# Distributed File Systems

Marco Aiello, Ilche Georgievski

University of Groningen

DS, 2014/2015

## Outline

## Distributed file systems

- Applications of an operating system access files transparently irrespectively whether they are being stored locally or remotely.
- Benefits
    - Sharing of resources
    - Remote access
    - Economy
    - Maintenance
    - Robustness

## Requirements for sharing

- *Persistent* storage of data
- *Consistent* distribution of up-to-date data

# Properties

| | *Sharing* | *Persistence* | *Distributed cache/replicas* | *Consistency maintenance* | *Example* |
|---|---|---|---|---|---|
| Main memory | ✗ | ✗ | ✗ | 1 | RAM |
| File system | ✗ | ✓ | ✗ | 1 | UNIX file system |
| Distributed File System | ✓ | ✓ | ✓ | ✓ | Sun NFS |
| Web | ✓ | ✓ | ✓ | ✗ | Web server |
| Distributed shared memory | ✓ | ✗ | ✓ | ✓ | Ivy |
| Remote objects (RMI/ORB) | ✓ | ✗ | ✗ | 1 | CORBA |
| Persistent object store | ✓ | ✓ | ✗ | 1 | CORBA Persistent Object Service |
| Persistent distributed object store | ✓ | ✓ | ✓ | ✓ | PerDiS, Khazana |
| Peer-to-peer storage system | ✓ | ✓ | ✓ | ✓ | OceanStore |

1 strict one-copy consistency

✓ approximate to one-copy consistency

✗ no automatic consistency

## One step back

What is a file system?

- API to disk storage
  - Access, create, update, organise, name, protect, retrieve and delete files
- Subsystem of an OS
- Hierarchical name space
- Access control
  - User authorisation
  - Access rights
- Concurrent access
  - Certainly for read-only access
  - What about updates?

Understand the importance of file systems in handling data!

## Modules

| | |
|---|---|
| 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 |

## Files

- Permanent storage
- Structure
  - data - a sequence of bytes
  - attributes

| |
|---|
| File length |
| Creation timestamp |
| Read timestamp |
| Write timestamp |
| Attribute timestamp |
| Reference count |
| Owner |
| File type |
| Access control list |

### Directory

A file that provides a mapping from text names to internal file identifiers.

# Operations (UNIX file system)

| | |
|---|---|
| *filedes = open(name, mode)* | Opens an existing file with the given name. |
| *filedes = create(name, mode)* | Creates a new file with the given name. |
| | Both operations deliver a file descriptor referencing the open file. |
| | The mode is read, write, or both. |
| *status = close(filedes)* | Closes the open file fileds. |
| *count = read(filedes, buffer, n)* | Transfers n bytes from the file referenced by filedes to buffer. |
| *count = write(filedes, buffer, n)* | Transfers n bytes to the file referenced by filedes from buffer. |
| | Both operations deliver the number of bytes actually transferred and advance the read-write pointer. |
| *pos = lseek(filedes, offset, whence)* | Moves the read-write pointer to offset (relative or absolute, depending on whence). |
| *status = unlink(name)* | Removes the file name from the directory structure. If the file has no other names, it is deleted. |
| *status = link(name1, name2)* | Adds a new name (name2) for a file (name1). |
| *status = stat(name, buffer)* | Gets the file attributes or file name into buffer. |

## Perfect!

Now, how do we share files?

## Definition

A distributed file system is a classical model of a file system distributed across multiple machines with aim to enable sharing of files.

