Operating Systems
Distributed File Systems

Shyan-Ming Yuan

CS Department, NCTU

smyuan@gmail.com

Office hours: Wed. 8~10 (EC442)

The Concept of Files

- A file is a named object in a computer system.
  - It is immune to temporary failures in the system.
  - It persists until explicitly destroyed.
- Files are used for the following purposes:
  - Permanent storage of information.
    - Files are stored on disks or other non-volatile storage media.
    - Files contain both data and metadata.
      - The data consists of a sequence of items that are accessible by operations such as read and write.
      - The metadata contains information such as the length of the file, timestamps, file type, owner's identity, access control lists, and more.
  - Sharing of information.
    - A file can be created by one application and then shared with different applications at a later time.

The Concept of File Systems

- A file system is a subsystem of an OS.
  - It is responsible for the organization, storage, retrieval, naming, sharing and protection of files.
  - File APIs are provided to free programmers from the details of storage allocation and layout.
- A file system also provides facilities for creating, naming and deleting files.
  - The file naming is usually supported by the use of directories.
- A directory is a special file that provides a mapping from text names to internal file identifiers.
- The metadata is referring to all of the extra information required by a file system for the management of files.

Conventional File System Layers

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Conventional File APIs

handler = open (name, mode)	Opens an existing file with the given name
handler = create (name, mode)	Creates a new file with the given name
	Both return a file handler referencing the opened file The mode may be read, write or both
read (handler, buffer, n)	Reads n bytes from the opened file into the buffer
write (handler, buffer, n)	Writes n bytes to the opened file from the buffer
lseek (handler, offset)	Moves the read-write pointer to the position offset
link (name1, name2)	Adds a new name (name2) to an existing file (name1)
unlink (name)	Removes the given name from the directory If the file has no other names, it is deleted
getattribute (handler, buffer)	Puts the file attributes of the opened file into the buffer
close (handler)	Close the opened file

Distributed File System Requirements (1)

- Important Transparencies:
  - Access transparency: client programs should not be aware of the distribution of files.
    - The same APIs are provided for accessing local and remote files.
  - Location transparency: client programs should see an uniform file name space.
    - Files may be relocated without changing their names.
    - Programs see the same name space wherever they are executed.
  - Mobility transparency: both client programs and administration configurations in client nodes are unchanged on file relocations.
  - Performance transparency: the performance of client programs should not be affected when the workload varies within a small range.
  - Scaling transparency: the DFS can be semi-linearly adjusted to deal with a wide range of loads and network sizes.

Distributed File System Requirements (2)

- Concurrent File Updates
  - Simultaneously accessing or changing the same file by more than one clients should not interfere with each other.
- File Replications
  - A file may have several copies stored at different locations.
    - It enhances scalability by enabling multiple servers to handle file accesses of the same file from different clients.
    - It enhances fault tolerance by enabling clients to locate other copies of the same file when one has failed.
- Hardware and OS Heterogeneity
  - The DFS APIs should be defined so that client and server software can be implemented for different OSs and computers.
  - This requirement is an important aspect of openness.

Distributed File System Requirements (3)

- Fault tolerance
  - It is essential that a DFS continues to operate in the face of client and server failures.
- File Sharing Semantics
  - One-copy update semantics (UNIX)
  - Session update semantics (AFS)
  - Append-only update semantics (HDFS)
  - Copy-on-write update semantics
- Security
- Efficiency
  - The performance of a DFS should be comparable to that of conventional file systems.

Naming of Distributed Files

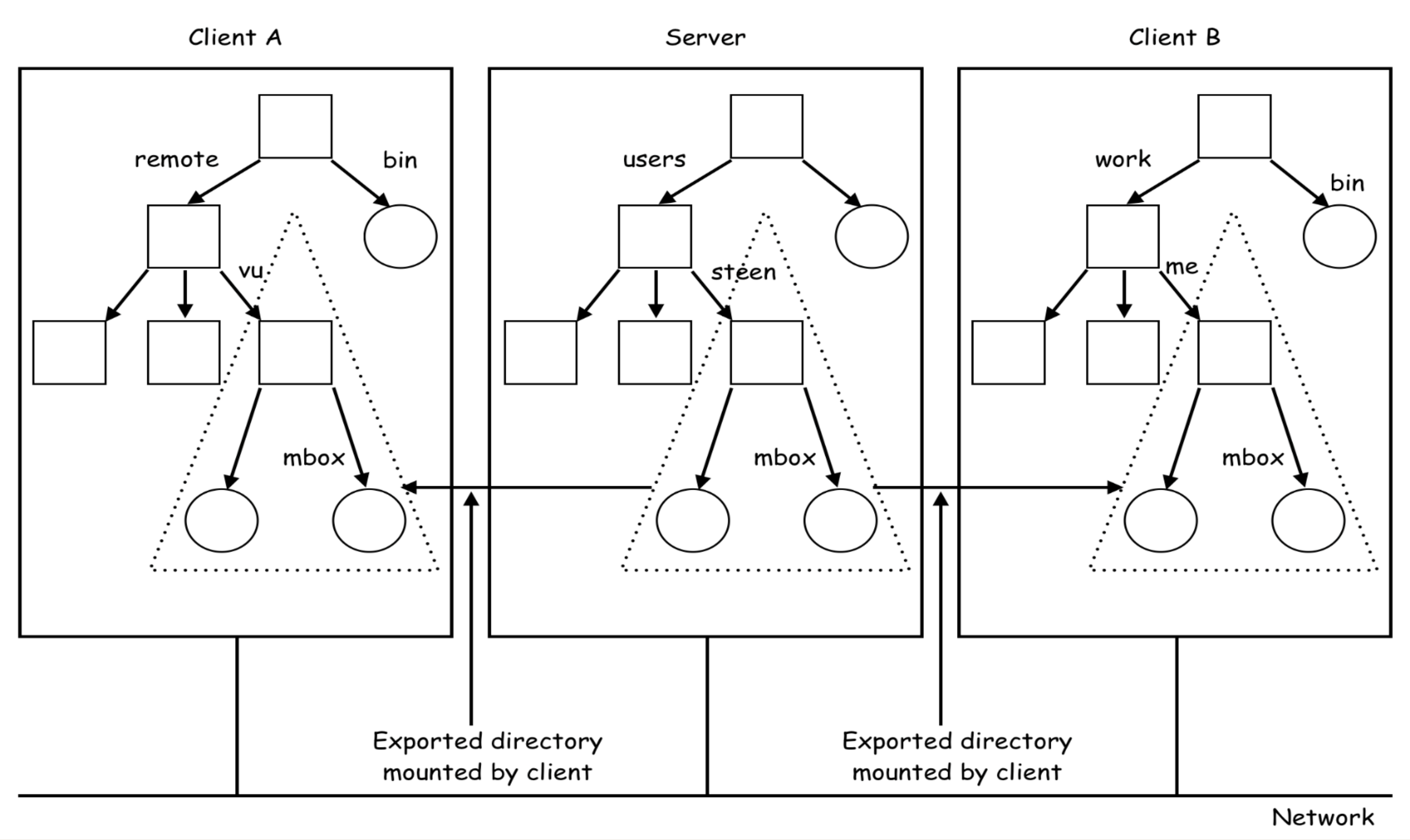
- Naming mechanism is handling the mapping between a logical file name and physical location of a file.
- A transparent DFS should hide the file location.
  - Location transparency: a file name should not reveal the file's physical storage location.
  - Location independence: a file name should not be changed when the file's physical storage location changes.
    - It provides better file abstraction.
    - It separates the naming hierarchy from the storage devices hierarchy.

Naming Schemes for DFSs

- Mount remote directories to local directories.
  - It gives the appearance of a coherent local directory tree.
  - Mounted remote directories can be accessed transparently.
  - Examples: NFS in Unix/Linux and mapped drives in MS Windows.
- Files named by combination of host name and local name.
  - It guarantees a system wide unique name.
  - Examples: Windows Network Places and Apollo Domain.
- Total integration of component file systems.
  - A single global name structure spans all the files in the system.

Mounting Remote Directories (NFS)

- File names are not unique in the system.
- File "names" cannot be passed around arbitrarily.

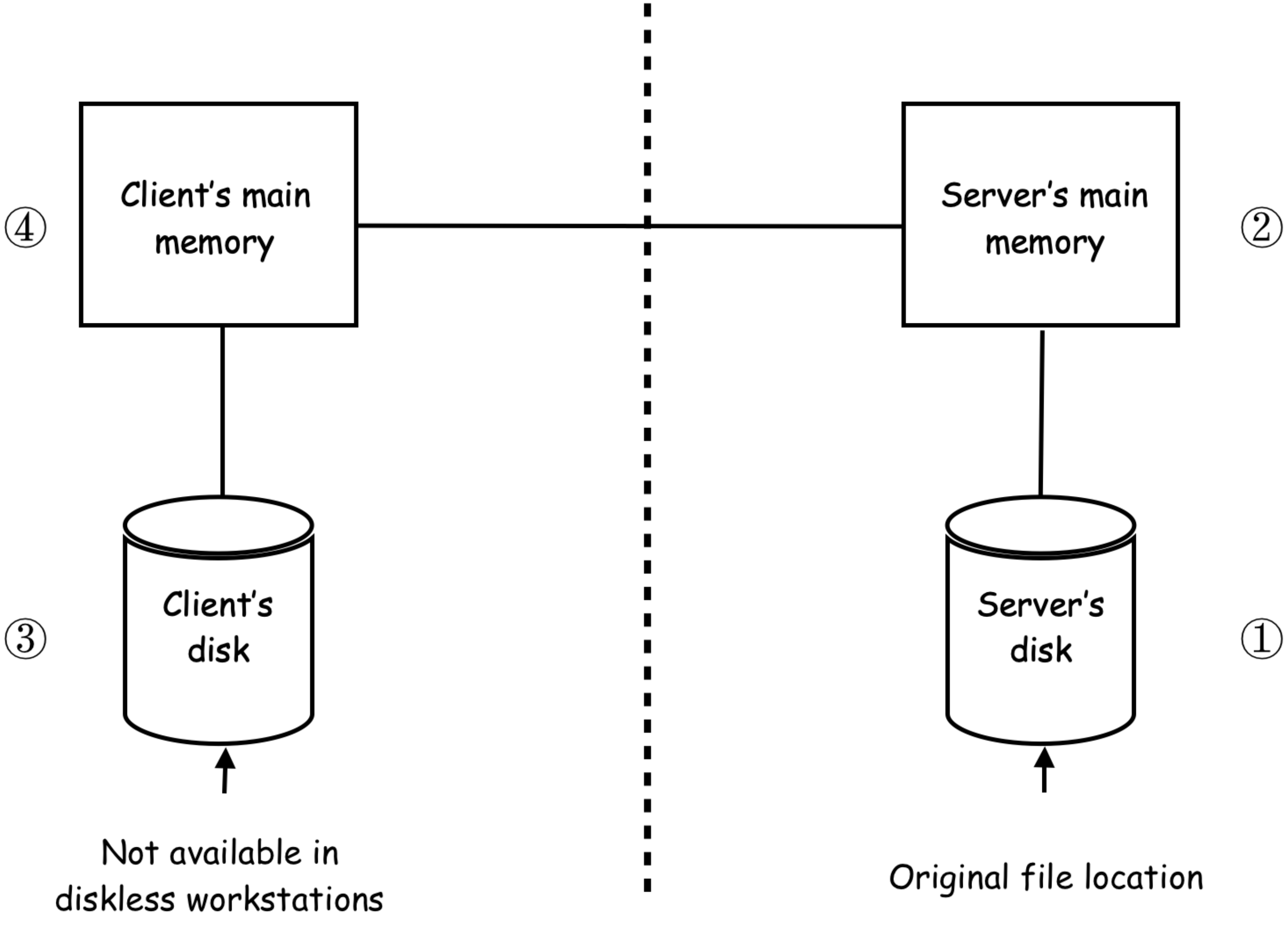

Client A
Server
Client B
remote
bin
vu
mbox
users
steen
mbox
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client
Network

File Accessing Models

- In remote service model, all file actions are performed in server side.
  - It is good for client machines with small memory or no local disk.
  - It is particularly preferred for large amounts of write operations.
- In data caching (upload/download) model, most file accesses are handled by the local caches of clients.
  - A server is contacted only when it is necessary.
  - It can reduces the server load and network traffic.
  - It may enhance scalability.
  - Cache consistency problem must be carefully handled.
    - The cached copies must be consistent with the master file.

Cache Location Policies

- There are following 3 possible locations for installing caches.
  - Server memory caching.
    - It is easy to implement and is totally transparent to the clients.
    - It is easy to support UNIX-like file-sharing semantics.
  - Client memory caching provides better performance speed up.
    - It is good when local usage is transient (or updates are rare).
    - It can be used for diskless workstations.
  - Client disk caching can cache larger files.
    - It is good when local usage dominates (e.g., AFS).
    - It can help protect clients from server crashes.



