Distributed File System

Slide set 8
Distributed Systems

Graduate Level

K. N. Toosi Institute of Technology
Dr. H. Khanmirza
h.khanmirza@kntu.ac.ir

Distributed File Systems

- DFS allows multiple processes to share data over long periods of time in a secure and reliable way

- Why they are useful?
  - Data sharing among multiple users
  - User mobility
  - Location transparency
  - Backups and centralized management

Challenges

- Transparency (1/2)
  - Access transparency
    - One unified interface to access remote and local files
  - Location transparency
    - Client programs should see a uniform file name space. Files may be relocated without change of path
  - Mobility transparency
    - Files or volumes of files maybe moved, but neither client programs nor system administration tables in client nodes need to be changed when files are moved

Challenges

- Transparency (2/2)
  - Performance transparency
    - Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
  - Scaling transparency
    - The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Challenges

- Hardware and operating system heterogeneity
  - Interface for various OS and platforms
- Fault tolerance
  - Service continuation after failure
- Geographic distance and high latency
- Security
  - Access control to files
  - Needs Authentication

Challenges

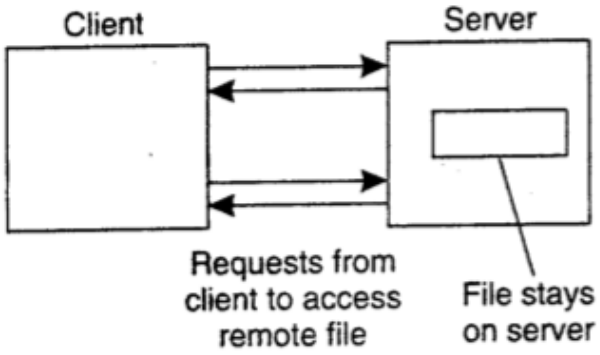
- File Replication for Scalability
  - Several copies of a file is kept in different locations for fault-tolerance or load-balancing
  - Concurrent file updates
  - Consistency
    - All readers should see the same content from a file
- Efficiency
  - Comparable with local file systems in performance and reliability

Architectures

- Client-server
- Cluster-Based
- Symmetric

Client-Server Architecture

- Remote Access Model
  - A file server provides a standardized view of its local file system
  - No matter what is the OS or the local file system of the server

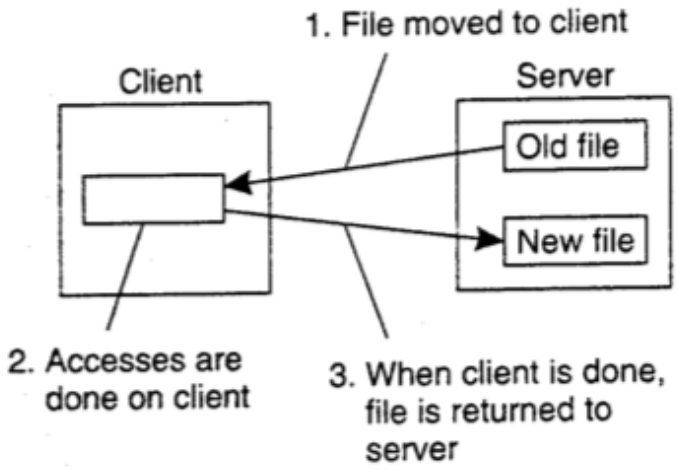


  - Client talks with server with a special protocol
  - Client can have any platform, OS, ⋯

  - Clients do RPC calls to access remote file system
  - This is transparent to client through an abstraction layer
  - The interface contains file operations

Client-Server Architecture

- Upload/Download Model
  - Client downloads files, works with it and then uploads, in case of any change
  - Like FTP



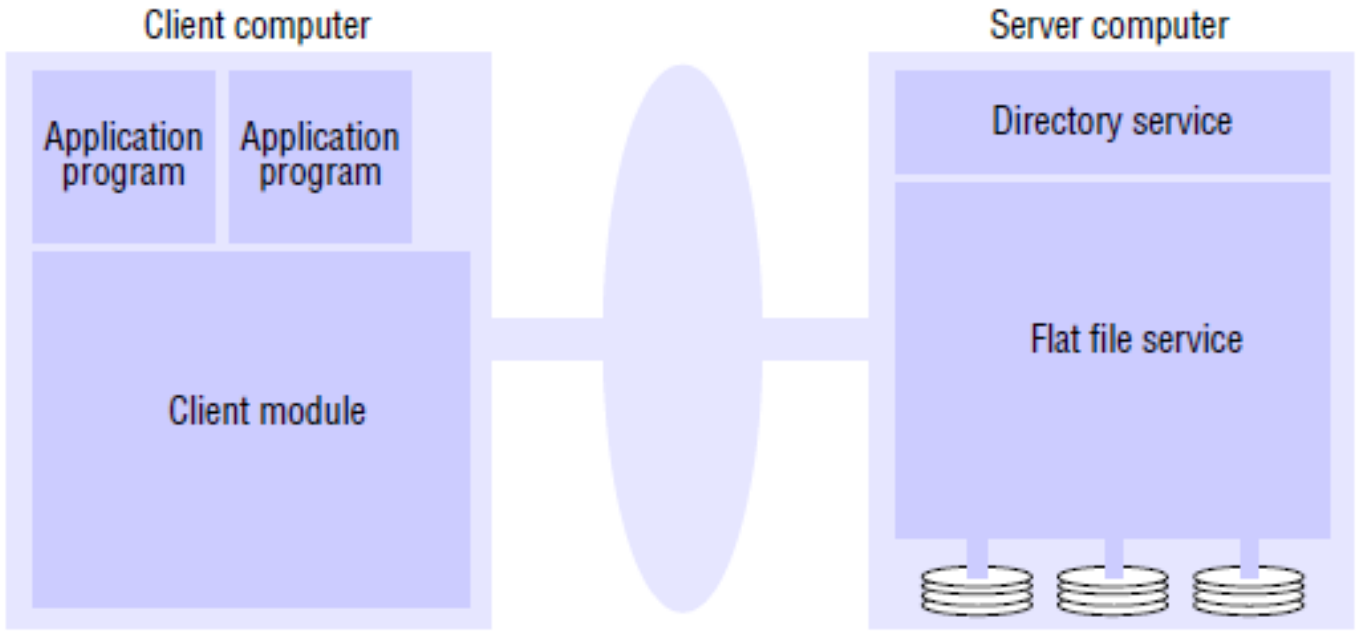
Cluster-Based DFS

- File server over a cluster of servers
- File stripping
  - Store a file in array of servers
  - Parallel access to one file
- Whole files across cluster
  - Store one file in one server, but distribute files over the cluster

Symmetric Architecture

- P2P & Distributed Hash Tables (DHT)

Main Modules



Flat File Service

- Concerned with implementing operations on the contents of files
- Each file have a Unique file identifier (UFID)
- UFID is generated by this module and is unique for a file in the whole distributed environment
- Flat file service provides an RPC interface for clients

Flat File Service

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

Flat File Service API

Directory Service

- This module is the client of flat file service
- Provides a mapping between text names for files and their UFIDs
- Each directory is stored as a conventional file with a UFID
- The module provides functions needed to work with directories
  - Generate directories
  - Add new file names to directories
  - Obtain UFIDs from directories
  - ...