## Transparency

*Access*: same operations for access to local and remote files

*Location*: same name space after relocation of files or processes

*Mobility*: automatic relocation of files is possible

*Performance*: satisfactory performance across a specified range of service loads

*Scaling*: service can be expanded to meet additional loads

## Concurrency

- Same view of the state of the file system
- File-level or record-level locking
- Other forms of concurrency control to minimise contention

# Replication

- File service maintains multiple identical copies of files
  - load-sharing between servers ➠ better service scalability
  - local access has better response ➠ lower latency
  - another server when one has failed ➠ enhanced fault tolerance
- Full replication is difficult to implement
  - Caching gives most of the benefits (except fault tolerance)

# Heterogeneity

- Operating system and hardware platform
- Compatible design of the file service
- Openness of file service interfaces

## Fault tolerance

- Communication failures - the file service must continue to operate even when clients make errors or crash.
  - *at-most-once* semantics
  - *at-least-once* semantics (requires idempotent operations)
- Server failures - the file service must resume after a server machine crashes.
  - No action if stateless
  - Otherwise, replication

## Consistency

- Conventional file systems (*e.g.*, UNIX) - *one-copy* update semantics ➠ cashing is completely transparent

☹ Distributed file systems - difficult to achieve the same while maintaining good performance and scalability

## Security

- Access control and privacy as for local files
  - Access rights of user making a request
  - Remote user authenticated
  - Privacy requires secure communication
- File service interfaces are open to all processes not excluded by a firewall
  - Vulnerable to impersonation and other attacks

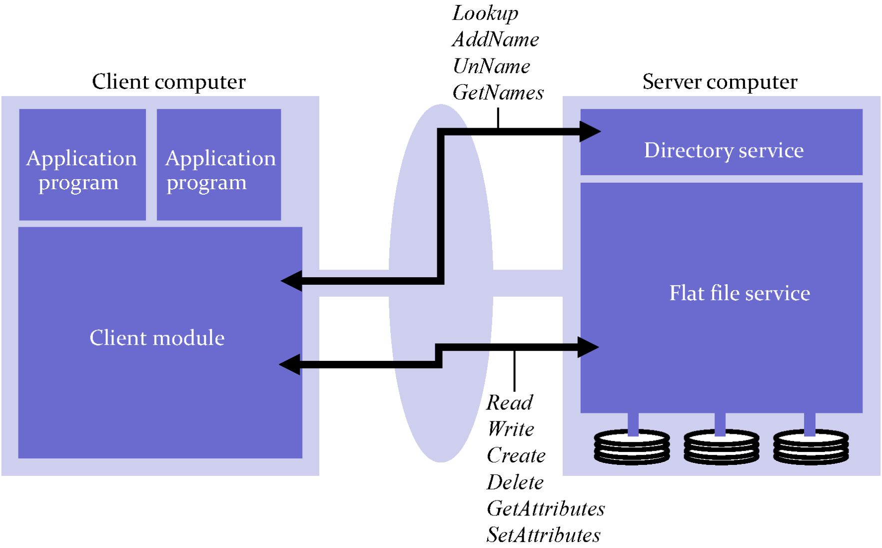
# Efficiency

- As powerful and general facilities as those in local file systems
- Performance comparable to a local file system

# Distributed file system

- Flat file service ➠ client + RPC interface
- Directory service ➠ client + RPC interface
- Client module ➠ application programming interface

# Distributed file system



# Interface

### Operations

- $Read(FileId, i, n) -> Data$
- $Write(FileId, i, Data)$
- $Create() -> FileId$
- $Delete(FileId)$
- $GetAttributes(FileId) -> Attr$
- $SetAttributes(FileId, Attr)$

### Unique File Identifier (UFID)

- A (long) sequence of bits
- Each file has a UFID

- *FileId* argument contains the UFID
- *FileId* is invalid if
  - the file is not present on the server, or
  - its access permissions are not sufficient

  ⇒ exception (except *Create*)
- $i$ - position of the first byte in the file
- $n$ - position of the last byte

# In comparison with the UNIX file system interface

- Functionally equivalent ➠ a client module can easily emulate the UNIX system calls
- No *open* and *close* operations ➠ files can be accessed by quoting the appropriate UFID
- UNIX *read* and *write* do not include starting position ➠ the current position of the read-write pointer

