
DISTRIBUTED SYSTEMS (COMP9243)

Lecture 9b: Distributed File Systems

Slide 1

- ① Introduction
 - ② NFS (Network File System)
 - ③ AFS (Andrew File System)
 - ④ Coda
 - ⑤ GFS (Google File System)
-

INTRODUCTION

Distributed File System Paradigm:

- File system that is shared by many distributed clients
- Communication through shared files
- Shared data remains available for long time
- Basic layer for many distributed systems and applications

Slide 2

Clients and Servers:

- Clients access files and directories
 - Servers provide files and directories
 - Servers allow clients to perform operations on the files and directories
 - Operations: add/remove, read/write
 - Servers may provide different views to different clients
-

CHALLENGES

Transparency:

- **Location**: a client cannot tell where a file is located
- **Migration**: a file can transparently move to another server
- **Replication**: multiple copies of a file may exist
- **Concurrency**: multiple clients access the same file

Slide 3

Flexibility:

- Servers may be added or replaced
- Support for multiple file system types

Dependability:

- **Consistency**: conflicts with replication & concurrency
 - **Security**: users may have different access rights on clients sharing files & network transmission
 - **Fault tolerance**: server crash, availability of files
-

Performance:

- Requests may be distributed across servers
- Multiple servers allow higher storage capacity

Scalability:

Slide 4

- Handle increasing number of files and users
 - Growth over geographic and administrative areas
 - Growth of storage space
 - No central naming service
 - No centralised locking
 - No central file store
-

THE CLIENT'S PERSPECTIVE: FILE SERVICES

Ideally, the client would perceive remote files like local ones.

File Service Interface:

- Slide 5**
- **File**: uninterpreted sequence of bytes
 - **Attributes**: owner, size, creation date, permissions, etc.
 - **Protection**: access control lists or capabilities
 - **Immutable files**: simplifies caching and replication
 - *Upload/download model versus remote access model*
-

FILE ACCESS SEMANTICS

UNIX semantics:

- Slide 6**
- A `READ` after a `WRITE` returns the value just written
 - When two `WRITES` follow in quick succession, the second persists
 - Trivial with a single file server and without caching, but...
 - A single file server contradicts scalability
 - Caches are needed for performance & write-through is expensive
 - UNIX semantics is too strong for a distributed file system
-

Session semantics:

- Changes to an open file are only locally visible
- When a file is closed, changes are propagated to the server (and other clients)
- But it also has problems:
 - What happens if two clients modify the same file simultaneously?
 - Parent and child processes cannot share file pointers if running on different machines.

Slide 7

Immutable files:

- Files allow only `CREATE` and `READ`
 - Directories can be updated
 - Instead of overwriting the contents of a file, a new one is created and replaces the old one
 - ✗ Race condition when two clients replace the same file
 - ✗ How to handle readers of a file when it is replaced?
-

Atomic transactions:

- Slide 8**
- A sequence of file manipulations is executed indivisibly
 - Two transaction can never interfere
 - Standard for databases
 - Expensive to implement
-

THE SERVER'S PERSPECTIVE: IMPLEMENTATION

Design Depends On the Use:

- Most files are small—less than 10k
- Reading is much more common than writing
- Usually access is sequential; random access is rare
- Most files have a short lifetime
- File sharing is unusual
- Most process use only a few files
- Distinct files classes with different properties exist

Slide 9

This was found by Satyanarayanan for the use of UNIX in a university.

There are also varying reasons for using a DFS:

- Big file system, many users
 - High performance
 - Fault tolerance
-

STATELESS VERSUS STATEFUL SERVERS

Advantages of *stateless* servers:

- Fault tolerance
- No OPEN/CLOSE calls needed
- No server space needed for tables
- No limits on number of open files
- No problems if server crashes
- No problems if client crashes

Slide 10

Advantages of *stateful* servers:

- Shorter request messages
 - Better performance
 - Read ahead easier
 - File locking possible
-

CACHING

We can cache in three locations:

- ① Main memory of the server: easy & transparent
- ② Disk of the client
- ③ Main memory of the client (process local, kernel, or dedicated cache process)

