CSE 487/587
Data Intensive Computing

Lecture 6/7: Parallel File Systems

Vipin Chaudhary

vipin@buffalo.edu

716.645.4740
305 Davis Hall

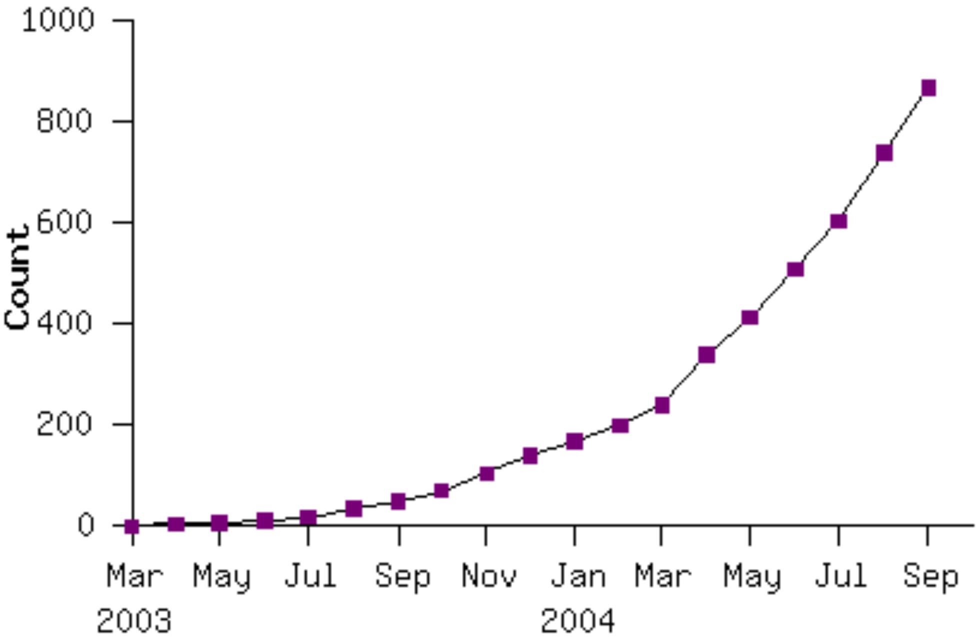
Overview of Today’s Lecture

- MapReduce Conclusion
- Description of Programming Assignment

- Network/Parallel/Distributed File systems
  - NFS and pNFS
  - PVFS
  - Lustre
  - GPFS

Model is Widely Applicable

MapReduce Programs In Google Source Tree



Example uses:

distributed grep

term-vector / host

document clustering

…

distributed sort

web access log stats

machine learning

…

web link-graph reversal

inverted index construction

statistical machine translation

…

Implementation Overview

Typical cluster:

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

Fault Tolerance / Workers

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks
- Re-execute in progress *reduce* tasks
- Task completion committed through master
-

Robust: lost 1600/1800 machines once → finished ok

Semantics in presence of failures: see paper

Master Failure

- Could handle, … ?
- But don't yet
    - (master failure unlikely)

Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement
Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:

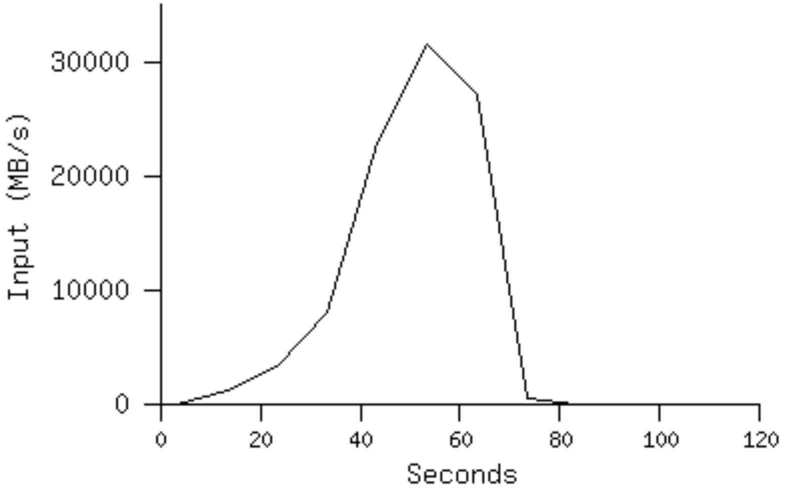
- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

MR_Grep 10^{10} 100-byte records to extract records matching a rare pattern (92K matching records)

MR_Sort 10^{10} 100-byte records (modeled after TeraSort benchmark)

MR_Grep



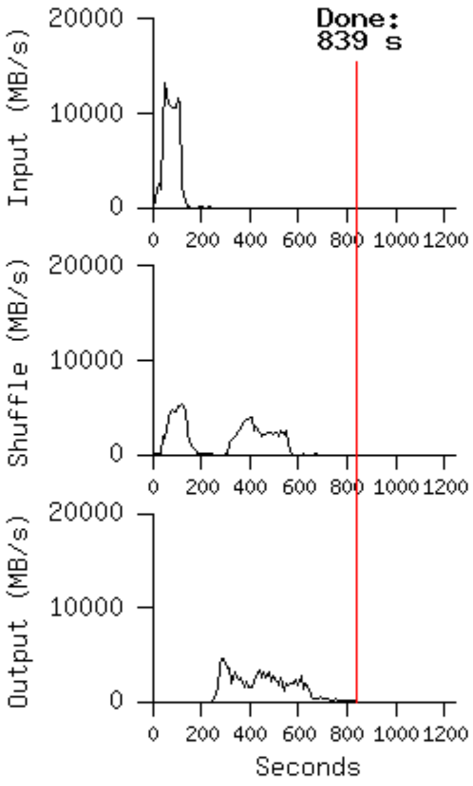
Locality optimization helps:

- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

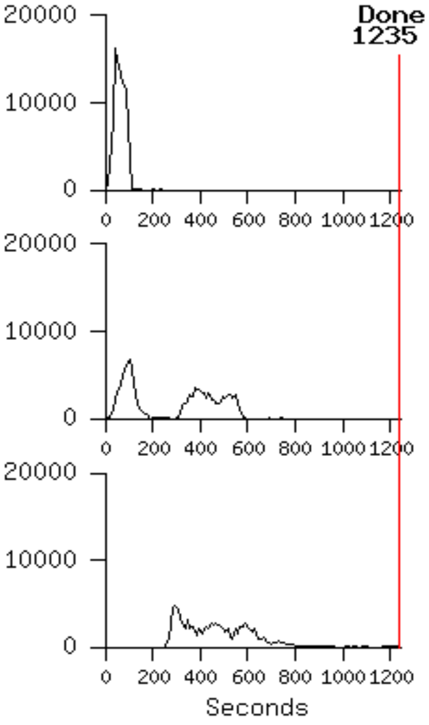
Startup overhead is significant for short jobs

MR_Sort

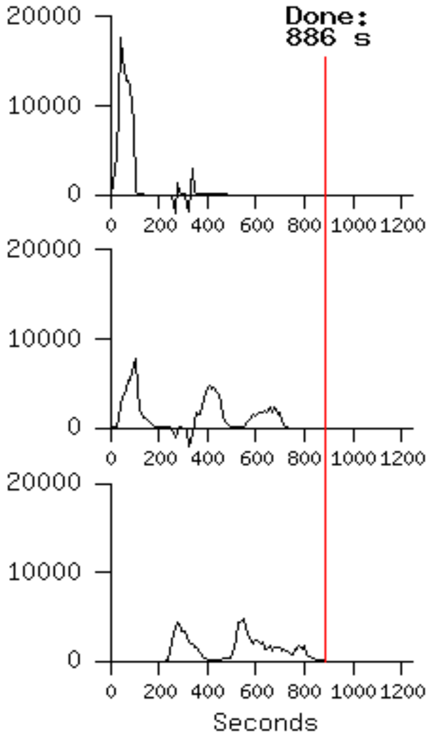
Normal



No backup tasks



200 processes killed



- Backup tasks reduce job completion time a lot!
- System deals well with failures

Experience

Rewrote Google's production indexing System using MapReduce

- Set of 10, 14, 17, 21, 24 MapReduce operations
- New code is simpler, easier to understand
  - 3800 lines C++ → 700

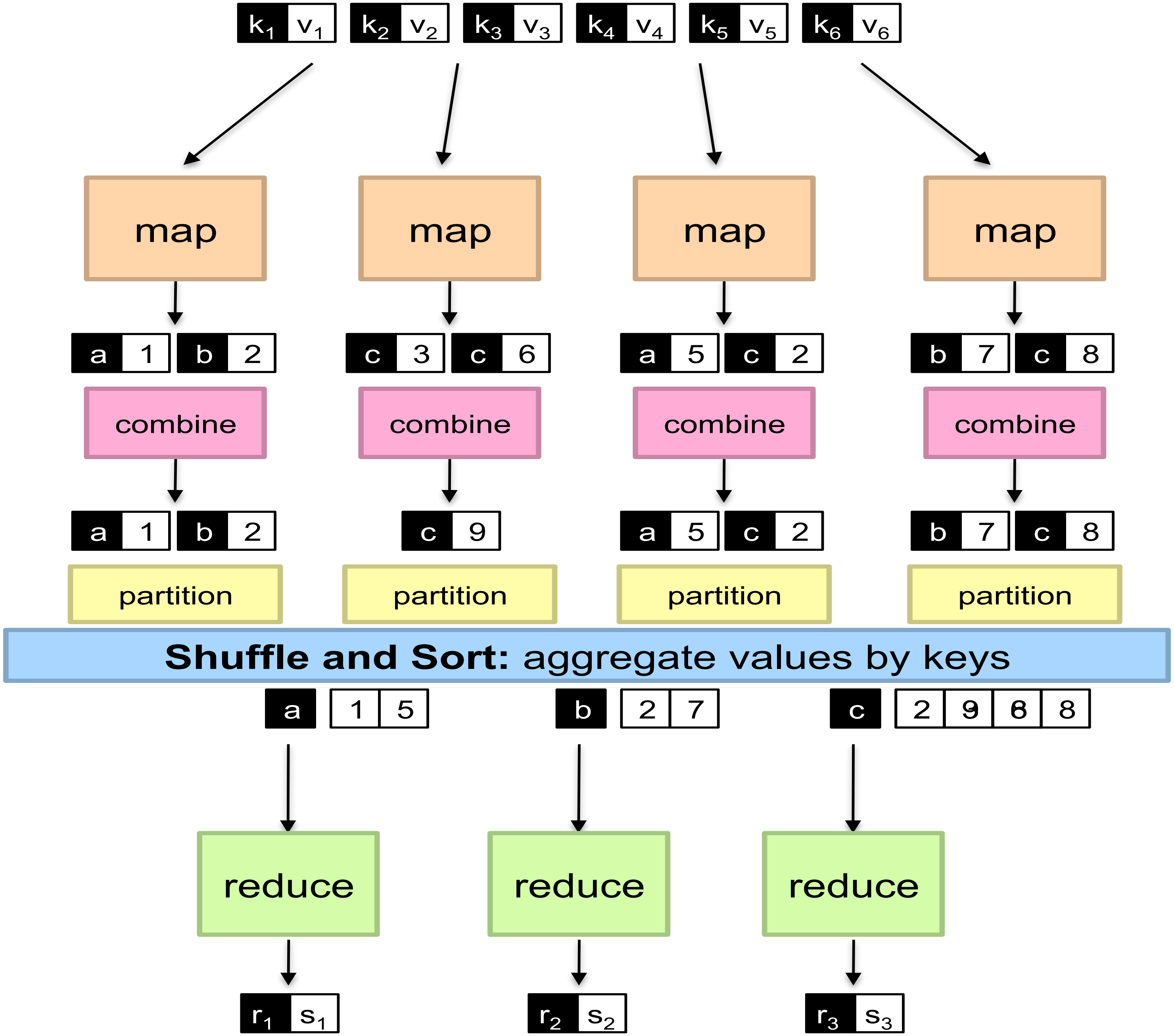
- MapReduce handles failures, slow machines
- Easy to make indexing faster by adding more machines

Usage in Aug 2004

Number of jobs	29,423	
Average job completion time	634	secs
Machine days used	79,186	days
Input data read	3,288	TB
Intermediate data produced	758	TB
Output data written	193	TB
Average worker machines per job	157	
Average worker deaths per job	1.2	
Average map tasks per job	3,351	
Average reduce tasks per job	55	
Unique *map* implementations	395	
Unique *reduce* implementations	269	
Unique *map*/*reduce* combinations	426	

Related Work

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River's distributed queues
  - River ['99]



k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6

map map map map

a 1 b 2 c 3 c 6 a 5 c 2 b 7 c 8

combine combine combine combine

a 1 b 2 c 9 a 5 c 2 b 7 c 8

partition partition partition partition

Shuffle and Sort: aggregate values by keys

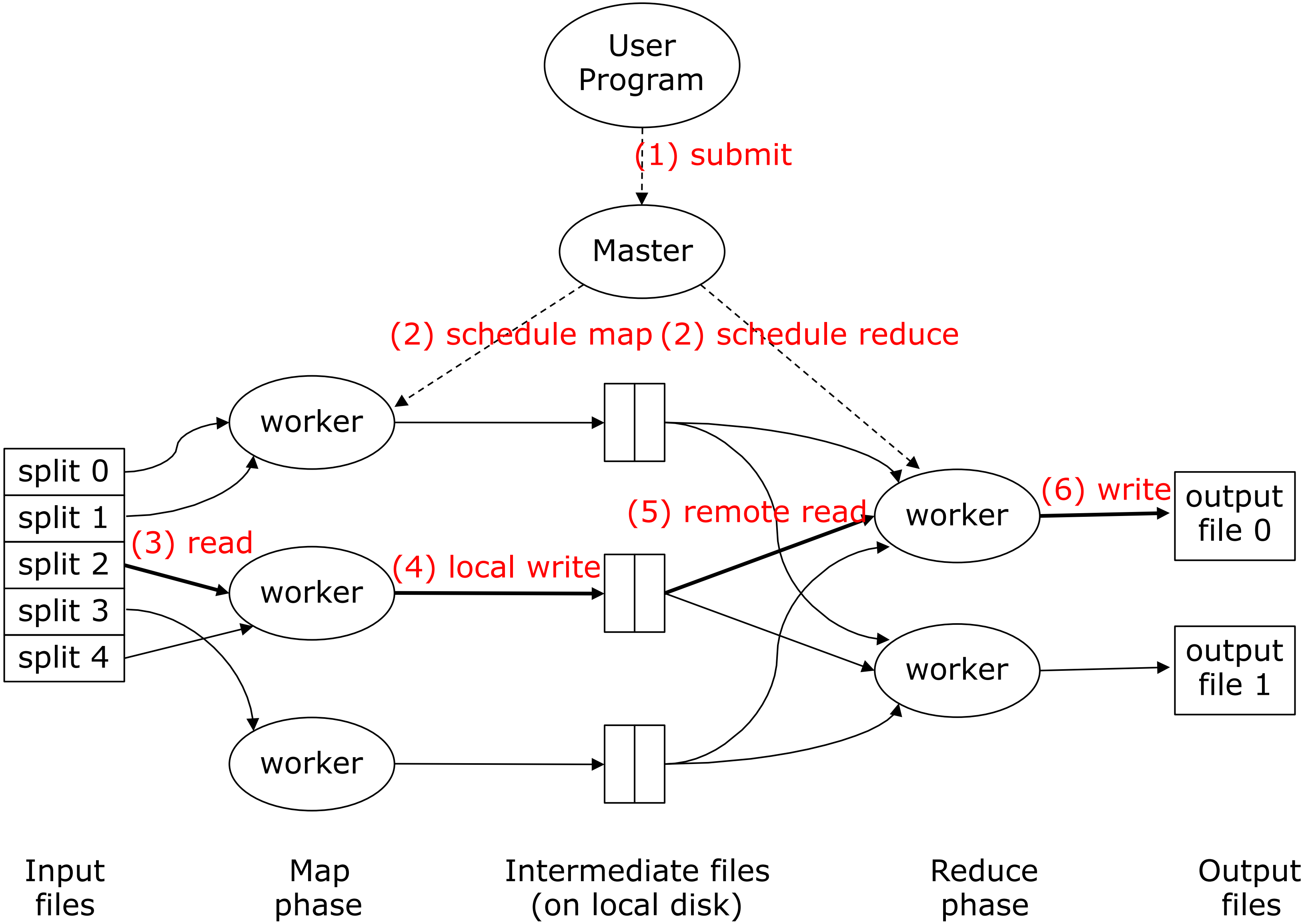
a 1 5 b 2 7 c 2 9 6 8

reduce reduce reduce

r_1 s_1 r_2 s_2 r_3 s_3

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



Adapted from (Dean and Ghemawat, OSDI 2004)

Conclusions

- MapReduce proven to be useful abstraction

- Greatly simplifies large-scale computations

- Fun to use:
    - focus on problem,
    - let library deal w/ messy details

Programming Assignment

- Stock Volatility
- Program submission before deadline
  - /gpfs/courses/cse587/Spring2015/username
  - UBLearns
  - /gpfs/courses/cse587/spring2015/data/hw1/small (approx.100M)
  - /gpfs/courses/cse587/spring2015/data/hw1/medium (approx.300M)
  - /gpfs/courses/cse587/spring2015/data/hw1/large (approx.1000M)

Basic File System Recap

- Must have
  - Name: like /gpfs/courses/cse587/vipin
  - Data: some sequence of bytes

- Additional Information
  - Size
  - Protection information
  - Location
  - Time

Sharing Files - 2 approaches

- Copy based
  - Application explicitly copies files between machines
  - FTP, RCP, SCP, …
- Access transparency by DFS