④ Client's main memory

② Server's main memory

③ Client's disk

① Server's disk

Not available in diskless workstations

Original file location

① No caching

② Cache located in server's main memory

③ Cache located in client's disk

④ Cache located in client's main memory

Cache Update Policies

- Write through scheme writes data to server disk ASAP.
  - Each write OP must propagate to server disks before returns.
  - It is more reliable but has poor performance.
- Delayed write scheme writes data in client cache first and then writes to the server later.
  - Some data may be overwritten in the cache to reduce network I/O.
  - Poor reliability
    - Unwritten data may be lost when client machine crashes.
    - Inconsistent data may occur.
  - There are 3 possible variations.
    - Write on ejection. (victim of cache replacement)
    - Periodic write.
    - Write on close. (session semantics)

Cache Validation Policies

- Client initiated approach.
  - A client initiates a validity check with the server.
  - Three possible checking variations are:
    - Checking before every access. (Unix semantics)
    - Periodic checking.
    - Checking when opens a file. (session semantics)
  - The validation is usually based on timestamp or version#.
- Server initiated approach.
  - The server records cache information for each client.
  - When the server detects a potential inconsistency, it reacts.
    - A file is opened for conflict accesses from different clients.
    - A file is closed with modification. (preferred and used in AFS)

File Replications

- File replication is the major mechanism to achieve high availability in a DFS.
- Replication vs Caching.
  - A replica is a copy in a server disk
    - A cache is a partial or whole copy of a replica.
    - A cache is dependent on the locality in file access patterns.
  - A replica depends on availability and performance requirements.
  - A replica is usually persistent and widely known.
    - The existence of a cache depends on the presence of a replica.
      - It must be periodically revalidated with respect to a replica.
- Transparency is an important issue in file replication.
  - A replicated file should be viewed as a single logical file to the users.

Replication Transparency Issues

- Two important issues related to replication transparency are naming of replicas and replication control.
- The naming scheme must map a replicated file name to a particular replica.
  - Existence of replicas should be invisible to higher levels.
  - Replicas must be distinguished from one another by different lower level names.
- Replication control includes determining the number and locations of replicas of a replicated file.
  - Explicit replication. Users can specify how many replicas are desired when creating a file.
  - Implicit/lazy replication. Auto configured by the DFS.

Multicopy Update Policies (1)

- Read Only Replication (ROR).
  - Only immutable files are replicated to avoid consistency issue.
- Read One Write All Protocol (R1Wn).
  - A read OP is performed by reading any copy of the file.
  - A write OP is performed by writing to all replicas of the file.
    - Some kind of locking has to be used to carry out a write operation.
  - It is suitable for implementing UNIX-like semantics.
  - It is very efficient when the read/write ratio is large.
- Available-Copies Protocol.
  - A write OP is performed by writing to all available copies.
    - Upon recovery from a failure, a server must 1st brings itself up to date by copying from other servers before accepting any user request.
  - It can tolerate site crashes.

Multicopy Update Policies (2)

- Primary-Copy Protocol.
  - For each replicated file, one copy is designated as the primary copy and all the others are secondary copies.
  - Read OPs can be performed using any copy.
  - All write OPs are 1st performed only on the primary copy.
    - Secondary copies are then updated later.
  - The consistency semantics depends on when the secondary copies are updated.
    - For UNIX semantics, secondary copies are updated by primary copy ASAP.
      - Some kind of locking has to be used to carry out these updates.
    - For other weaker semantics, a write OP returns as soon as the primary copy has been updated.
      - Secondary copies are then lazily updated either in the background or when a read OP is requesting the latest version.

Multicopy Update Policies (3)

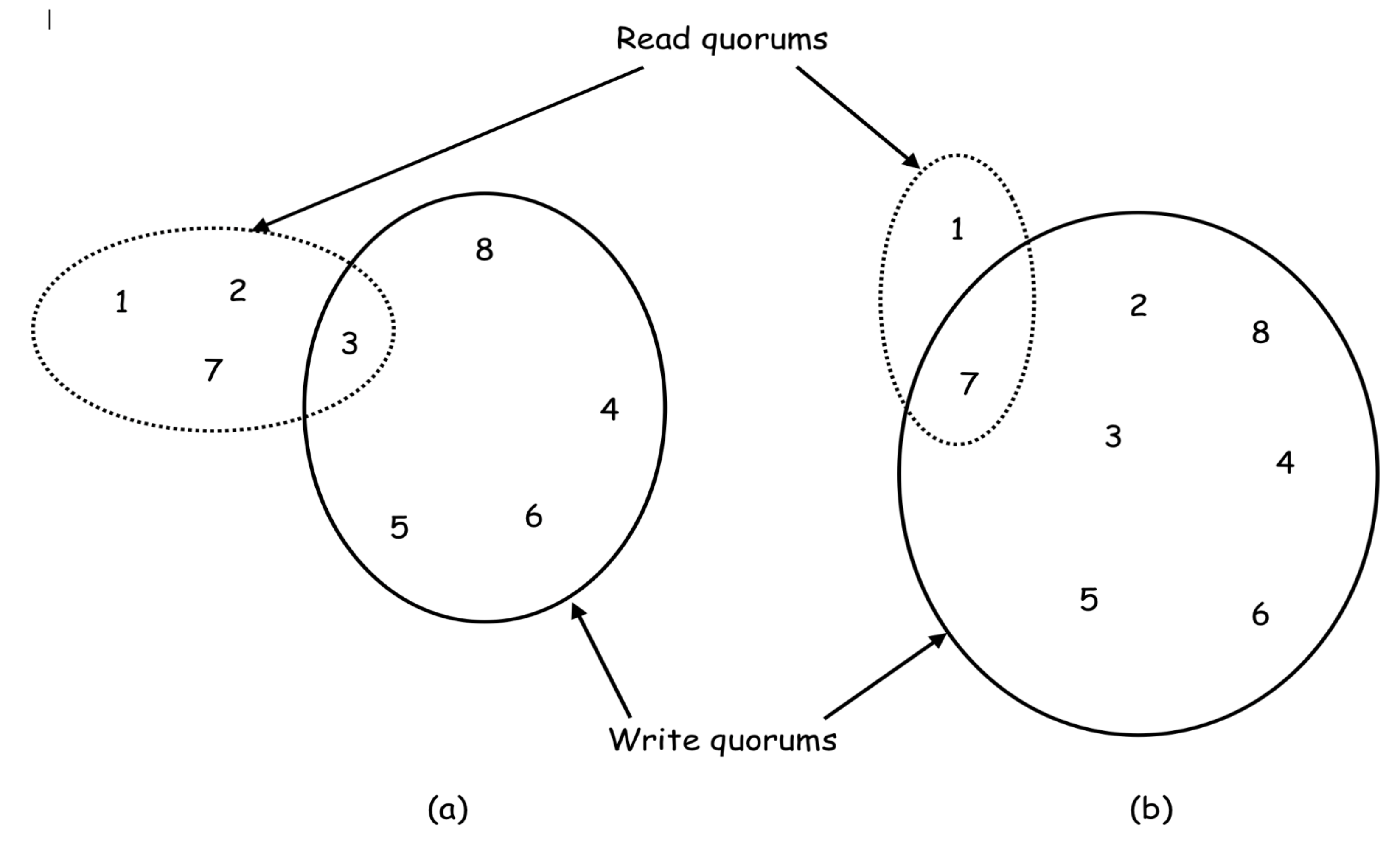
- Quorum-Based Protocols.
  - In 1979, Gifford presented a simple quorum protocol that is capable of handling the network partition problem and can increase the availability of write OPs at the expense of read OPs.
  - Assume that a replicated file F has a total of **n** copies.
    - For each read OP, a set of **r** replicas (read quorum) must be contacted to get the latest version.
    - For each write OP, a set of **w** replicas (write quorum) must be locked to perform the updates.
  - If (**r+w > n and 2w>n**) then
    - Every read OP always gets the result of the latest write.
    - Every write OP is always performed on top of the latest write.

Quorum Consensus (QC) Protocol

- A read is executed as follows:
  - Retrieve a read quorum (any r copies) of the file F.
    - Of the r copies retrieved, select the copy with the largest version#.
  - Perform the read operation on the selected copy.
- A write is executed as follows:
  - Retrieve a write quorum (any w copies) of the file F.
    - Of the w copies retrieved, get the copy with the largest version#.
  - Increment the version#.
  - Write the new value and the new version# to all the w copies in the write quorum.

Examples of QC Protocol

(a) n =8, r =4, w = 5 (b) n = 8, r =2, w = 7.



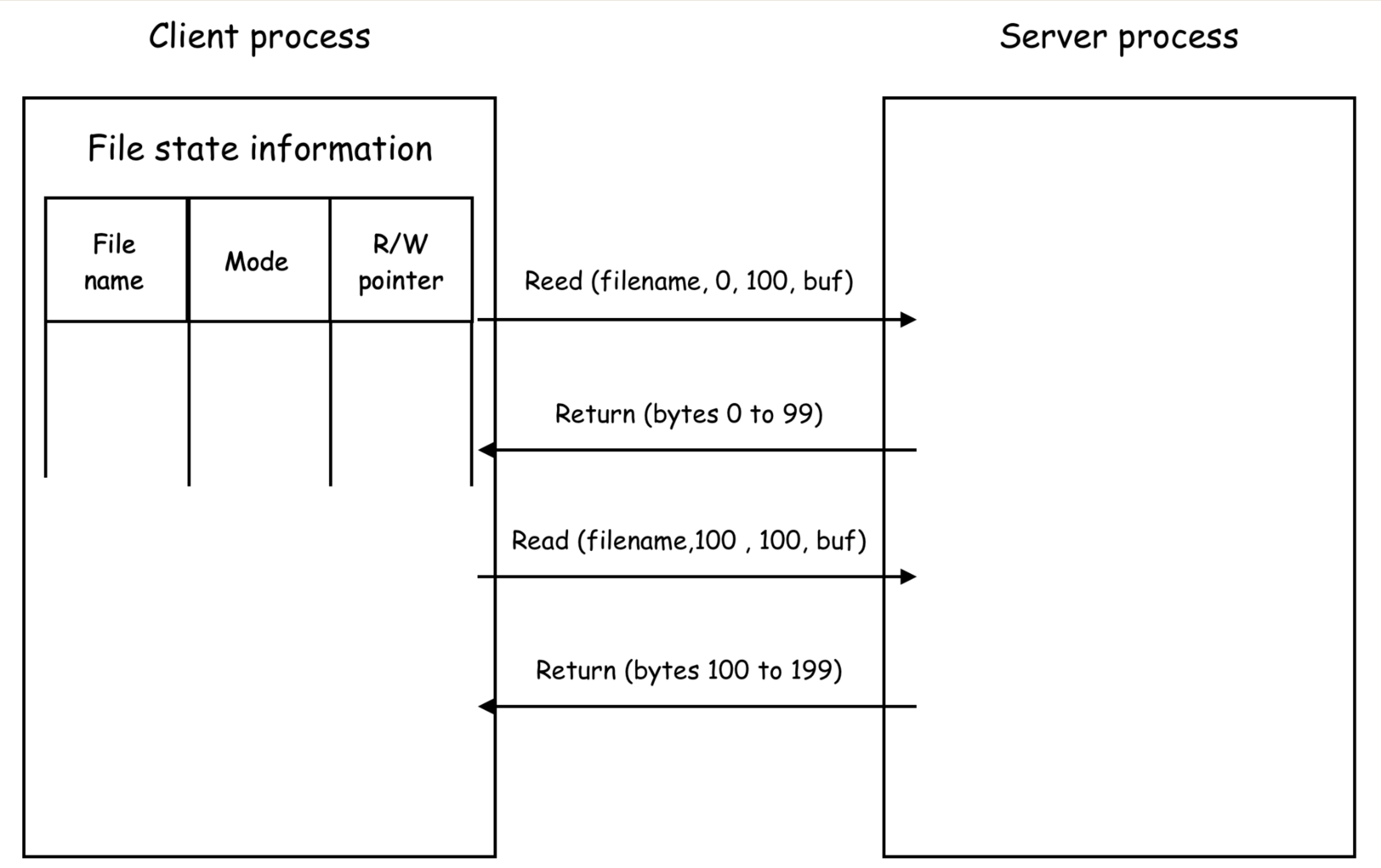
Variations of the QC Protocol

- Read One Write All Protocol. ($r=1$ and $w=n$)
  - It is commonly used when the read/write ratio is large.
- Majority Consensus protocol. ($r=w=\lfloor n/2 \rfloor+1$)
  - It is commonly used when the read/write ratio is nearly 1.
- Consensus with weighted voting.
  - If the importance of each replica is vary, each replica can then be assigned to a different votes.
  - A read quorum of r votes is needed to read a file.
  - A write quorum of w votes is needed to write a file.
  - If the total number of votes is v, then ($r+w > v$ and $2w>v$) must hold.

Stateless vs Stateful Service

- A stateless server does not keep any state information of clients.
  - This leads to self-contained requests.
    - Each request must identify the file and position in the file.
    - No explicit open() and close() before actual accessing a file.
  - Example: NFS v3 and earlier versions.
- A stateful server maintains clients' state information from one access request to the next.
  - A client must explicitly open a file to start an access session and close a file to end a session.
    - The server maintains the state information of a client for each access session.
  - Example: AFS and NFS v4.