Cache consistency:

- Obvious parallels to shared-memory systems, but other trade offs
 - No UNIX semantics without centralised control
 - Plain *write-through* is too expensive; alternatives: delay WRITES and agglomerate multiple WRITES
 - *Write-on-close*; possibly with delay (file may be deleted)
 - Invalid cache entries may be accessed if server is not contacted whenever a file is opened
-

Slide 11

REPLICATION

Multiple copies of files on different servers:

- Prevent data loss
- Protect system against down time of a single server
- Distribute workload

Slide 12

Three designs:

- **Explicit replication**: The client explicitly writes files to multiple servers (not transparent).
 - **Lazy file replication**: Server automatically copies files to other servers after file is written.
 - **Group file replication**: WRITES simultaneously go to a group of servers.
-

Slide 13

Update protocols:

- *Primary copy replication*: Primary logs changes; then, applies them locally; and finally, instructs secondaries.
- *Voting*: Multiple servers have to give permissions before a client can read or write.

CASE STUDIES

Slide 14

- Network File System (NFS)
- Andrew File System (AFS)
- Coda
- Google File System (GFS)

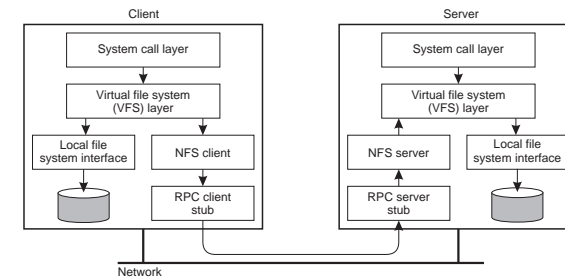
NETWORK FILE SYSTEM (NFS)

Properties:

Slide 15

- Introduced by Sun
- Fits nicely into UNIX's idea of mount points, but does **not** implement UNIX semantics
- Multiple clients & servers (a single machine can be a client and a server)
- Stateless servers (no OPEN & CLOSE) (changed in v4)
- File locking through separate server
- No replication
- ONC RPC for communication

Slide 16



Slide 17

Server side:

- NFS protocol independent of underlying FS
- NFS server runs as a daemon
- `/etc/export`: specifies what directories are exported to whom under which policy
- Transparent caching

Slide 18

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

Slide 19

Client side:

- Explicit mounting versus automounting
- Hard mounts versus soft mounts
- Supports diskless workstations
- Caching of file attributes and file data

Caching:

- Implementation specific
- Caches result of `read`, `write`, `getattr`, `lookup`, `readdir`
- Consistency through polling and timestamps
- Cache entries are discarded after a fixed period of time
- Modified files are sent to server asynchronously (when closed, or client performs sync)
- *Read-ahead* and *delayed write* possible

Slide 20

Security:

Traditionally (NFS v3), clients were trusted. Three ways of authentication:

- System authentication
 - Client passes user ID and group ID to server. (i.e., almost no security)
- Secure NFS using Diffie-Hellman (public key crypto)
 - ✗ More complex (implementation and key management)
 - ✗ Very short key length
- Kerberos
 - ✓ Secure
 - ✗ Entry costs

Enhanced security with NFS v4:

- RPCSEC_GSS as general security framework
- Provides hooks for different authentication systems
- Message integrity and confidentiality

ANDREW FILE SYSTEM (AFS)

Properties:

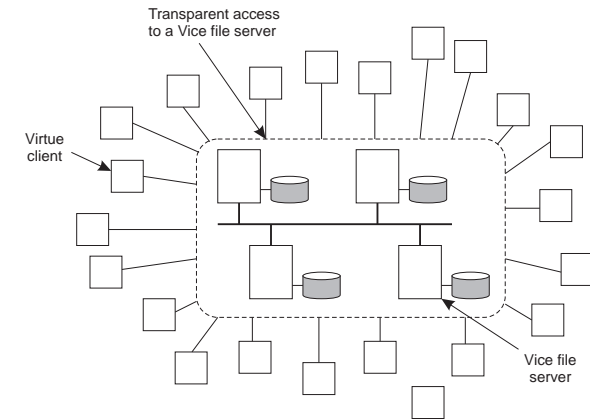
- From Carnegie Mellon University (CMU) in the 1980s.
- Developed as campus-wide file system: Scalability
- Global name space for file system (divided in *cells*, e.g. `/afs/cs.cmu.edu`, `/afs/ethz.ch`)
- API same as for UNIX
- UNIX semantics for processes on one machine, but globally write-on-close
- Developed to product by Transarc
- Open source version OpenAFS

