bandwidth**
  - Absorb short-lived or repetitive metadata updates
  - Journal used for recovery after MDS failures
- Long-term storage
  - On-disk layout **optimized for future read access**
    - Group metadata by directory
    - Embed inodes—good locality without large, awkward inode tables

Metadata Server Performance

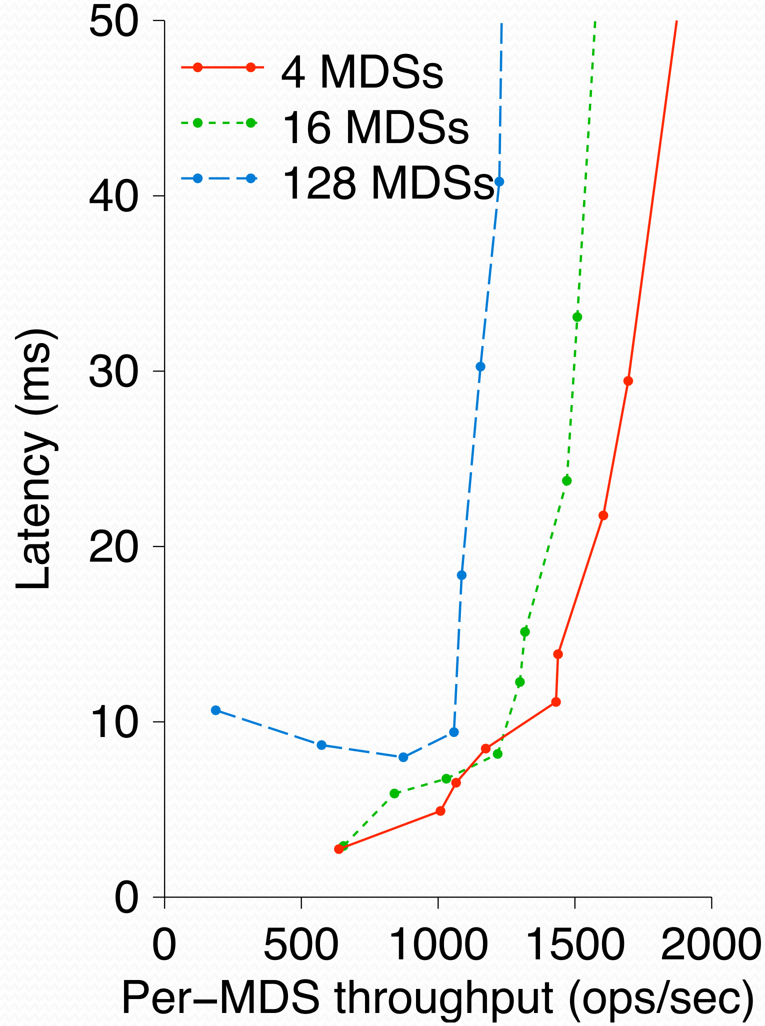
- Performance is pretty good on a single server
  - 1800–5000 ops/second
- Performance scales well
  - At 128 nodes, total performance is 320,000 ops/second!
- Scaling isn't linear, but not too far below
- Over 250,000 operations per second is very good!



Metadata Latency

- Metadata servers have relatively low latency until saturation
  - Workload: creating directories
  - Latency about 10ms with 128 servers at 1000 ops/sec each
- Result: 128,000 directory creates per second with 10 ms latency

➡ Metadata servers won't be the bottleneck for Ceph!

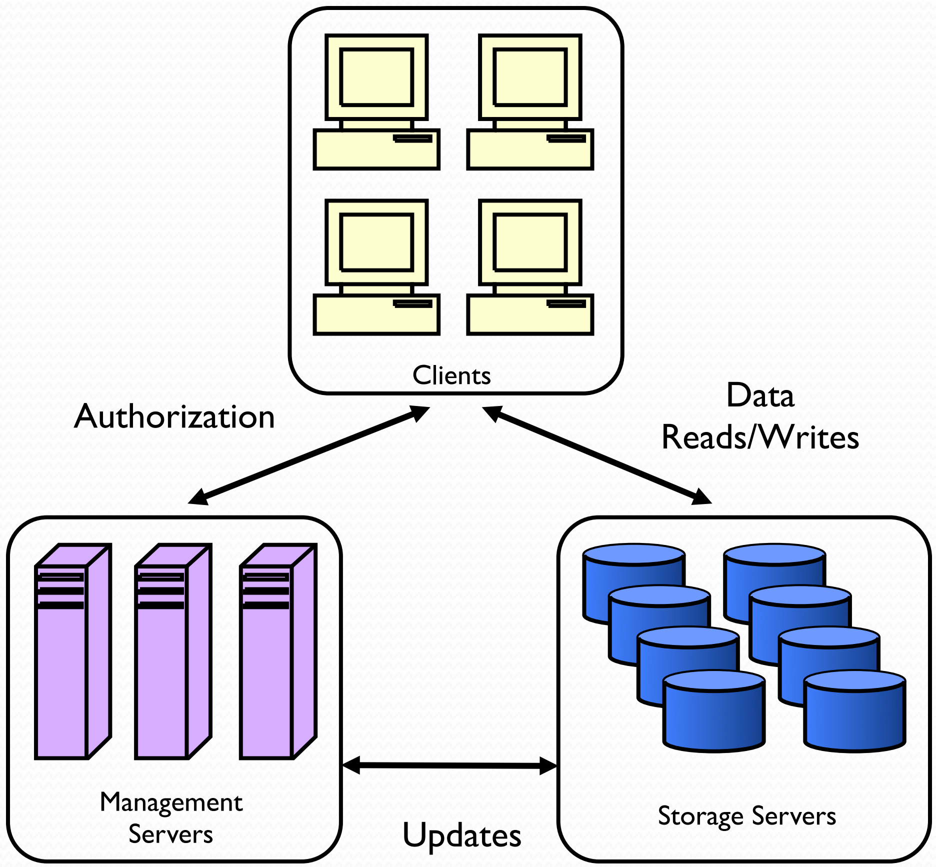


Outline

1. Maximal separation of data and metadata
   - Object-based storage
   - Independent metadata management
   - CRUSH – data distribution function
2. Dynamic metadata management
   - Adaptive and scalable
3. Intelligent disks
   - Reliable Autonomic Distributed Object Store
4. Scalable security
5. Scalable quota management
6. High-performance local disk file system