A Stateless File Server Scenario



Client process

Server process

File state information

File name	Mode	R/W pointer

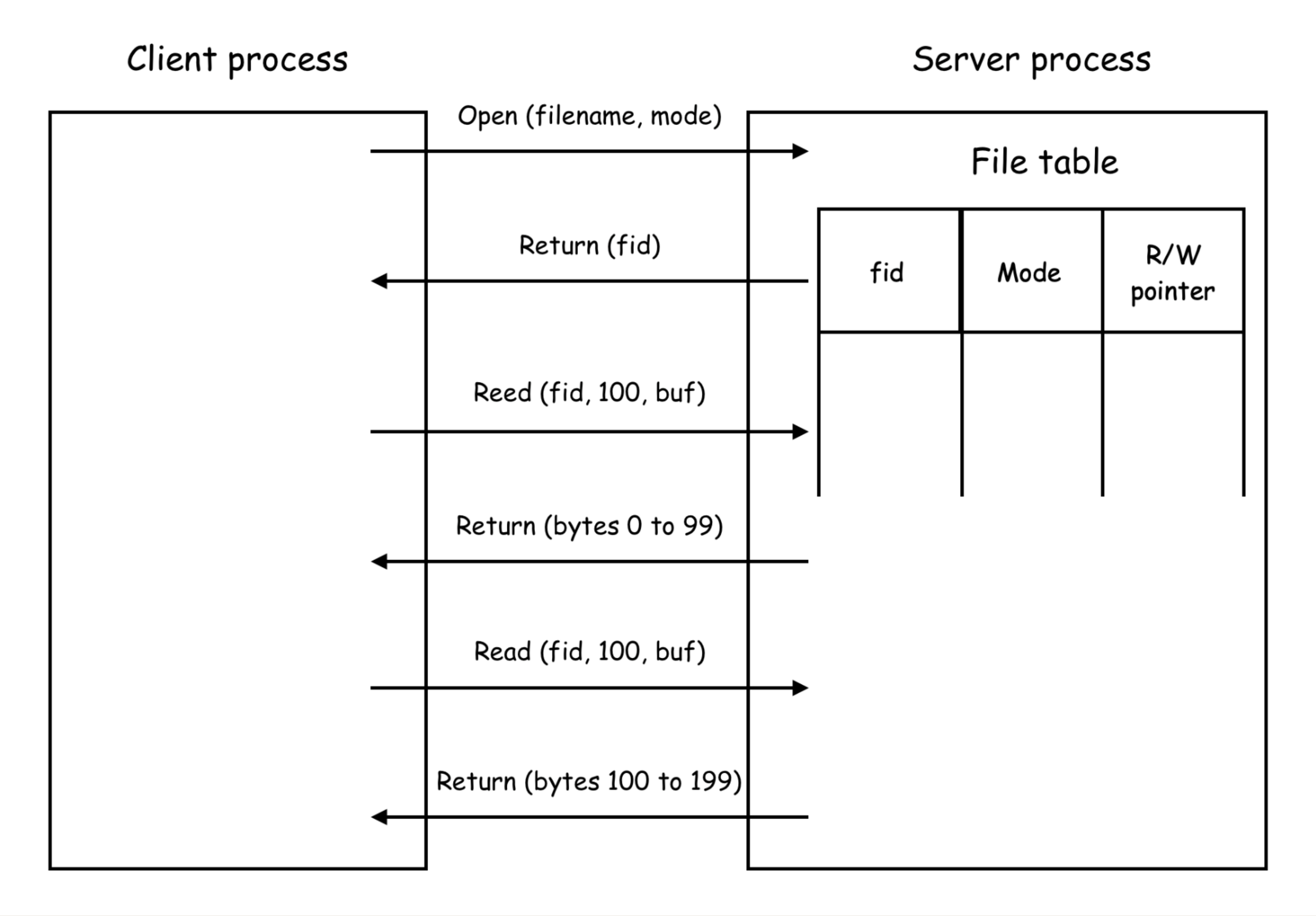
Reed (filename, 0, 100, buf)

Return (bytes 0 to 99)

Read (filename,100 , 100, buf)

Return (bytes 100 to 199)

A Stateful File Server Scenario

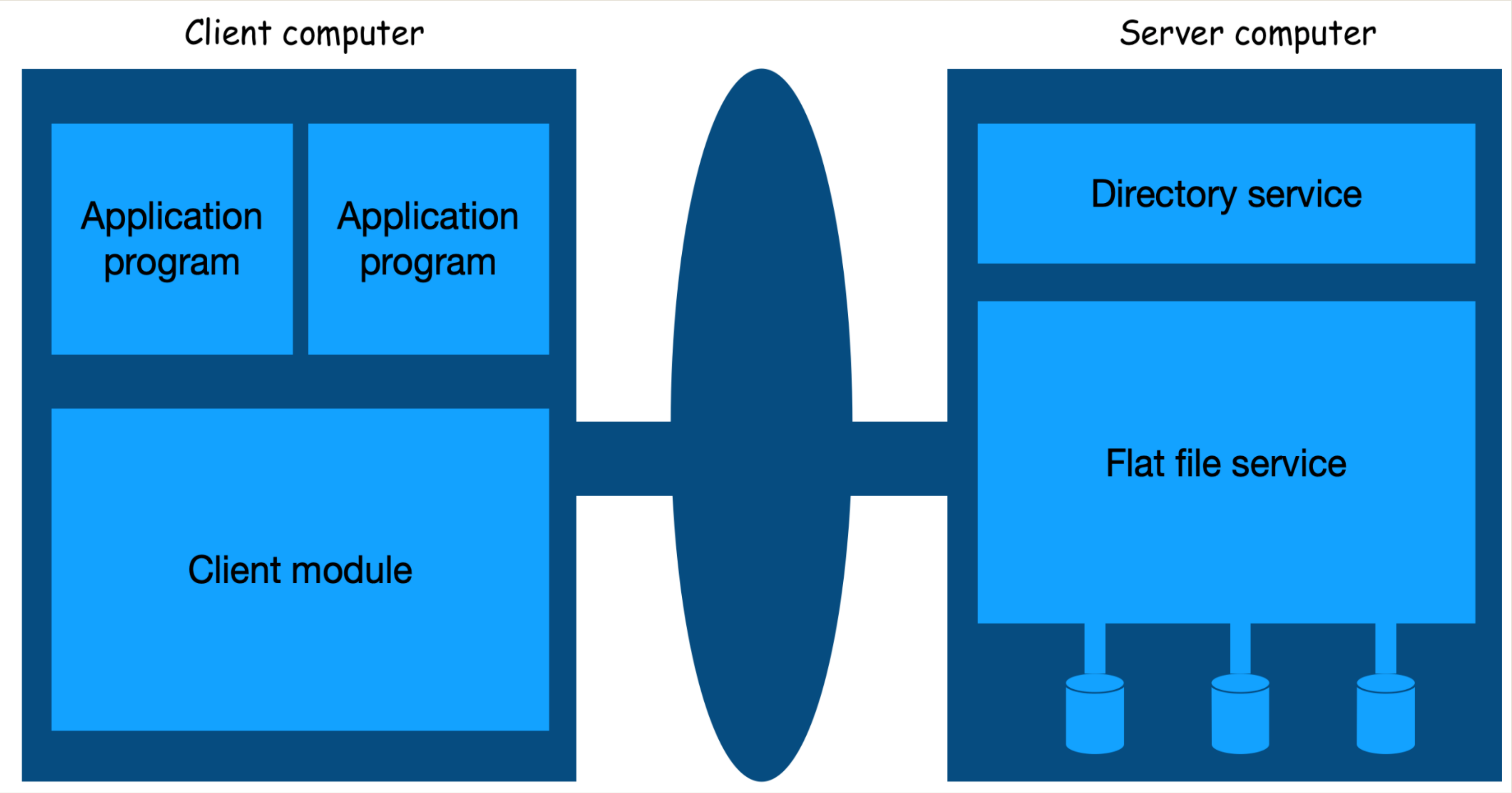


fid	Mode	R/W pointer

Stateless vs Stateful

- A stateful server loses all volatile state in a crash.
  - Restore state by recovery protocol based on a dialog with clients.
  - The server needs to be aware of crashed client processes.
    - Orphan detection and elimination are needed.
- Some environments require stateful service.
  - Server initiated cache validation cannot provide stateless service.
  - File locking needs stateful service.
- Failure and recovery are almost unnoticeable in a stateless server .
  - A newly restarted server can respond to self-contained requests without difficulty.
- Penalties for using the robust stateless service are:
  - longer request messages and slower request processing

A Simple File Service Architecture



File Service Components

- The Flat File Service is handling accesses on the file contents.
  - Unique file identifiers (UFIDs) are used to refer to files in all requests for flat file service operations.
- The Directory Service provides a mapping between text names of files and their UFIDs.
  - The directory service provides the functions needed
    - to generate directories,
    - to add new file names to directories, and
    - to obtain UFIDs of files from their parent directories.
  - It can be a client of the flat file service.
    - Directory files may be stored as files in the flat file service.
- A Client Module runs in each client computer.
  - It integrates and extends the operations of the flat file service and the directory service under a single API that is available to user-level programs in client computers.

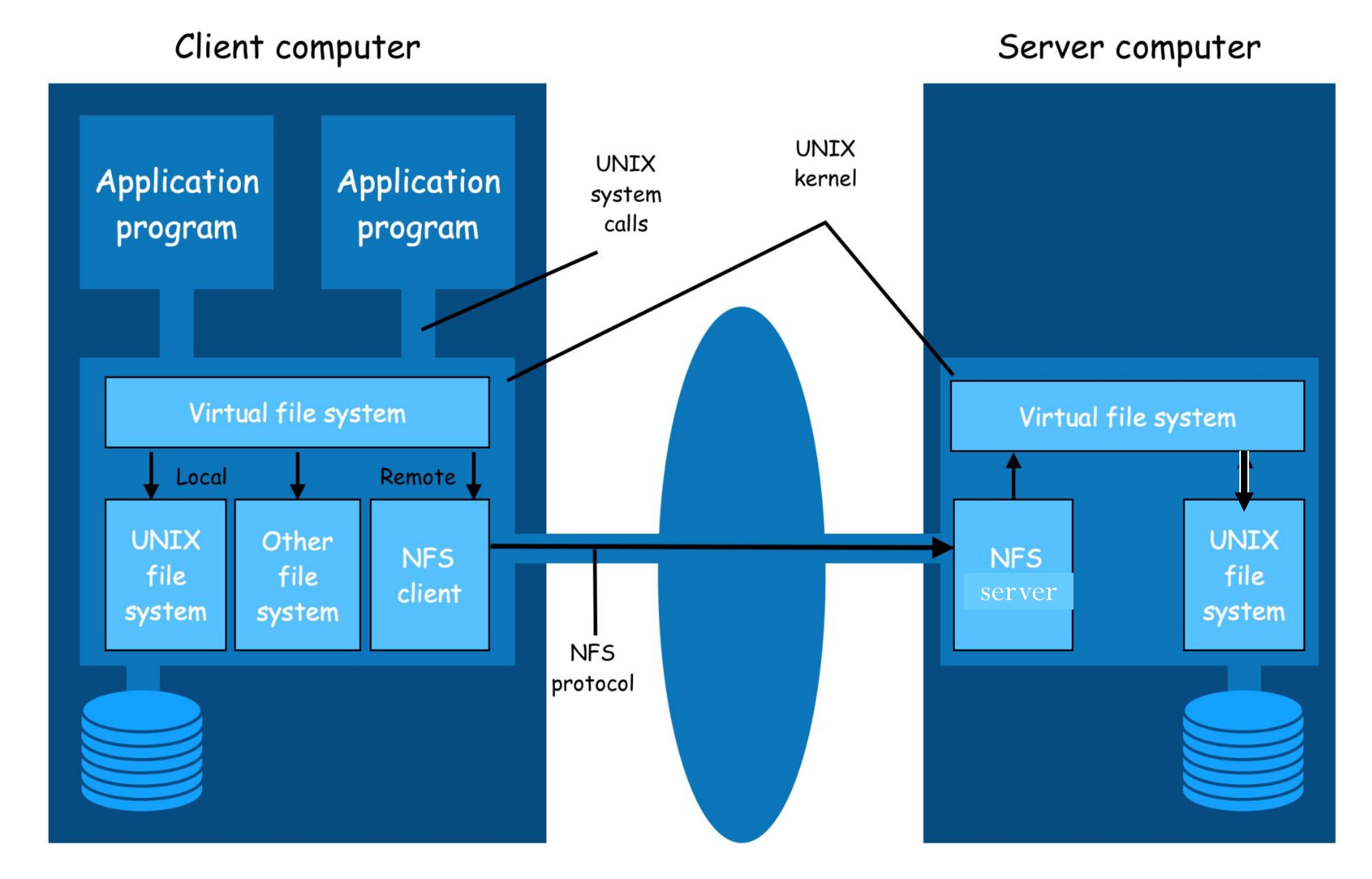
Flat File Service APIs

UFID ← Create ()	Creates a new file of length 0 and returns a UFID for it
data ← Read (UFID, i, n)	Reads *n* items from file UFID starting at item i and returns them into data
Write (UFID, data, i)	Writes data to file UFID starting at item i
Delete (UFID)	Removes file UFID from the file store
attr ← GetAttribute (UFID)	Returns the file attributes of file UFID into attr
SetAttribute (UFID, attr)	Sets the file attributes of file UFID from attr

Directory Service APIs

UFID ← LookUp (dir, name)	Locates the filename name in the directory dir and returns its UFID.
AddName (dir, name, UFID)	Adds the (name, UFID) to the directory dir and updates the attributes of file UFID if necessary.
RemoveName (dir, name)	Removes the entry containing name from the directory dir.
NameSeq ← FindNames (dir, pattern)	Returns all the filenames in the directory dir that match the regular expression pattern into NameSeq.