- Main difference in fault tolerance
  - Repeatable operations: clients may repeat calls to which they receive no reply ➠ UNIX operations are not idempotent
  - Stateless servers: no need for restoring states ➠ UNIX server needs to store the read-write pointer as long as a relevant file is open

## Function and interface

- Maps the text names of files to their UFIDs
- A client of the flat file service

**Operations**

- $Lookup(Dir, Name) -> FileId$
- $AddName(Dir, Name, FileId)$
- $UnName(Dir, Name)$
- $GetNames(Dir, Pattern) -> NameSeq$

## Access control

Access rights checks <u>at the server</u>

- A file name is mapped to a UFID
- Every file operation accompanied with a user identity check

# File group

A collection of files that can be located on any server or moved between servers while maintaining the same names.

- Similar to a UNIX filesystem
- Helps with allocation of files to file servers in larger logical units
- File groups have identifiers which are unique throughout the system (for an open system, they must be globally unique)

**Note**

- filesystem: a set of files held in a storage device
- file system: a software component that provides access to files

To construct a globally unique ID, some unique attribute of the machine on which it is created is used, e.g., IP number, even though the file group may move subsequently.

File Group ID:

| 32 bits | 16 bits |
|---|---|
| IP address | date |

- On each client computer
- Integrates and extends flat file service and directory service operations (single API)
- Maintains information about the network locations of the flat file service and directory service
- Helps in achieving satisfactory performance ➠ caches recently used file blocks

## Name resolution

- Hierarchical file system (UNIX-like file-naming system)
  - Files at the leaves, directories at other nodes
  - The root is a directory with known UFID
  - Multiple names for files (*AddName* operation and a reference count field in the file attribute record)
- All machines should have the exact same view of the hierarchy
  - ✓ location transparency, *e.g.*, *//server1/dir/file*
  - ✗ location independence: what about *server2*?

# Time for examples

- Sun Network File System
- Andrew File System
- Google File System

# Sun Network File System

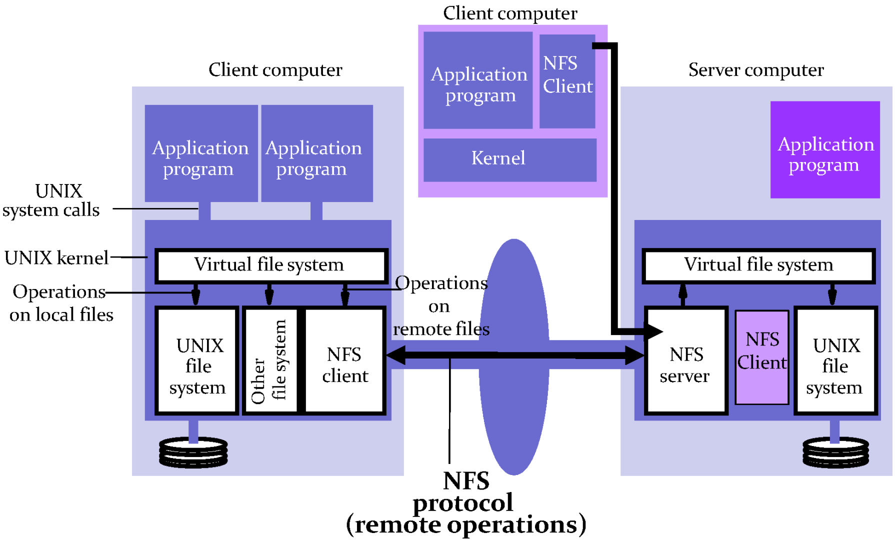
An open standard for file sharing on local networks with clear and simple interfaces, widely used in industry and academy.