The Problem

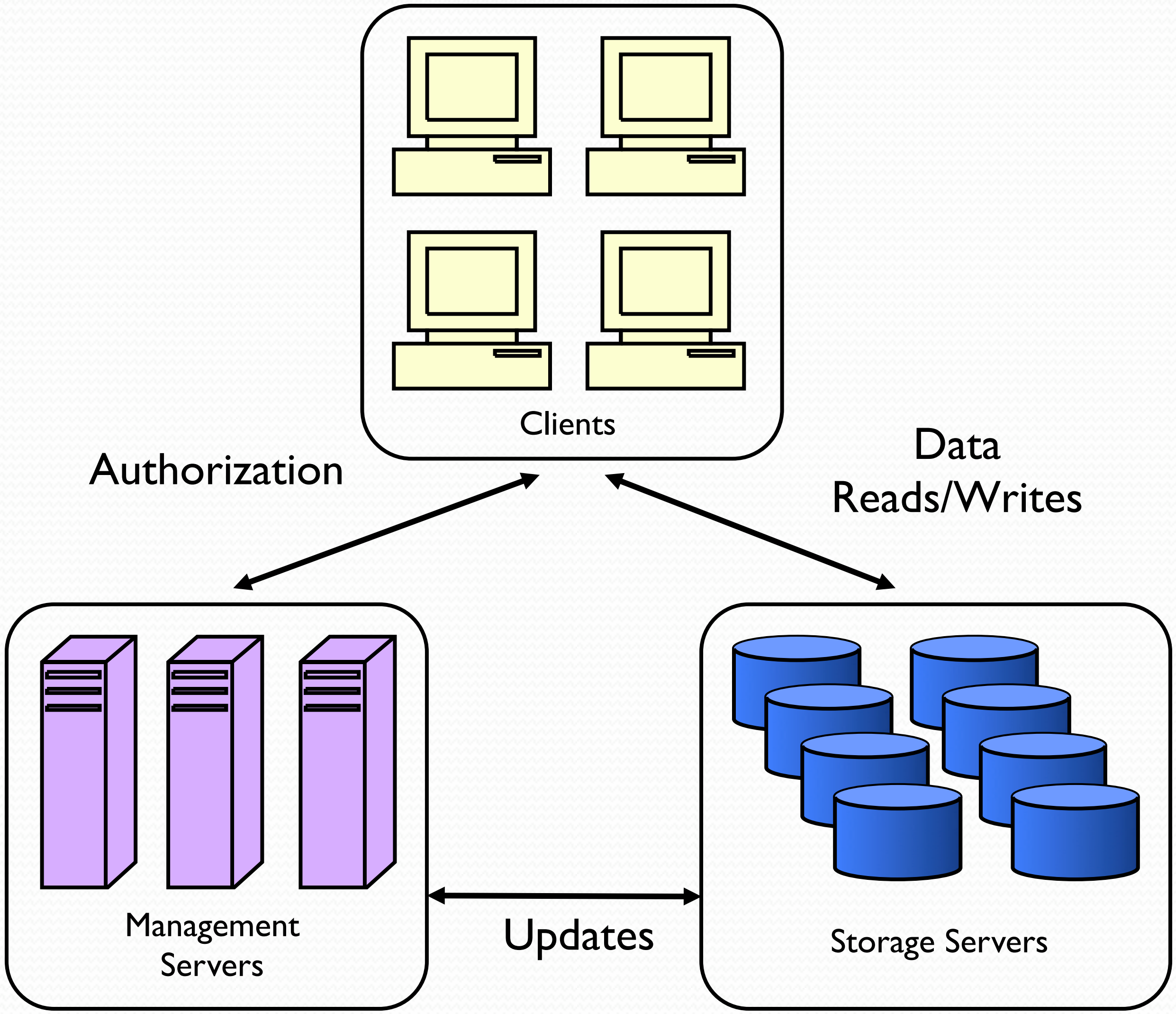
How do we track allocation and enforce quotas in a system where allocation decisions are made in a distributed fashion?