Directory Service

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

Directory Service API

Client Modules

- Runs in each client computer and integrates and extends the operations of the flat file and the directory service under a single API
- In UNIX hosts, this module emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service.
- Holds information about the network locations of the flat file server and directory server processes
- It may provide caching service

Implementations ...

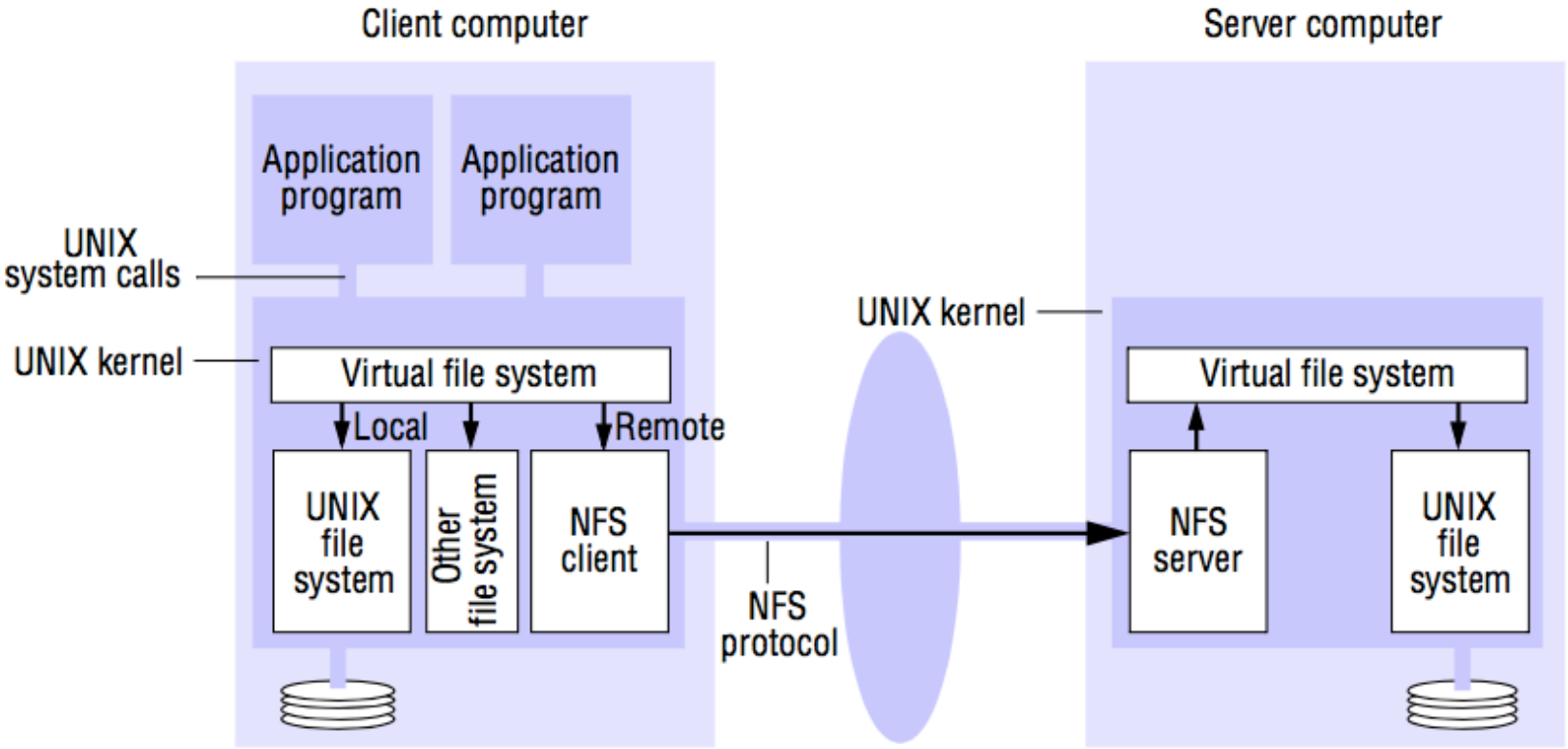
- NFS: Network File System
- AFS: Andrew File System
- Coda

NFS – Network File System

- Created by Sun Microsystems in 1984
- The first DFS product

- Features
  - Client-Server Arch.
  - Access transparency
  - Symmetric client-server architecture
    - A host can be both client and server
  - The main design principal is heterogeneity
    - It should work with all hardware platforms and operating systems

NFS



VFS: Virtual File System

NFS

- Operations on file
  - Open/close a file, check status of a file
  - Read/Write data from a file
  - Lock a file or part of a file
  - List files in a directory
  - Create/delete a directory
  - Delete/Rename a file, add a symlink
  - …

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

NFS

- NFS server in v[1-3] was stateless

- Advantages
  - Simplicity, easy to implement
  - No need to recovery in case of failure

- Disadvantages
  - Some operations cannot be implemented like file locking, authentication, ..
  - Separate daemons handled such circumstances

NFS

- NFSv4 is stateful
  - It is designed to work in WAN → need for caching due to delays
  - Addition of callbacks → server can do RPC on clients

AFS

- Andrew File System developed in CMU, designed as a campus computing and information system
- Supports sharing on a large scale by minimizing client-server communication
- Performs well with larger numbers of active users than other DFSs
- It is compatible with NFS as it uses the same file handles as NFS
- It is available in Linux (Linux AFS) and is used as the base in the Open Software Foundation's Distributed Computing Environment (DCE)

Design Principals

1. Files are small; most are less than 10 kilobytes in size
2. Read operations on files are more common than writes (~6 times)
3. Sequential access is common, and random access is rare
4. Most files are read and written by only one user.
   1. When a file is shared, it is often only one user who modifies it
5. Files are referenced in bursts
   1. If a file has been referenced recently, with high probability it will be referenced again in the near future

- These observations is not valid for databases, DB files does not fit in any class of regular files