- 1984 (v1): internal purposes of Sun
- 1989 (v2): outside Sun, UDP, *stateless* server, 2GB of a file
- 1995 (v3): TCP, files larger than 2GB, asynchronous writers on the server
- 2000 (v4): *stateful* server, performance improvements, strong security
- 2010 (v4.1): clustered servers, scalable parallel access

## Design goals

- Any machine can be a client and/or a server
- Diskless workstations
- Heterogeneous systems
- High performance

# Architecture



## Does the implementation have to be in the system kernel?

No

There are examples of NFS clients and servers that run at application-level as libraries or processes (*e.g.*, early Windows and MacOS implementations, current PocketPC, *etc.*)

But, for a UNIX implementation, there are advantages

- Binary code compatible - no need to recompile applications ➠ standard system calls that access remote files can be routed through the NFS client module by the client
- Shared cache of recently-used blocks at client
- Kernel-level server can access i-nodes and file blocks directly ➠ but, a privileged (root) application program could do almost the same
- Security of the encryption key used for authentication

- One VFS structure per mounted file system
- Differentiates between local and remote files
    - One *v-node* per open file ➠ show whether the file is local (*i-node*) or remote (*fh*)
    - Translates between NFS file identifiers and the UNIX internal file identifiers
- Keeps track of currently available filesystems
- Passes each request to the appropriate local system module

## File handles (fh)

- Information for identification and maintenance of a file
- In UNIX implementation of NFS:

| | | |
|---|---|---|
| Filesystem identifier | i-node number | i-node generation |

- Access control and authentication
  - Stateless
  - Does not keep files open
  - On each request checks:
    - user identity
    - access rights
  - Every client request is accompanied by the *userID* and *groupID*
  - Security loophole
    - DES encryption
    - Kerberos
- Interface
  - NFS *read*, *write*, *getattr*, *setattr* ≈ flat file service *Read*, *Write*, *GetAttributes*, *SetAttributes*
  - *lookup* and most of the directory operations ≈ directory service
  - Integrated in a single service

## Interface (v3)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh,name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

- Provides an interface used by applications
- Emulates standard UNIX operations
- Integrated with UNIX kernel
- Transfers blocks of files to/from server
- Caches blocks in the local memory

## NFS protocol

- Mounting protocol ➠ request an access to an exported directory
- Directory and file protocol ➠ access files and directories

## Static mounting

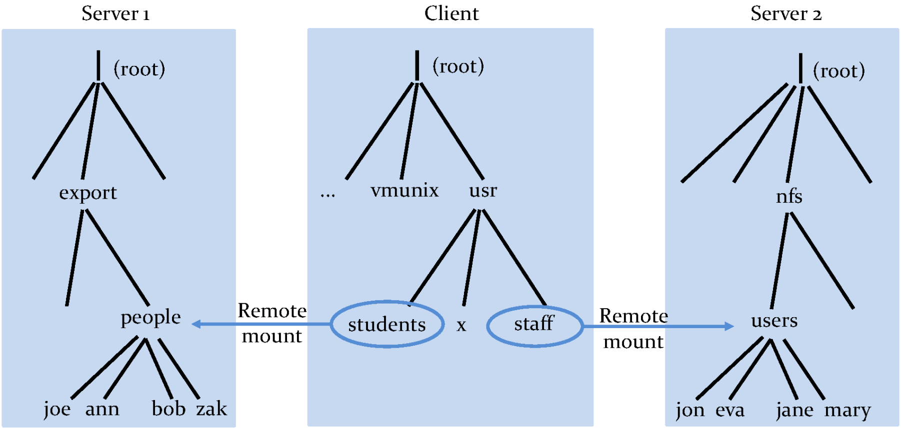
- Each server has a list of filesystems available for remote mounting
- Each server maintains a table of clients who have mounted filesystems at that server

Filesystems are mounted with *mount* command:

*mount(remotehost, remotedirectory, localdirectory)*

- Each client maintains a table of mounted filesystems
- Types of mounting:
  - *hard mounts* - client blocks until success
  - *soft mounts* - returns a failure after a small number of retries

# Static mounting
## Local and remote file systems




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

# Automounter

- ✗ Static mounting
  - Boot-time excessive ➠ any of the remote filesystems is not responding
  - Client accesses an ‘empty’ mount point
- Table of mount points and multiple candidate servers for each mount point
- It sends a probe message to each candidate server and then uses the mount service to mount the filesystem at the first server to respond
- Small mount table
- A simple form of replication for read-only filesystems

## Example

If there are several servers with identical copies of */usr/lib*, then each server will have a chance of being mounted at some clients.