The Network File System (NFS)



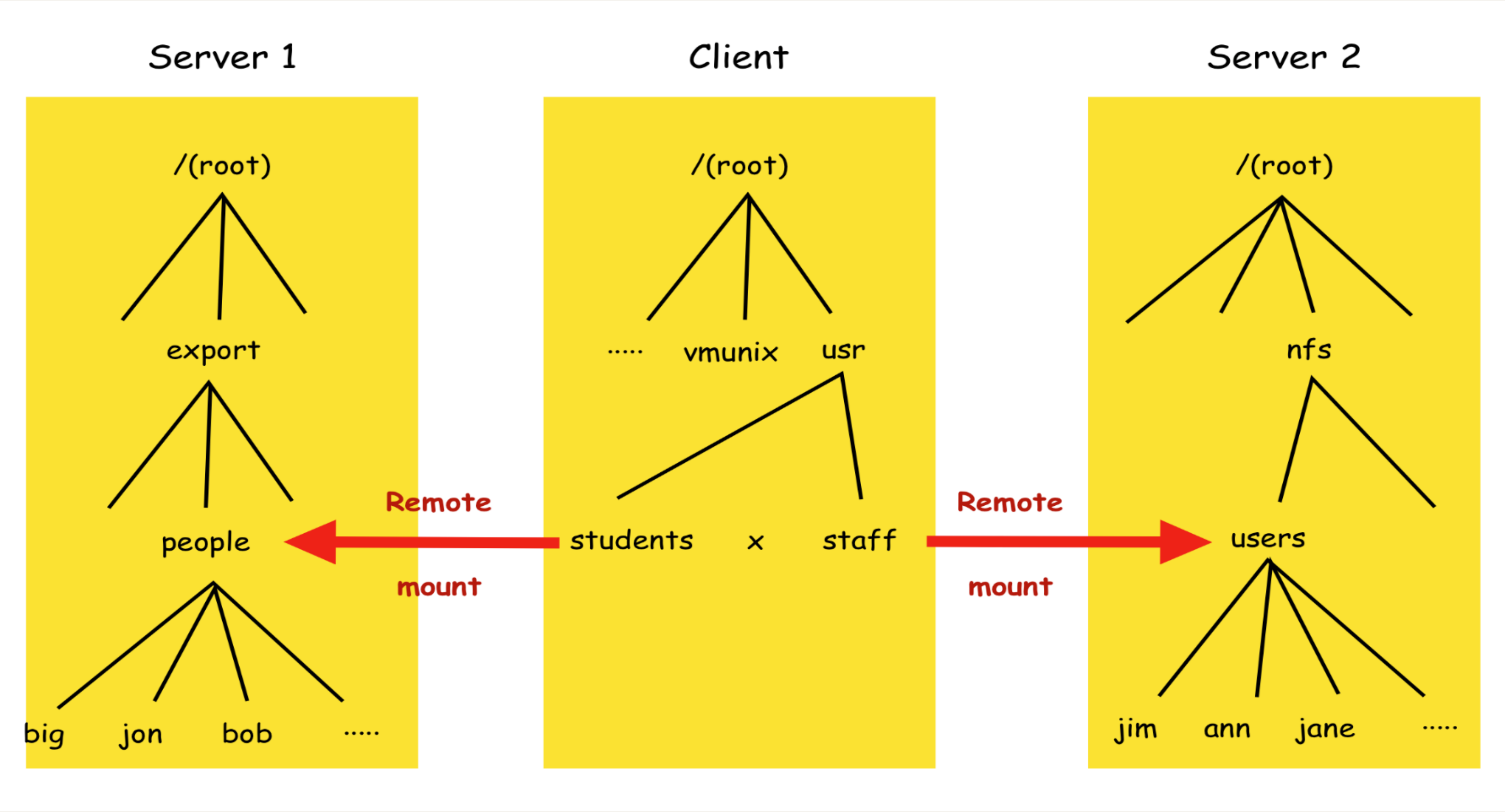
Subset of NFS v3 APIs (1)

lookup (dirfh, name) → fh, attr	Returns file handle fh and attributes attr for the file name in the directory dirfh.
create (dirfh, name, attr) → fh, newattr	Creates a new file name in directory dirfh with attributes attr and returns the new file handle fh and new attributes newattr.
remove (dirfh, name)	Removes file name from directory dirfh.
getattr (fh) → attr	Returns file attributes of file fh into attr.
setattr (fh, attr) → newattr	Sets file attributes of file fh by attr and returns the new file attributes into newattr.
read (fh, offset, count) → attr, data	Returns up to count bytes of data from the file fh starting at offset. Also returns the new file attributes attr.
write (fh, offset, count, data) → attr	Writes count bytes of data to the file fh starting at offset. Returns the new file attributes attr.
rename (dirfh, name, todirfh, toname)	Changes the name of file name in directory dirfh to toname in directory todirfh.
link (newdirfh, newname, dirfh, name)	Creates an entry newname in the directory newdirfh which refers to the file name in the directory dirfh.

Subset of NFS v3 APIs (2)

symlink (newdirfh, newname, string)	Creates an entry newname in the directory newdirfh of type symbolic link with the value string. The server creates a symbolic link file to hold the string.
readlink (fh) → string	Returns the string stored in the symbolic link file fh.
makedir (dirfh, name, attr) → newdirfh, newattr	Creates a new directory name with attributes attr in the directory dirfh and returns the new directory newdirfh and new attributes newattr.
rmdir (dirfh, name)	Removes an empty directory name from the parent directory dirfh.
readdir(dirfh, cookie, count) → entries	Returns up to count bytes of directory entries from the directory dirfh. Each entry contains a file name, a file handle, and a cookie point to the next directory entry. The cookie is used in subsequent readdir calls to start reading from the following entry. If the value of cookie is 0, reads from the first entry in the directory.
statfs(fh) → fsstats	Returns file system information into fsstats (such as block size, number of free blocks and so on) for the file system containing the file fh.

Remote Files Accesses on an NFS Client



1. The file system mounted at /usr/students in the client is actually the sub-tree located at /export/people in Server 1
2. The file system mounted at /usr/staff in the client is actually the sub-tree located at /nfs/users in Server 2.

Pathname Translation in NFS

- In NFS, pathnames are not translated at server nodes.
  - Because the name may cross a "mount point" at the client.
  - Directories holding different parts of a multi-part name may reside in file systems at different servers.
- Pathnames are translated in an iterative manner by the client.
  - Each name that refers to a remote-mounted directory is translated to a file handle using a separate lookup request to the remote server.
    - The lookup operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes.
    - The file handle returned in the previous step is used as a parameter in the next lookup step.
- Caching of the results of each step in pathname translations can compensate the inefficiency of this iterative process.
  - Due to locality of reference to files and directories, users and programs typically access files in only a small number of directories.

Server Caching in NFS

- Caching is used in a NFS server for both read and write operations.
  - Read-ahead anticipates read accesses and fetches the pages following those that have most recently been read.
  - Delayed-write optimizes writes by holding updated contents in the server cache as long as possible.
    - However, clients must be ensured that the results of the write operations are persistent, even when server crashes occur.
- In the NFS v3 protocol, the write operation offers two options:
  - (Write-Through Caching) Data is stored in the memory cache and written to disk before a reply is sent to the client.
    - The clients can be sure that the data is stored persistently upon receiving the reply.
  - (Committed-Write Caching) Data is stored only in the memory cache.
    - It is written to disk when a commit operation is received for the relevant file.
    - The client can be sure that the data is persistently stored only upon receiving the reply for a commit operation of the relevant file.
  - Standard NFS clients use the 2nd option by issuing a commit operation whenever an open-for-write file is closed.
    - This overcomes a performance bottleneck caused by the write-through caching in servers that receive large numbers of write operations.

Client Caching in NFS (1)

- The NFS client module can caches the results of read, write, getattr, lookup and readdir operations to reduce the number of requests transmitted to servers.
  - However, client caching may lead to data inconsistency.
    - Writes by a client may not be immediately reflected in caches of other clients.
    - Clients are responsible for polling the server to check the validity of the cached data that they hold.
- A timestamp-based method is used to validate cached blocks before they are used.
  - Each data or metadata item cached uses two timestamps:
    - Tc is the time when the cache entry was last validated.
    - Tm is the time when the block was last modified at the server.

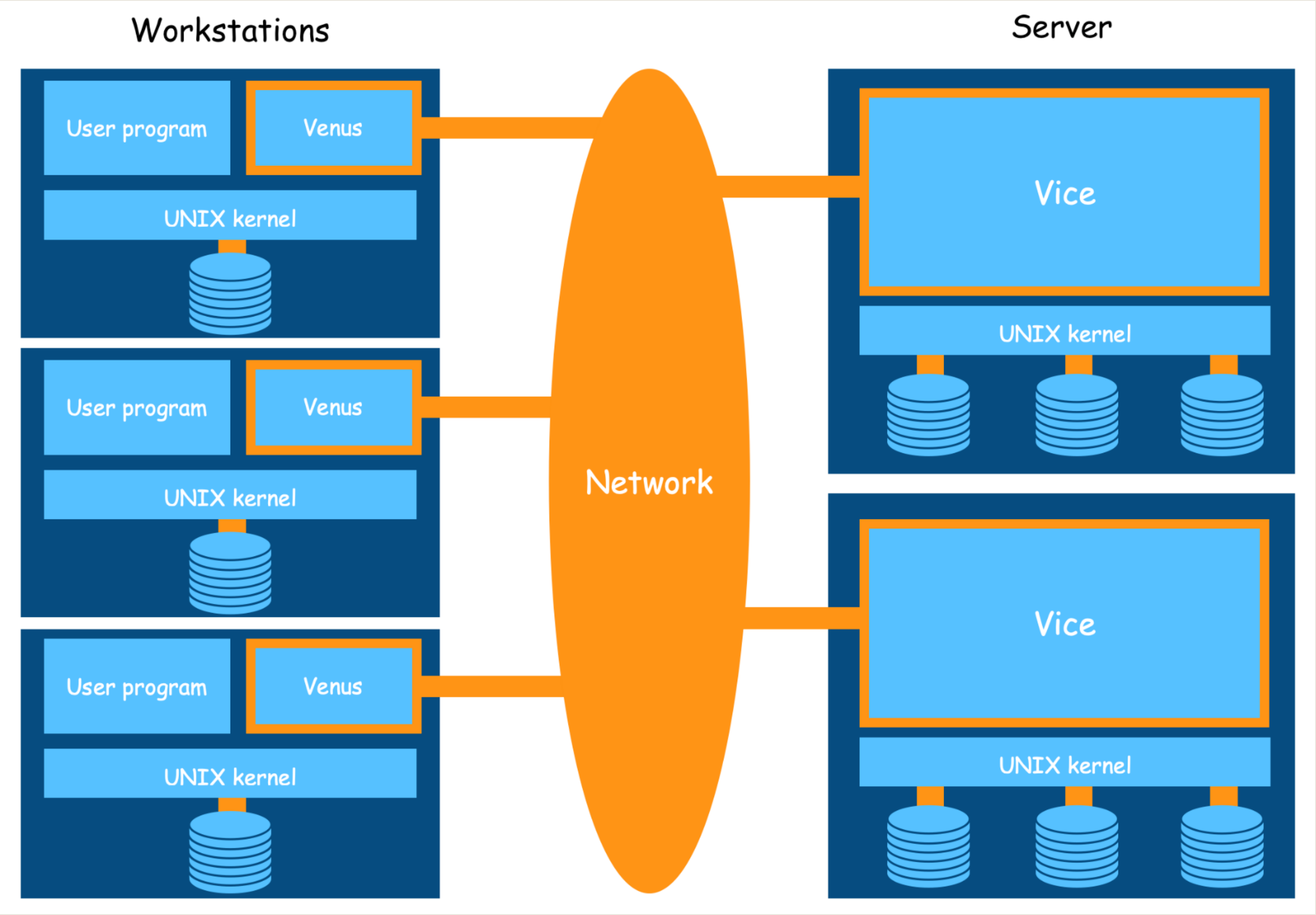
Client Caching in NFS (2)

- A cache entry is valid at time T
  - if (T-Tc) is less than a freshness interval t (predefined), or
  - if the Tm recorded at the client matches the Tm at the server.
- Validation condition: (T-Tc < t) or (Tm(client) == Tm(server))
  - The value of t is determined by compromising consistency and efficiency.
  - There is a Tm(server) for each data block in a file and the file attributes of a file also has its own Tm(server).
- NFS clients do not know whether a file is being shared or not.
  - A validity check must be performed whenever a cache entry is used.
  - Getattr operation is used to obtain the Tm(server) of a file block.
- Methods for reducing the traffic of getattr calls to the server:
  - Upon receiving a new value of Tm(server), it is applied to all cache entries for the file.
  - The attribute values are "piggybacked" in the results of each operation on a file.
    - If the value of Tm(server) has changed, the client uses it to update the cache entries of the relevant file.
- Using adaptive algorithm for setting the interval t can reduce the traffic for most files.
  - For files, the range is 3 to 30 seconds depending on the update frequency of the file.
  - For directories, the range is 30 to 60 seconds reflecting the lower risk of concurrent updates.