Our Approach

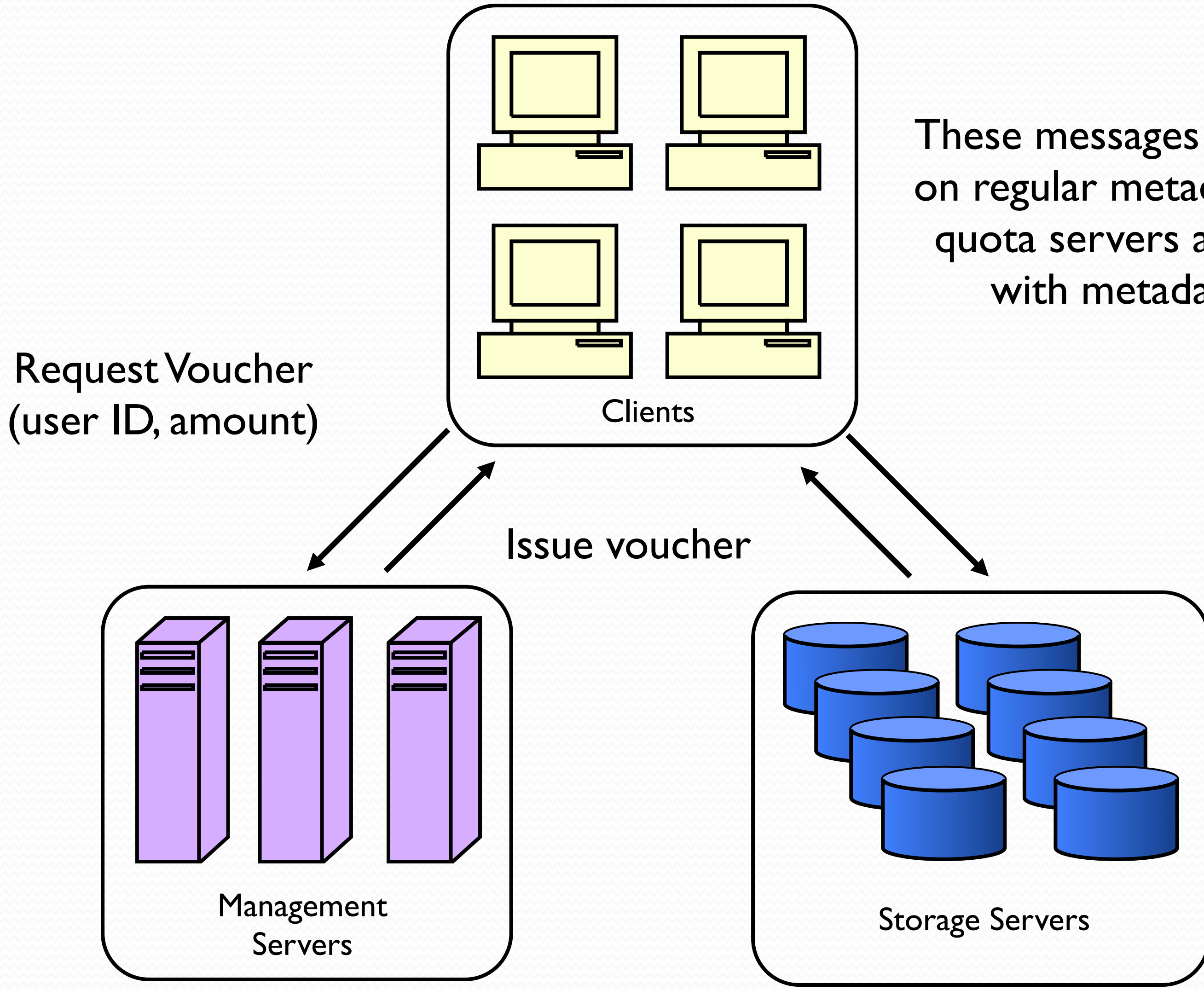
- Separate allocation and quota management using a monetary system model.
  - Quota management server acts as a bank.
    - Clients withdraw vouchers from the quota server for a user and store for later use.
    - Clients spend vouchers for users in order to purchase storage from storage servers.
    - Storage servers periodically update the quota server about user storage.
    - Cheaters are caught by the bank at defined intervals.
- Vouchers are cryptographically-protected byte sequences.
  - {epoch, expiry, user, amount, serial}auth