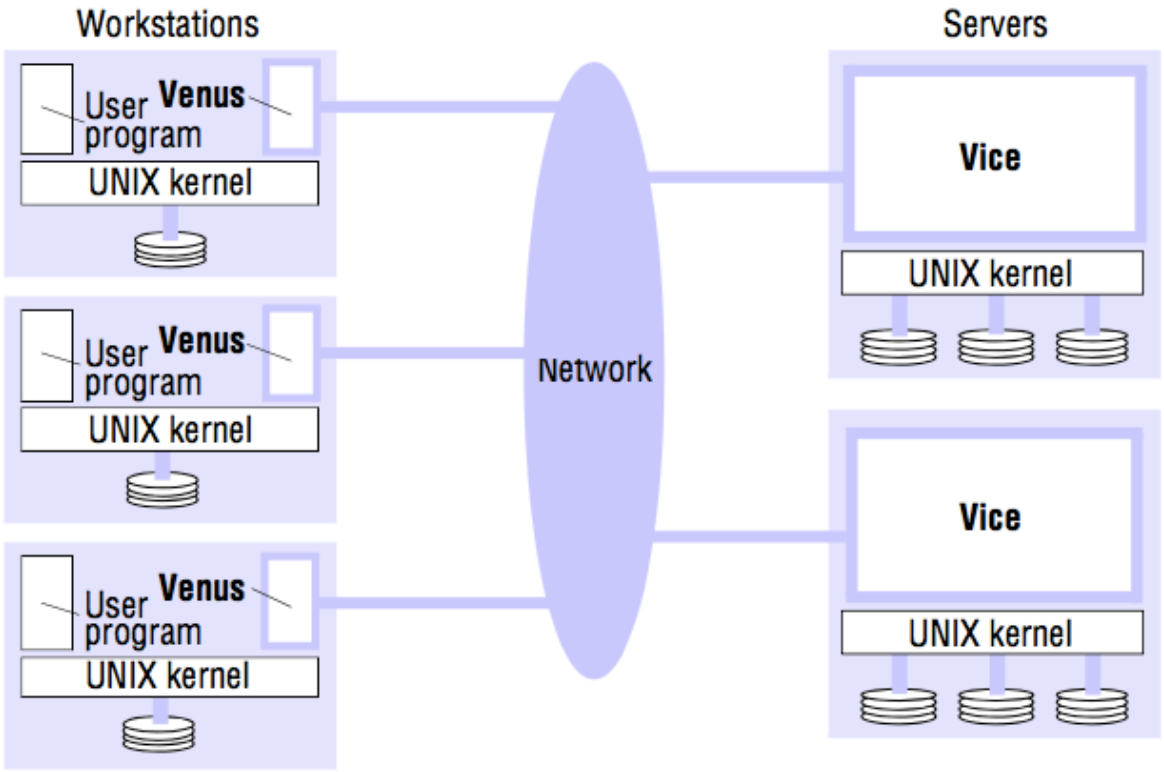
Design Characteristics

- Whole-file serving
  - The entire contents of directories and files are transmitted to client computers
- Whole-file caching
  - Copy of the downloaded file is stored in a cache on the local disk
  - Cache contains several hundred of the files most recently used
  - Cache is permanent, surviving reboots of the client computer
  - Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible.

Sample Scenario

- User process of a client issues an open system call for a file
  - File is located in the shared file space
  - The server holding the file is located and a request is sent
- The copy is downloaded and stored in the local file system
  - The copy is then opened and the resulting file descriptor is returned
- All read, write and other operations on the file are done locally
- The process issues a close system call
  - If the local copy has been updated the file is uploaded 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 subsequent use

Modules



- Vice: user level software in server
- Venus: user-level software in client

Other Characteristics

- AFS is a multi-thread FS
  - Tables are held in memory and shared between threads
- AFS supports read-only replicas
- AFSv3 supports partial file caching
- Performance:
  - For a 18-client benchmark while AFS server load was 40%, for NFS it was 100%
- AFSv3 supports WAN deployment

Coda

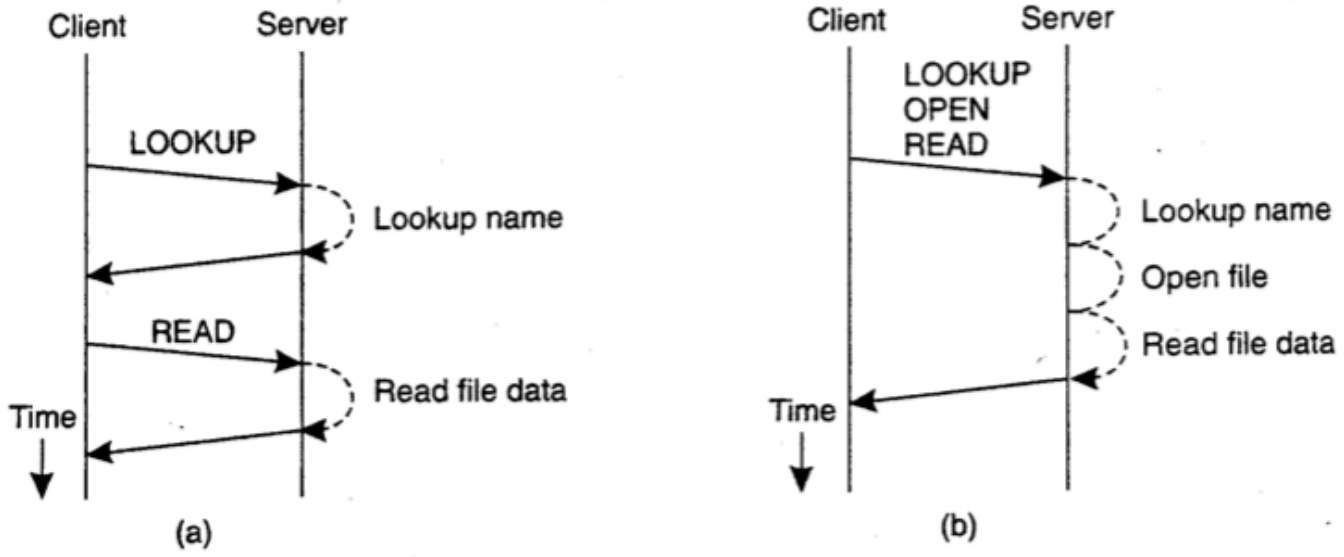
- Based on AFS2
- Main difference with AFS
  - AFS allows only one write server, all other servers are read-only replicas
  - Coda allows all servers to receive updates
- Coda benefits
  - Better Availability
  - Sharing in large scale
  - Disconnected Operations (network partitions)
    - Support for portable devices