- Pathnames translated at the client
- If a name refers to a remote-mounted directory, the client sends an RPC message to the server ➠ a file is accessed by a *lookup* operation: pathnames are resolved by iterative calls to *lookup()*, one call for each component of the path, starting with the ID of the root directory "/". Each *lookup()* returns a file handle.

## Server caching

- Similar to UNIX file caching for local files
  - Pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. *Read-ahead* and *delayed-write* optimisations.
  - For local files, writes are deferred to next sync event (30 second intervals)
  - Works well in local context, where files are always accessed through the local cache, but in the remote case it does not offer necessary synchronisation guarantees to clients.
- NFS (v3) servers offer two strategies for updating disk
  - *write-through* - altered pages are written to disk as soon as they are received at the server. When a *write()* operation returns, the NFS client knows that the page is on the disk.
  - *delayed commit* - pages are held only in the cache until a *commit()* call is received for the relevant file. This is the default mode used by NFS (v3) clients. A *commit()* is issued by the client whenever a file is closed.

## Client caching

Reduces RPC traffic between client and server

- Caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations

✗ Client writes do not result in the immediate updating of cached files of the same file in other clients

✓ Timestamp-based validity check ➠ reduces inconsistency, but does not eliminate it

  - $(T - Tc < t) \vee (Tm_{client} = Tm_{server})$

| | |
|---|---|
| $t$ | freshness guarantee |
| $Tc$ | time when cache entry was last validated |
| $Tm$ | time when block was last updated at server |
| $T$ | current time |

  - $t$ is configurable (per file) but is typically set to 3 seconds for files and 30 seconds for directories

- Writes: a modified cached page is marked as 'dirty' ➠ scheduled to be flushed to the server (file is closed or a *sync* occurs)

## Reports

- Early measurements (1987):
  - poor performance of the *write* operation, but writes are responsible for only 5% of server calls in typical UNIX environments ➠ hence *write-through* at server is acceptable
  - *lookup* accounts for 50% of operations due to step-by-step pathname resolution necessitated by the naming and mounting semantics
- Measurements from 1993:
  - single CPU configuration: $\approx$ 12000 server ops/sec
  - large multi-processor configuration: $\approx$ 300000 server ops/sec
- More recent measurements (2001):
  - 1x450 MHz Pentium III: >5000 server ops/sec, <4 milliseconds average latency
  - 24x450 MHz IBM RS64: >29000 server ops/sec, <4 milliseconds average latency

☺ See `www.spec.org` for even more recent measurements

## Summary

| | |
|---|---|
| **Access transparency** | excellent ☺ |
| **Location transparency** | not guaranteed, but achieved ☺ |
| **Mobility transparency** | hardly achieved ☹ |
| **Scaling transparency** | good ☺ |
| **Replication** | limited ☺ |
| **Heterogeneity** | good ☺ |
| **Fault tolerance** | limited, but effective ☺ |
| **Concurrency** | limited, but adequate ☺ |
| **Security** | good ☺ |
| **Performance** | good ☺ |