The Protocol

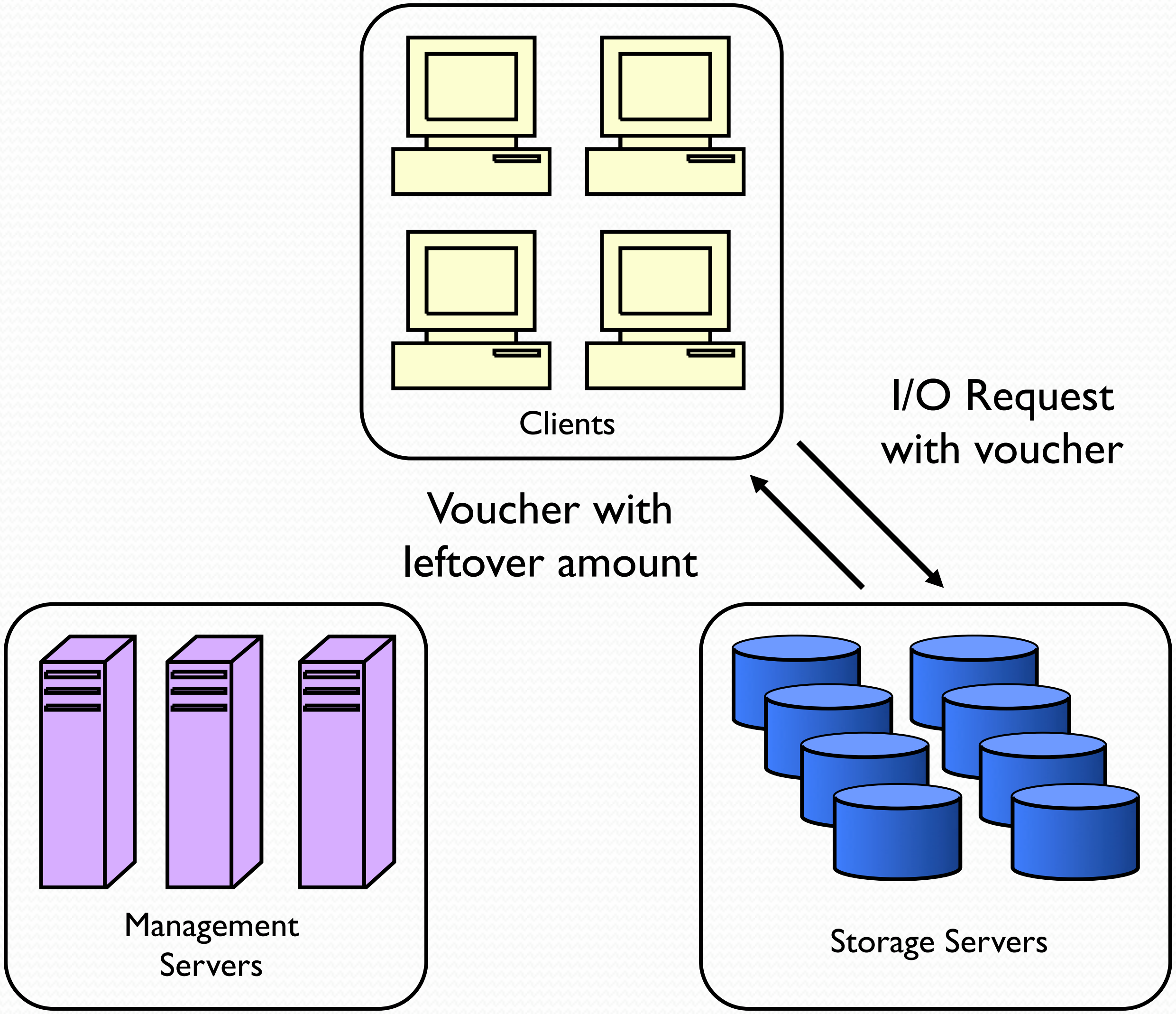


The Protocol

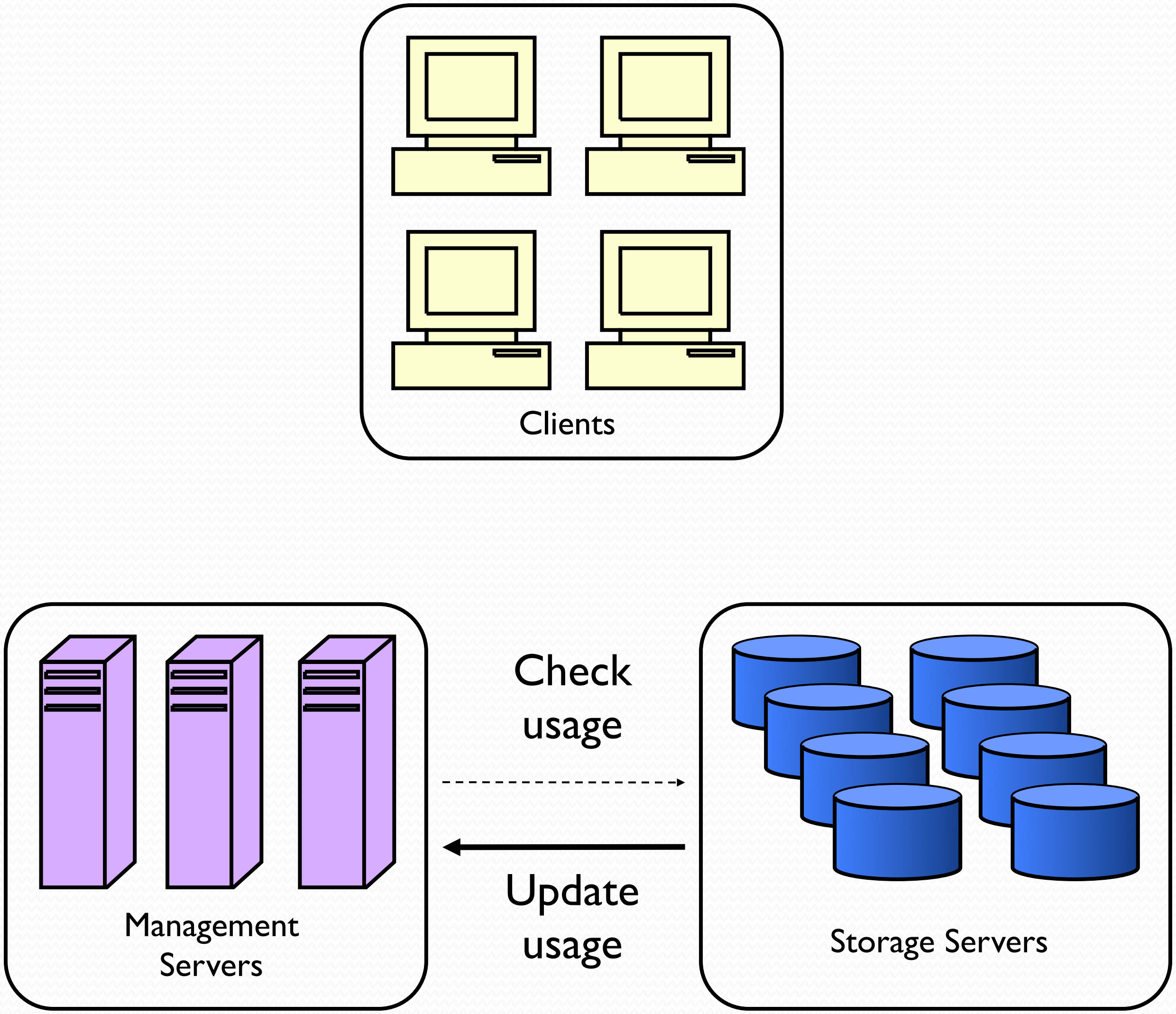
These messages can piggy-back on regular metadata requests if quota servers are co-located with metadata servers.