Copy Based Flow

- Find a copy
  - Naming based on machine name of source, rush.ccr.buffalo.edu, username, path
- Transfer file to local file system
  - Scp username@rush.ccr.buffalo.edu:test.pdf
- Read/write
- Copy back if modified

Copy Based Benefits

- Pros
  - Semantics are clear
  - No OS/library modifications
- Cons
  - Involved process
  - Have to copy entire file
  - Inconsistency possible
  - Inconsistent copies distributed at many places

Access Transparency

- Process of accessing remote data same as local data (using file system hierarchy) using DFS
- DFS stores files on more than one system
  - Network File Systems
  - Parallel File Systems
  - Distributed File Systems

Advantages of DFS

- Sharing files with other users
  - Easy for others to access your files and vice versa
- Redundant copies of your files elsewhere
- Small amount of local resources suffice
- High failure rate of local resources ok
- Eliminates version problem

Naming

- Location Transparency
  - Names do not reveal physical location
  - Mount like protocols /../ mntLocationofSharedFolder/myFile are not transparent

- Location Independence
  - File name does not change with change in physical location of file

Performance for PFS/DFS

- Caching
  - most important factor to improve performance
  - Biggest cause of problems for developers
    - Maintaining consistency
    - Impact on scalability
  - Size of Cache
    - Large => efficient use of network: spatial locality, latency
    - Caching complete files simple but lack of resources
    - Small files: too much overhead
  - Pull/Push of data

Consistency & Replication

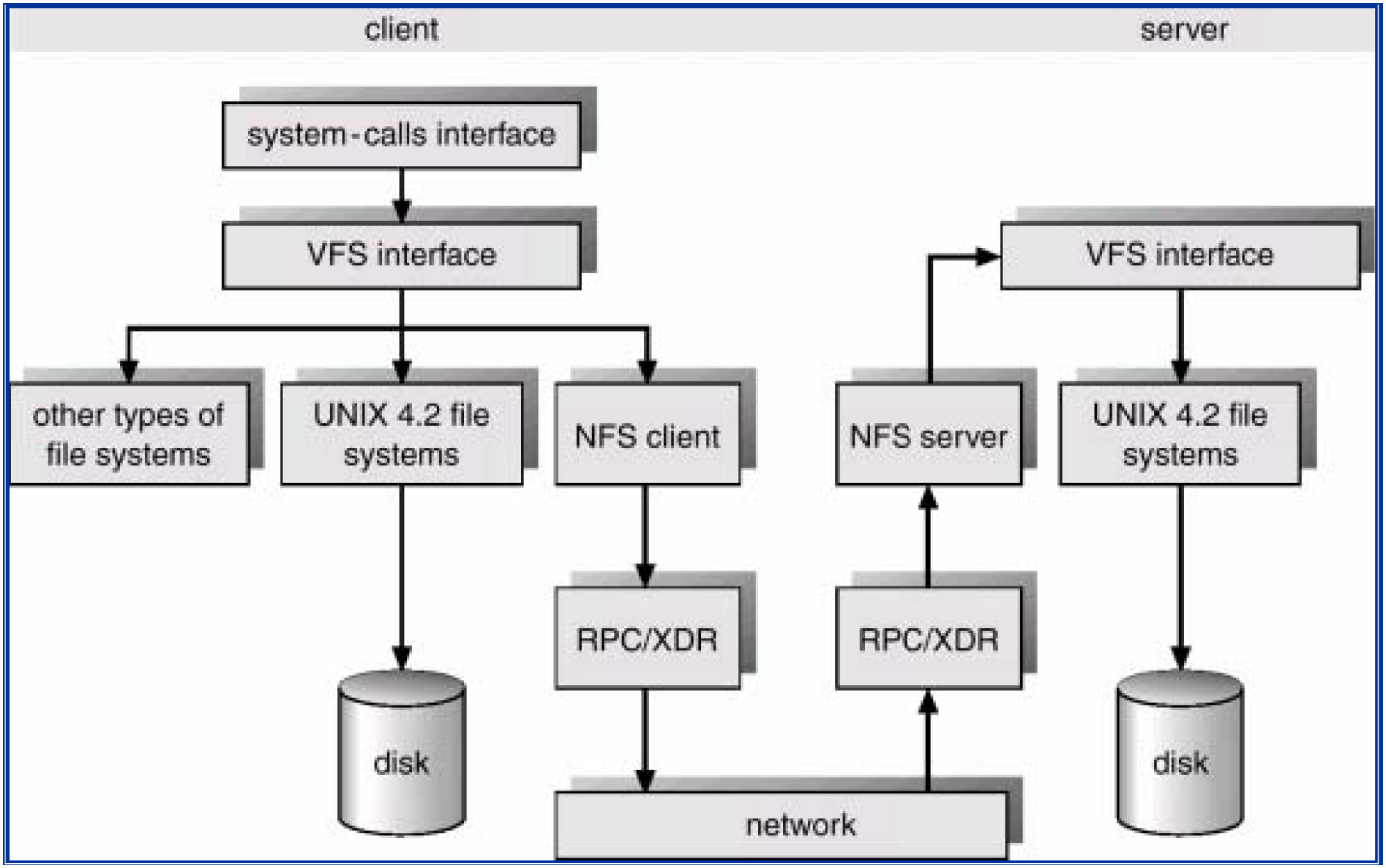
- Consistency varies with DFS
- Replication
  - Improves Fault tolerance
  - Improves Performance
- Need location independence in naming
- Different forms of replication mechanisms lead to varying consistency semantics

NFS (Network File System)

- *NFS (Network File System)* allows hosts to mount partitions on a remote system and use them as though they are local file systems. This allows the system administrator to store resources in a central location on the network, providing authorized users continuous access to them.

- NFSv3 very successful, changes in NFSv4 and pNFS

NFS (Network File System)



NFS (Network File System)

- There are multiple ways to implement network file system:
- Important aspect of NFS implementation – implementing effective cache mechanism to boost performance.

NFS (Network File System)

Implementations:

- CIFS (Microsoft Common Internet File System based on SMB protocol). Widely used in Microsoft Windows Networks and in heterogeneous environment.
- NFS (SUN Microsystems initial implementation). Widely used in Unix environment.
- Andrew file system (Carnegie-Mellon university implementation). Was used in academic environment.

Objectives (I)

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files

Objectives (II)

- ***UNIX semantics should be maintained on client***
  - Best way to achieve transparent access
- ***“Reasonable” performance***
  - Robustness and preservation of UNIX semantics were much more important

Basic design

- Three important parts
  - The protocol
  - The server side
  - The client side

The protocol (I)

- Uses the Sun RPC mechanism and Sun eXternal Data Representation (XDR) standard
- Defined as a set of remote procedures
- Protocol is ***stateless***
  - Each procedure call contains ***all the information necessary to complete the call***
  - Server maintains no “between call” information

Advantages of statelessness

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server
- Client can always ***repeat any request***

Consequences of statelessness

- Read and writes must specify their start offset
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes
- Open system call translates into several lookup calls to server
- No NFS equivalent to UNIX close system call

The lookup call (I)

- Returns a **file handle** instead of a file descriptor
  - File handle specifies unique location of file
- **lookup(dirfh, name)** *returns* **(fh, attr)**
  - Returns file handle **fh** and attributes of named file in directory **dirfh**
  - Fails if client has no right to access directory **dirfh**

The lookup call (II)

- One single open call such as

  fd = open(“/usr/vipin/cse587/nfs.txt”)

will be result in several calls to lookup

lookup(rootfh, “usr”) returns (fh0, attr)
lookup(fh0, “vipin”) returns (fh1, attr)
lookup(fh1, “cse587”) returns (fh2, attr)
lookup(fh2, “nfs.txt”) returns (fh, attr)

The lookup call (III)

- Why all these steps?
  - Any of components of **/usr/vipin/cse587/nfs.txt** could be a ***mount point***
  - Mount points are ***client dependent*** and mount information is kept above the lookup() level

Server side (I)

- Server implements a ***write-through*** policy
  - Required by statelessness
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

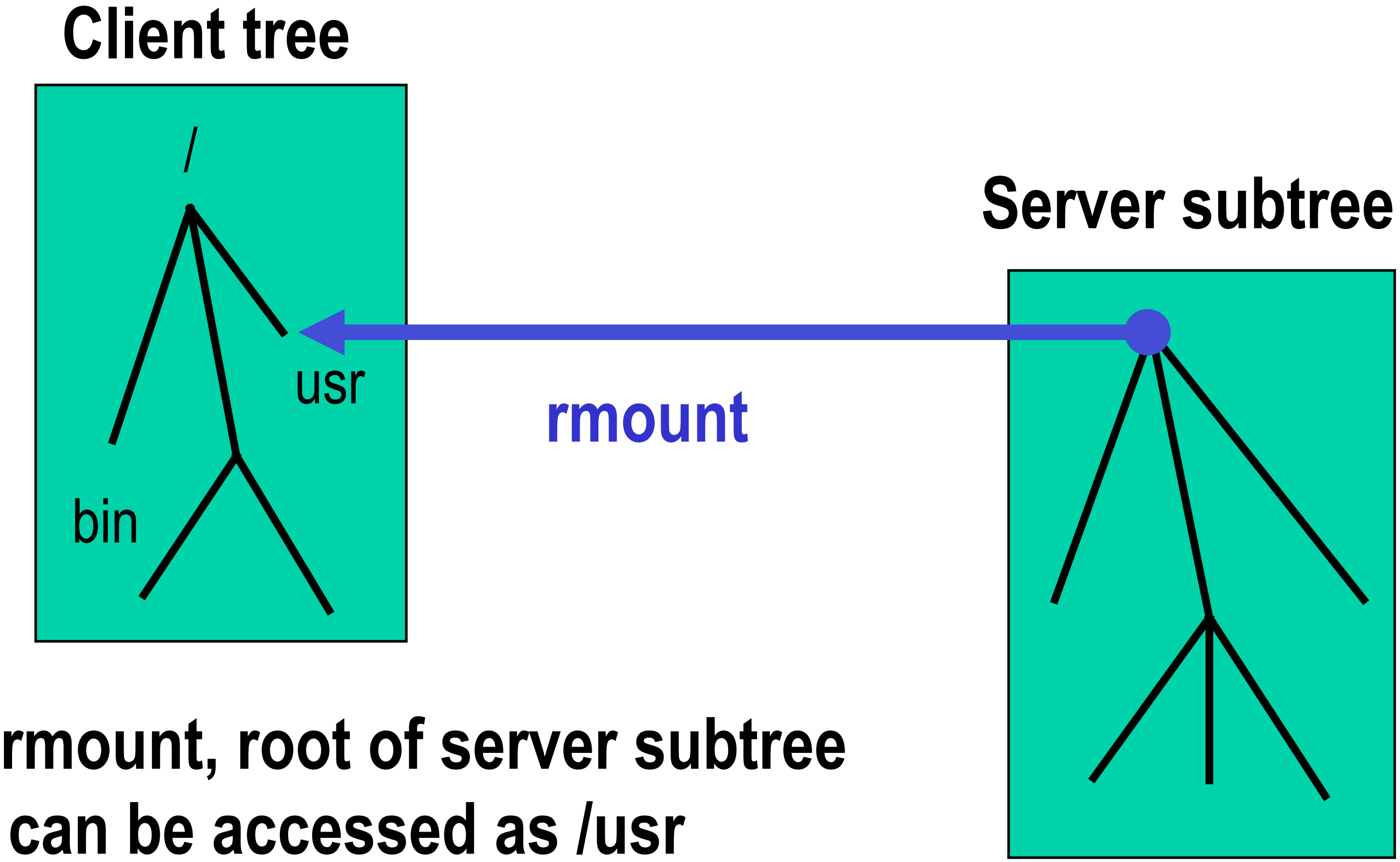
Server side (II)

- ***File handle*** consists of
  - ***Filesystem id*** identifying disk partition
  - ***I-node number*** identifying file within partition
  - ***Generation number*** changed every time i-node is reused to store a new file
- Server will store
  - ***Filesystem id*** in filesystem **superblock**
  - I-node ***generation number*** in **i-node**

Client side (I)

- Provides transparent interface to NFS
- Mapping between remote file names and remote file addresses is done a server boot time through ***remote mount***
  - Extension of UNIX mounts
  - Specified in a ***mount table***
  - Makes a remote subtree appear part of a local subtree

Remote mount

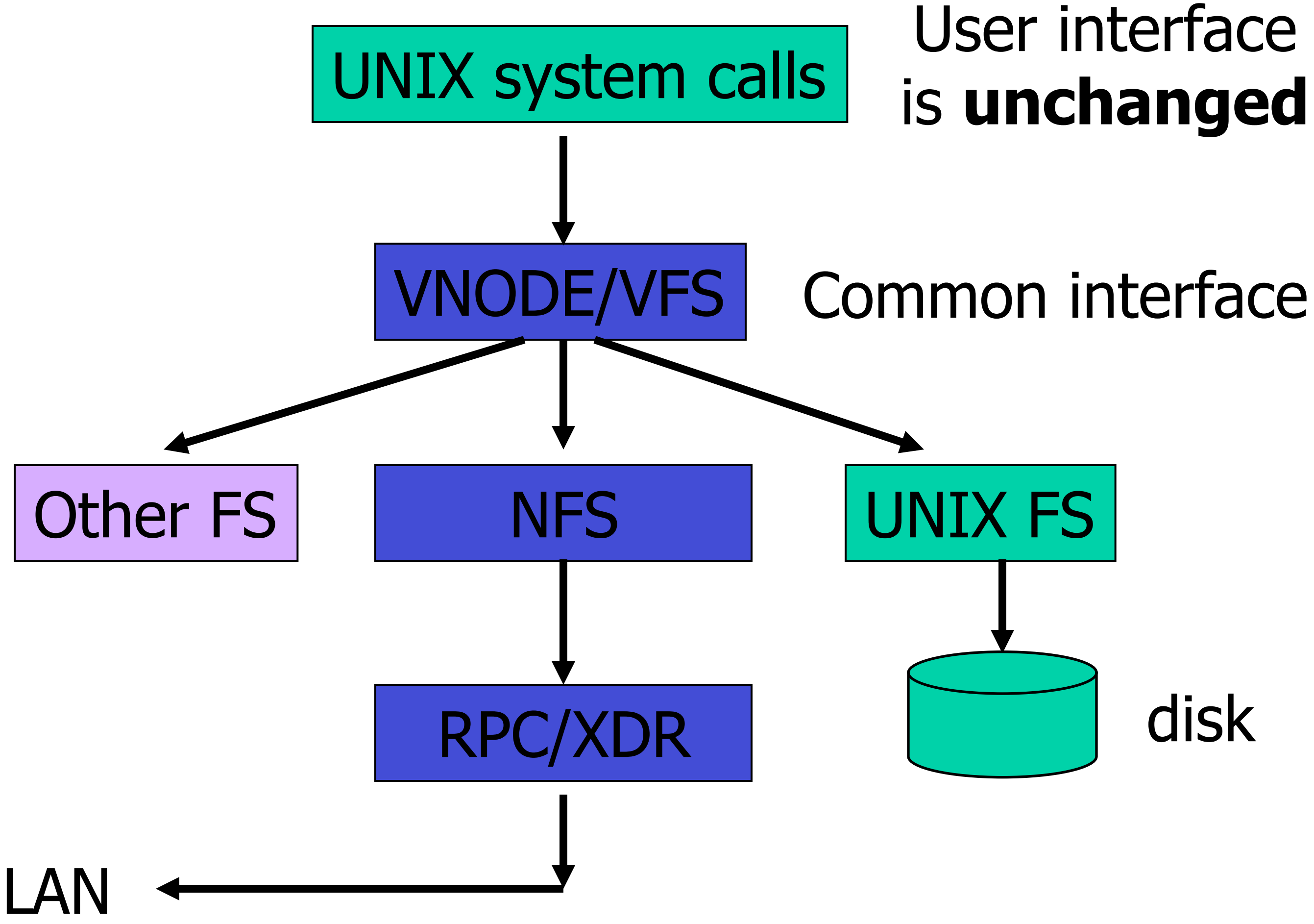


After rmount, root of server subtree can be accessed as /usr

Client side (II)

- Provides transparent access to
  - NFS
  - Other file systems
- New virtual filesystem interface supports
  - VFS calls, which operate on whole file system
  - VNODE calls, which operate on individual files
- Treats all files in the same fashion

Client side (III)