Andrew File System

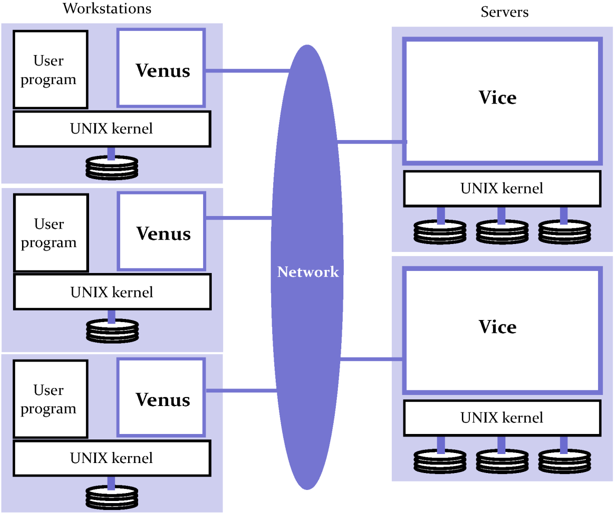
- 1983 (v1): internal purposes for the campus at CMU
- 1989 (v2): commercialised version (Transarc)
- 1993 (Arla): free version (Arla)
- 2000 (OpenAFS): open source distribution (IBM)
- 2013 (kAFS): (Red Hat)

## Design goal

- Scalability
  - Most files are small
  - Reads are more common than writes
  - Most files are read/written by one user
  - Files are referenced in bursts (once referenced, a file will be probably referenced again)
- Upload/download model:
  - *whole-file serving* - the server sends the entire file when opened
  - *whole-file caching* - the client saves the entire file on a local disk

- NFS compatible
- UNIX file system interface to access files
- Client machine = *Venus*, and dedicated server machine = *Vice*
- Servers and clients are interconnected by internet of LANs
- Kerberos authentication protocol
- Dramatically reduced loads on servers: a server load of 40% with 18 client nodes versus a load of 100% for NFS

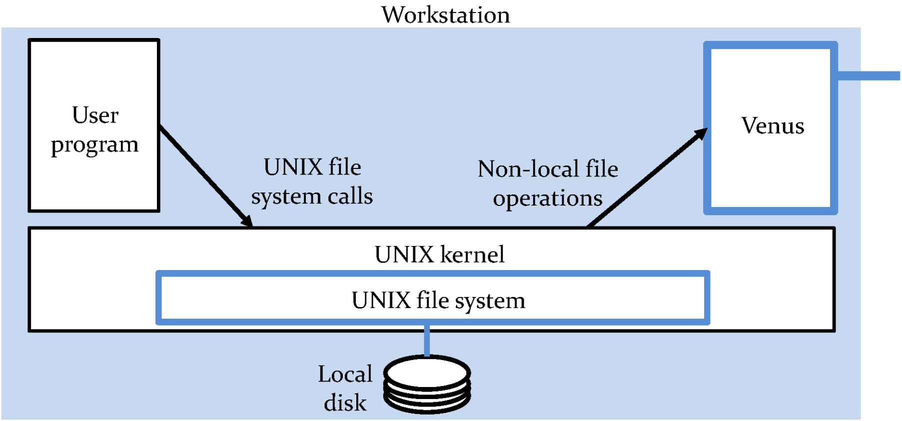
# Architecture



## Venus

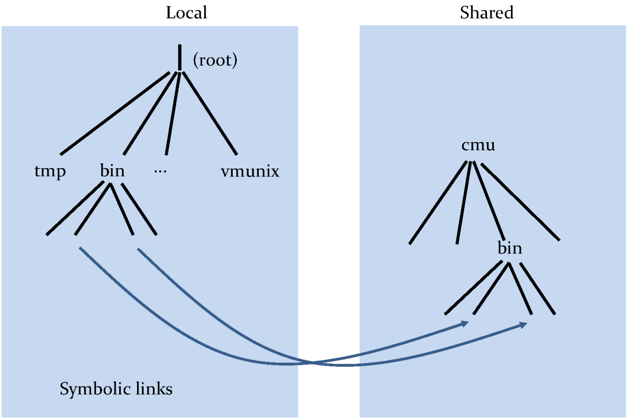
- Manages cache
- Intercepts operations to shared files
- Implements the hierarchical directory structure
- Translates pathnames to file identifiers (*fid*s)

# System call interception



# File name space

- *Local* files - stored on the local disk and handled as standard UNIX files
- *Shared* files - stored on servers, while copies of them are cached on the local disk.