The Protocol



The Protocol

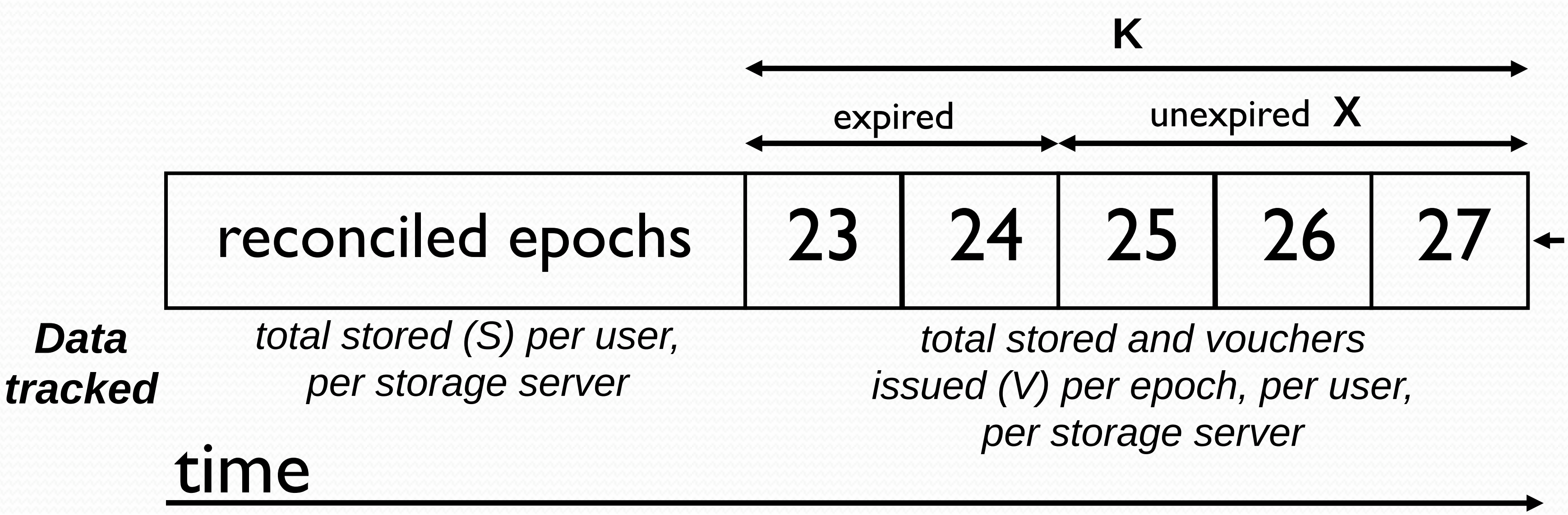


Voucher Splitting and Change

- If a client sends a voucher to a storage server that is greater than the amount allocated the storage server makes change by appending the leftover amount to the voucher.
  - {{epoch, expiry, user, amount, serial}auth, amount′}
- A client may also split a voucher if it needs to use portions of the voucher to satisfy multiple requests asynchronously.
- A client may receive change from a storage server if the entire amount is not spent on storage.
  - Only the storage server knows for certain how much space will be consumed.

Time is Divided into Epochs

- Vouchers are reconciled by epochs.
  - If the system is quiescent, the true state is known after K epochs.
  - This is purely for performance, the larger the K, the less traffic to the bank.
  - Expired vouchers are simply no longer valid.



Calculating a Client's Current Allocation

- Total amount of space client allocated with vouchers created K epochs ago (guaranteed expired since K > X) and the total amount of vouchers issued in the last K epochs.

  or $\sum_{\forall servers} S_{c-K} + \sum_{0 \leq i < K} V_{c-i}$ *where c is the current epoch*

Deletes

- When a user deletes files the storage server issues a refund voucher for the amount deleted.
- The refund voucher is set to the current epoch c, and the refunding storage server *d* decrements its value for $S_c(d)$.
- If the voucher is spent at another storage server d′ it will be added to that server's value for $S_c(d')$.

Catching Cheaters

- How to catch clients using vouchers more than once?
  - Storage servers check voucher serial number for duplicates.
  - When storage servers update quota server for an epoch, the total amount a user stored using vouchers from the epoch is compared to the total amount of vouchers issued for that epoch.
  - Quota server can check for duplicate serial numbers used across storage servers to find misbehaving client.

Bounds on Cheating

- Clients can cheat, but they will be caught, and the amount of cheating is bounded.
- K_μ , where μ is the maximum throughput available to the cheating user (this depends on the number of clients it could corrupt to cheat).
- $(Q – A)n$, where $Q – A$ is the amount of quota the cheating user has left according to the quota server, and n is the number of storage servers it has permission to store data on.