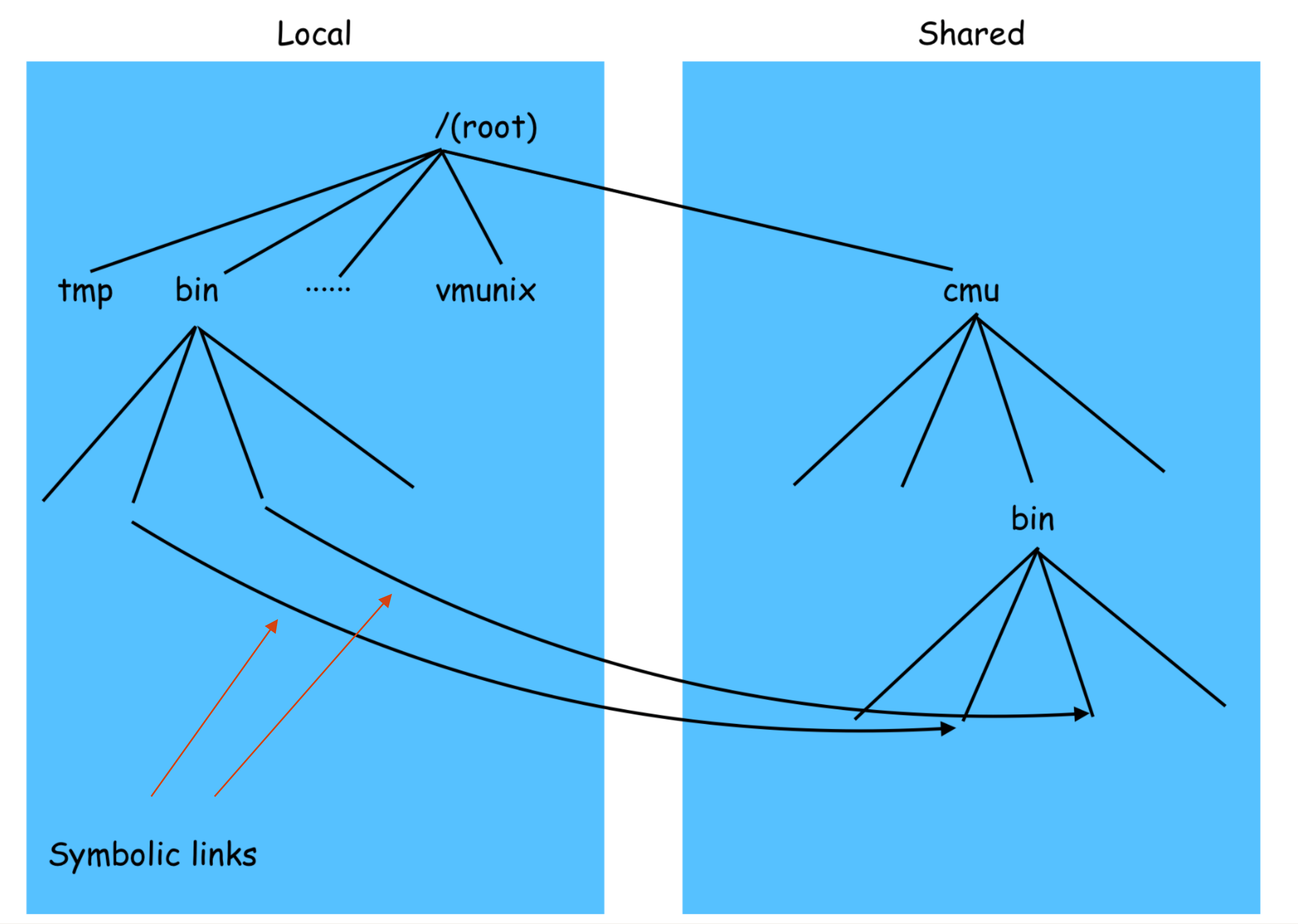
Client Caching in NFS (3)

- When a cached page is modified it is marked as "dirty" and is scheduled to be flushed to the server asynchronously.
  - Modified pages are flushed when the file is closed or a sync occurs at the client.
  - This does not provide the same persistence guarantee as the server cache, but it emulates the behavior for local writes.
- To implement read-ahead and delayed-write, the NFS client needs to perform some reads and writes asynchronously.
  - This is achieved in UNIX implementations of NFS by the inclusion of one or more bio-daemon processes at each client.
    - Here, bio stands for block input-output.
- A bio-daemon is notified after receiving the reply for each read request.
  - It then requests the following file block from the server to the client cache.
- A bio-daemon will send a block to the server whenever a block has been filled by a client operation.
- Directory blocks are sent whenever a modification has occurred.

The Andrew File System (AFS)



File Name Space in an AFS Client



Enhanced Scalability in AFS

- AFS has two unusual design characteristics to achieving high scalability.
  - (Whole-file serving) The entire contents of directories and files are transmitted to client computers by AFS servers.
    - In AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks.
  - (Whole-file caching) Once a copy of a file has been transferred to a client computer it is stored in a cache on the local disk.
    - The cache contains several hundred of the files most recently used on the client computer.
    - The cache is permanent and can survive reboots of the client computer.
    - Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible.

Reasons behind the Design Strategy

- The design strategy is based on some assumptions about average and maximum file size and locality of reference to files in UNIX systems.
  - Files are small: most are less than 10 kilobytes in size.
  - Read operations on files are much more common than writes.
    - Approximates to 6:1
  - Sequential access is common and random access is rare.
  - Most files are read and written by only one user.
    - When a file is shared, it is usually only one user who modifies it.
  - Files are referenced in bursts.
    - If a file has been referenced recently, there is a high probability that it will be referenced again in the near future.

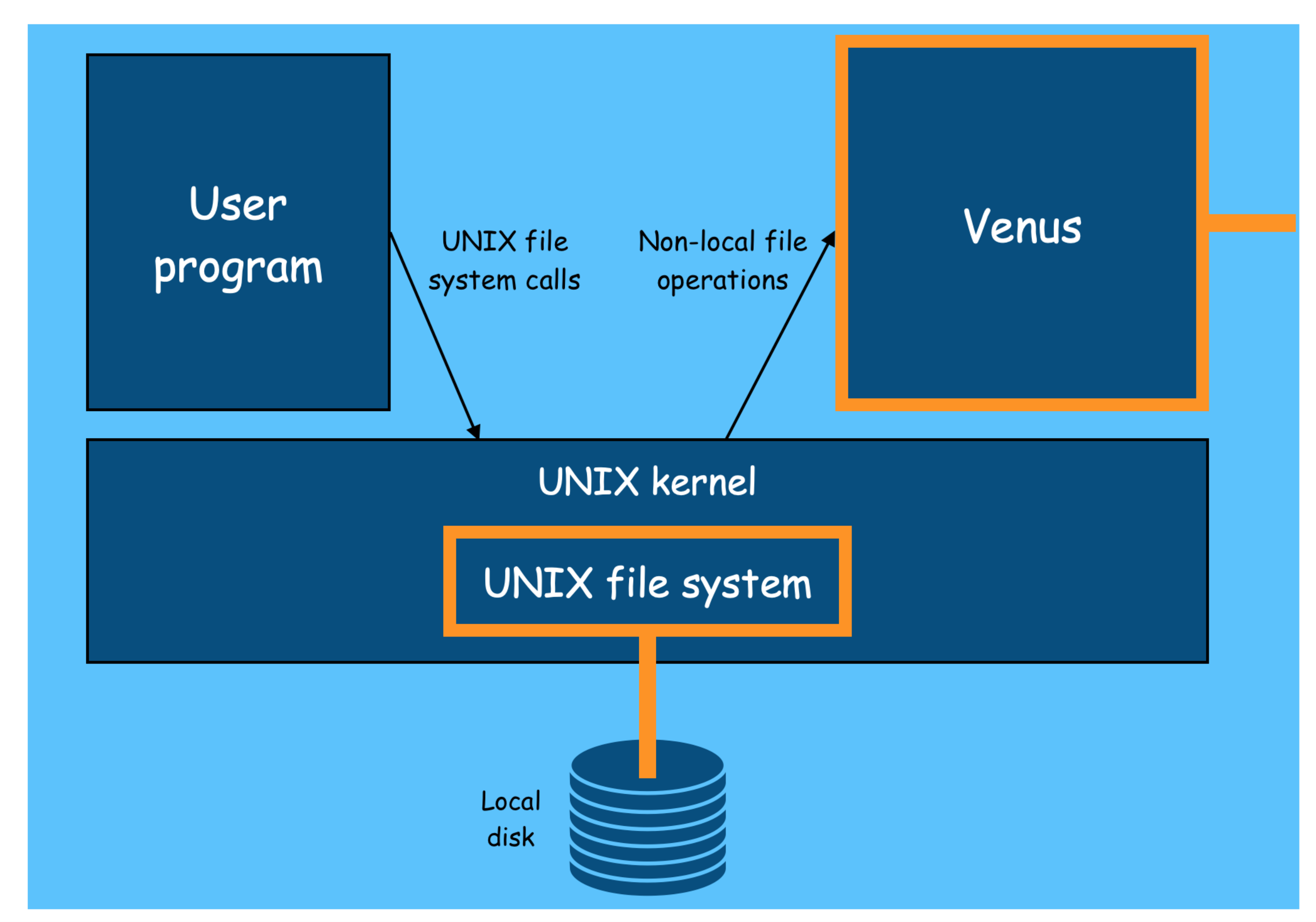
An Operational Scenario in AFS

- When a user process in a client computer issues an open call for a file in the shared file space.
  - If there is not a current copy of the file in the local cache, the client module 1st locates the server holding the file and then sends a request for getting a copy of the file.
- The local copy of the file is opened and the resulting UNIX file descriptor is returned to the user process.
  - Subsequent read, write and other operations on the file by the user process in the client computer are applied to the local copy.
- When the user process in the client issues a close call.
  - If the local copy has been updated, its contents are sent back to the server.
    - The server updates the file contents and the timestamps on the file.
  - The copy on the client's local disk is retained for possible future usage by other user processes on the client computer.

Basic Concepts in AFS

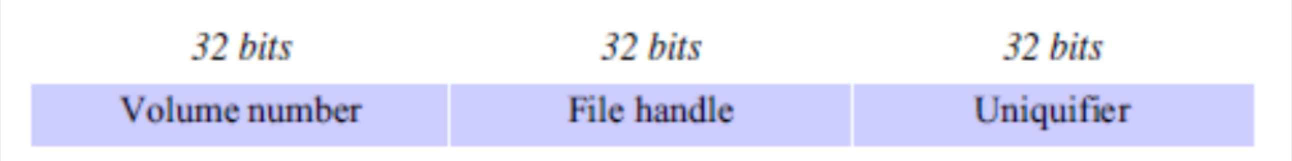
- AFS is implemented as two software components that exist as UNIX processes called Vice and Venus.
  - Vice is the server software that runs as a user-level UNIX process in each server computer.
  - Venus is the client module that runs as a user-level UNIX process in each client computer.
- The files are available to user processes as either local or shared.
  - Local files are handled as normal UNIX files.
    - They are stored on a local disk and are available only to local user processes.
    - Local files are used only for temporary files (/tmp) and files that are essential for client computers startup.
    - Other standard UNIX files (/bin, /lib and so on) are implemented as symbolic links from local directories to files held in the shared space.
  - Shared files are stored on servers and copies of them are cached on the local disks of client computers.
    - Users' home directories are in the shared space for enabling users to access their files from any client computer.

System Call Interception in AFS



AFS Implementation (1)

- The UNIX kernel in each client and server is a modified version of BSD UNIX.
  - The open, close and some other file system calls are intercepted and passed to the Venus when they refer to shared space files.
- Each client computer uses a local file partition as a cache for holding the cached copies of files from shared space .
  - Venus manages the cache.
    - It may remove the least recently used files when a new file is acquired from a server and the partition is full.
- Each file and directory in the shared file space is identified by a unique 96-bit file identifier (fid).



32 bits	*32 bits*	*32 bits*
Volume number	File handle	Uniquifier

AFS Implementation (2)

- Files are grouped into volumes for ease of location and movement.
  - Each user's personal files are usually in a separate volume.
  - Other volumes are allocated for system binaries, documentations and library codes.
- User programs use pathnames to refer to files, but AFS uses fids in the communication between the Venus and Vices.
  - The Vice servers accept requests only in terms of fids.
  - Venus translates the pathnames supplied by clients into fids.
    - It uses a step-by-step lookup calls to obtain the information from the file directories held in the Vice servers.

File System Calls in AFS

User process	UNIX kernel	Venus	Net	Vice
open(FileName, mode)	If FileName refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid callback promise, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.		Transfer a copy of the file and a callback promise to the Venus. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.		Replace the file contents and send a callback to all other clients holding callback promises on the file.