Slide 21

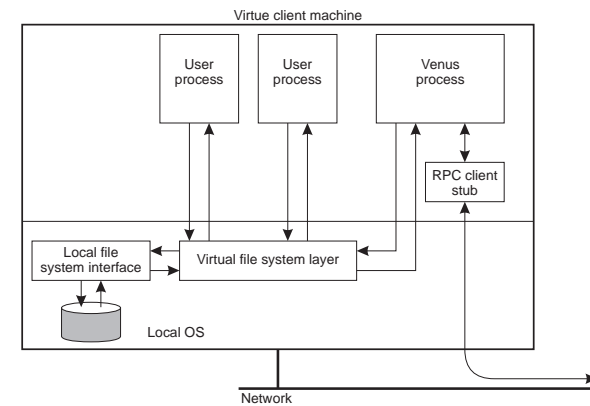
System Architecture:

- Client: User-level process *Venus* (AFS daemon)
- Trusted servers collectively called *Vice*
- Cache on local disk

Slide 22



Slide 23



Slide 24

Slide 25

Scalability:

- Server serves whole files
- Clients cache whole files
- Server invalidates cached files with callback (stateful servers)
- Clients do not validate cache (except on first use after booting)
- Modified files are written back to server on `close()`
- Result: Very little cache validation traffic
- Flexible *volume* per user (resize, move to other server)
- Read-only volumes for software

Slide 26

Security – Authentication:

- AFS does not trust UNIX user IDs
- AFS IDs managed at cell level
- Users authenticate to cells with Kerberos (`klog` command)
- Users get *token* (required for AFS commands and file accesses, stored in local cache manager)
- Tokens have time stamp and expire

Security – Authorisation:

- Access Control Lists (ACLs)
 - Directory based
 - Finer grained access modes than UNIX
-

CODA

Slide 27

- Developed at CMU by M. Satyanarayanan's group
- Successor of the Andrew File System (AFS)
- Supports disconnected, mobile operation of clients
- System architecture quite similar to AFS

Slide 28

Client Side View:

- Single mount point `/coda`
 - Files names are location transparent (servers cannot be distinguished)
 - Client caching in a user-level cache process called *Venus*
 - Downloads and caches whole files; directories are handled as files
 - UNIX semantics for processes on one machine, but globally write-on-close
 - High availability: access to a cached copy of a file is only denied if it is known to be inconsistent
-

DESIGN & ARCHITECTURE

Disconnected operation:

- All client updates are logged in a *Client Modification Log (CML)*
- On re-connection, the operations registered in the CML are replayed on the server
- CML is optimised (e.g. file creation and removal cancels out)
- On weak connection, CML is reintegrated on server by *trickle reintegration*
- Trickle reintegration tradeoff: Immediate reintegration of log entries reduces chance for optimisation, late reintegration increases risk of conflicts
- **File hoarding**: System (or user) can build a user hoard database, which it uses to update frequently used files in a hoard walk
- **Conflicts**: Automatically resolved where possible; otherwise, manual correction necessary
- Conflict resolution for temporarily disconnected servers

Slide 29

Servers:

- Read/write replication servers are supported
- Replication is organised on a per volume basis
- Group file replication (multicast RPCs); read from any server
- Version stamps are used to recognise server with out of date files (due to disconnect or failure)

Slide 30

GOOGLE FILE SYSTEM

Motivation:

- 10+ clusters
- 1000+ nodes per cluster
- Pools of 1000+ clients
- 350TB+ filesystems
- 500Mb/s read/write load
- Commercial and R&D applications