Distributed File System

Communication

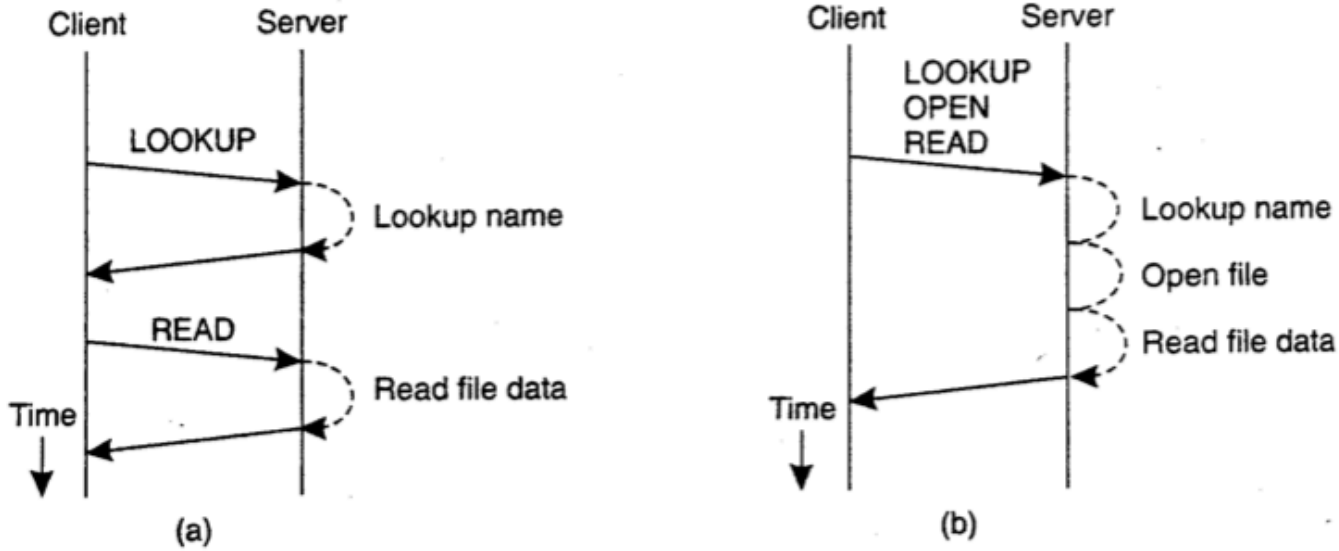
NFS

- NFS uses simple RPC
  - Open Network Computing RPC standard
- Before NFSv4
  - Implementation of server was easy by keeping requests relatively simple
  - The strategy is not good for WAN deployment



NFS

- NFSv4 supports compound procedures
  - Several RPCs can be grouped into a single request
  - No transactional concepts
    - Executed up until a fault occur and reported back the achieved result to the client
  - No protection against concurrent requests



Coda

- Uses RPC2 protocol
  - Reliable RPC on UDP
  - Support for side effects
    - Opening another connection for exchanging application-specific messages
  - Multicasting / MultiRPC
    - For notifying a file change for all of the owners of the file

Distributed File System

Naming

Read yourself!

Distributed File System

Sharing, Replication & Consistency

No sharing no problem!

Sharing & Replication

- Replication in DFS is naturally done through caching files in clients and leads to consistency issues

- Sharing
  - In some conditions several processes use the same file read or write
  - A file is used by one process, but it is cached in several clients

  - More general concept than replication

File Sharing Semantics

- When several clients read or write the same file the semantics should be defined precisely
  - UNIX semantics
  - Session semantics
  - Immutable semantics
  - Transaction semantics

UNIX Semantics

- When a read follows a write, read operation must return the recent data
- After two successive writes, read operation must return the last write data
- It is easy in one machine

UNIX Semantics

- Implementation in distributed system is easy if
  - Network has one write file server
  - Clients do not cache anything
    - All reads and writes go directly to the file server
    - Requests are processed strictly sequentially
  - Very poor performance!
    - Increasing performance with allowing clients keep local caches write updates immediately to the server

Session Semantics

- Changes to an open file are initially visible only to the process (or possibly machine) that has modified the file
- Only when the file is closed the changes are made visible to other processes (or machines)
  - Download/Upload model
- What happens when two clients download a file and change their local caches?
  - Just overwrite server file with the last update! (have order or non-deterministic?)

Immutable Semantics

- Client can create or open a file for reading, not writing!
- When need to change
  - Client creates a new file with the same name
  - Uploads the file to the same directory (atomically replaces the old file)
  - Very easy to implement
- What happens if two processes attempt to replace file?
  - The last one or any of them (non-deterministic) will be visible
- Issue: file is replaced while another process is reading it!
  - Let the process continue reading the old file
  - Detect file change and return error for subsequent attempts for reader process

Transactional Semantics

- When a process starts to use a file, issues a begin transaction
- At the end, issues an end transaction command (commit)
- In case of parallel transactions, the system ensures that the final result is the same as if they were all run in some (undefined?) sequential order
  - Distributed Commit Semantic

File Sharing Issues

- File Locking
- Client-Side Caching
- Server-Side Replication & Consistency

NFS File Locking

- Synchronizing access to shared files

- NFSv4 Lock Models (1/2)
  - Explicit Lock Model
    - It supports read & write locks
    - Operations are done for a byte-range not the whole file
    - Simultaneous read locks are supported
    - Each lock has four operation
      - Lock: non-blocking, it returns error if lock is granted to other process
      - Lockt: test lock availability
      - Locku: unlock
      - Renew: Client requests if lock is not available, server put it in a queue. When lock is available it is granted in an order to clients, but client must renew its request periodically

NFS File Locking

- NFSv4 Lock Models (2/2)
  - Implicit Lock model or Share Reservation
    - When opening a file, client requests type of access (READ, WRITE, BOTH) and type of access the server should deny other clients (NONE, READ, WRITE, BOTH)

- Locks are different problem than sharing semantics
  - When a file is locked and changed by a process then when clients must see the results?

NFS Server Caching

- NFS servers use the cache at the server machine just as it is used for other file accesses
  - Serve the read operations from the cache without disk access

- Disk write strategies
  - Write-through
    - Modified data is written into the disk
  - Commit
    - Modified data is stored in disk if client issue a commit command
    - Commit is issued when a file is closed in the client
  - All read request are served from the disk data

NFS Server Caching

- UNIX supports Delayed-Write
  - When a page has been altered, its new contents are written to disk only when the buffer page is required for another page.
  - To guard against loss of data in a system crash, the UNIX sync operation flushes altered pages to disk every 30 seconds
  - Delayed-Write optimizes write operation

NFS Client Caching

- Client caches results of operations in mem or/and disk cache
  - Like read, write, getAttr, lookup, readdir operations
- Clients are responsible for polling server for fresh data
  - Changes in a client is not reflected immediately to others
    - This leads to existence of inconsistent data in clients
- NFSv4 uses two different approaches for caching
  - Simple timestamp-based method
  - Open delegation

Timestamp-Based

- A request for opening a file is issued from a user
- If the requested block of the file isn't in cache, it requests from the server
- NFS caches blocks in client, not the whole file
- If the block is in cache, it must be revalidated for possible modification
  - If time of previous checking has not passed a threshold just serve from cache
  - If time is passed check modification from the server
  - If not modified, open the file from the cache

Timestamp-Based

- Write operations is applied to cache, after the file is closed, modified blocks are flushed-back to the server
  - This approach corresponds to the session semantics
  - NFSv4 does not inform other clients about the file modification

- NFS is not recommended for cooperative or interactive applications
  - Delay to write the changes from updating client
  - Delay of other clients to be notified of changes

Open Delegation

- Only in NFSv4
- The client machine is allowed to locally handle open and close operations on the same machine
- The server is in charge of
  - Checking should opening the file succeed or not (to check share reservations)
  - Handle locking requests from clients on other machines
- If file is delegated for write, other write requests are denied by the server (according to share reservation)
- After some time server recalls delegation
  - Needs callback from server to client → needs stateful server
- It partly corresponds to Transaction semantics