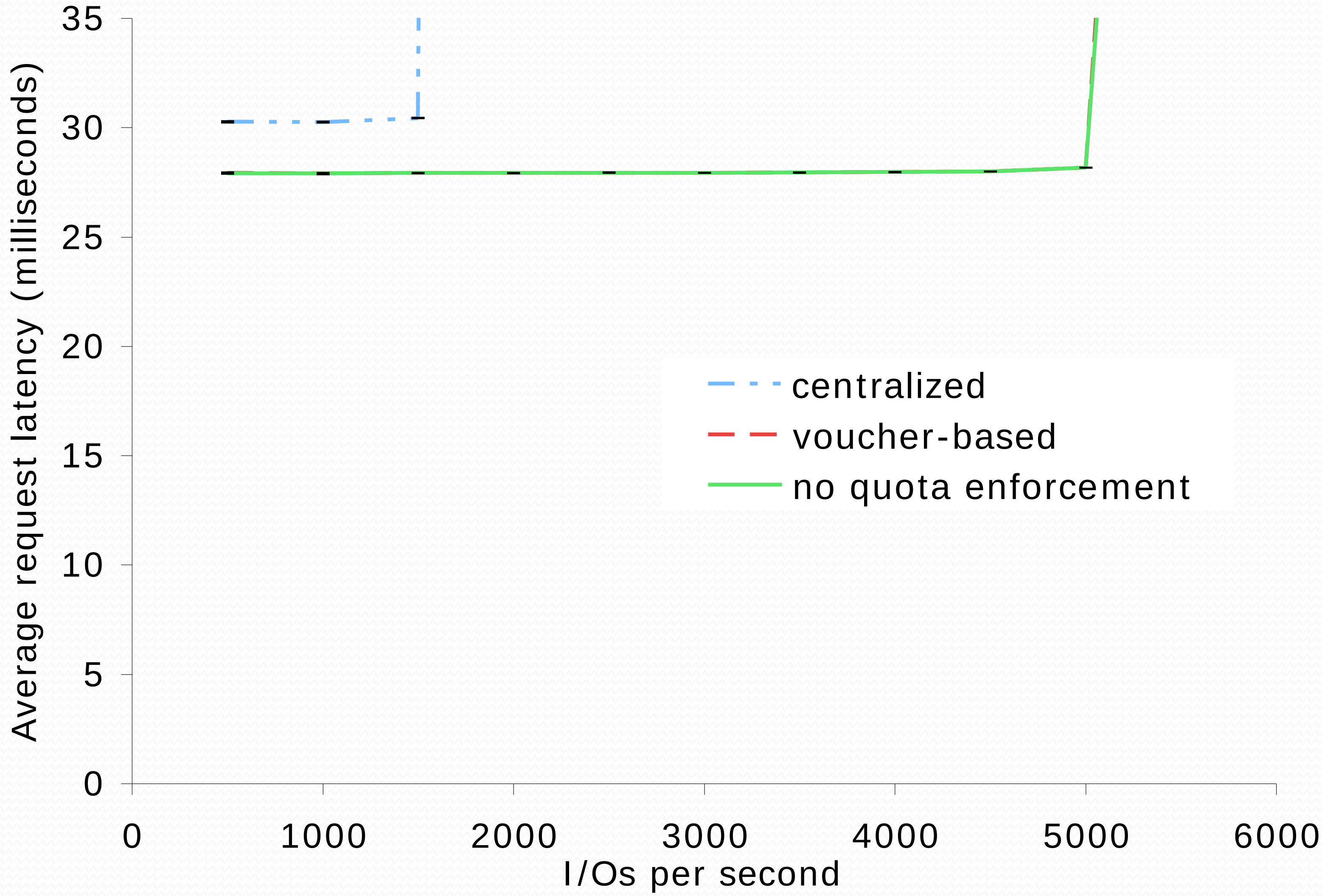
Failures

- Client failure
  - If client is holding unused vouchers they will be accounted for when storage servers update quota server.
- Storage server failure
  - Exclude storage consumed on failed storage server when calculating quotas. This will allow space for rebuild.