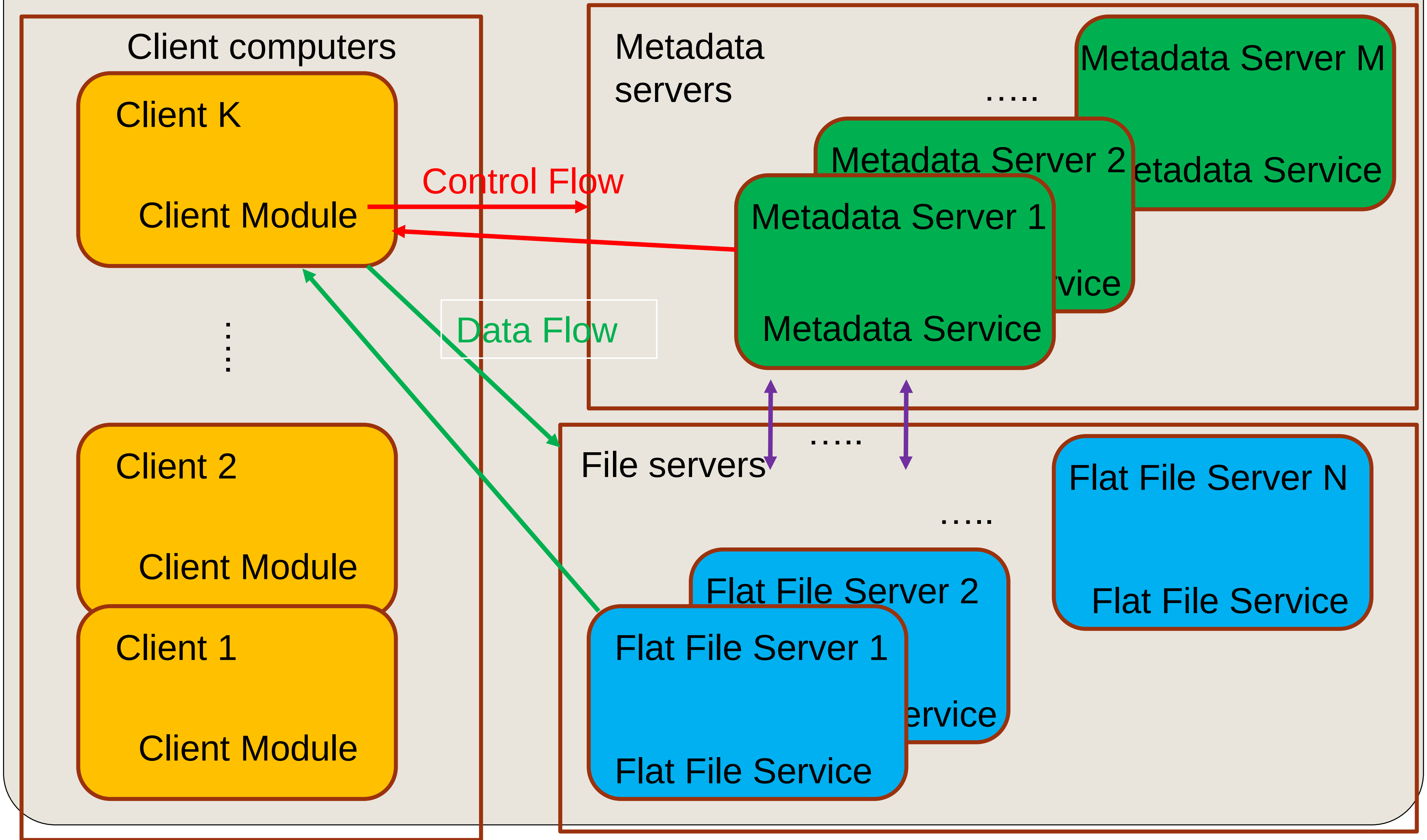
Cache Consistency in AFS

- When Vice supplies a copy of a file to a Venus, it also provides a callback promise.
  - A callback promise is a token issued by the Vice who is the custodian of the file.
    - It guarantees to notify the Venus when any other client modifies the file.
  - Callback promises are stored with the cached files on the client's local disks and have two states: valid or cancelled.
  - When a Vice performs a request to update a file it notifies all of the Venus who have been issued callback promises.
    - A callback is a remote procedure call from a Vice to a Venus.
  - When the Venus receives a callback, it sets the callback promise token for the relevant file to cancelled.
- Whenever Venus handles an open call, it checks the cache.
  - If the required file is found in the cache, then its callback promise is checked.
  - If it is cancelled, then a fresh copy of the file must be fetched from the Vice.
- When a client computer is restarted after a failure or a shutdown, the Venus must ensure all callback promises are still valid.
  - It generates a cache validation request containing the file modification timestamp for each cached file.
  - Callbacks must be renewed before an open if a freshness interval T has elapsed since the file was cached without communication from the server.

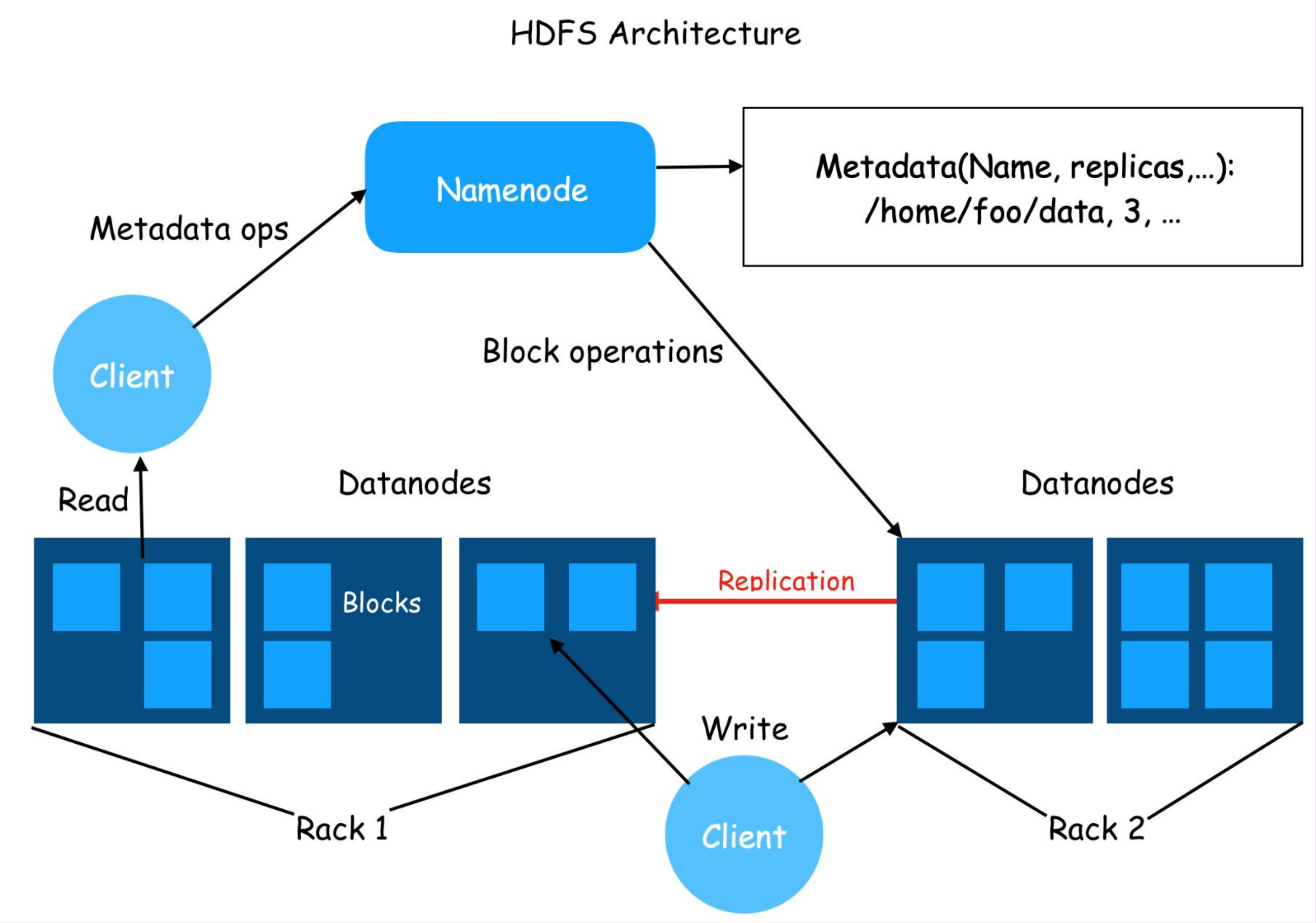
The Vice service APIs (subset)

Fetch(fid) → attr, data	Returns the attributes of the file fid into attr and the contents of fid into data. Also records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and the contents of the file fid.
Create() → fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the file fid.
SetLock(fid, mode)	Sets a lock on the file (or directory) fid. The mode of the lock may be shared or exclusive. Locks will automatically expire after 30 minutes.
ReleaseLock(fid)	Unlocks the file (or directory) fid.
RemoveCallback(fid)	Informs the server that a Venus has flushed a file from its cache.
BreakCallback(fid)	Callback is sent from a Vice to a Venus. Venus cancels the callback promise on the relevant file.

A New File Service Architecture



Hadoop Distributed File System (HDFS)



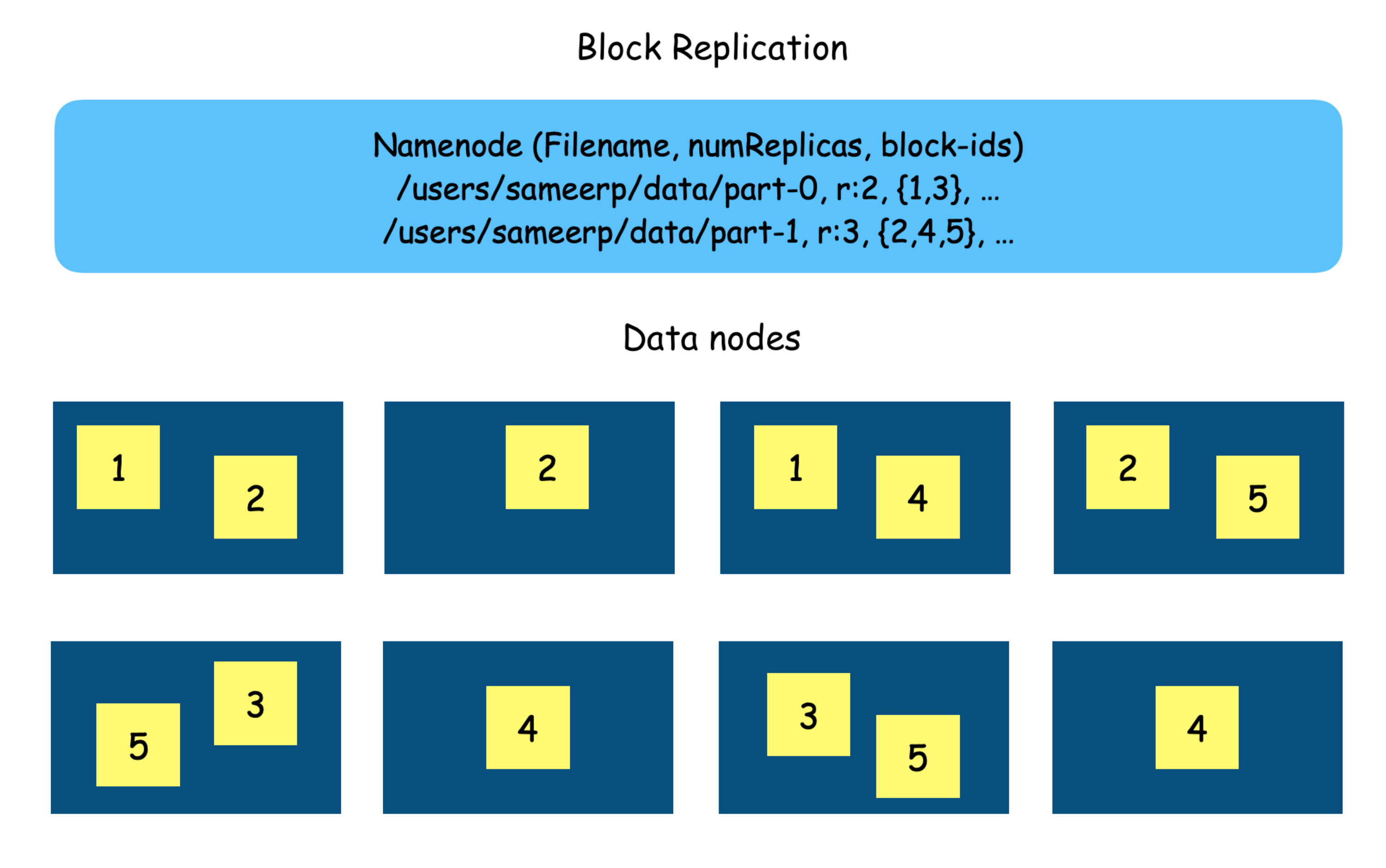
HDFS Architecture

- Master / Slave Architecture
- NameNode as the Master
    - File system namespace (metadata) management
    - Name to file mapping
        - Operation: open, close, and renaming
    - Data blocks allocation/deallocation
- DataNodes as Slaves
    - Data blocks management
        - Block creation, deletion and replication requests for NameNode
    - Data block services
        - Operation: read and write

Files in HDFS

- Each file is stored as a sequence of blocks.
  - All blocks except the last block are the same size.
  - File configuration parameters:
    - block size, replication factor, …
  - Blocks of the same file are allocated separately.
  - Blocks of the same file can have different Replication factors.
- All files are write-once only and can only be written by one writer at a time.
  - All writes are appended to the end of a file.

Block Replication (1)



Block Replication

Namenode (Filename, numReplicas, block-ids)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...

Data nodes

Block 1,3 have 2 replicas.
Block 2,4,5 have 3 replicas.

Block Replication (2)

- Simple policy
  - All replicas are in different racks.
  - Disadvantage: the cost of write is high.
- Rack locality awareness policy
  - 1st replica is placed on a nearest data node from the requester.
  - 2nd replica is placed on a data node in the same rack as the 1st replica.
  - All other replicas are placed randomly on the nodes outside the rack.
- Other concerns
  - High disk usage nodes should not be used for allocation of new blocks.
  - High workload nodes should be avoided for allocation of new blocks.
  - Fast and slow nodes should be treated differently.
- Conditions for Re-Replication
  - A data node becomes unavailable.
  - A replica becomes corrupted.
  - A hard disk on a data node fails
  - The replication factor of a file is increased.