NFS Client Caching

- Callback vs. Timestamp
  - The timestamp approach
    - Number of interactions between client and server is high
    - Needs client and server to be time-synchronized
  - The callback- based approach
    - Results in less communication between client and server
    - Activity takes place in the server only when the file has been updated
  - Knowing that
    - Majority of files are not accessed concurrently
    - Read operations predominate over writes in most applications
  - Callback dramatically reduces the number of client-server interactions
  - Callback is more scalable

NFS Server-Side Replication

- NFS does not support file replication with updates
  - One write server with multiple read-only replicas
  - A separate daemon, NIS, adds replication feature to NFS

AFS Client Caching

- AFS uses the same caching strategies as NFS
  - Timestamp based
  - Callback-based with some differences

AFS Client Caching

- When a file is copied to a client a callback promise is also issued
  - It can be Valid or Canceled.
    - When a file is updated by another client, server callbacks all other clients having the file and cancels their promise
    - When opening a file this status is checked, if it is in Cancel mode a fresh copy is downloaded from the server
  - In concurrent write condition, the last write silently overwrites others!
  - In case of client failure, or passing a time T, callback promises must be renewed
  - Server needs to keep some state about promises

Coda File Sharing Semantics

- Uses session semantics
- It also uses a technique very close to NFS Share Reservation
  - Client specifies opening mode when requests for opening a file
    - READ or WRITE
  - When a process opens a file for writing, other write requests will fail but reads will proceed, only one process can write to a file at a time (= locking)
  - When write is completed and file is closed, it is transferred to server
  - Server sends invalidation messages to all others having the file
    - They may decide to continue reading or re-open or report error

CODA Server-Side Replication

- Volume is the unit of replication which is a collection of files
  - In UNIX volume corresponds to disk partition
- Volume Storage Group (VSG)
  - The collection of Coda servers that have a copy of a volume
- Accessible Volume Storage Group (AVSG)
  - The volume's VSG that the client can contact at the moment
  - If the AVSG is empty, the client is said to be disconnected

CODA Server-Side Replication

- Coda uses a replicated-write protocol to maintain consistency of a replicated volume
  - Optimistically, it allows clients modify the file locally
  - Coda uses a variant of Read-One, Write-All (ROWA)
    - When requesting a file, it uses one of the servers in AVSG
    - When writing, the updated file is sent to all AVSG servers using multiRPC
- In case of failures some parts of network may not receive the updated version of a file which leads to inconsistency

CODA Server-Side Replication

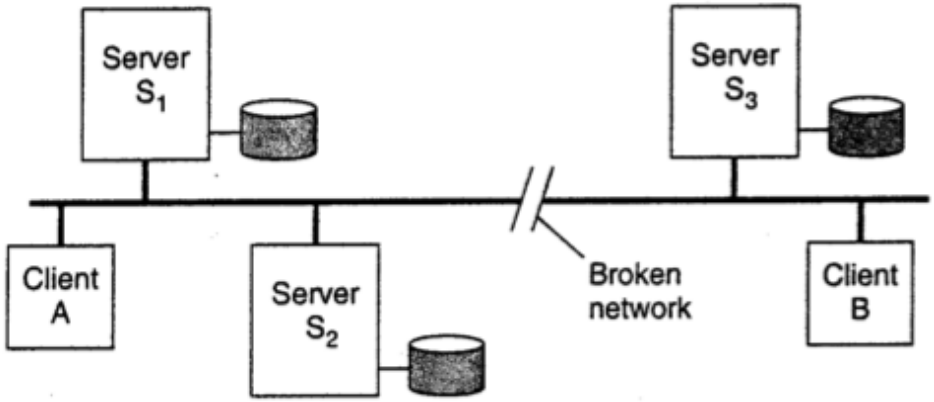
- Inconsistency solution
  - Coda Version Vector (CVV) is attached to each version of a file
  - CVV in i[th] server: $cvv_i = [v_1, \dots, v_i, \dots, v_n]$.
  - $CVV_i[j]$ shows the version of a file on S_j from the S_i viewpoint
    - Looks familiar??
  - If $\exists i, \forall j \neq i \; CVV_i > CVV_j$ then there is no conflict
    - There is a conflict if $CVV_i \not> CVV_j \; and \; CVV_j \not> CVV_i$
  - Coda does not resolve conflicts automatically 🤯

CODA Server-Side Replication

- Inconsistency solution
  - When a modified file is closed in a client
  - Client sends an updated content and the current CVV of the file
  - Receiving members, check the CVV and if the CVV is greater than their CVV, store the content and reply with positive ack
  - Client computes new CVV and sends to AVSG members
  - Only AVSG members receive new update not the whole VSG

CODA Server-Side Replication

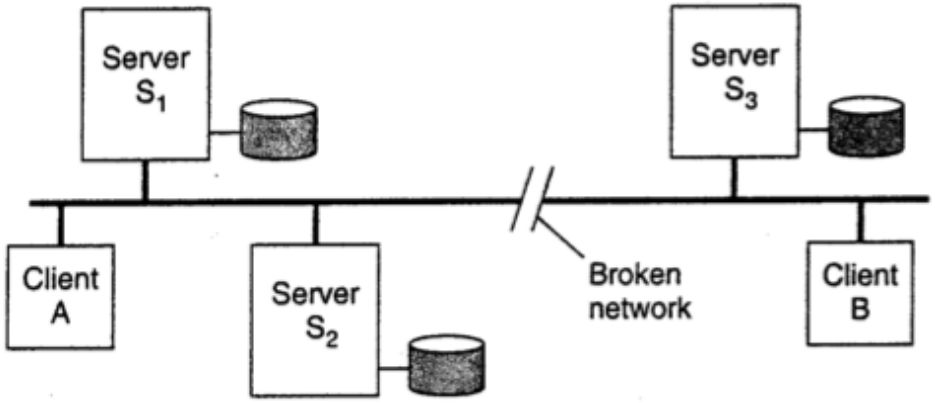
1. Initially $CVV_1 = CVV_2 = CVV_3 = [1,1,1]$
2. Failure occurs
3. A reads file from S_1.
4. A updates the file and sends to S_1 and S_2.
5. $CVV_1 = CVV_2 = [2,2,1]$
6. B reads the same file from S_3



CODA Server-Side Replication

7. B updates file and sends it to S_3.
8. $CVV_3 = [1,1,2]$
9. Link is established
10. B is notified change of AVSG – requests CVV
11. Finds two [2,2,1] and a [1,1,2] vector → a conflict

If items 6-7 is removed CVVs are [2,2,1] and [1,1,1] which is easily resolved by copying S_1 or S_2 copy on S_3



CODA Server-Side Replication

- Coda enhances availability by
  - Replication of files across servers
  - Ability of clients to operate entirely out of their caches

Distributed File System

Google File System (GFS)

Why another FS?

- Crawl the whole web
  - Store it all on one big disk
  - Does not scale!!