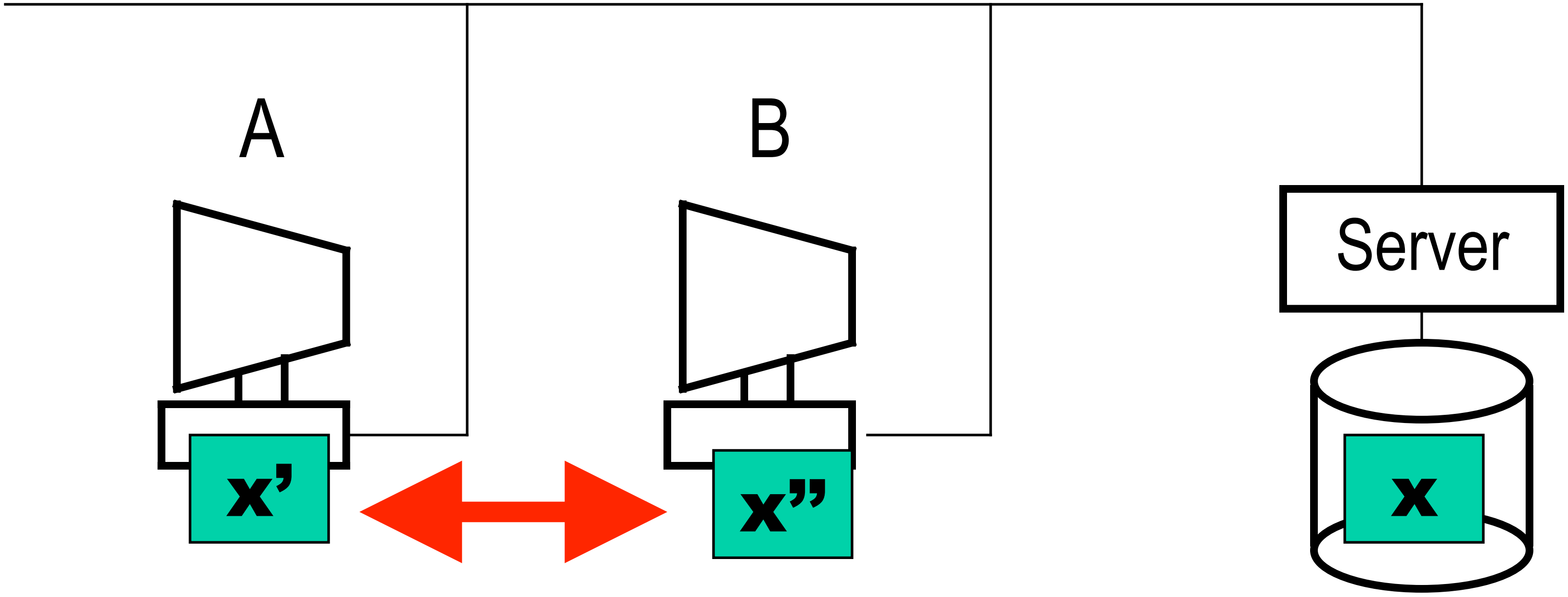
File consistency issues

- Cannot build an efficient network file system without ***client caching***
  - *Cannot send each and every read or write to the server*
- ***Client caching introduces <u>consistency issues</u>***

Example

- Consider a one-block file X that is concurrently modified by two workstations
- If file is cached at ***both*** workstations
  - A will not see changes made by B
  - B will not see changes made by A
- We will have
  - Inconsistent updates
  - Non respect of UNIX semantics

Example



Inconsistent updates X' and X" to file X

UNIX file access semantics (I)

- Conventional timeshared UNIX semantics guarantee that
  - All writes are executed in strict sequential fashion
  - Their effect is immediately visible to all other processes accessing the file
- Interleaving of writes coming from different processes is left to the kernel discretion

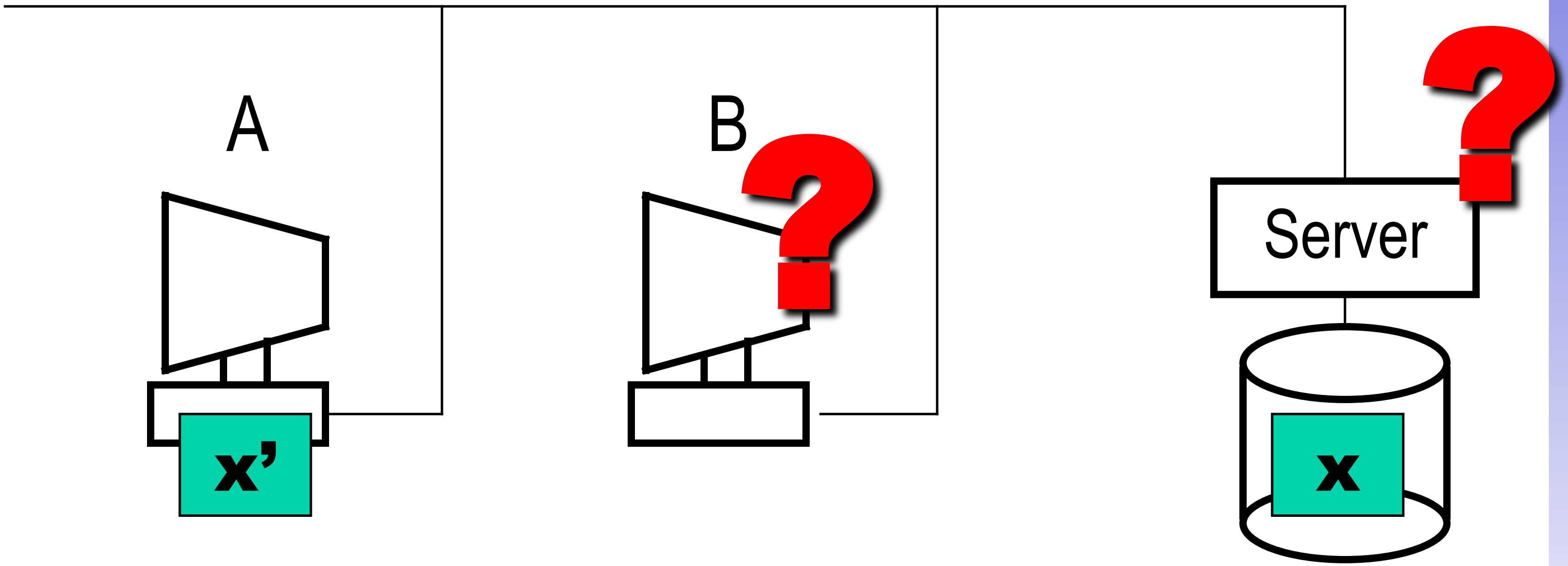
UNIX file access semantics (II)

- UNIX file access semantics result from the use of a single I/O buffer containing all cached blocks and i-nodes
- Server caching is not a problem
- Disabling client caching is not an option:
  - Would be too slow
  - Would overload the file server

NFS solution (I)

- Stateless server does not know how many users are accessing a given file
  - ***Clients do not know either***
- Clients ***must***
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache

NFS solution (II)



Better to propagate my updates and refresh my cache

Implementation

- VNODE interface only made the kernel 2% slower
- Few of the UNIX FS were modified
- MOUNT was first included into the NFS protocol
  - Later broken into a separate user-level RPC process

Hard issues (I)

- UNIX allows removal of opened files
  - File becomes **nameless**
  - Processes that have the file opened can continue to access the file
  - Other processes cannot
- NFS cannot do that and remain stateless
  - NFS client detecting removal of an opened file renames it and deletes renamed file at close time

Hard issues (II)

- In general, NFS tries to preserve UNIX open file semantics but does not always succeed
  - If an opened file is removed by a process on another client, file is immediately deleted

Problems with NFS

- Write performance is slow
  - Could add non-volatile RAM to the server
    - Was expensive

- Poor scalability: data at one place
- Poor availability: failure of server

First programming assignment

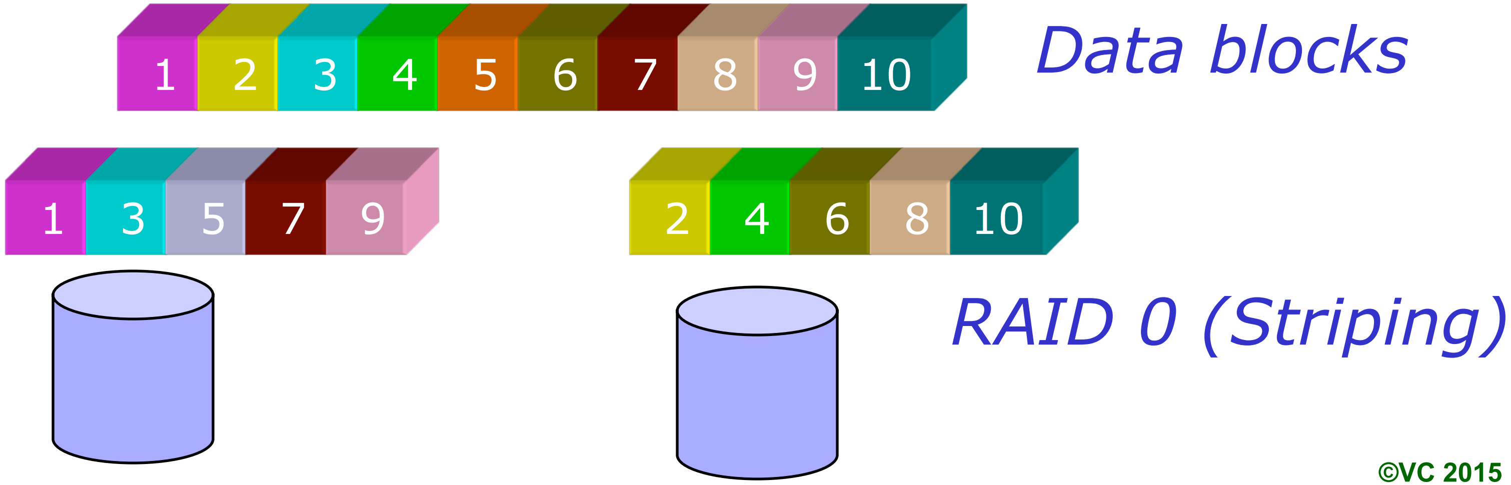
- Due on Feb 28

PVFS: A Parallel File System for Linux Clusters

- Design Goals
  - Provide high bandwidth for concurrent read/write operations
  - Support multiple APIs
    - Native PVFS API
    - Unix/Posix/ and MPI-IO API
  - Common Unix commands must work with PVFS Files (ls, cp, rm, …)
  - Apps developed with Unix I/O API must be able to access PVFS files without recompiling
  - Robust and Scalable
  - Easy to install and use

PVFS…

- N servers exporting portions of a file to parallel application tasks running on multiple processing nodes over an existing network
  - Instead of one server exporting a file via NFS,
  - Aggregate bandwidth exceeds that of a single machine exporting the same file to all processing nodes.
- This works much the same way as RAID 0 – file data is striped across all I/O nodes.



PVFS Features

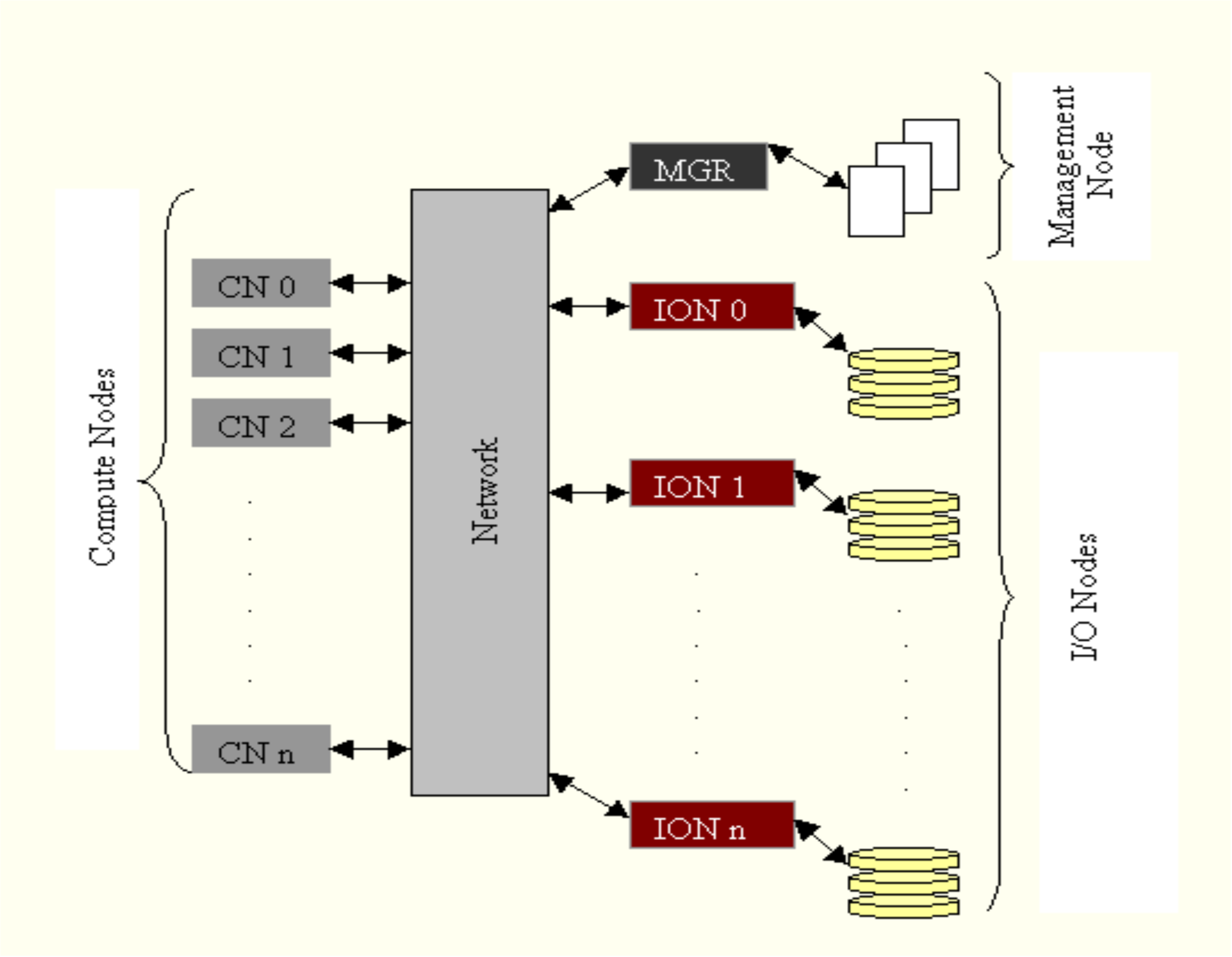
- allows existing binaries to operate on PVFS files without the need for recompiling
- enables user-controlled striping of data across disks on the I/O nodes
- provides high bandwidth for concurrent read/write operations from multiple processes to a common file
- easily used - provides a cluster wide consistent name space,
- PVFS file systems may be mounted on all nodes in the same directory simultaneously, allowing all nodes to see and access all files on the PVFS file system through the same directory scheme.
- Once mounted PVFS files and directories can be operated on with all the familiar tools, such as ls, cp, and rm.

PVFS Design & Implementation

- PVFS spreads data out across multiple cluster nodes, called I/O nodes
  - High performance access
  - applications have multiple paths to data through the network and multiple disks on which data is stored.
  - eliminates single bottlenecks in the I/O path and thus increases the total potential bandwidth for multiple clients, or aggregate bandwidth.
- Roles of nodes in PVFS:
  - COMPUTE NODES - on which applications are run,
  - MANAGEMENT NODE - which handles metadata operations
  - I/O NODES - which store file data for PVFS file systems.

Note:- nodes may perform more than one role

PVFS System Architecture



PVFS Components

- Four major components to the PVFS system:
  1. Metadata server (mgr)
  2. I/O server (iod)
  3. PVFS native API (libpvfs)
  4. PVFS Linux kernel support

  - The first two components are daemons (server types) which run on nodes in the cluster

1. <u>The metadata server (or mgr)</u>
   - File manager; it manages metadata for PVFS files.
   - A single manager daemon is responsible for the storage of and access to all the metadata in the PVFS file system
   - <u>Metadata</u> - information describing the characteristics of a file, such as permissions, the owner and group, and, more important, the physical distribution of the file data

PVFS Components

- PVFS files are striped across a set of I/O nodes in order to facilitate parallel access.
- The specifics of a given file distribution are described with three metadata parameters:
  - base I/O node number
  - number of I/O nodes
  - stripe size
- These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified

PVFS Components

Example:

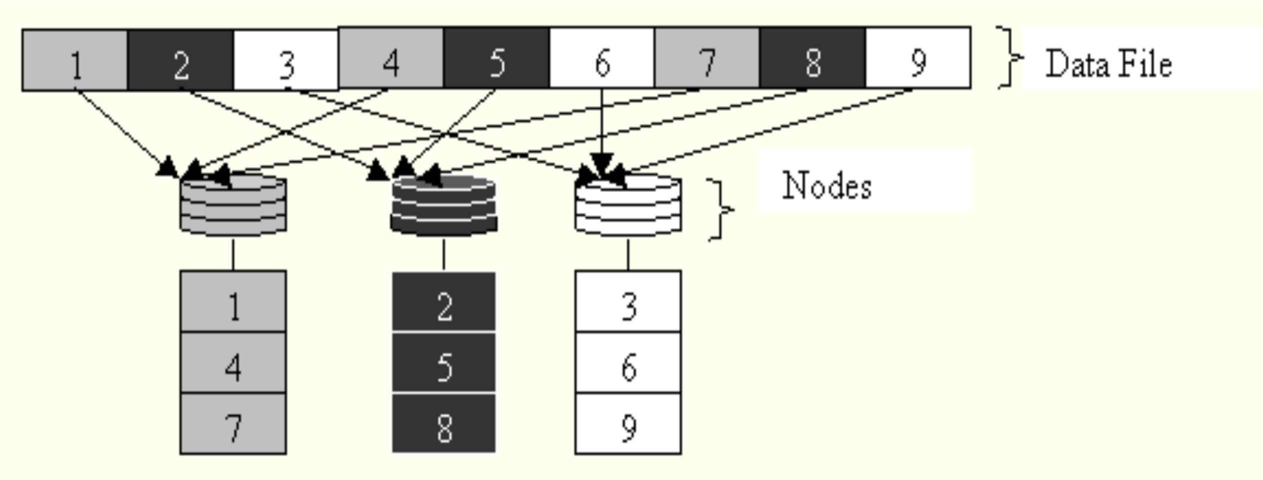
- pcount - field specifies that the the number of I/O nodes used for storing data
- base - specifies that the first (or base) I/O node (is node 2 here)
- ssize - specifies that the stripe size--the unit by which the file is divided among the I/O nodes—here it is 64 Kbytes
- The user can set these parameters when the file is created, or PVFS will use a default set of values

inode	1092157504
base	2
pcount	3
ssize	65536

Meta data example for file

PVFS Components

PVFS file striping done in a round-robin fashion



- Though there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a striping.
- Each I/O daemon stores its portion of the PVFS file in a file on the local file system on the I/O node.
- The name of this file is based on the inode number that the manager assigned to the PVFS file (in our example, 1092157504).