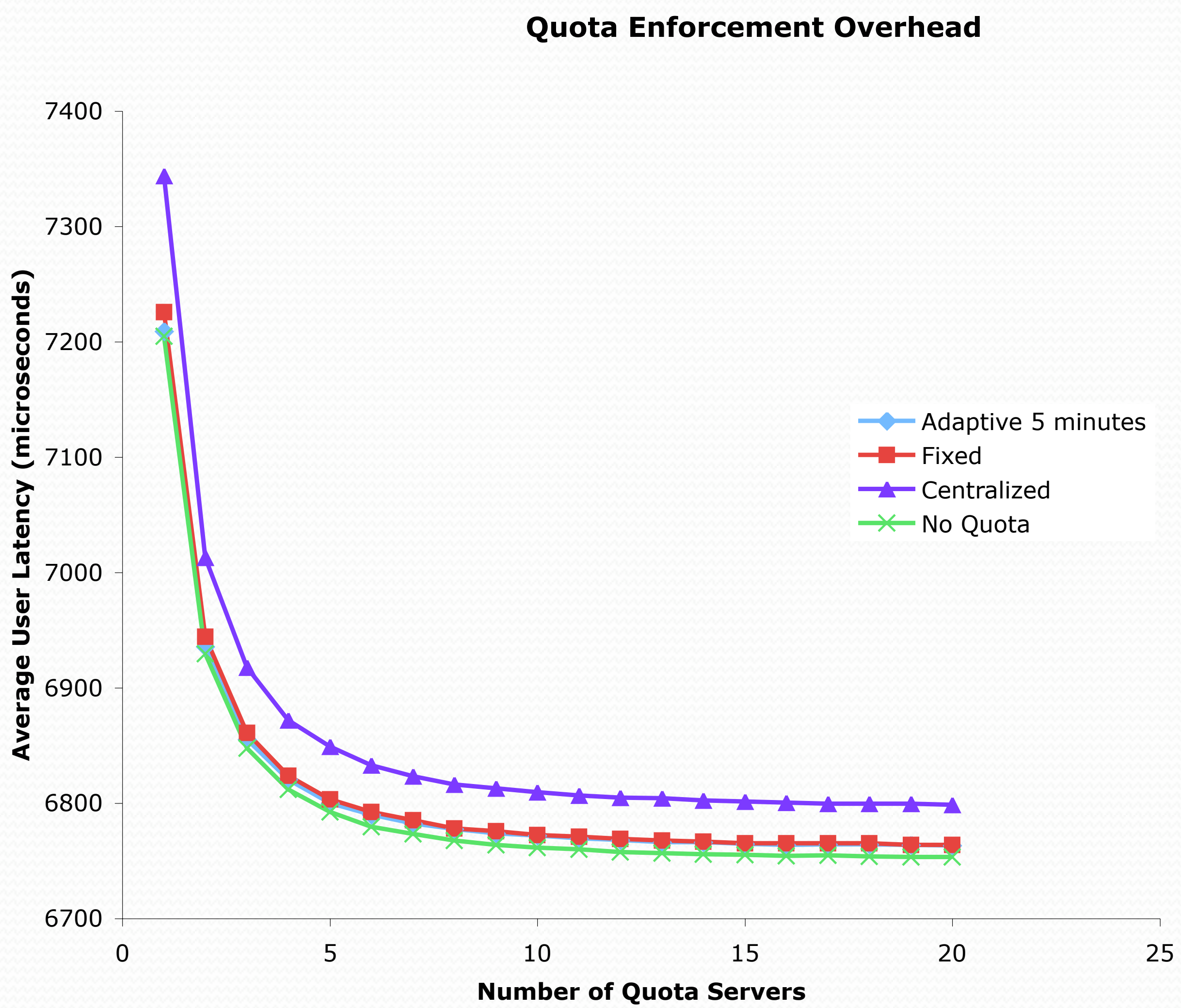
Failures

- Quota server failure
  - After *K* epochs the storage servers will have reported the total reconciled storage for all of the vouchers issued before the management server failure
  - If the management servers actively query all of the storage servers for the purpose of recovery it is possible to recover after *X* epochs by forcing the reconciliation to happen immediately after the last vouchers issued by the management server expire.

Average I/O request latency for the ***scientific workload*** under increasing I/Os per second.



Quota Enforcement

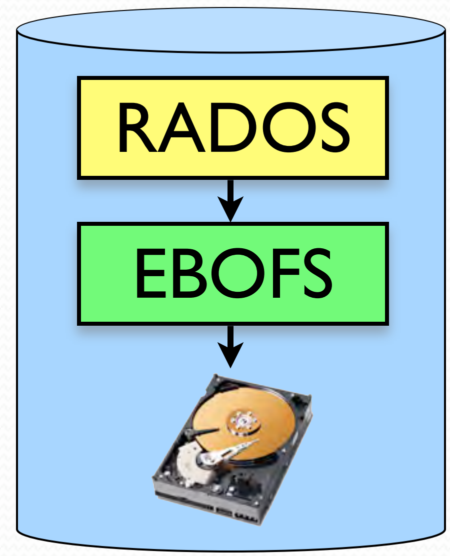


Outline

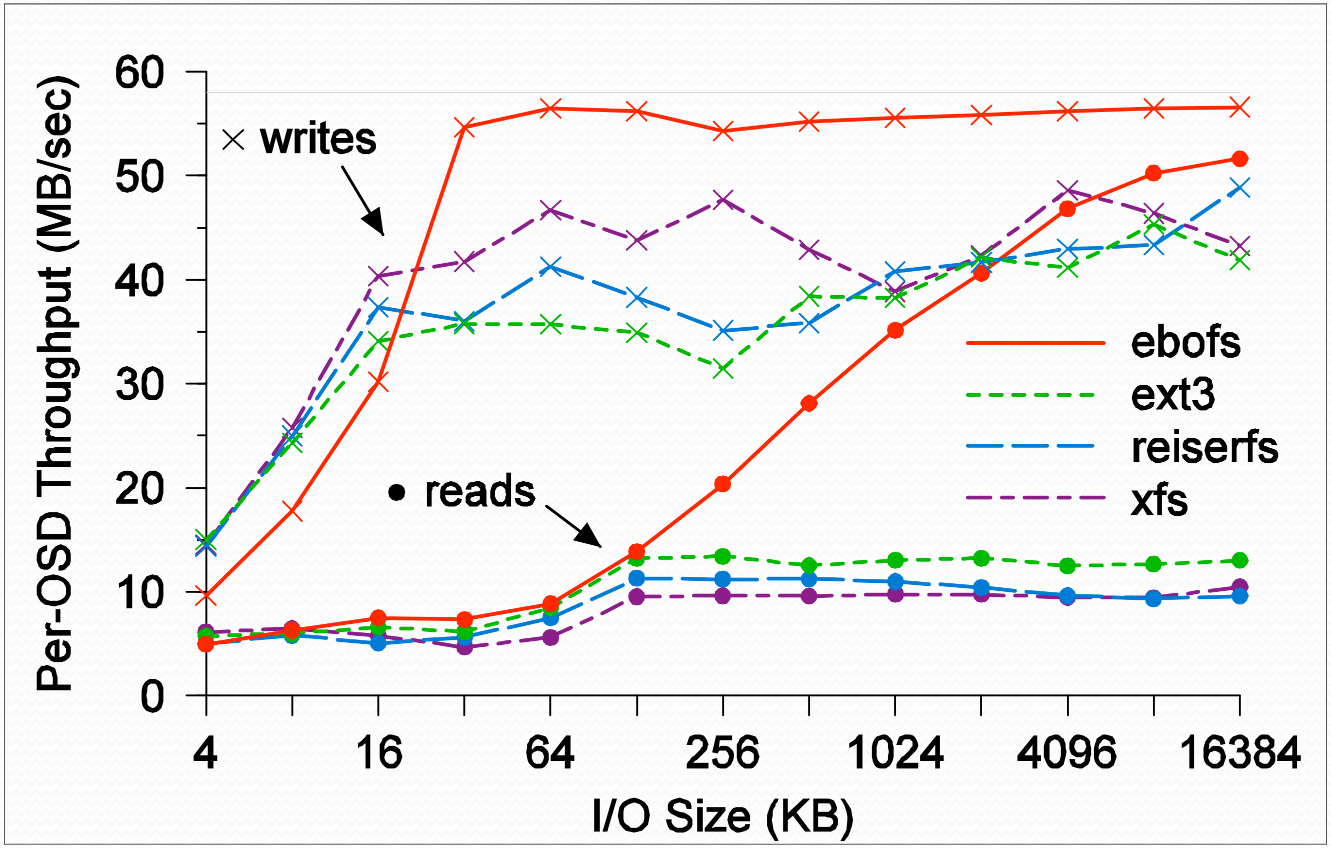
1. Maximal separation of data and metadata
   - Object-based storage
   - Independent metadata management
   - CRUSH – data distribution function
2. Dynamic metadata management
   - Adaptive and scalable
3. Intelligent disks
   - Reliable Autonomic Distributed Object Store
4. Scalable security
5. Scalable quota management
6. High-performance local disk file system