Slide 31

Assumptions:

- Failure occurs often
- Huge files (millions, 100+MB)
- Large streaming reads
- Small random reads
- Large appends
- Concurrent appends
- Bandwidth more important than latency

Interface:

No common standard like POSIX.
Provides familiar file system interface:

- Create
- Delete
- Open
- Close
- Read
- Write

Slide 32

In addition:

- *Snapshot*: low cost copy of a whole file with copy-on-write operation
- *Record append*: Atomic append operation

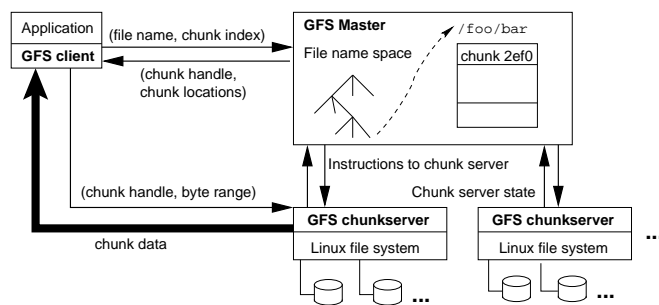
Slide 33

Design Overview:

- Files split in fixed size *chunks* of 64 MByte
- Chunks stored on *chunk servers*
- Chunks replicated on multiple chunk servers
- GFS master manages name space
- Clients interact with master to get *chunk handles*
- Clients interact with chunk servers for reads and writes
- No explicit caching

Slide 34

Architecture:



GFS Master:

- Single point of failure
- Keeps data structures in memory (speed, easy background tasks)
- Mutations logged to *operation log*
- Operation log replicated
- Checkpoint state when log is too large
- Checkpoint has same form as memory (quick recovery)
- Note: Locations of chunks *not* stored (master periodically asks chunk servers for list of their chunks)

Slide 35

GFS Chunkservers:

- Checksum blocks of chunks
- Verify checksums before data is delivered
- Verify checksums of seldomly used blocks when idle

Slide 36

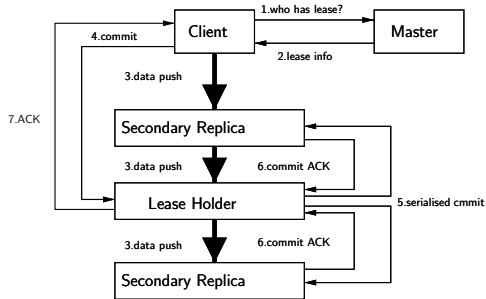
Data Mutations:

- Write, atomic record append, snapshot
- Master grants *chunk lease* to one of a chunk's replicas
- Replica with chunk becomes *primary*
- Primary defines serial order for all mutations
- Leases typically expire after 60 s, but are usually extended
- Easy recovery from failed primary: master chooses another replica after the initial lease expires

Example: Write:

Write(filename, offset, data)

Slide 37



Example: Append:

RecordAppend(filename, data)

→ Primary has extra logic

Slide 38

→ Check if fits in current chunk

- If not pad and tell client to try again
- Otherwise continue as with write

→ Guarantees that data is written *at least once* atomically

Consistency:

→ Relaxed

Slide 39

	Write	Record Append
Serial success	defined	defined/inconsistent
Concurrent success	consistent & unde- fined	defined/inconsistent
Failure	inconsistent	inconsistent

RE-EVALUATING GFS AFTER 10 YEARS

Workload has changed → changed assumptions

Single Master:

- ✗ Too many requests for a single master
- ✗ Single point of failure

Slide 40

- ✓ Tune master performance
- ✓ Multiple cells
- ✓ Develop distributed masters

File Counts:

- ✗ Too much meta-data for a single master
- ✓ applications rely on Big Table (distributed)

File Size:

- ✗ Smaller files than expected
- ✓ Reduce block size to 1MB

Slide 41 Throughput vs Latency:

- ✗ Too much latency for interactive applications (e.g. Gmail)
 - ✓ Automated master failover
 - ✓ Applications hide latency: e.g. multi-homed model
-