PVFS Components

- When application processes (clients) open a PVFS file, the PVFS manager informs them of the locations of the I/O daemons

- Clients then establish connections with the I/O daemons directly

- When a client wishes to access file data, the client library sends a descriptor of the file region being accessed to the I/O daemons holding data in the region

- Daemons determine what portions of the requested region they have locally and perform the necessary I/O and data transfers.

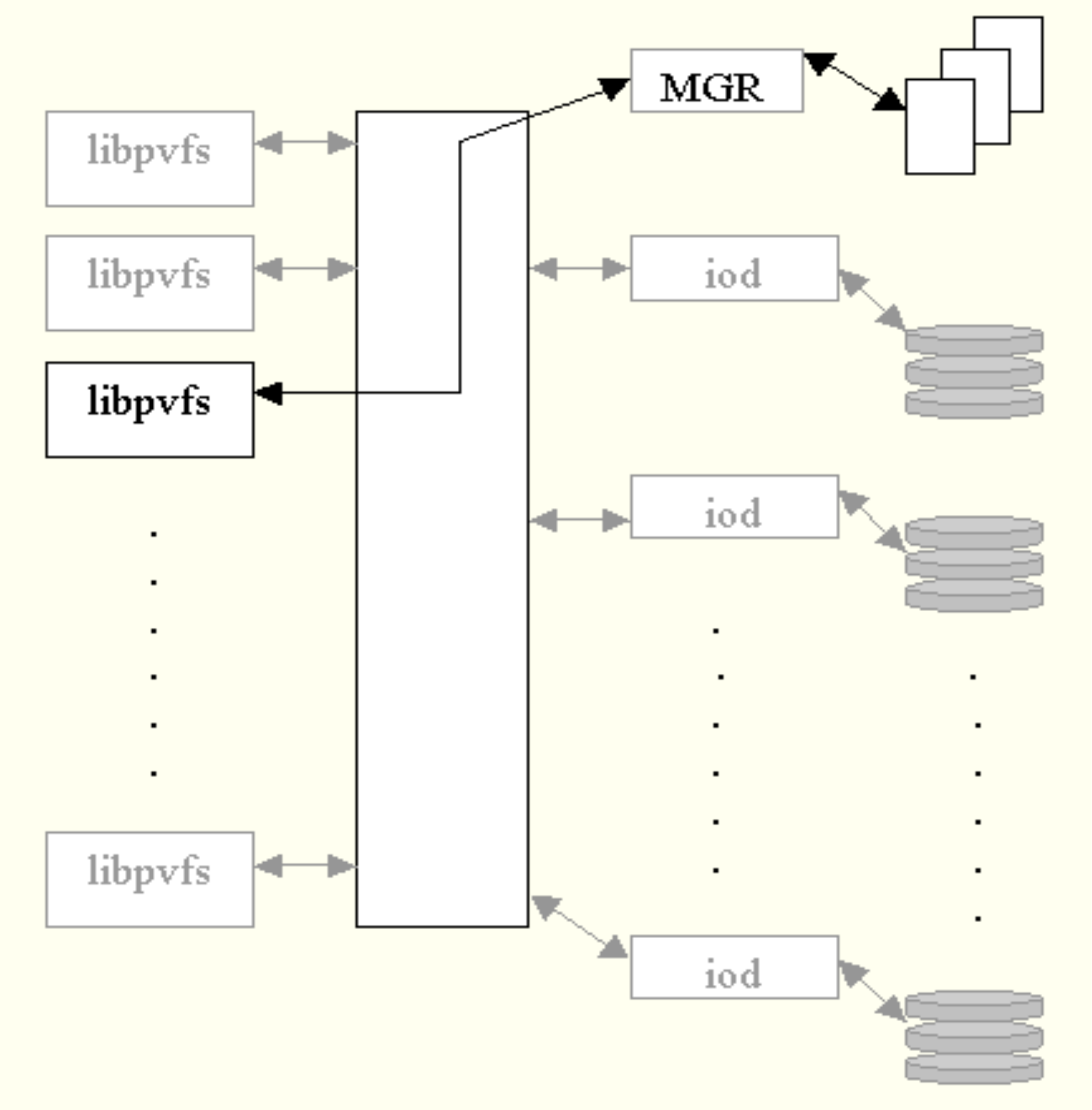
PVFS Components

2. The I/O server (or iod): It handles storing and retrieving file data stored on local disks connected to the node.

3. PVFS native API (libpvfs)

    - It provides user-space access to the PVFS servers
    - libpvfs handles the scatter/gather operations necessary to move data between user buffers and PVFS servers, keeping these operations transparent to the user
    - For metadata operations, applications communicate through the library with the metadata server
    - For data access the metadata server is eliminated from the access path and instead I/O servers are contacted directly. This is key to providing scalable aggregate performance

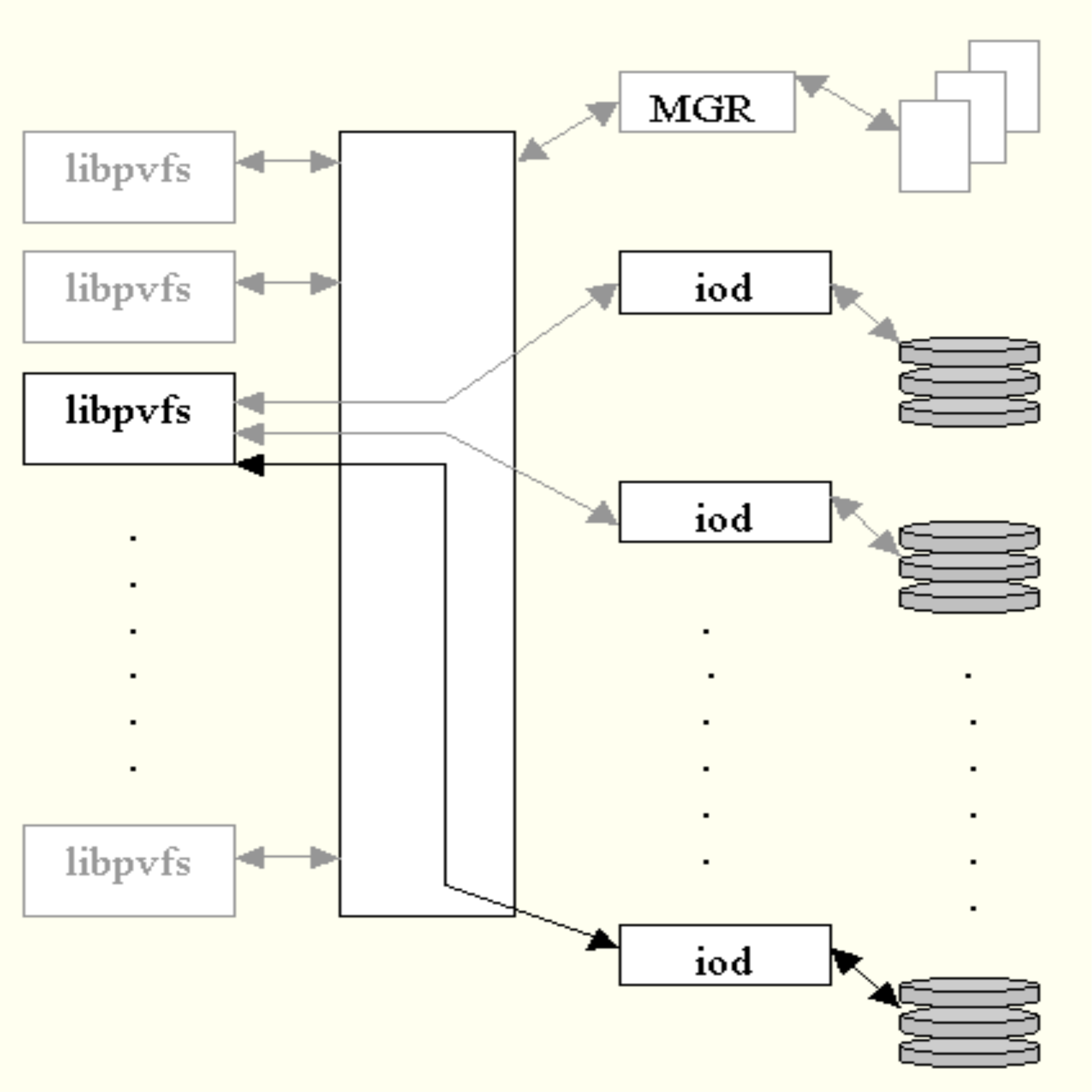
PVFS Components



Metadata access

- For metadata operations applications communicate through the library with the metadata server

PVFS Components



Data access

- Metadata server is eliminated from the access path
- instead I/O servers are contacted directly
- libpvfs reconstructs file data from pieces received from IODS

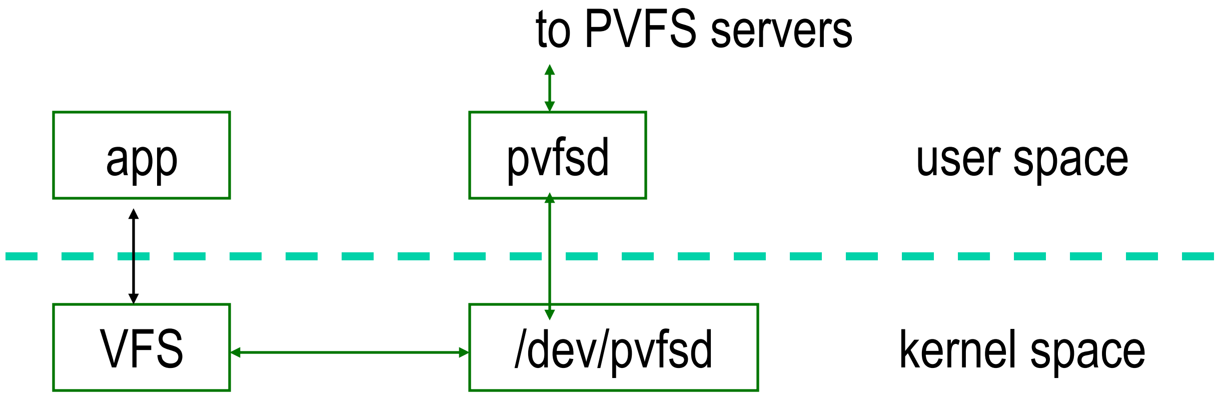
PVFS Components

4. PVFS Linux kernel support

- PVFS Linux kernel support provides the functionality necessary to mount PVFS file systems on Linux nodes
  - allows existing programs to access PVFS files without any modification
  - not necessary for PVFS use by applications, but it provides an extremely convenient means for interacting with the system
- The PVFS Linux kernel support includes:
  - a loadable module
  - an optional kernel patch to eliminate a memory copy
  - a daemon (pvfsd) that accesses the PVFS file system on behalf of applications
- It uses functions from libpvfs to perform these operations.

PVFS Components

Data flow through kernel



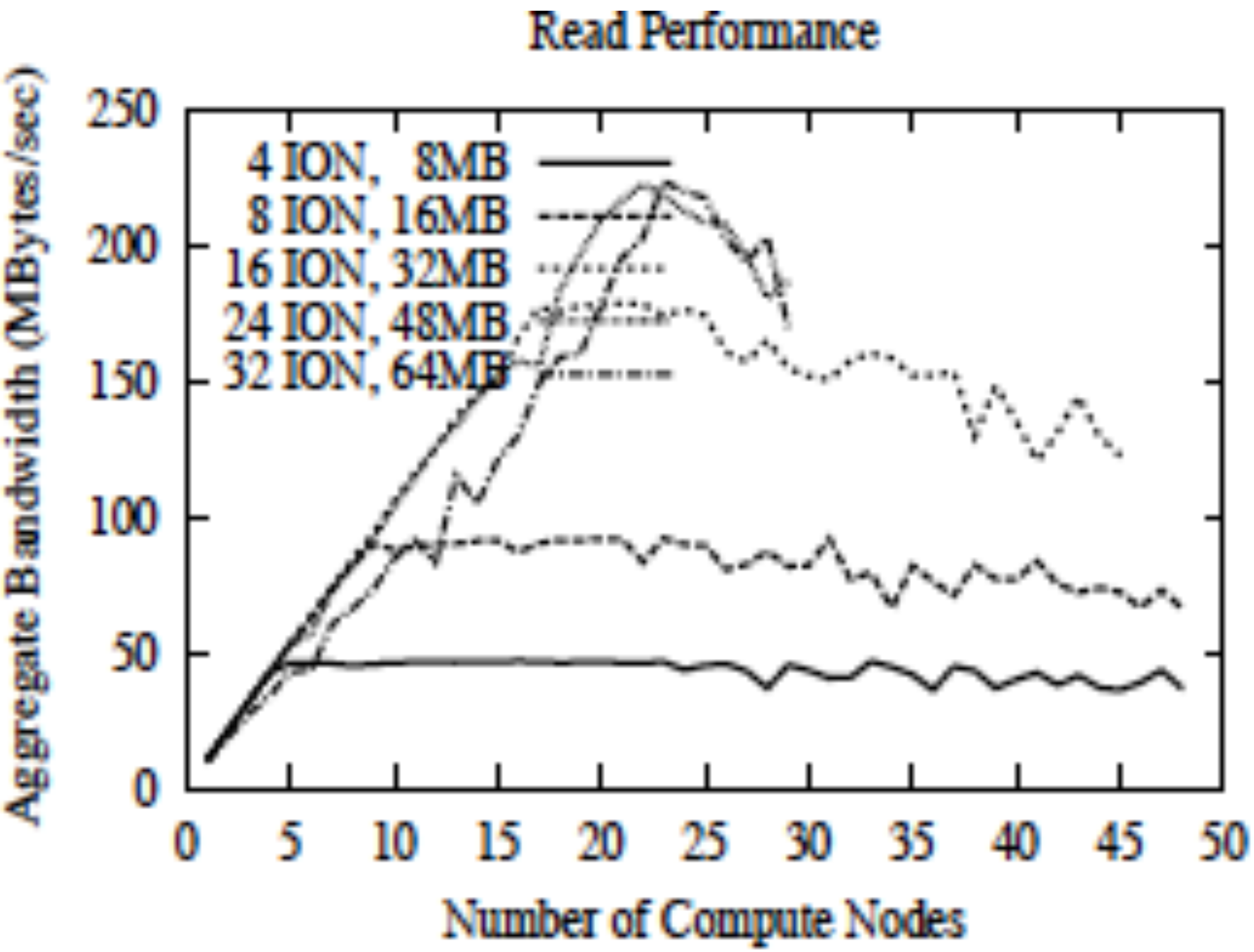
- Operations are passed through system calls to the Linux VFS layer.
- They are queued for service by the pvfsd, which receives operations from the kernel through a device file
- It then communicates with the PVFS servers and returns data through the kernel to the application

PVFS Application Interfaces

Applications can access PVFS data on I/O nodes using:

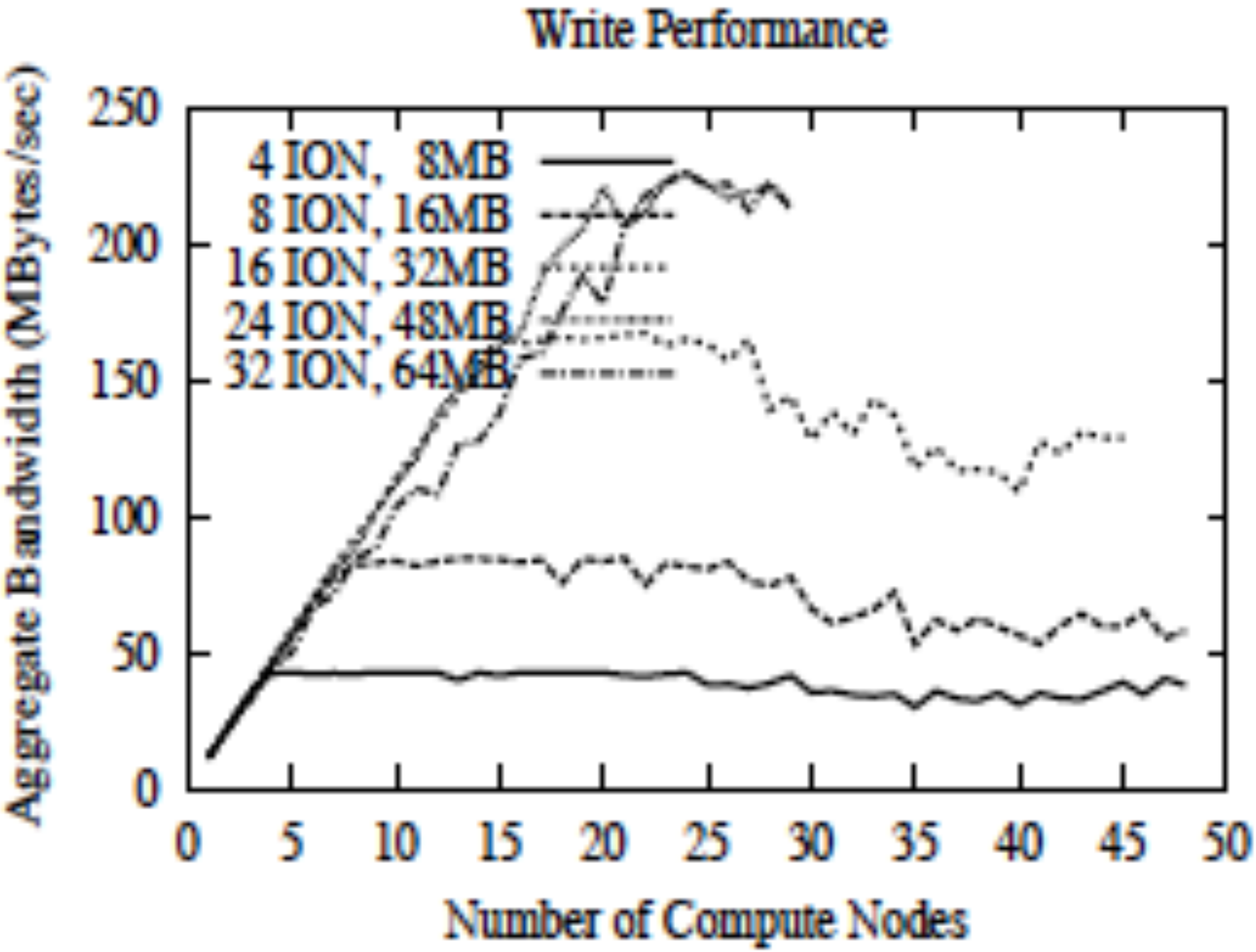
1. PVFS native API
   - Provides a UNIX-like interface for accessing PVFS files.
   - Allows users to specify how files will be striped across the IO nodes in the PVFS system.
2. Linux kernel interface
   - Provides functionality for adding new file-system support via loadable modules without recompiling the kernel.
   - Allow PVFS file systems to be mounted in a manner similar to NFS. Once mounted, the PVFS file system can be traversed and accessed with existing binaries just as any other file system.
3. ROMIO MPI-IO interface
   - ROMIO implements the MPI2 I/O calls in a portable library.
   - Allows parallel programmers using MPI to access PVFS files through the MPI-IO interface