EBOFS—Low-level object storage

- **E**xtent and **B**-tree-based **O**bject **F**ile **S**ystem
- Non-standard interface and semantics
  - Asynchronous notification of commits to disk
  - Atomic compound data+metadata updates
- Extensive use of copy-on-write
  - Revert to consistent state after failure
- User-space implementation
  - We define our own interface—not limited by ill-suited kernel file system interface
  - Avoid Linux VFS, page cache—designed under different usage assumptions



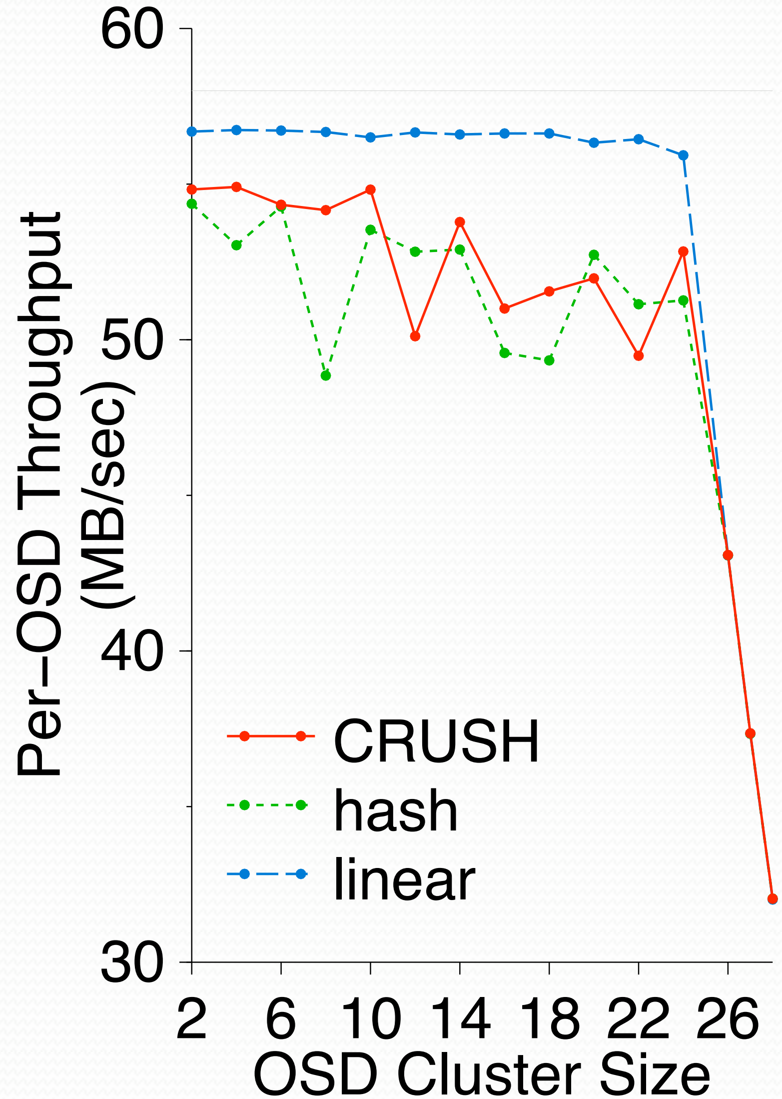
OSD Performance— EBOFS vs. ext3, ReiserFSv3, XFS



- EBOFS writes saturate disk for request sizes over 32k
- Reads perform significantly better for large write sizes

Overall System Performance

- Performance without security
- Performance is per-OSD
  - Limited by bandwidth of the switch!
- Max performance is 1.3 GB/second
  - Done on just 24 nodes!
- Scaling is very close to linear
  - 2400 nodes would run at 130 GB/second!



On-going and Future Work

- Completion of prototype
  - MDS failure recovery
- Rich, scalable metadata
  - Our MDS is built around 30-year old POSIX file system interface
  - Next generation file systems will likely diverge from a single hierarchy
- Archival storage
  - Managing data hot spots, idle data
- More research on scalable security: scalable encryption on disk
- In-flight data management
  - Tracking data as it moves around the system
  - Retrieving cached copies from clients that have the data
- Quality of service
- Time travel (snapshots)

Conclusions

- **High performance and reliability with excellent scalability!**
- CRUSH distribution function makes it possible
  - Global knowledge of complete data distribution
  - Data locations calculated when needed
- Decoupled metadata improves scalability
  - Eliminating allocation lists makes metadata simple
  - MDS stays out of I/O path
- Dynamic metadata management
  - Preserve locality, improve performance
  - Adapt to varying workloads, hot spots
- Intelligent OSDs
  - Manage replication, failure detection, and recovery
- Scalable security
  - Keeps data secure without compromising performance

Ceph Contributors

- Martin Arnberg
- Bo Hong
- R.J. Honicky
- Stephanie Jones
- Eric Lalonde
- Andrew Leung
- Christopher Olson
- Kristal Pollack
- Feng Wang
- Sage Weil
- Joel Wu
- Qin Xin
- Lan Xue

Sponsors

- Lawrence Livermore National Laboratory
- Los Alamos National Laboratory
- Sandia National Laboratory
- Department of Energy Office of Science
- National Science Foundation
- SSRC industrial sponsors

Open Source (go work on it!)

- http://ceph.newdream.net/
- http://sourceforge.net/projects/ceph