GFS – Google File System

- Appears as a single disk
- Runs on top of a native filesystem
- Fault tolerant: can handle disk crashes, machine crashes,…
- Hadoop Distributed File System (HDFS) is an open source Java product similar to GFS

Design Assumptions

- The system is built from many inexpensive commodity components that often fail
- The system has large files > 100 MB
- Workload consists
  - Large sequential reads – Large sequential appends
  - Small random reads – Small random writes (overwrites)

GFS Designed for

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file
- Streaming data access
  - Data is written once and read many times
  - Optimized for batch reads rather than random reads
- Cheap commodity hardware
  - No need for super-computers, use less reliable commodity hardware

GFS is not good for

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data
- Large amount of small files
  - Better for millions of large files instead of billions of small files
- Multiple writers
  - Single writer per file
  - Writes only at the end of file, no-support for arbitrary offset

GFS

- GFS provides standard operations create, delete, open, close, read, and write
- Append operation allows multiple clients do append simultaneously
- GFS is not POSIX-compliant

GFS

- Files are split in chunks

- Chunks are
  - Single unit of storage: a contiguous piece of information on a disk.
  - Chunks are traditionally either 64MB or 128MB: default is 64M

- Why chunks are large?
  - Reduces the number interactions between client & master
  - Client can do more on large chunks, reducing network overhead
  - Having smaller seek tables/time

GFS Architecture

- Components
  - GFS master
  - GFS chunk servers
  - GFS client

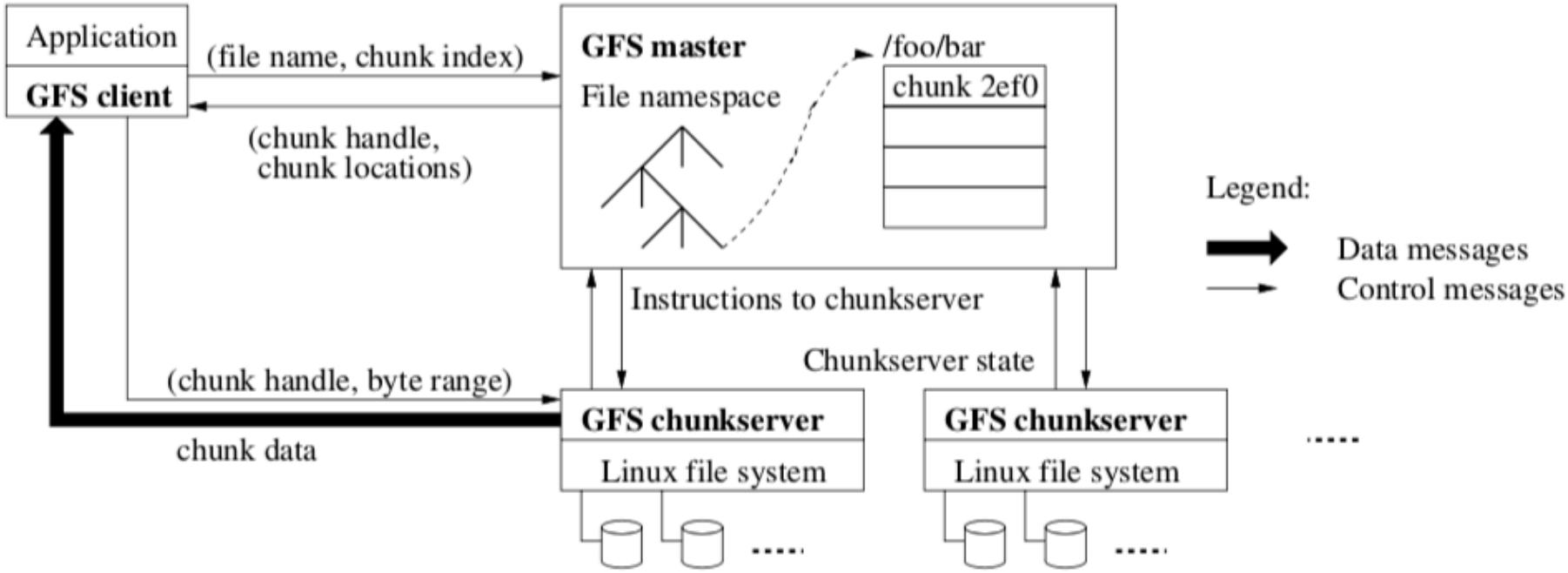


Figure 1: GFS Architecture

GFS Master

- Manages file namespace operations
- Manages file metadata (holds all metadata in memory)
  - Access control information
  - Mapping from files to chunks
  - Locations of chunks
- Manages chunks in chunk servers
  - Creation/deletion
  - Placement
  - Load balancing
  - Maintains replication
  - Garbage collection

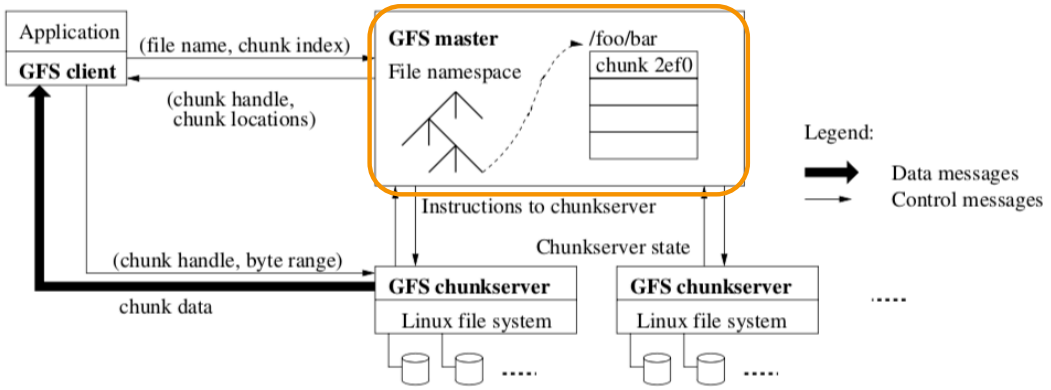


Figure 1: GFS Architecture

GFS Chunk Server

- Store chunks as a regular Linux file
- Talks with master
  - Manages chunks
  - Maintains consistency
  - Reports what chunks has

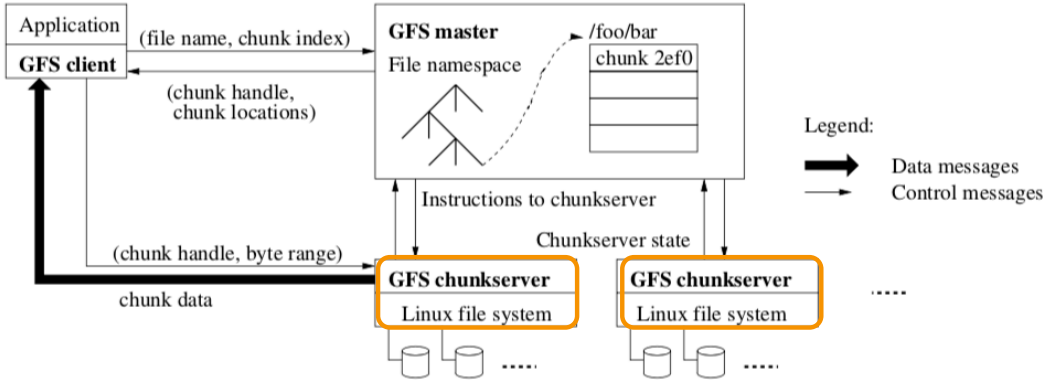


Figure 1: GFS Architecture

GFS Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunk servers
- Caches metadata, but does not cache data.