Performance: Fast Ethernet (1)

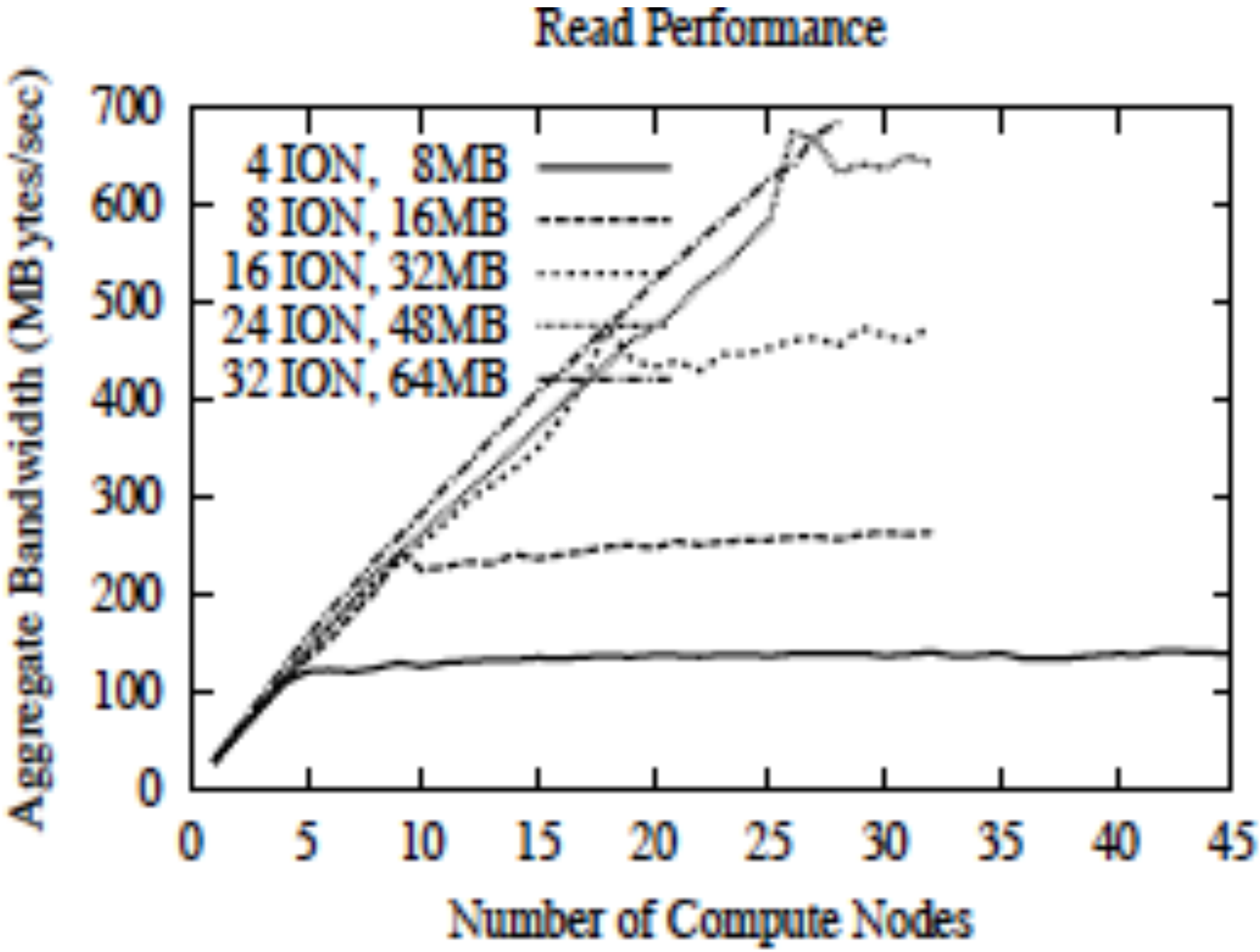


26 Applications accessing 8 IO nodes, each with 16MB for total of 416 MB file

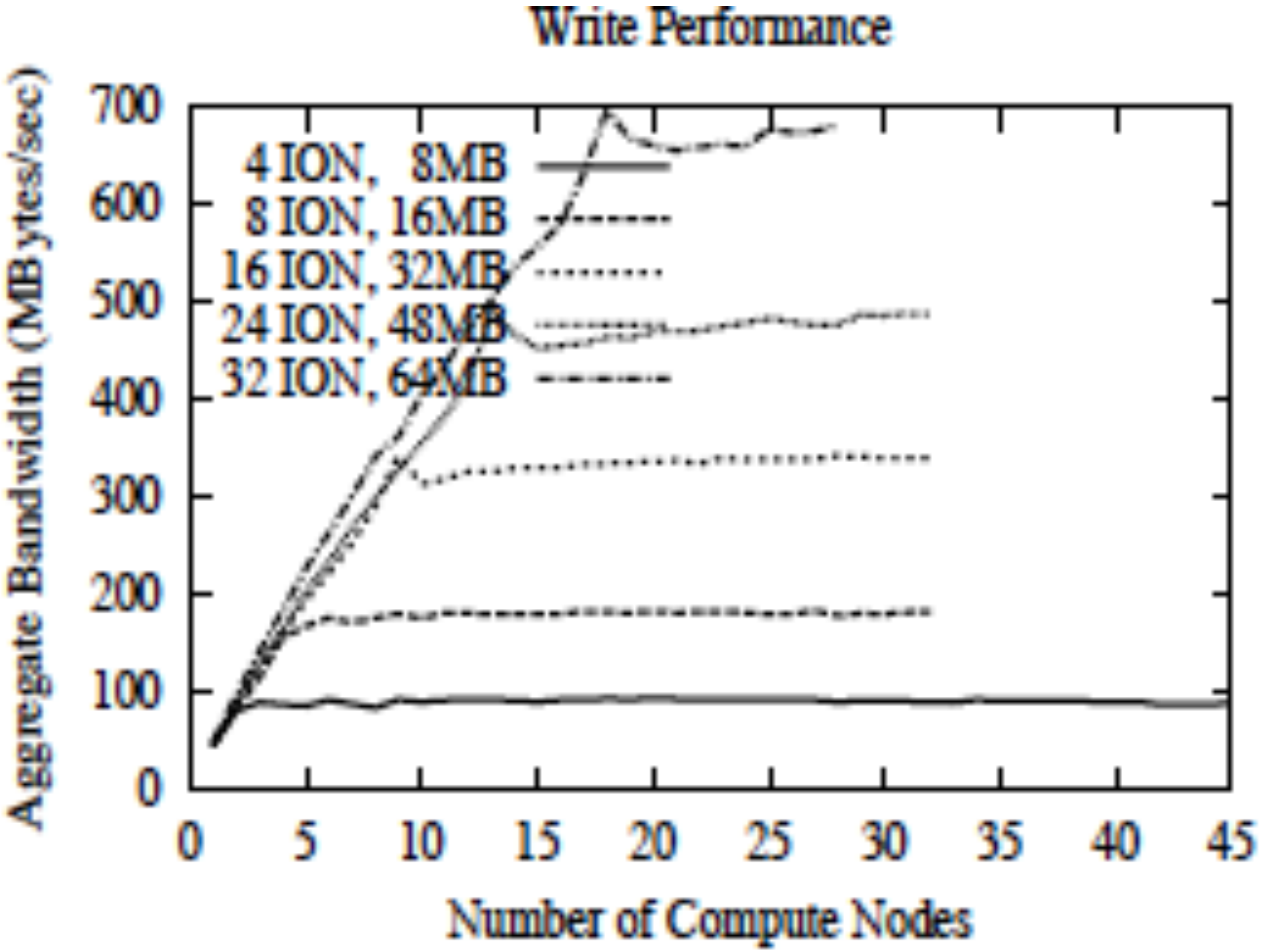
Performance: Fast Ethernet (2)



Performance: Myrinet (3)



Performance: Myrinet (4)



Performance (5)

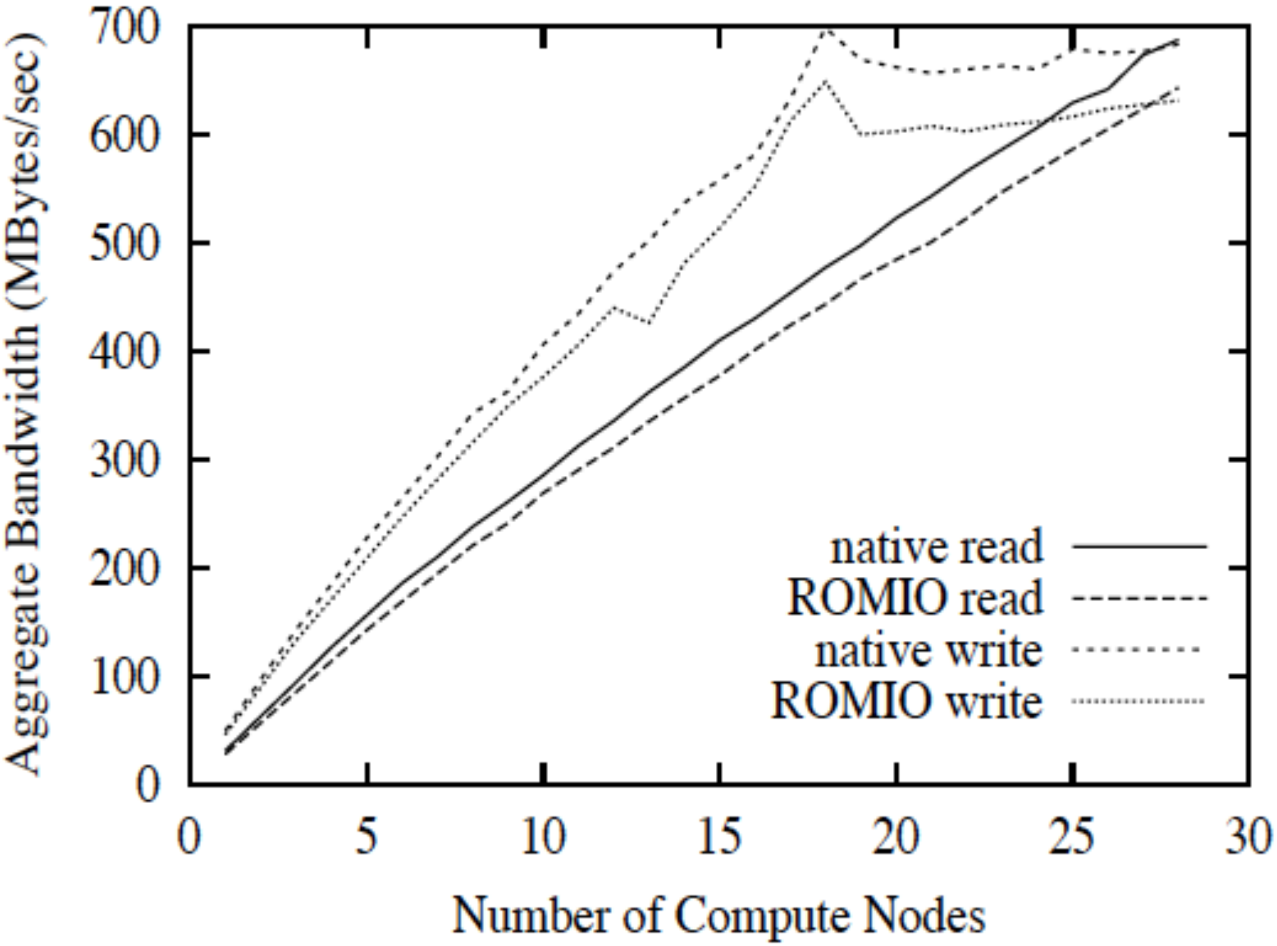


Figure 7: ROMIO versus native PVFS performance with Myrinet and 32 I/O nodes

Performance (6)

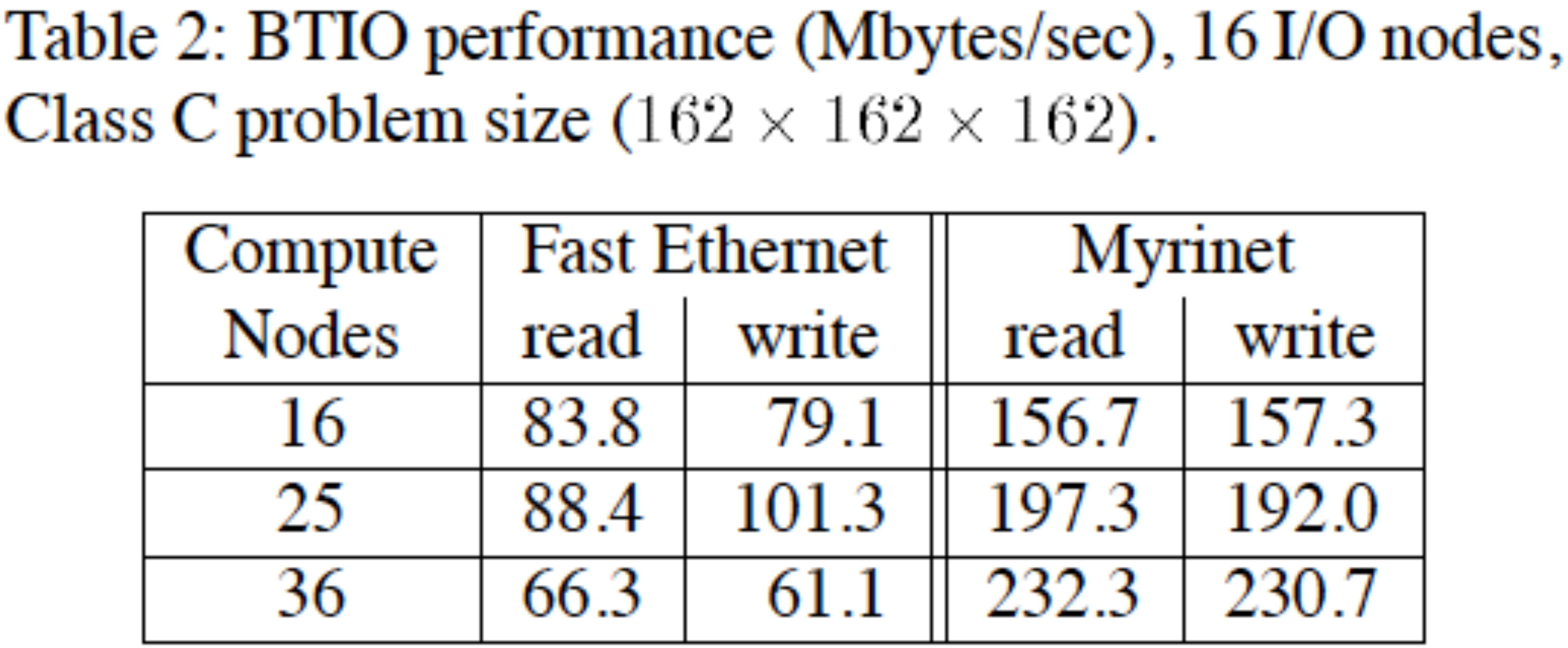


Table 2: BTIO performance (Mbytes/sec), 16 I/O nodes, Class C problem size (162 × 162 × 162).

Compute	Fast Ethernet		Myrinet	
Nodes	read	write	read	write
16	83.8	79.1	156.7	157.3
25	88.4	101.3	197.3	192.0
36	66.3	61.1	232.3	230.7

BTIO is NASA Benchmark

Conclusion for PVFS

Pros:

- Higher cluster performance than NFS.
- Many hard drives to act a one large hard drive.
- Works with current software.
- Best when reading/writing large amounts of data

Cons:

- Multiple points of failure.
- Poor performance when using kernel module.
- Not as good for “interactive” work.