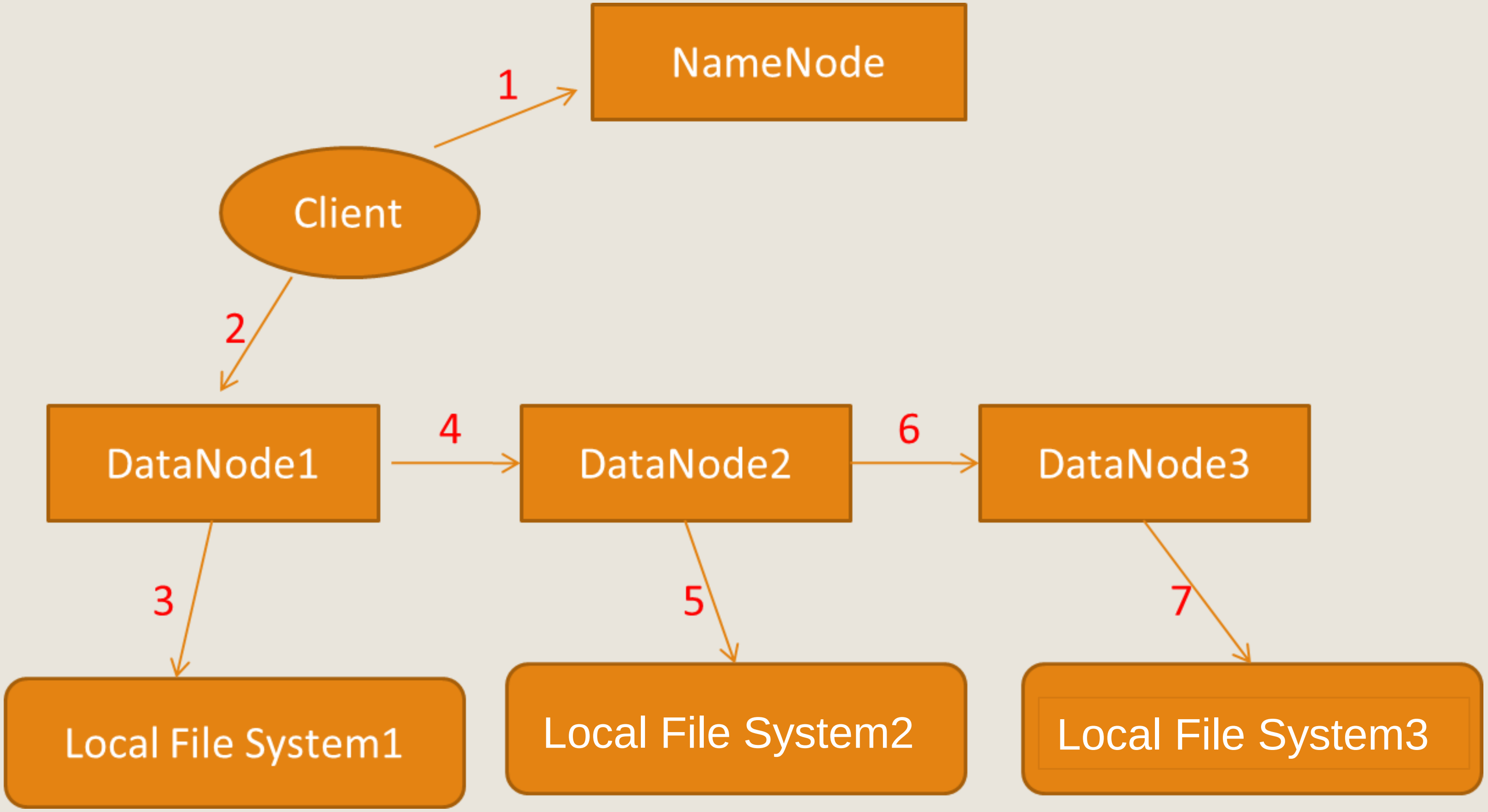
Metadata in HDFS

- EditLog
  - Records every change made to file system metadata since last checkpoint of FsImage.
- FsImage (metadata)
  - Contains entire file system namespace, mapping of blocks to files, and file system properties.
  - FsImage is check pointed periodically for failure recovery.
- If the name node fails, a new name node module can be launched manually with last checkpoint of FsImage.
  - The updates in EditLog can then be applied for recovery.

File Creation Scenario in HDFS

- A file creation request by a client may not reach NameNode immediately.
  - The file was created in the client's local disk first.
  - Write operations are performed in local until accumulated data reaches the block size of the HDFS file.
    - The default data block size is 64 MB.
- Client contacts the NameNode.
  - The NameNode inserts the file name into the file system.
  - The NameNode makes allocation assignment for the new block.
    - It replies the IDs of the DataNodes and the IDs of data blocks back to the client.
- Client flushes the data block from its local disk to the DataNodes.
- Client continues to buffer subsequent writes.
  - When the file is closed, the remaining un-flushed data is transferred to the DataNodes.
  - If the NameNode crushes before the file is closed, the file my be lost.

Replication Scenario in HDFS



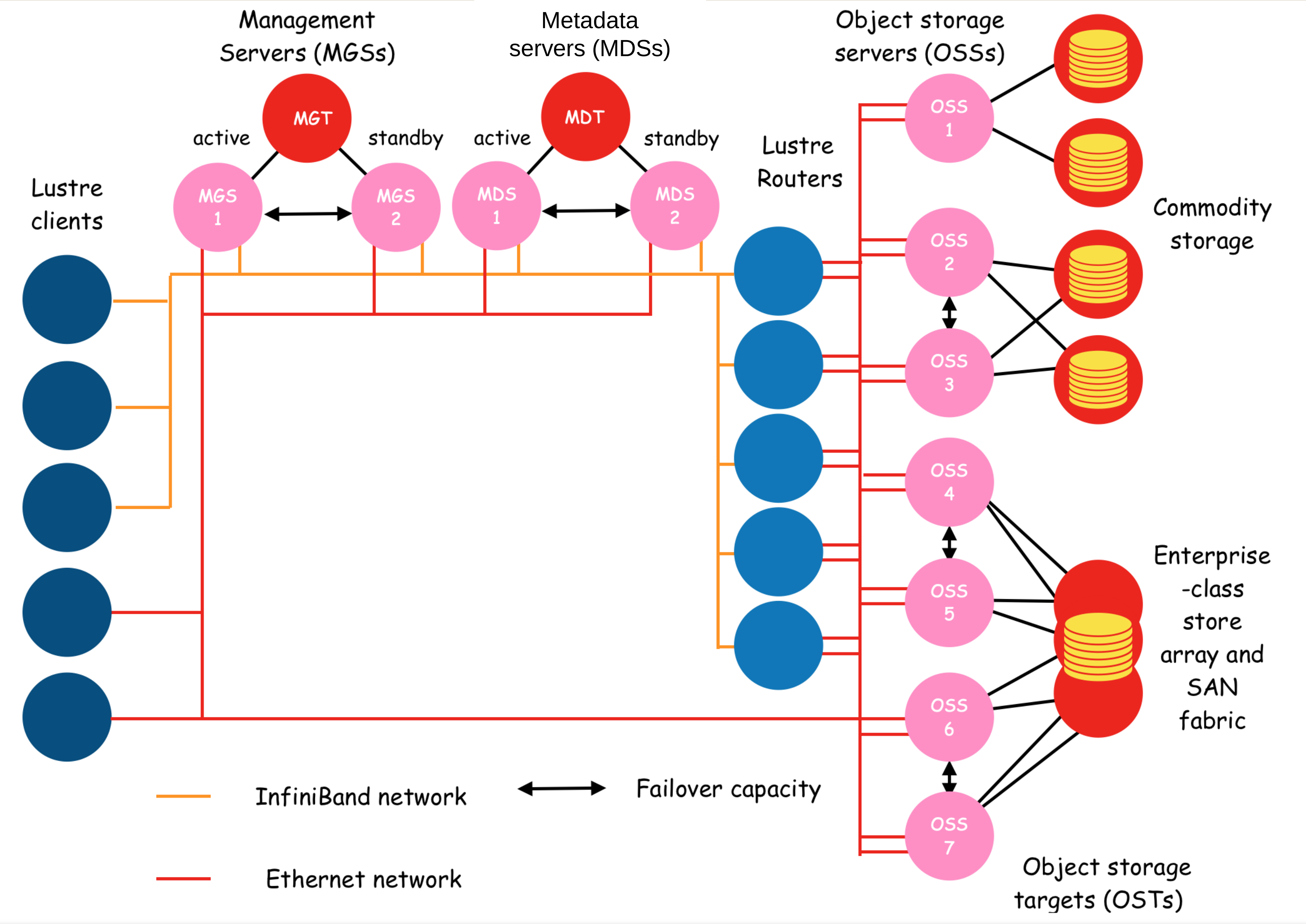
Replication Pipeline

- A client first gets a list of DataNodes from the NameNode.
- A client then flushes the data block to the first DataNode.
  - Upon receiving a data block (or a portion of a block) from a client,
    - DataNode1 writes each portion to its local file system and transfers that the portion to the DataNode2.
  - DataNode2 does the same thing as DataNode1 and transfer the received portion to DataNode3.
  - DataNode3 stores the data to its local file system.
- Data blocks are checksumed for failure detection.
  - A DataNode containing a bad block can retrieve a good replica from other DataNodes for block recovery.

Space Reclamations

- File deletion is not immediate performed in HDFS.
  - HDFS first renames it to a file in the "/trash" directory.
  - After the expiry of its life in "/trash", NameNode deletes the metadata of the file and asks relevant Datanodes to delete all datablocks.
- File undelete is possible.
  - If the deleted file is still in "/trash", it then can be undeleted.
- Replication Factor can be reduced to save space.
  - The NameNode selects excess replicas to be deleted.

The Lustre Distributed File System



Management Servers (MGSs)

Metadata servers (MDSs)

Object storage servers (OSSs)

Lustre clients

Lustre Routers

Commodity storage

Enterprise -class store array and SAN fabric

Object storage targets (OSTs)

InfiniBand network

Ethernet network

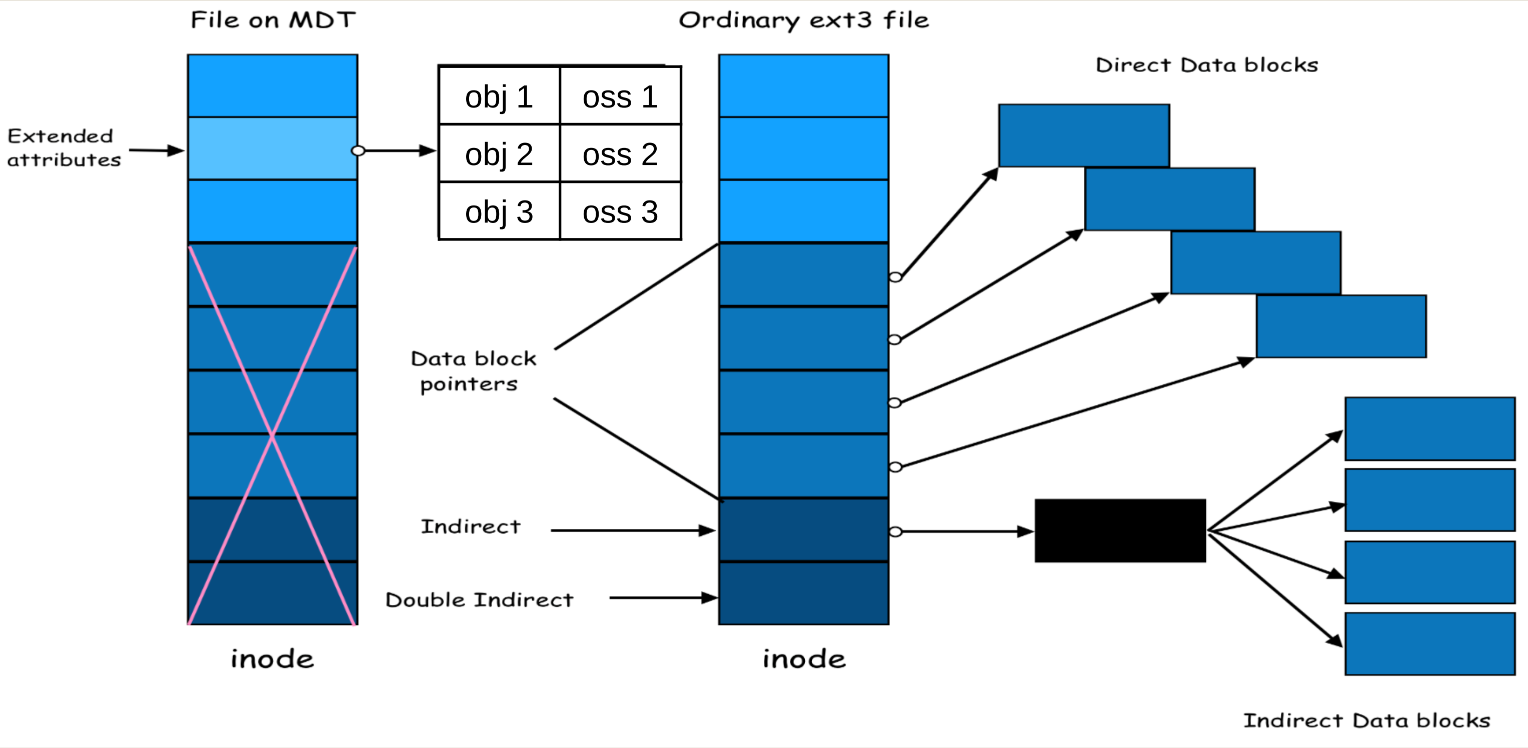
Failover capacity

System Components in Lustre

- MDSs (Metadata Servers)
  - They manage names and directories in Lustre.
  - MDT (Metadata Storage Target)
    - It provides physical storage for MDSs.
- OSSs (Object Storage Servers)
  - They provide file services for clients.
  - OSTs (Object Storage Targets)
    - They provide physical storages for OSSs.
- FSCs (File System Clients)
  - They access files through MDSs and OSSs.

