Distributed Systems (COMP9243)

Lecture 8b: Distributed File Systems

① Introduction

② NFS (Network File System)

③ AFS (Andrew File System) & Coda

④ 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

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

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:

- ➜ 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:

- ➜ 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:

- ➜ A READ after a WRITE returns the value just written
- ➜ When two WRITEs follow in quick succession, the second persists
- ➜ 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.

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:

- ➜ 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:

- ➜ Satyanarayanan, 1980's university UNIX 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

Is this still valid?

There are also varying reasons for using a DFS:

- ➜ Big file system, many users, inherent distribution
- ➜ 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

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

REPLICATION

Multiple copies of files on different servers:

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

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.

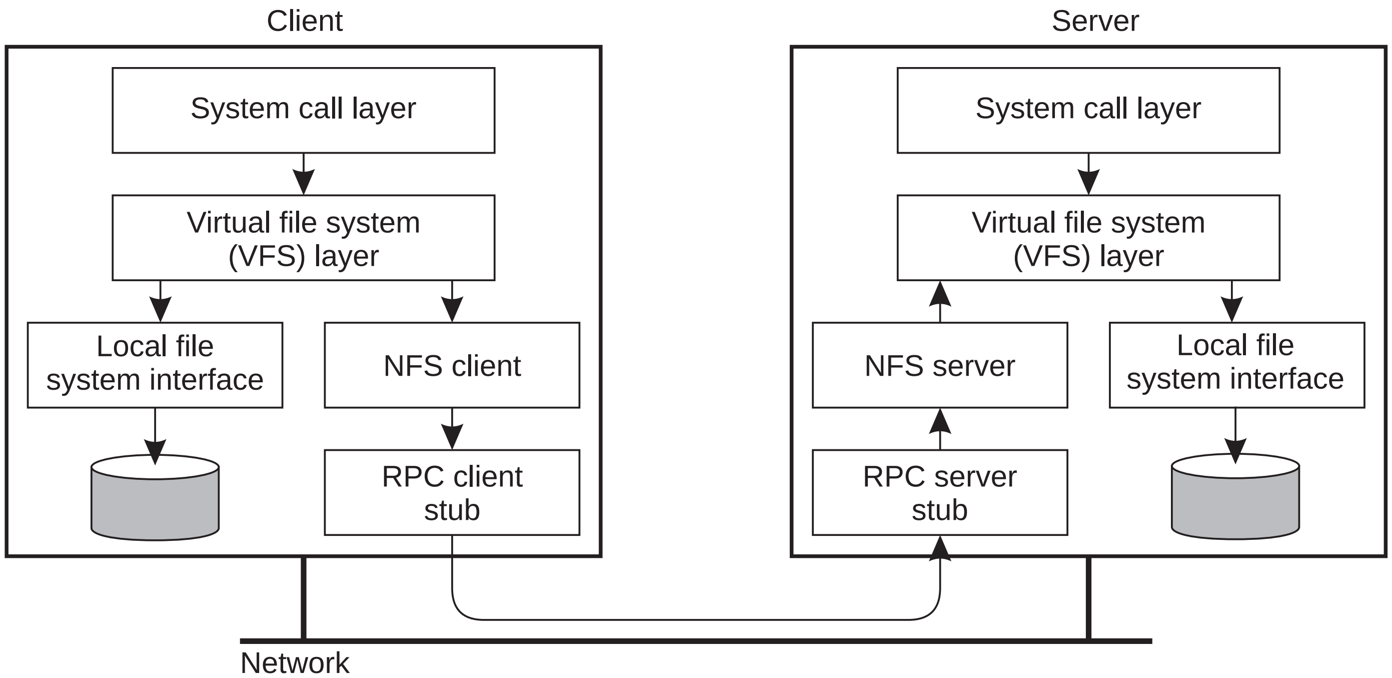
CASE STUDIES

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

NETWORK FILE SYSTEM (NFS)

Properties:

- ➜ 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
- ➜ Caching: local files copies
  - consistency through polling and timestamps
  - asynchronous update of file after close


Client
Server
System call layer
Virtual file system
(VFS) layer
Local file
system interface
NFS client
RPC client
stub
System call layer
Virtual file system
(VFS) layer
NFS server
Local file
system interface
RPC server
stub
Network

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

Andrew File System (AFS) & Coda

Properties:

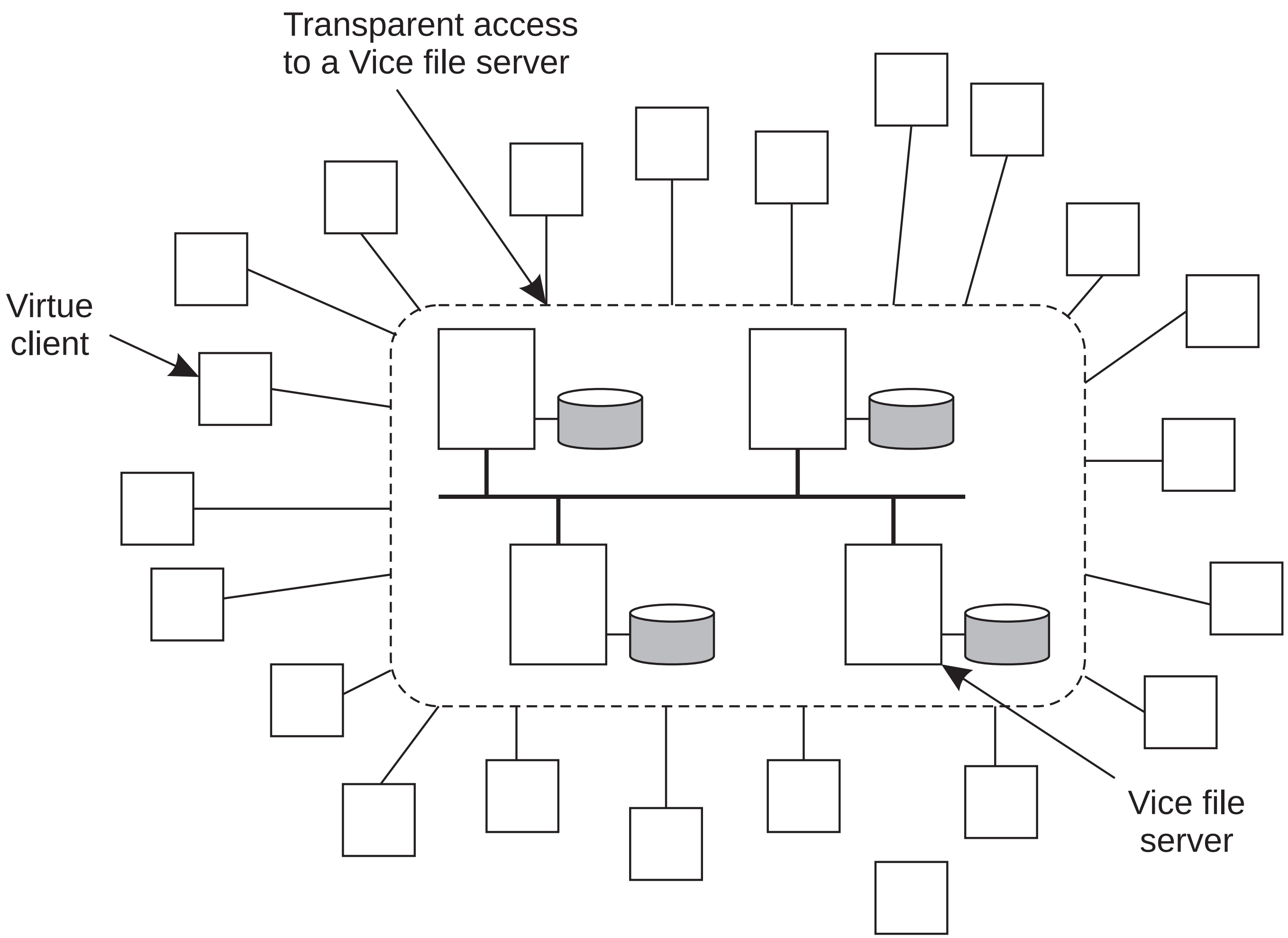
- ➜ From Carnegie Mellon University (CMU) in the 1980s.
- ➜ Developed as campus-wide file system: Scalability
- ➜ Global name space for file system (divided in *cells*, e.g. `/afs/cs.cmu.edu`, `/afs/ethz.ch`)
- ➜ API same as for UNIX
- ➜ UNIX semantics for processes on one machine, but globally write-on-close

System Architecture:

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

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)
- ➜ Result: Very little cache validation traffic


Transparent access
to a Vice file server
Virtue
client
Vice file
server

CODA

- ➜ Successor of the Andrew File System (AFS)
  - System architecture quite similar to AFS
- ➜ Supports disconnected, mobile operation of clients
- ➜ Supports replication

Design & Architecture

Disconnected operation:

- ➜ All client updates are logged in a *Client Modification Log (CML)*
- ➜ On re-connection, CML operations are replayed on the server
- ➜ 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

Servers:

- ➜ Read/write 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)

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

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`

In addition:

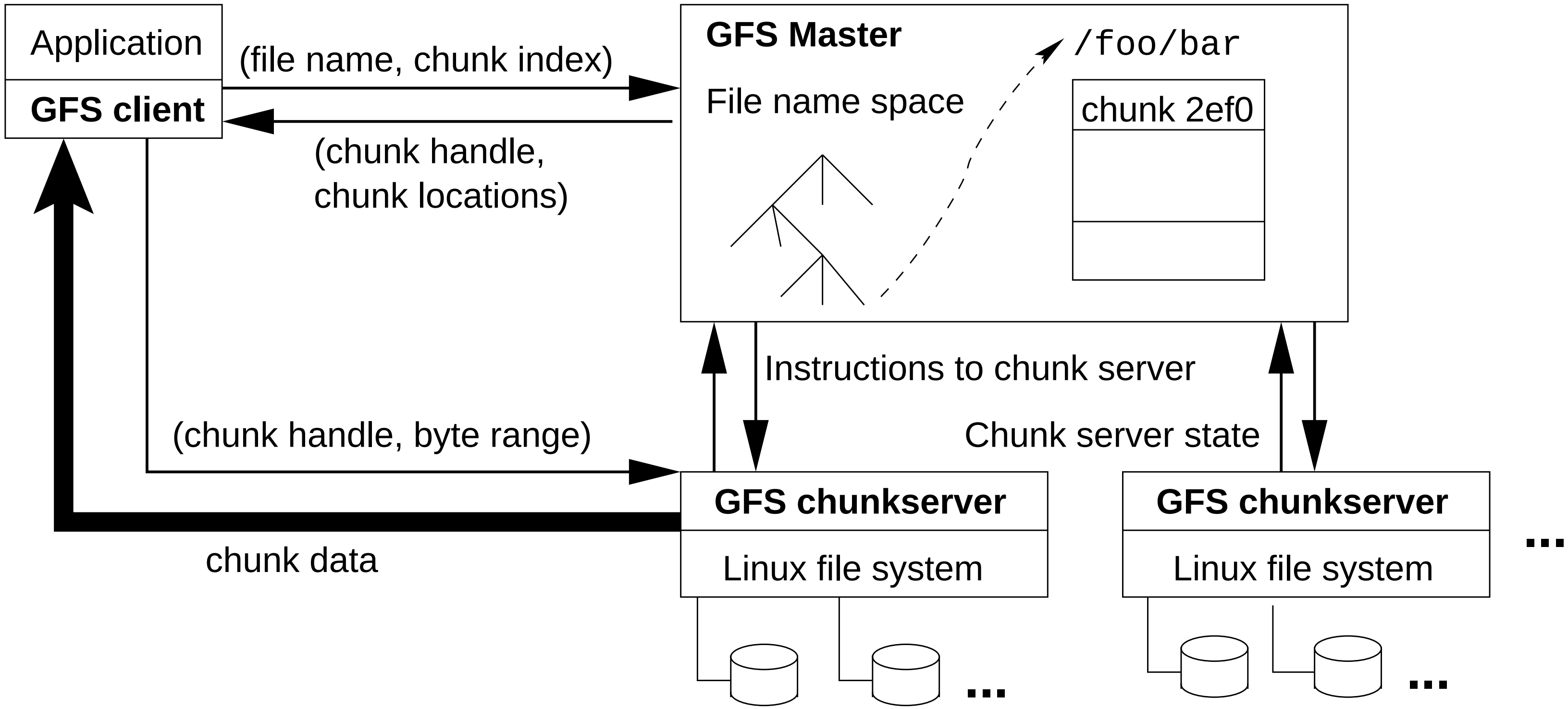
➜ *Snapshot*: low cost copy of a whole file with copy-on-write operation

➜ *Record append*: Atomic append operation

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

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)

GFS Chunkservers:

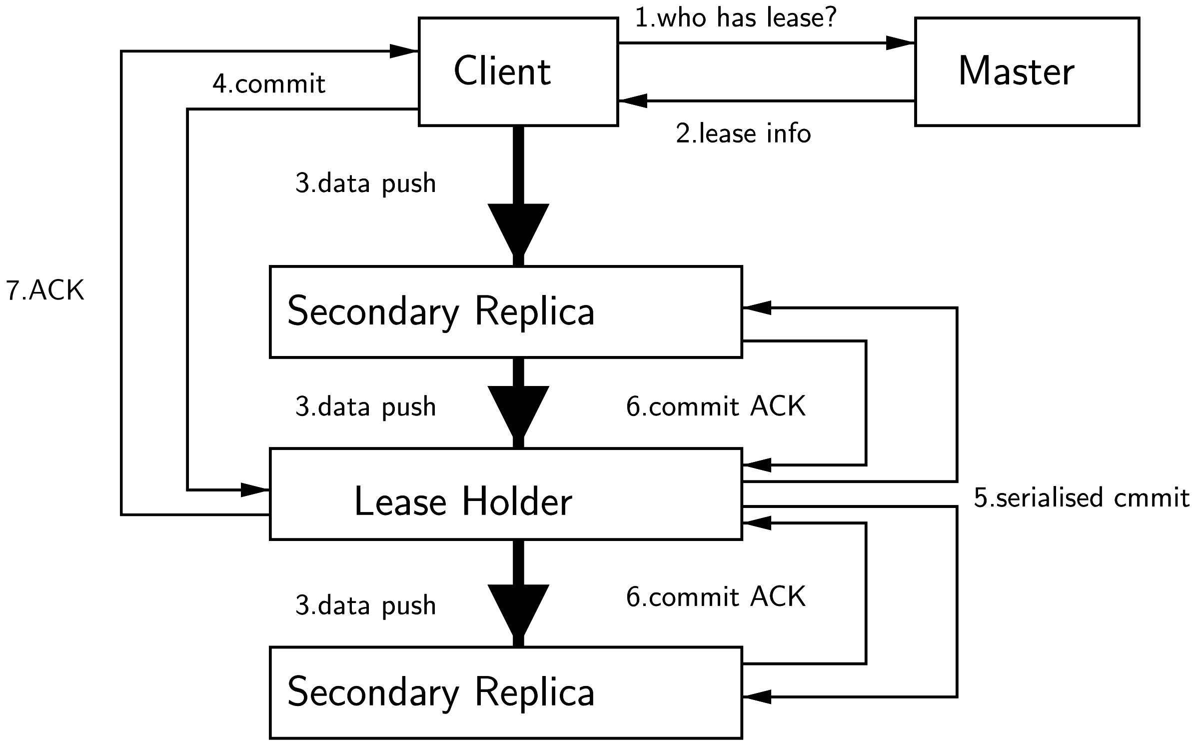
- ➜ Checksum blocks of chunks
- ➜ Verify checksums before data is delivered
- ➜ Verify checksums of seldomly used blocks when idle

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)`



RE-EVALUATING GFS AFTER 10 YEARS

Workload has changed → changed assumptions

Single Master:

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

Throughput vs Latency:

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

CHUBBY

Chubby is...:

- ➜ Lock service
- ➜ Simple FS
- ➜ Name service
- ➜ Synchronisation/consensus service

Architecture:

- ➜ Cell: 5 replicas
- ➜ Master:
  - gets all client requests
  - elected with Paxos
  - master lease: no new master until lease expires
- ➜ Write: Paxos agreement of all replicas
- ➜ Read: local by master

API:

- ➜ Pathname: `/ls/cell/some/file/name`
- ➜ Open (R/W), Close, Read, Write, Delete
- ➜ Lock: Acquire, Release
- ➜ Events: file modified, lock acquired, etc.

Using Chubby: electing a leader:

```
if (open("/ls/cell/TheLeader", W)) {
  write(my_id);
} else {
  wait until "/ls/cell/TheLeader" modified;
  leader_id = read();
}
```

What else ... ?

Colossus:

- ➜ follow up to GFS

BigTable:

- ➜ Distributed, sparse, storage map
- ➜ Chubby for consistency
- ➜ GFS/Colossus for actual storage

Megastore:

- ➜ Semi-relational data model, ACID transactions
- ➜ BigTable as storage , synchronous replication (using Paxos)
- ➜ Poor write latency (100-400 ms) and throughput

Spanner:

- ➜ Structured storage, SQL-like language
- ➜ Transactions with TrueTime, synchronous replication (Paxos)
- ➜ Better write latency (72-100ms)

READING LIST

Scale and Performance in a Distributed File System File system properties

NFS Version 3: Design and Implementation NFS

Disconnected Operation in the Coda File System Coda

The Google File System GFS

HOMEWORK

- ➜ Compare Dropbox, Google Drive, or other popular distributed file systems to the ones discussed in class.

Hacker's edition:

- ➜ See *Naming* slides