Lustre Parallel File System

- Storage architecture for Clusters – Cluster File System
- Scales to 10s of thousands clients, PBs of data, GB/s IO
- Object based storage
- Software-only solution
- No single point of failure
- 100% POSIX compliant
- Open Source under GNU GPL

Lustre Parallel File System

Lustre components:

- MDS(Meta Data Server):
  - Manages the names and directories in the filesystem, not real data
  - Exports one or more MetaData Targets (MDTs – disk partitions)

- OSS(Object Storage Servers)
  - Provides file IO service; Does real work to store, receive, and send data
  - Exports one or more Object Storage Targets (OSTs – disk partitions)

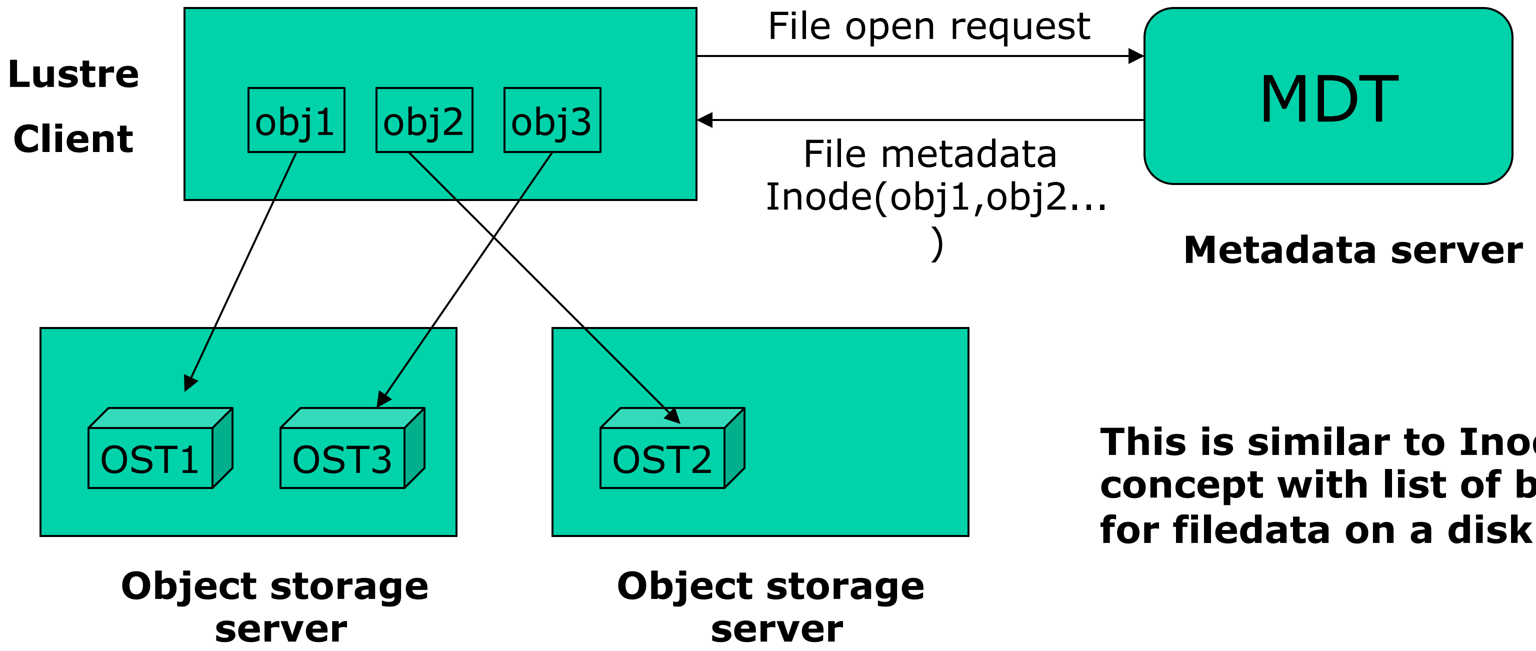
- Lustre Clients
  - Mount and use Filesystem
  - Computation or desktop nodes

Key Design Issue : Scalability

- I/O throughput
  - How to avoid bottlenecks
- Metadata scalability
  - How can 10,000’s of nodes work on files in same folder
- Cluster Recovery
  - Transparent recovery on failure
- Management
  - Adding, removing, replacing, systems; data migration & backup

Lustre features

- Lustre achieves high I/O performance through distributing the data objects across OSTs and allowing clients to directly interact with OSSs

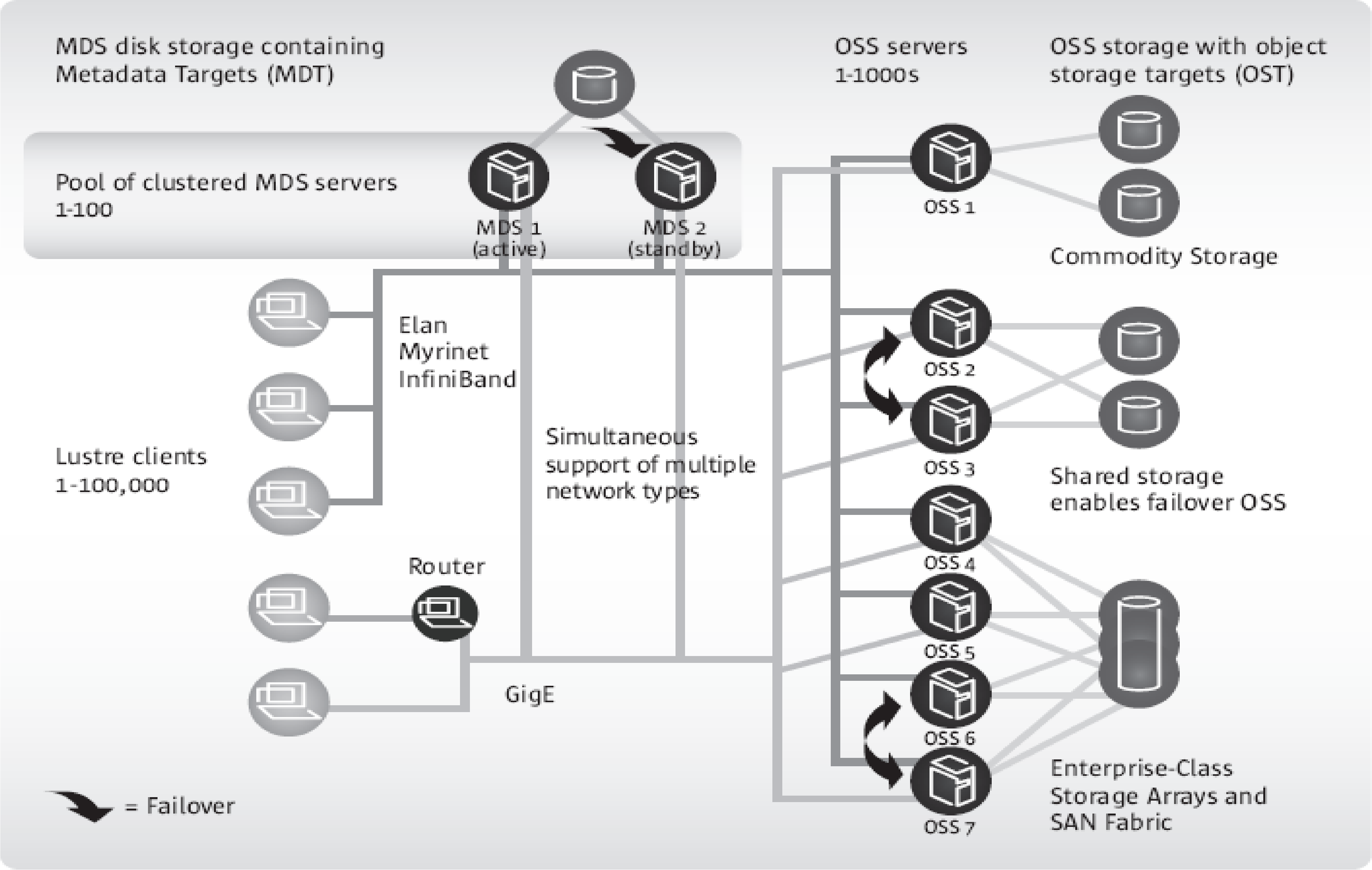


This is similar to Inode concept with list of blocks for filedata on a disk.

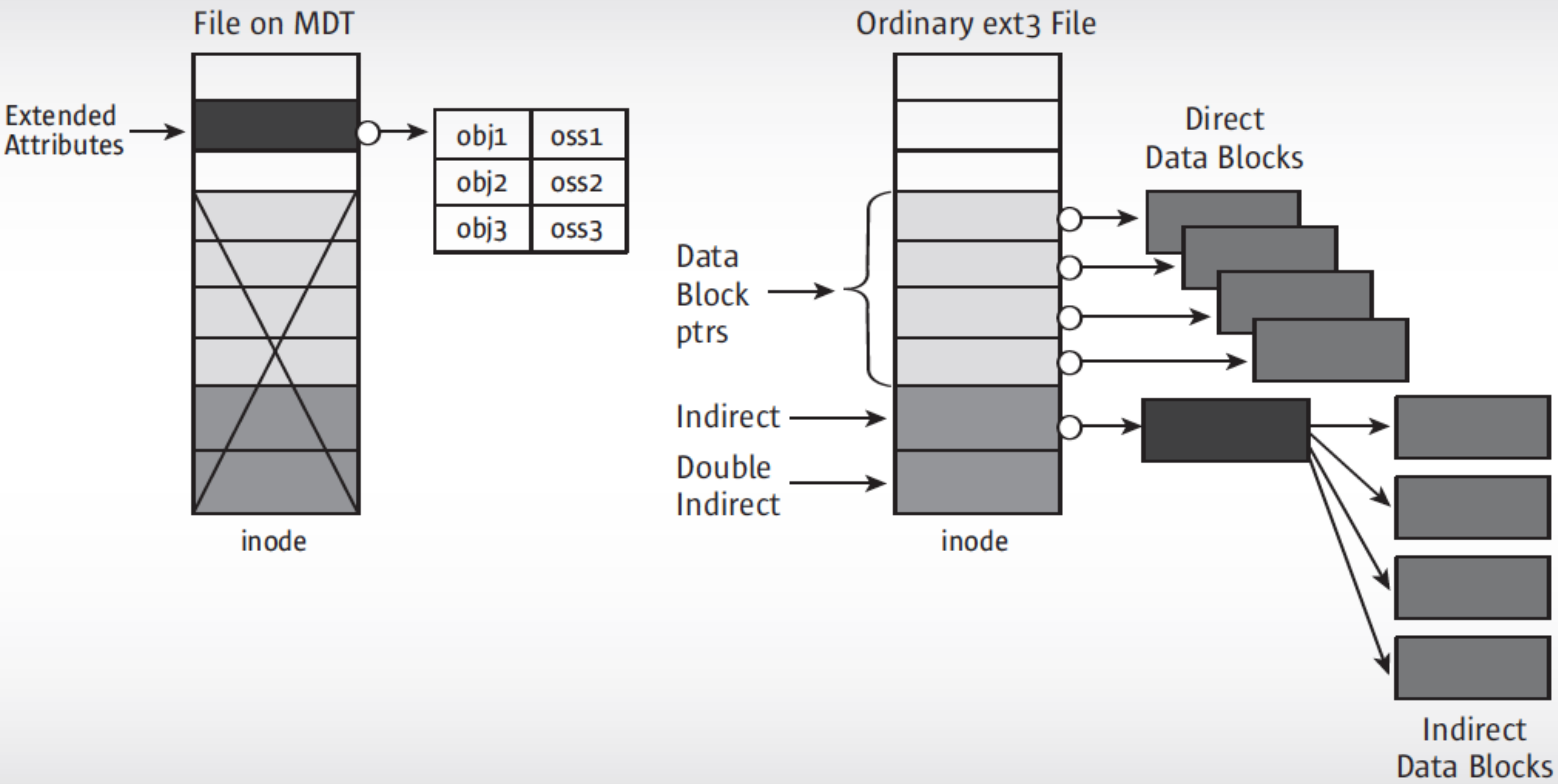
Scalability Example

- File IO as a percent of raw bandwidth: >90%
- Achieved single object storage server IO: > 6GB/s
- Achieved single client IO: > 3GB/s
- Achieved aggregate sustained IO: 1.2TB/s write and 2 TB/s read
- Metadata transaction rate: 60K Ops/s
- Maximum file size: 32PB
- Maximum file system: > 512 PB
- Maximum number of clients: > 50K

Lustre Architecture


MDS disk storage containing Metadata Targets (MDT)
Pool of clustered MDS servers 1-100
MDS 1 (active)
MDS 2 (standby)
OSS servers 1-1000s
OSS storage with object storage targets (OST)
OSS 1
Commodity Storage
Lustre clients 1-100,000
Elan
Myrinet
InfiniBand
Simultaneous support of multiple network types
OSS 2
OSS 3
Shared storage enables failover OSS
OSS 4
OSS 5
OSS 6
OSS 7
Router
GigE
Enterprise-Class Storage Arrays and SAN Fabric
= Failover

Where are the Files?

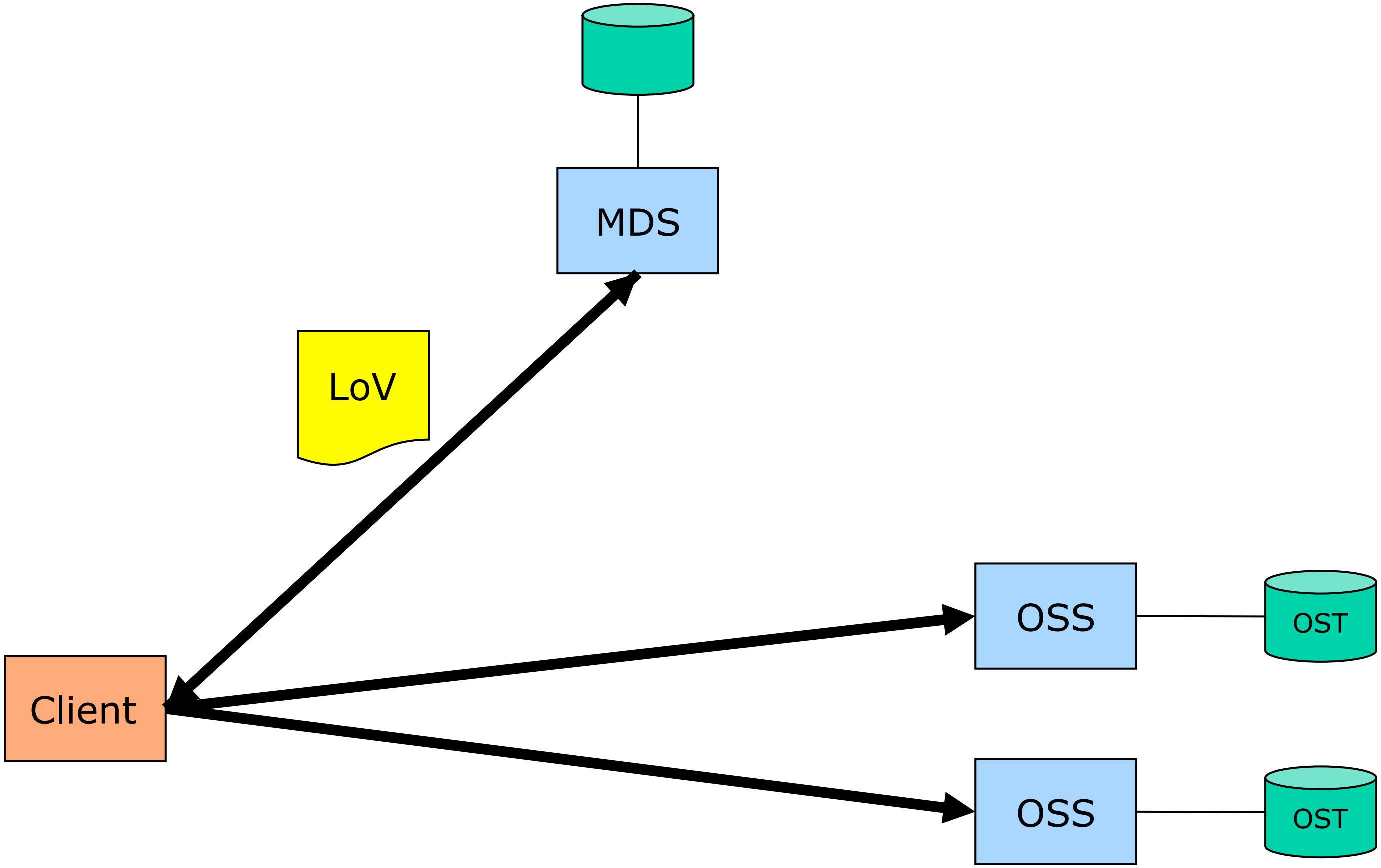


obj1	oss1
obj2	oss2
obj3	oss3

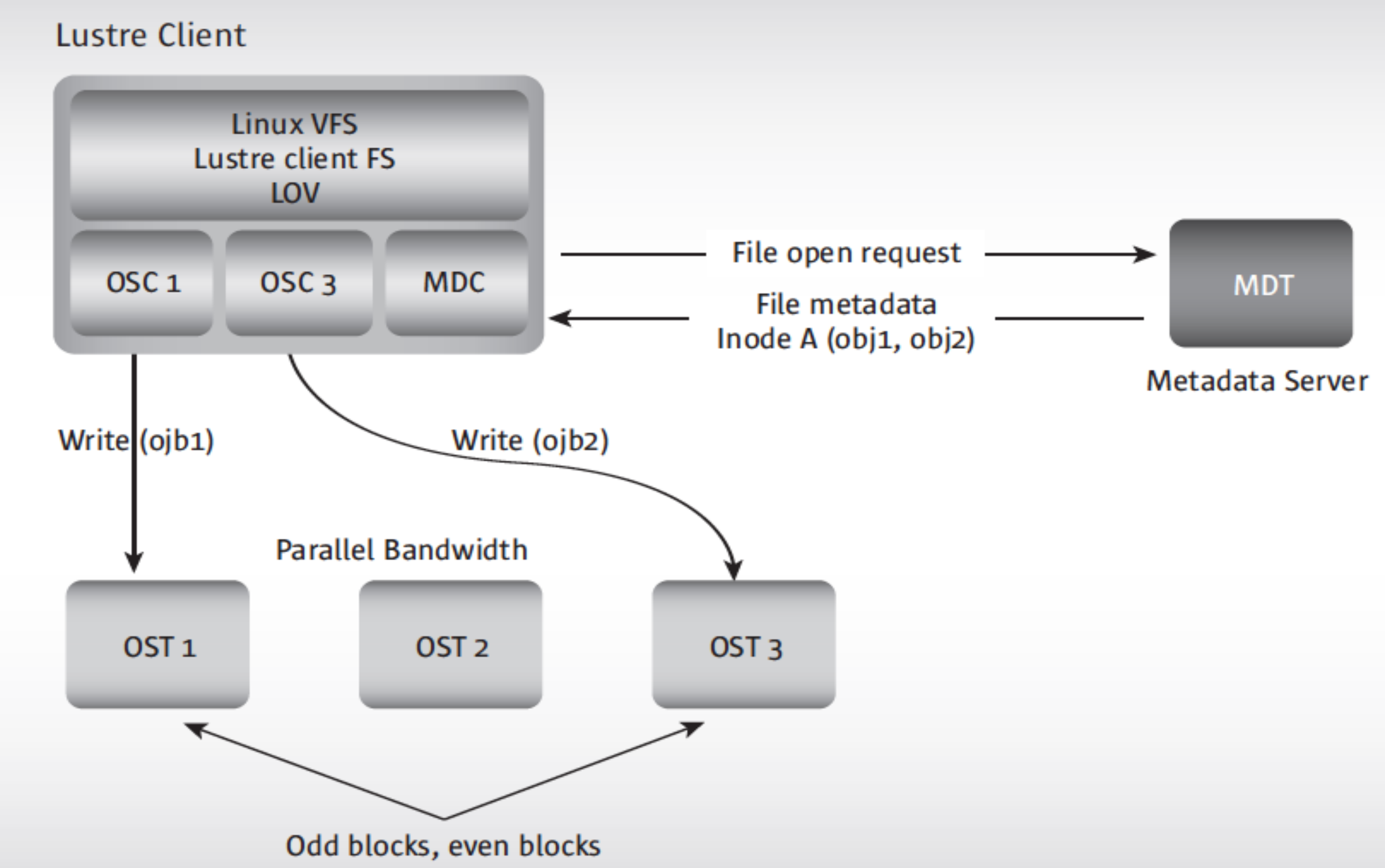
MDS inodes point to objects; ext3 inodes point to data