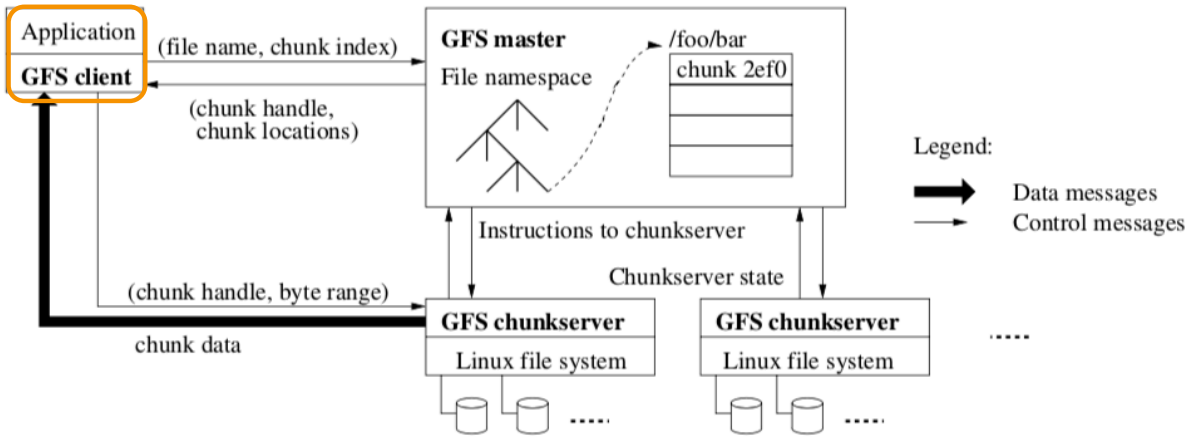


Figure 1: GFS Architecture

Namespace & Locking

- No per-directory data structure
- No hard or symbolic links
- Represents its namespace as a lookup table mapping full pathnames to metadata

Namespace & Locking

- Each master operation acquires a set of locks before it runs
  - Read lock on internal nodes and read/write lock on the leaf
  - Allowed concurrent mutations in the same directory
  - Read lock on directory prevents its deletion, renaming or snapshot

- Appending to a file “/home/usr/docs/file” Needs
  - read lock for home, usr, docs directories
  - write lock for “file”

Replication

- Each chunk is kept in multiple chunk servers (typically 3)
  - Causes Reliability, Availability, Bandwidth utilization
- Replica location is decided by Master
  - Local rack close to the creator of file
  - Another machine in the same rack
  - Another machine in another rack

File Operations

- Creation
  - Place new replicas on chunk servers with below-average disk usage
  - Limit number of recent creations on each chunk servers
- Re-replication
  - When number of available replicas falls below a user-specified goal
- Rebalancing
  - Distribution of replicas is analyzed periodically for better disk utilization and load balancing

File Operations

- Deletion
  - File deletion logged by master
  - File renamed to a hidden name with deletion timestamp
  - Master regularly deletes files older than 3 days (configurable)
    - Deletion is done lazily
    - Until then, hidden file can be read and undeleted
    - When a hidden file is removed, its in-memory metadata is erased

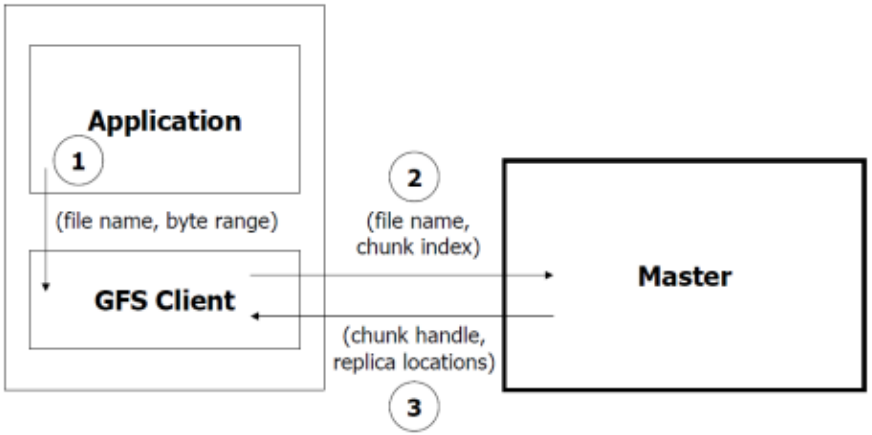
Stale Replica

- Chunk replicas may become stale
  - If a chunk server fails and misses changes to the chunk while it is down
  - Need to distinguish between up-to-date and stale replicas
  - Chunk version number:
    - Increased when master grants new lease on the chunk
    - Not increased if replica is unavailable
- Stale replicas deleted by master in regular garbage collection
  - Some clients may read stale data (It is OK!)

File Operations

Read

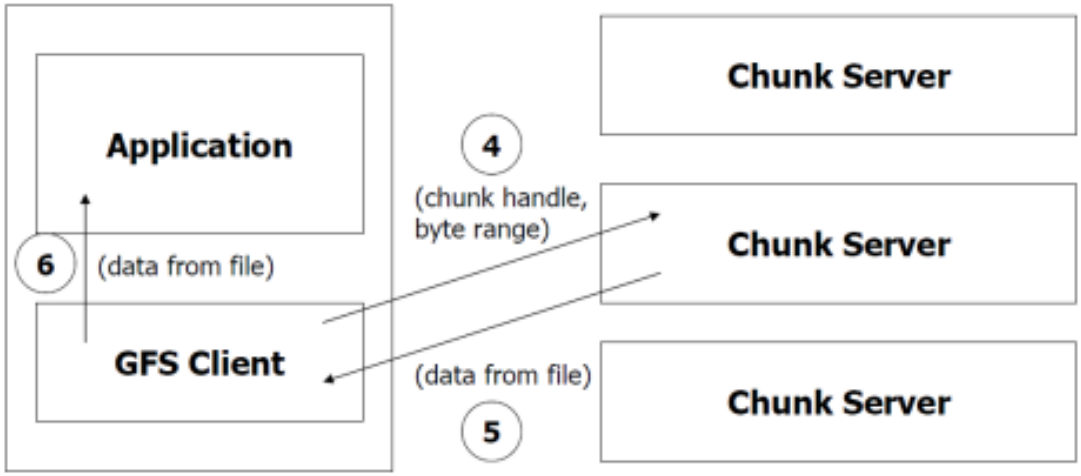
1. Application originates the read request
2. GFS client translates request and sends it to the master
3. The master responds with chunk handle and replica locations



File Operations

Read

4. The client picks a location and sends the request
5. The chunk server sends requested data to the client
6. The client forwards the data to the application