➠ location transparency loss, but every client will see the same name space (*/afs*)

## File identifiers

Each file and directory is identified by three 32-bit numbers:

- *volumeID* - identifies the volume to which file belongs. A *volume* is a group of files, e.g., user's home directory. The client caches the binding between volumeID and server, but the server maintains the bindings.
- *vnodeID* - a file handle that refers to a file within the volume
- *uniquifier* - a unique number to ensure that the same *vnodeID*s are not used.

## Access control

- Directory level: all files in a particular directory share the same permissions
- Directory has a set of *user permission* pairs (Access Control List)
  - *user*: single username or AFS groupname
  - *permission*: read, write, insert, delete, lookup, lock and administer
- More flexible than traditional UNIX permissions

## Caching

### Server

- *Callback promise* - the token guarantees that the server will notify the client upon file modification by other clients (*valid* and *cancelled*)
- *Callback* - RPC that notifies all clients about file update

### Client

- Receives a *callback* and sets the *callback promise* to *cancelled*
- Validity check - if the requested file is in the cache
  - if its value is *cancelled* ➠ a fresh copy from Vice
  - otherwise, cached copy opened
- Failure ➠ cache validation request based on timestamp

## Summary

| | |
|---|---|
| **Access transparency** | good ☺ |
| **Location transparency** | poor ☹☺ |
| **Mobility transparency** | good ☺ |
| **Scalability** | excellent ☺ |
| **Replication** | limited ☺ |
| **Heterogeneity** | good ☺ |
| **Fault tolerance** | limited ☺ |
| **Concurrency** | limited ☺ |
| **Security** | limited ☺ |
| **Performance** | excellent ☺ |

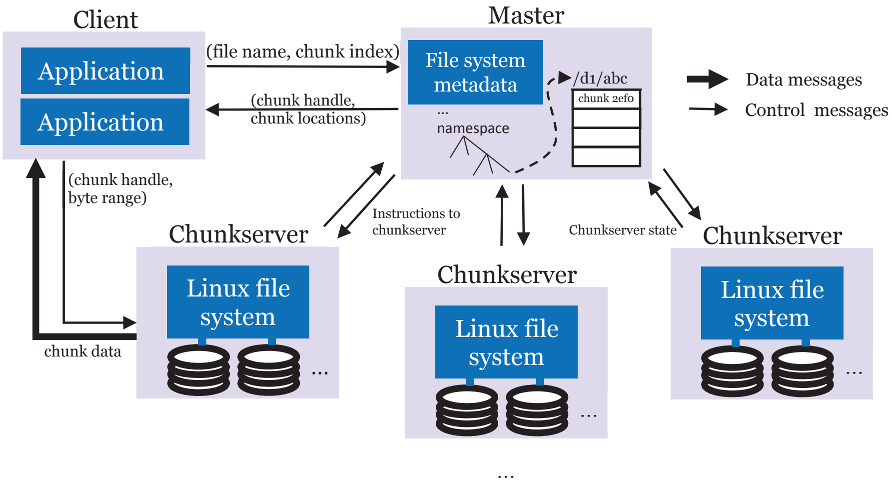
# Google File System

- 2001: revolution for the Web; for batch applications with large files (web crawling, indexing)
- 2011: Colossus for real-time operations

## Design goals

- Inexpensive commodity hardware
- Scalability
  - Large number of clients
  - Large files ($> 100$MB)
  - Two types of reads: large streaming reads ($> 1$MB) and small random reads (a few KB)
  - Large and sequential writes
- Fault tolerance: component failures are the norm rather than the exception
- Well-defined concurrency: atomicity with minimal synchronisation overhead is essential
- High bandwidth $>$ low latency