Interaction between systems

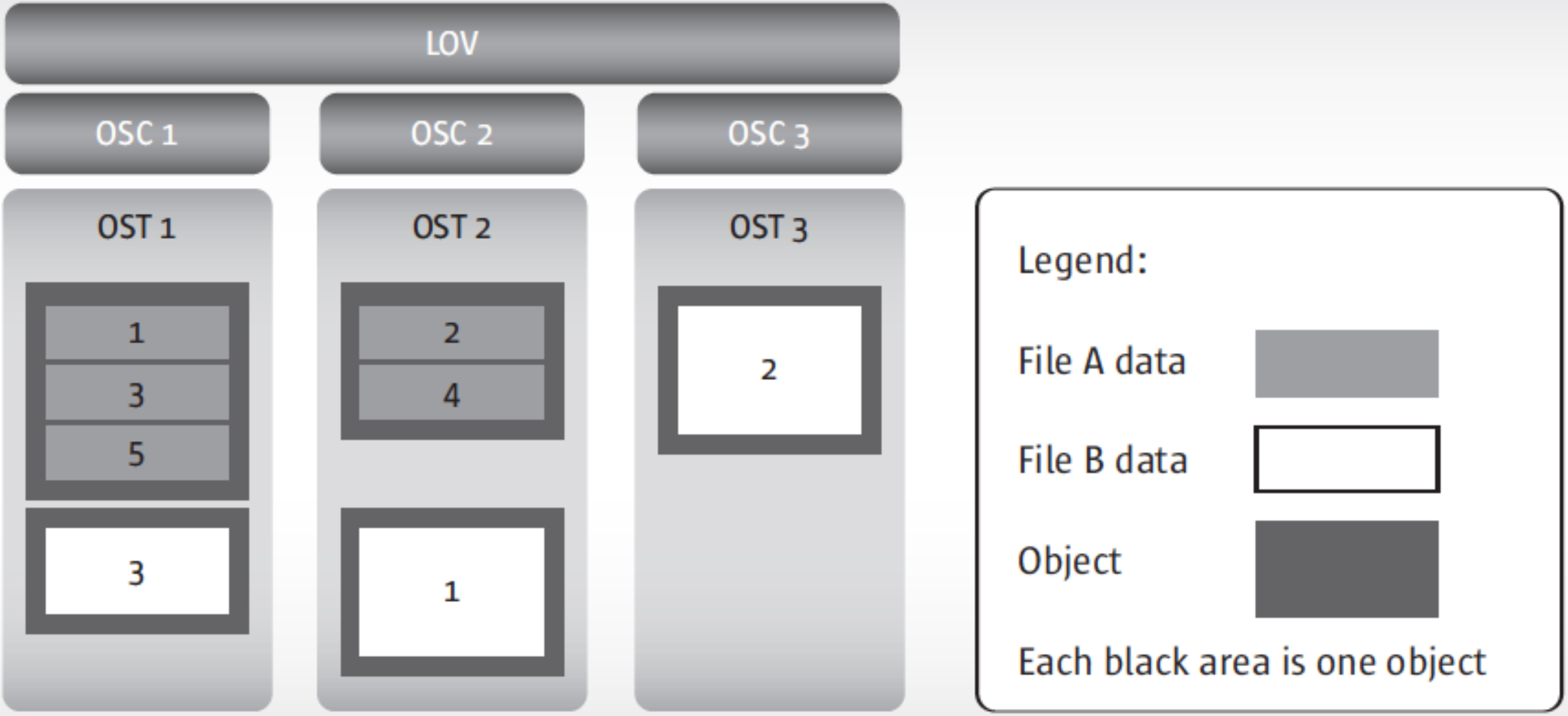
- Clients retrieve layout data from MDS: LoV
- Clients directly communicate with OSS (not OST)



File Open and IO

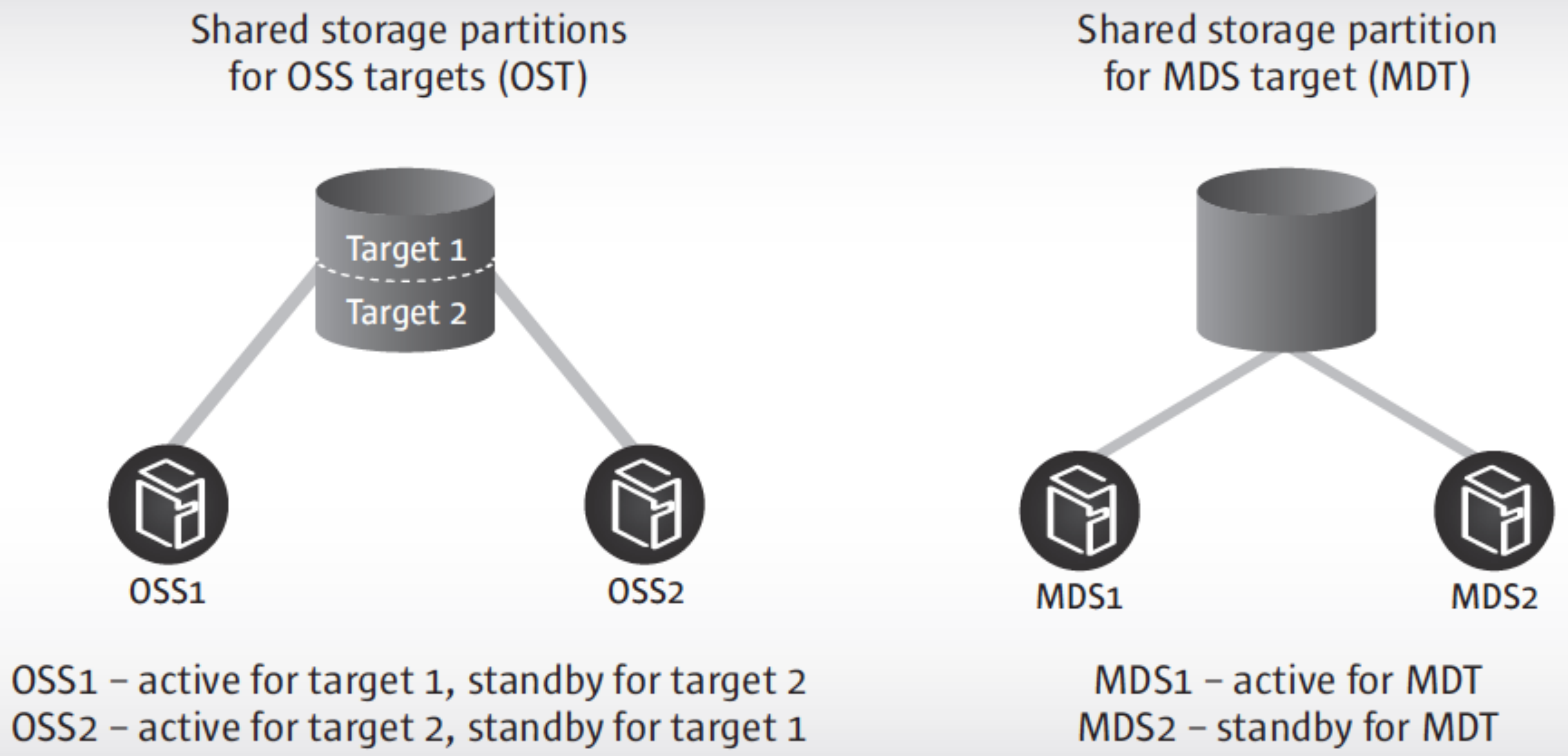


File Open and IO



File striped with a stripe count of 2 and 3
with different stripe sizes

High Availability



Shared storage partitions for OSS targets (OST)

Shared storage partition for MDS target (MDT)

Target 1

Target 2

OSS1

OSS2

MDS1

MDS2

OSS1 – active for target 1, standby for target 2
OSS2 – active for target 2, standby for target 1

MDS1 – active for MDT
MDS2 – standby for MDT

- OSS is active/active
- MDS is active/passive

Adaptive Timeouts

- On large clusters (>10K nodes) extreme server load is indistinguishable from server down
- Modify RPC timeouts based on server load
  - Timeouts increase as server load increases and vice-versa
- Track service time histories on all servers and estimates for future RPCs are reported back to clients
- Server sends early replies to clients asking for more time if RPCs queued on the server near their timeouts
- Adaptive Timeout Benefits:
  - Relieves users from having to tune the obd_timeout value
  - Reduces RPC timeouts and disconnect/reconnect cycles

OST Pools

- Allows administrator to name a group of OSTs for file striping
- Pools can separate heterogeneous OSTs within the same filesystem
  - Fast vs slow disks
  - Local network vs remote network
  - Type of RAID storage
  - Specific OSTs for user/group/application
- Pool usage defined and stored along with rest of striping info
- OSTs can be added and removed from a pool at any time and can be associated with multiple pools.

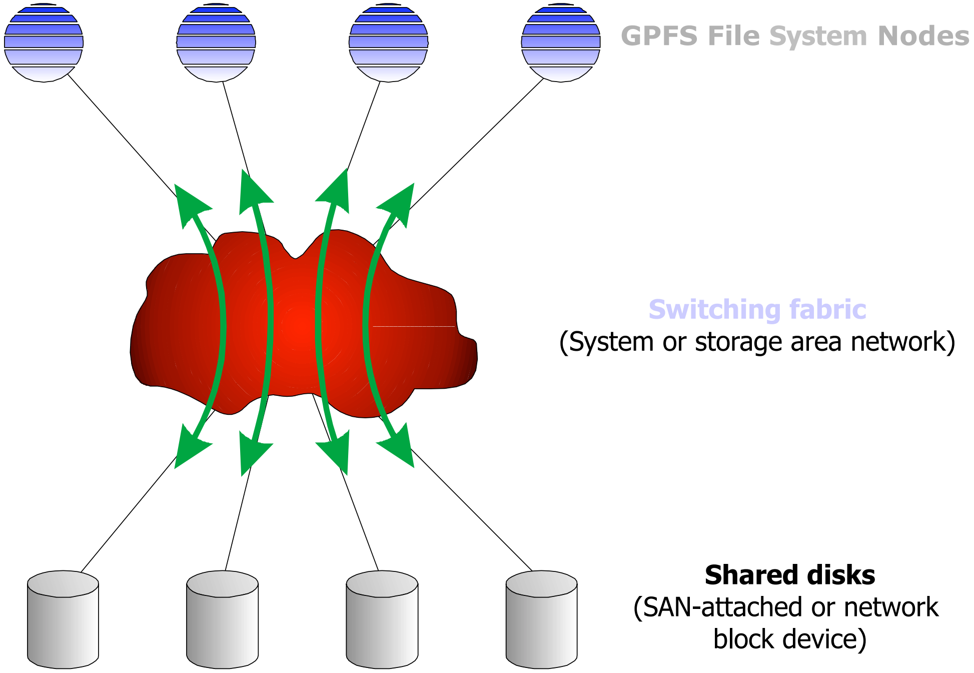
GPFS

Extreme Scalability	Proven Reliability	Performance
File system		
2^{63} files per file system	No Special Nodes	High Performance Metadata
Maximum file system size: 2^{99} bytes	Add/remove on the fly	Striped Data
Maximum file size equals file system size	Nodes	Equal access to data
	Storage	Integrated Tiered storage
Production 5.4 PB file system	Rolling Upgrades	Quotas
Number of nodes	Administer from any node	
1 to 8192	Data replication	
	Journaling, snapshots	

GPFS: parallel file access

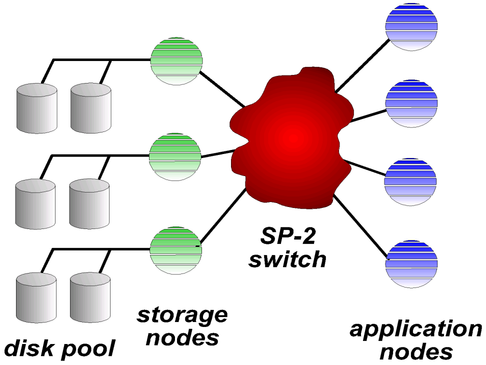
Parallel Cluster File System Based on Shared Disk (SAN) Model

- *Cluster* - fabric-interconnected nodes (IP, SAN, …)
- *Shared disk* - all data and metadata on fabric-attached disk
- *Parallel* - data and metadata flows from all of the nodes to all of the disks in parallel under control of distributed lock manager.


GPFS File System Nodes
Switching fabric
(System or storage area network)
Shared disks
(SAN-attached or network block device)

Shared Disks - Virtual Shared Disk architecture

- **File systems consist of one or more shared disks**
  - Individual disk can contain data, metadata, or both
  - Disks are designated to failure group
  - Data and metadata are striped to balance load and maximize parallelism
- **Recoverable Virtual Shared Disk for accessing disk storage**
  - Disks are physically attached to SP nodes
  - VSD allows clients to access disks over the SP switch
  - VSD *client* looks like disk device driver on client node
  - VSD *server* executes I/O requests on storage node.
  - VSD supports JBOD or RAID volumes, multi-pathing (where physical hardware permits)
- **GPFS only assumes a conventional block I/O interface**



General architecture

GPFS Architecture Overview

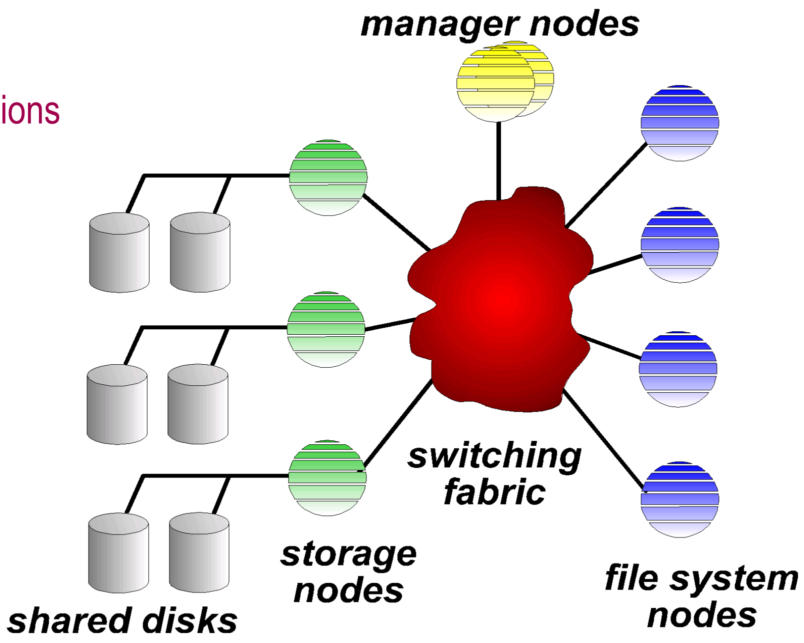
- **Implications of Shared Disk Model**
  - All data and metadata on globally accessible disks (VSD)
  - All access to permanent data through disk I/O interface
  - Distributed protocols, e.g., distributed locking, coordinate disk access from multiple nodes
  - Fine-grained locking allows parallel access by multiple clients
  - Logging and Shadowing restore consistency after node failures

- **Implications of Large Scale**
  - Supports very large systems
  - Failure detection and recovery protocols to handle node failures
  - Replication and/or RAID protect against disk / storage node failure
  - On-line dynamic reconfiguration (add, delete, replace disks and nodes; rebalance file system)