File Operations

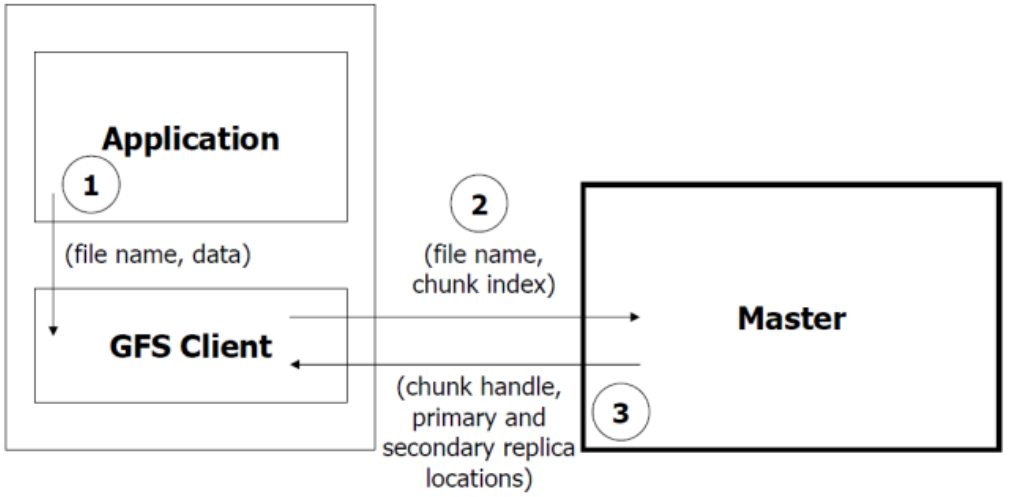
- Update
  - An operation that changes the contents or metadata of a chunk
  - For consistency, updates to each chunk must be ordered in the same way at the different chunk replicas
  - Consistency means that replicas will end up with the same version of the data and not diverge

Update

- For each chunk, one replica is designated as the Primary
  - The other replicas are designated as secondaries
- Primary defines the update order, all secondaries follows this order
- Primary is selected by Master
  - It grants a lease for a chunk for a chunk server
  - Chunk-server holds the lease for a period T
  - Chunk-server can refresh the lease endlessly, but if the chunk server can not successfully refresh lease from master, stops being a primary and master gives lease to another replica

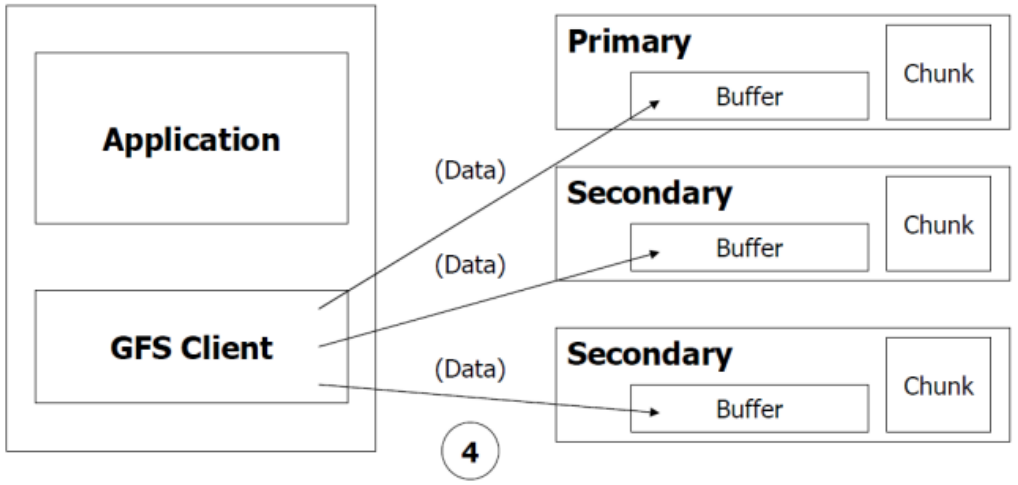
Update

1. Application originates the request
2. The GFS client translates request and sends it to the master
3. The master responds with chunk handle and replica locations



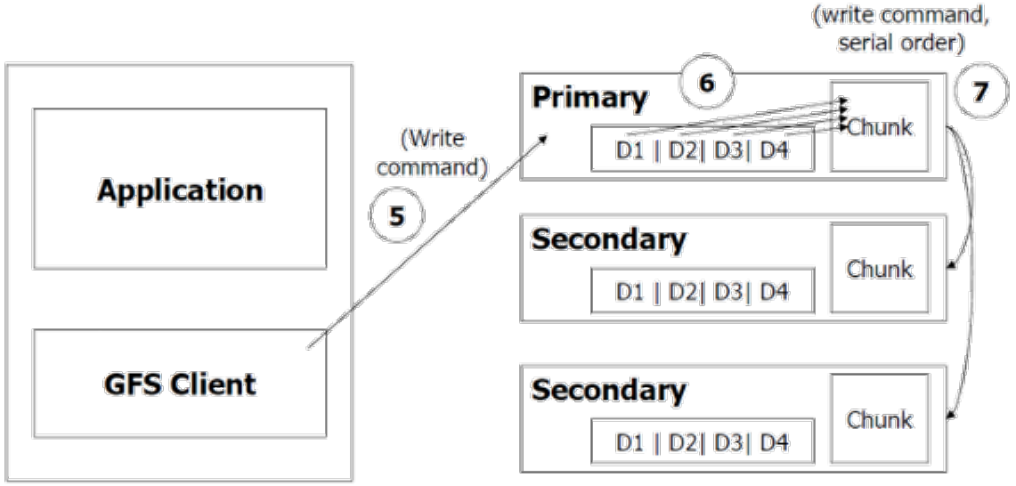
Update

4. The client pushes write data to all locations. Data is stored in chunk-server's internal buffers



Update

5. The client sends write command to the primary
6. The primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk
7. The primary sends the serial order to the secondaries and tells them to perform the write.



Update

- Append is similar to write but offset is determined by Master
- Append operations follow append-at-least-once semantics
- Readers can detect duplicates with extra information like checksums or unique identifiers

Consistency

- Primary enforces
  - One update order across all replicas for concurrent writes
  - Waits until a write finishes at the other replicas before it replies

- File regions may end up containing mingled fragments from different clients since writes to different chunks may be ordered differently by their different primary chunk-servers

Fault Tolerance

- Data integrity
  - Checksum for each chunk divided into 64KB blocks
  - Checksum is checked every time an application reads the data

- Chunk
  - All chunks are versioned
  - Version number updated when a new lease is granted
  - Chunks with old versions are not served and are deleted

Fault Tolerance

- Master Machine
  - Logs every operation
  - In some points, a checkpoint is created to compress logs and to reduce initialization time
  - Master state replicated for reliability on multiple machines
- Heartbeat messages
  - Checking liveness of chunk-servers
  - Piggybacking garbage collection commands
  - Lease renewal

The End!