File Creation Scenario in Lustre

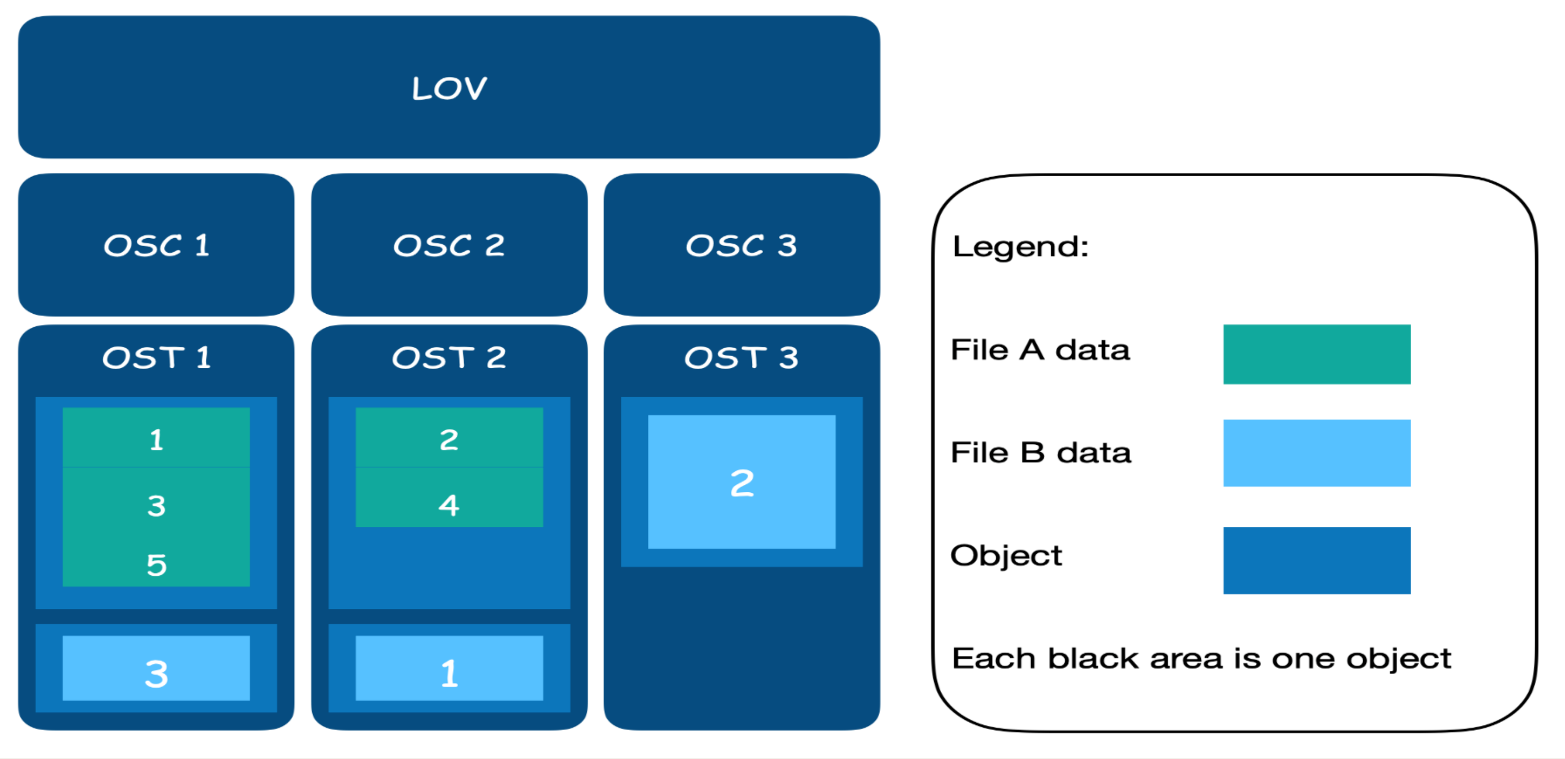
- Upon receiving a file creation request from a client,
  - a MDS creates an inode in the MDT.
    - The inode (metadata of the new file) records information about file objects and their associated OSSs.
  - The MDS then asks these OSSs to create file objects for future operations.



obj 1	oss 1
obj 2	oss 2
obj 3	oss 3

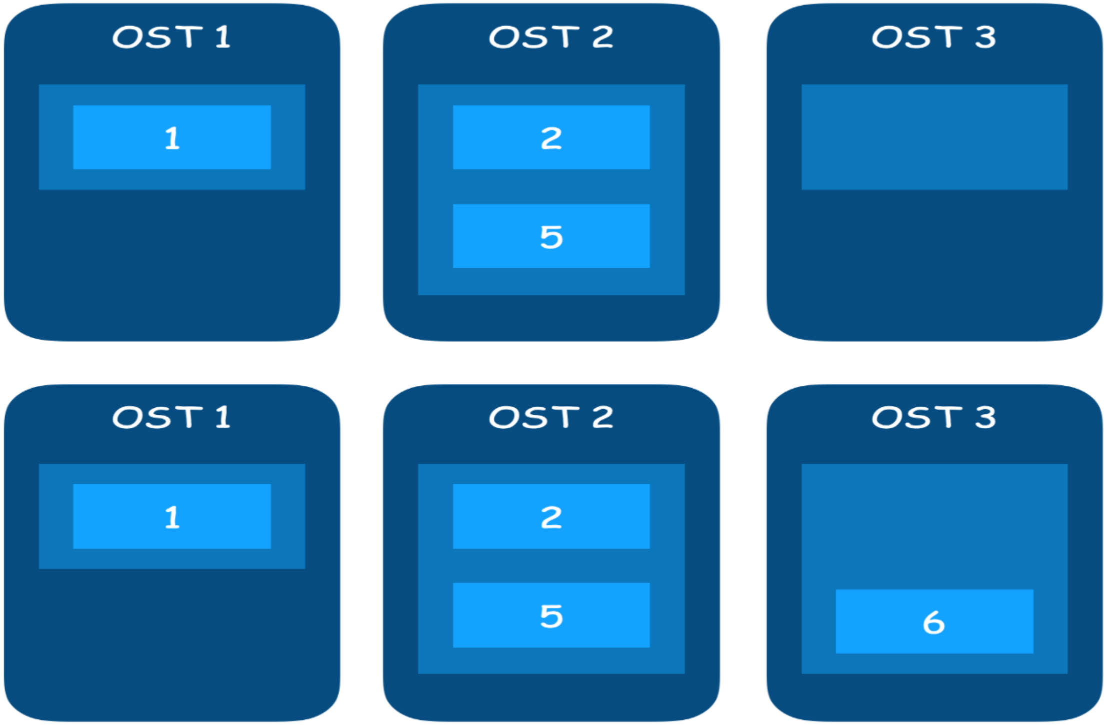
File Stripping in Lustre

- Each file in Lustre can be stripped into several objects.
  - In general, an object is served by an OSS.
  - Data accesses in Lustre are performed by blocks.
- Each object may contain several blocks.



A File Stripping Example

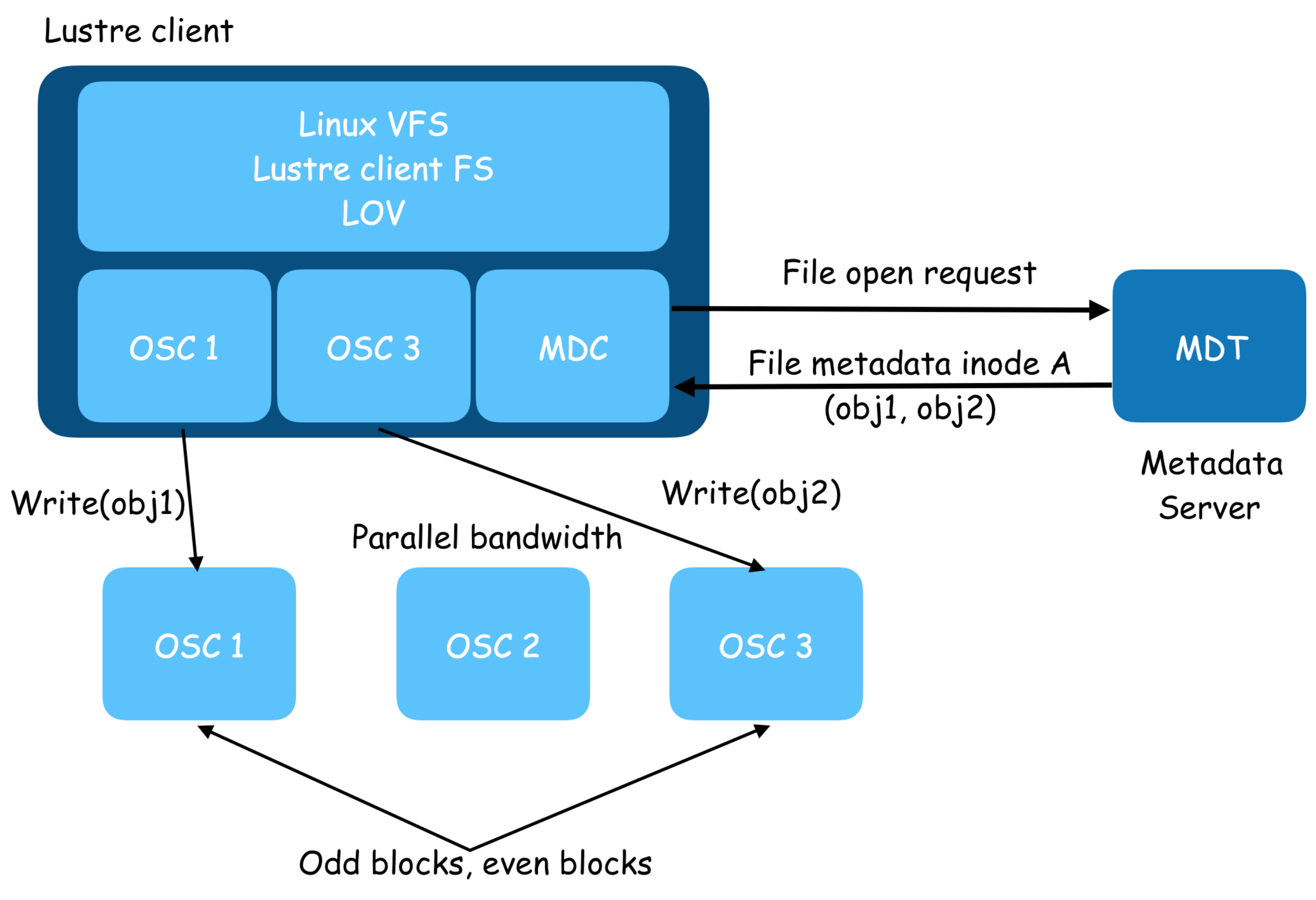
- A file of 6 blocks is stripped into 3 objects.
  - OSS1 handles object1 for block1 and block4.
  - OSS2 handles object2 for block2 and block5.
  - OSS3 handles object3 for block3 and block6.



Client 1,2,5 done
File size 5
One hole in the file (3,4)
1 empty and 1 short object

Client 1,2,5,6 done
File size 6
One hole in the file (3,4)
1 sparse and 1 short object

File Operations in Lustre



Lustre client

Linux VFS
Lustre client FS
LOV

OSC 1

OSC 3

MDC

File open request

MDT

File metadata inode A (obj1, obj2)

Metadata Server

Write(obj1)

Write(obj2)

Parallel bandwidth

OSC 1

OSC 2

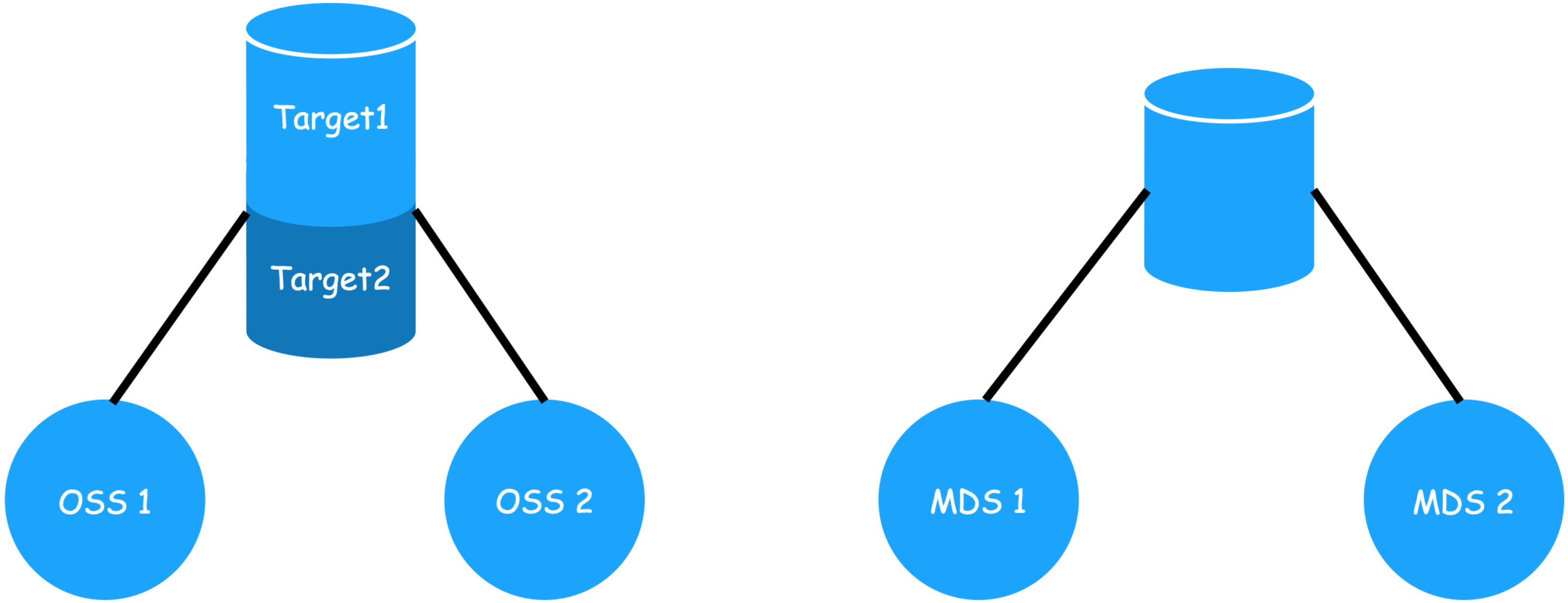
OSC 3

Odd blocks, even blocks

Failover in Lustre

Shared storage partitions
for OSS targets (OST)

Shared storage partitions
for MDS target (MDT)



OSS1 - active for target1, stand by for target2
OSS2 - active for target2, stand by for target1

MDS1 - active for MDT
MDS2 - standby for MDT