# Architecture



## Architecture

- Single master
  - Maintains: the namespace, access control information, the mapping from files to chunks, and the current locations of chunks
  - Controls system-wide activities: chunk lease management, garbage collection, chunk migration
- Chunk servers
  - Store chunks on local disks as Linux files, and chunk handle and byte range
  - No caching: Linux's buffer cache already keeps frequently accessed data in memory
- Clients
  - Implements the file system API and communicates
    - with the master for the metadata operations
    - with the chunkservers for all data-bearing operations
  - No caching: most applications stream through huge files (working sets too large to be cached)

## Interface

- Standard operations: *create*, *delete*, *open*, *close*, *read*, and *write*
- Additional operations:
  - *snapshot* - creates a copy of a file or a directory tree at low cost
  - *record append* - allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each append

## File namespace

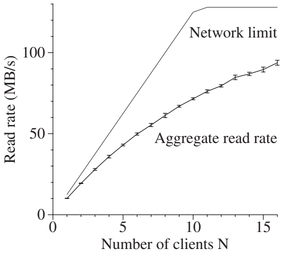
- The master executes all name-space related operations
- Non-conventional file system (no directory structure, no aliases for the same file/directory): the name space is a lookup table that maps full pathnames to metadata
- Read-write lock: before each operation, the master acquires a set of locks (prevents creation, rename or deletion of a file while some directory being snapshotted; concurrent mutations in the same directory)
  - if `/d1/d2/.../dn/leaf`, then acquire read-locks on the directory names `/d1`,`/d1/d2`,...,`/d1/d2/.../dn`, and either a read lock or a write lock on the full pathname

## Fault tolerance

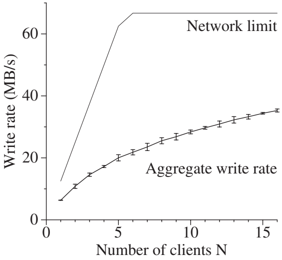
- Keep the overall system *highly available*
    - Fast recovery
    - Replication (chunks and master)

- Data integrity: chunkservers use checksumming to detect data corruption

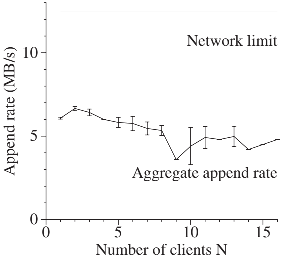
# Performance




(a) Reads




(b) Writes




(c) Record appends

One master, two master replicas, 16 chunkservers, and 16 clients. All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection. All 16 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

## Summary

| | |
|---|---|
| **Access transparency** | good ☺ |
| **Location transparency** | good ☺ |
| **Mobility transparency** | good ☺ |
| **Scalability** | excellent ☺ |
| **Replication** | good ☺ |
| **Heterogeneity** | poor ☹ |
| **Fault tolerance** | good ☺ |
| **Concurrency** | good ☺ |
| **Security** | limited ☺ |
| **Performance** | excellent ☺ |

## Tips

When designing a distributed file system, ask yourself:

- What is the size of distributed files?
- What are the relative and absolute frequencies of different file operations?
- How often does the data in the file change?
- How often do users share files for reading and for writing?
- Does the type of a file substantially influence these properties?

## References


- Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design (5th Edition), Chapter 15, Addison-Wesley Longman Publishing Co., Inc., 2012
- R. Brittain, Notes on OpenAFS, Dartmouth College, 2010, url: `http://caligari.dartmouth.edu/classes/afs/index.html`
- Internet Engineering Task Force, NFS Version 4 Minor Version 1 Protocol, IESG, 2010, url: `http://tools.ietf.org/pdf/rfc5661.pdf`
- S. Ghemawat, H. Gobioff, and S. Leung, The Google File System, In the *Proceedings of the 19th ACM symposium on Operating Systems Principles*, pp.29-43, 2003