General architecture

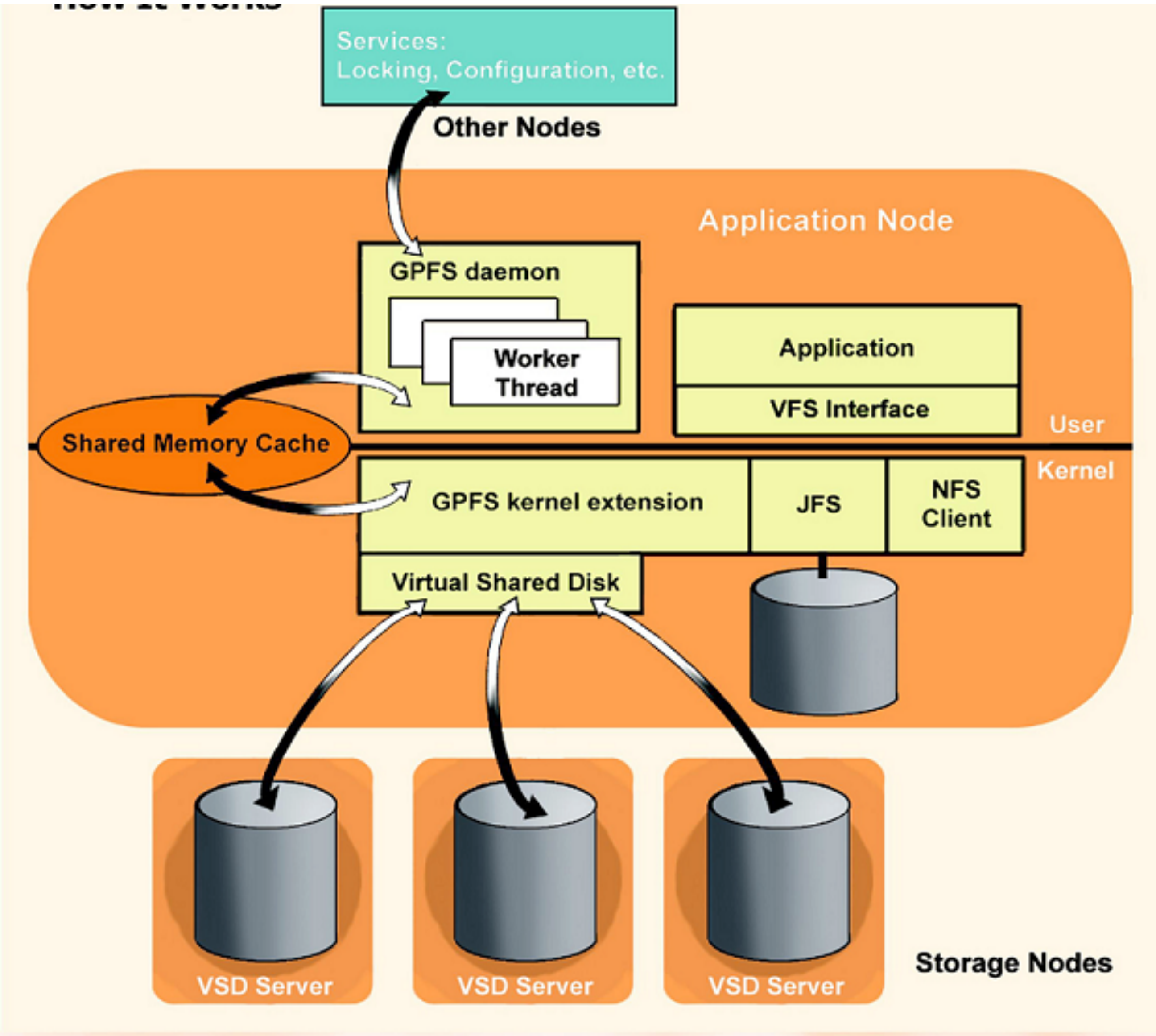
GPFS Architecture - Node Roles

- **Three types of nodes:** file system, storage, and manager
  - Each node can perform any of these functions
  - File system nodes
    - run user programs, read/write data to/from storage nodes
    - implement virtual file system interface
    - cooperate with manager nodes to perform metadata operations
  - Manager nodes (one per "file system")
    - global lock manager
    - recovery manager
    - global allocation manager
    - quota manager
    - file metadata manager
    - admin services fail over
  - Storage nodes
    - implement block I/O interface
    - shared access from file system and manager nodes
    - interact with manager nodes for recovery (e.g. fencing)
    - file data and metadata striped across multiple disks on multiple storage nodes



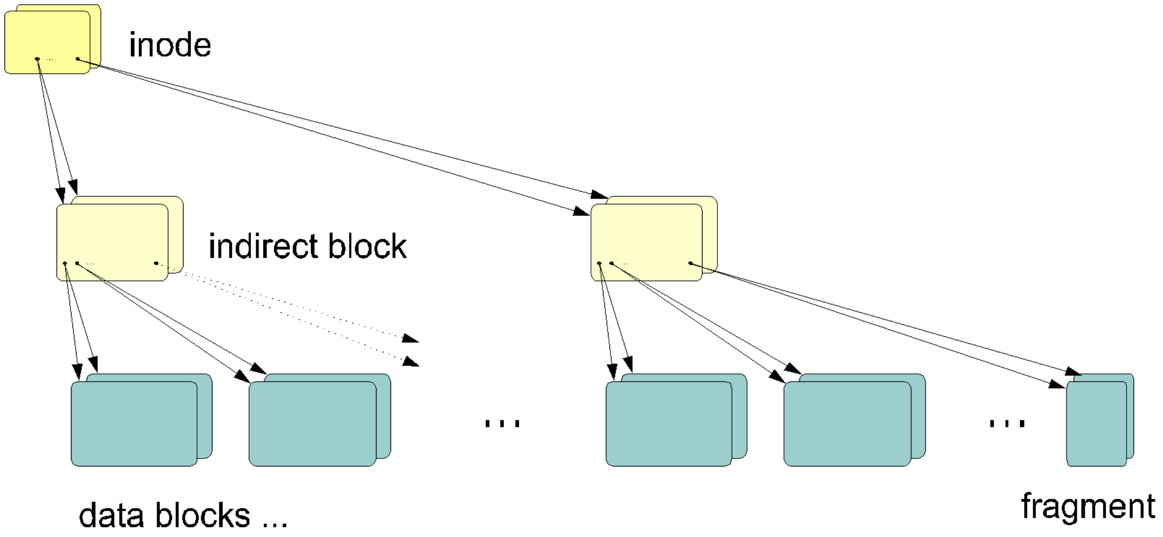
General architecture

GPFS Software Structure



Disk Data Structures: Files

- Large block size allows efficient use of disk bandwidth
- Fragments reduce space overhead for small files
- No designated "mirror", no fixed placement function:
  - Flexible replication (e.g., replicate only metadata, or only important files)
  - Dynamic reconfiguration: data can migrate block-by-block
  - Multi level indirect blocks



Each disk address:

- list of pointers to replicas

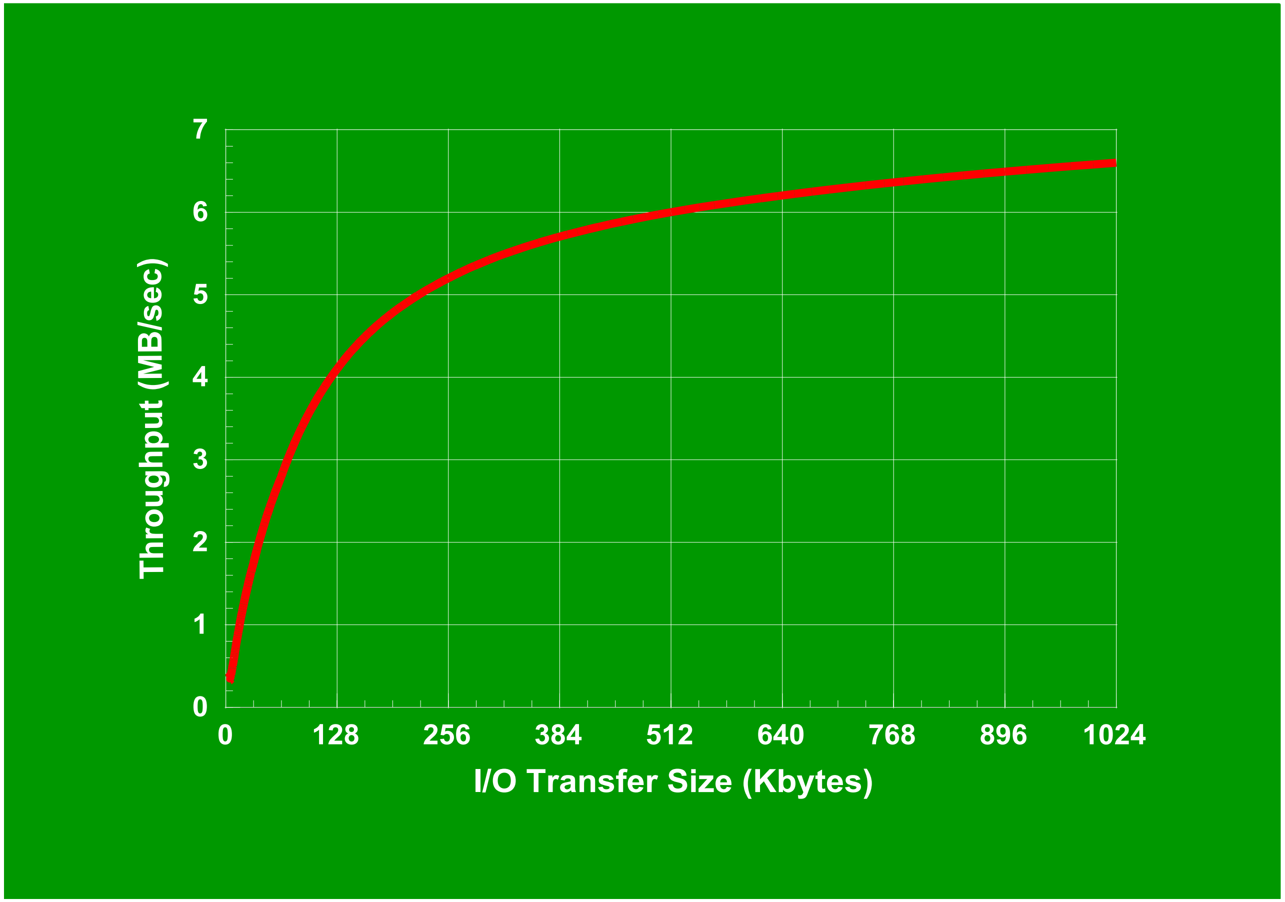
Each pointer:

- disk id + sector no.

General architecture

Large File Block Size

- Conventional file systems store data in small blocks to pack data densely
- GPFS uses large blocks (256KB default) to optimize disk transfer speed



Performance

Parallelism and consistency

- **Distributed locking** - acquire appropriate lock for every operation - used for updates to user data
- **Centralized management** - conflicting operations forwarded to a designated node - used for file metadata
- **Distributed locking + centralized hints** - used for space allocation
- **Central coordinator** - used for configuration changes

I/O slowdown effects
Additional I/O activity
rather than token server
Overload as shown in
this Byte Range lock

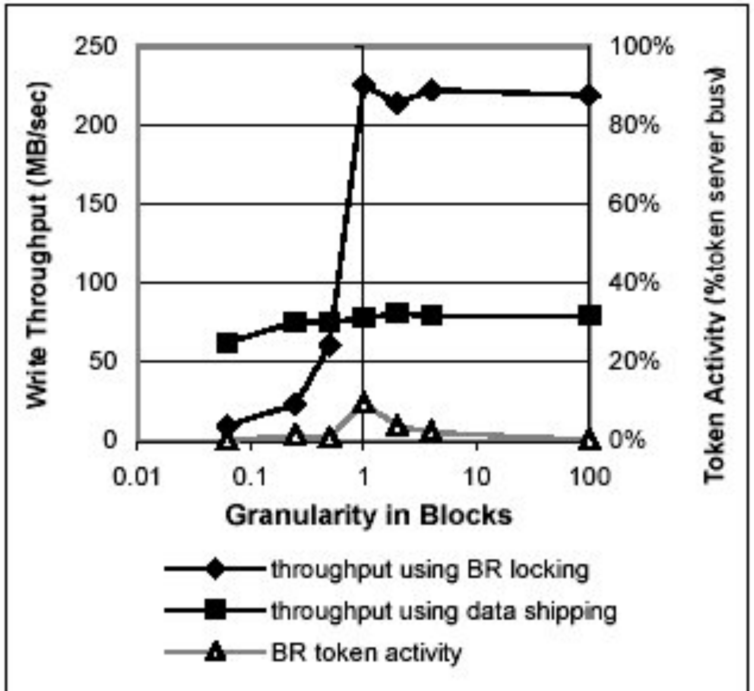
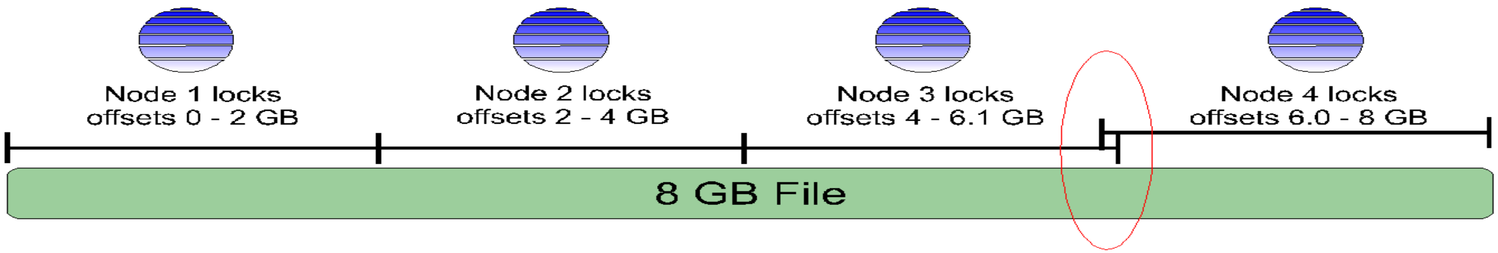


Figure 3: Effect of Sharing Granularity on Write Throughput

Parallel File Access From Multiple Nodes

- GPFS allows parallel applications on multiple nodes to access non-overlapping ranges of a single file with no conflict
- Global locking serializes access to overlapping ranges of a file
- Global locking based on "tokens" which convey access rights to an object (e.g. a file) or subset of an object (e.g. a byte range)
- Tokens can be held across file system operations, enabling coherent data caching in clients
- Cached data discarded or written to disk when token is revoked
- Performance optimizations: required/desired ranges, metanode, data shipping, special token modes for file size operations

Parallel Application on Nodes 1-4



Node 1 locks offsets 0 - 2 GB

Node 2 locks offsets 2 - 4 GB

Node 3 locks offsets 4 - 6.1 GB

Node 4 locks offsets 6.0 - 8 GB

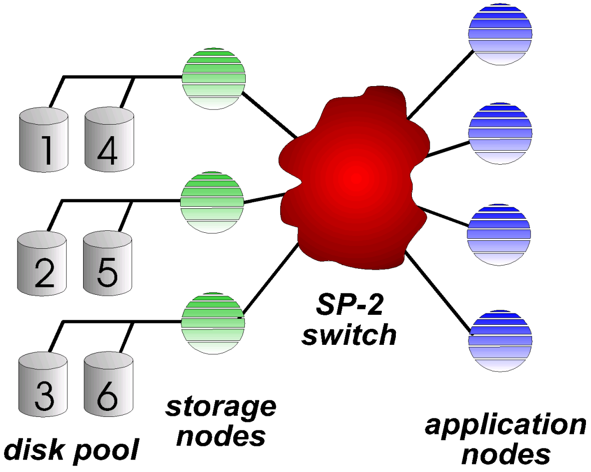
8 GB File

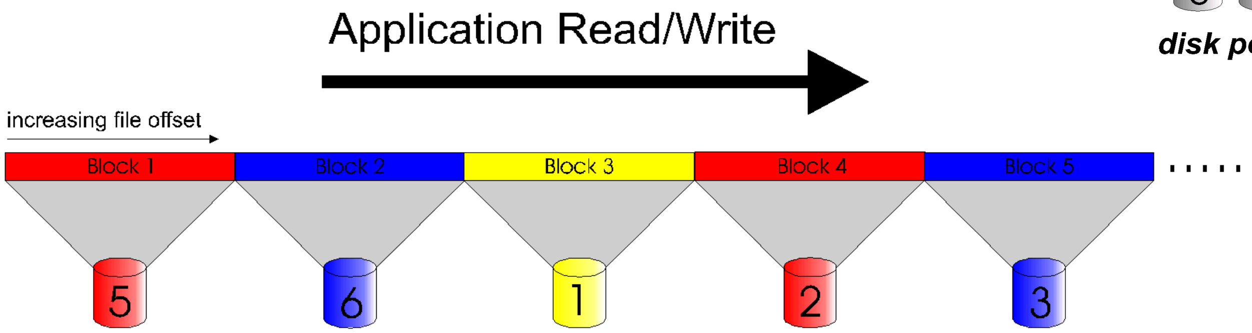
File offsets 0 - 6.0 GB and 6.1 - 8.0 GB are accessed in parallel with no conflict

Conflict for 6.0-6.1 GB detected by lock manager, only one node may access this region at any time.

Deep Prefetch for High Throughput

- GPFS stripes successive blocks across successive disks
- Disk I/O for sequential reads and writes is done in parallel
- GPFS measures application "think time" ,disk throughput, and cache state to automatically determine optimal parallelism
- Accepts hints
- Write-behind policy





Application reads at 15 MB/sec
Each disk reads at 5 MB/sec

Three I/Os executed in parallel

Thank you