

# Distributed Systems Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science  
steen@cs.vu.nl

## Chapter 11: Distributed File Systems

Version: December 10, 2012



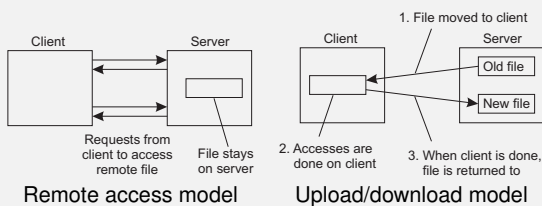
1 / 14

Distributed File Systems 11.1 Architecture

## Distributed File Systems

### General goal

Try to make a file system transparently available to remote clients.



2 / 14

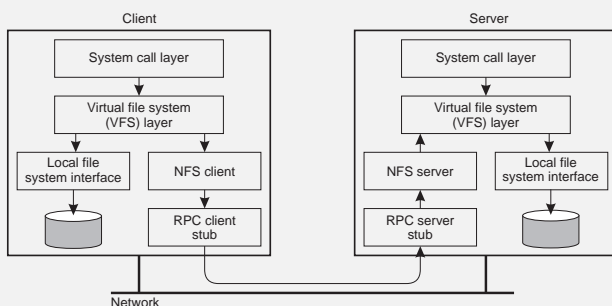
Distributed File Systems 11.1 Architecture

Distributed File Systems 11.1 Architecture

## Example: NFS Architecture

### NFS

NFS is implemented using the [Virtual File System](#) abstraction, which is now used for lots of different operating systems.



3 / 14

3 / 14

## Example: NFS Architecture

### Essence

VFS provides standard file system interface, and allows to hide difference between accessing local or remote file system.

### Question

Is NFS actually a file system?

4 / 14

4 / 14

## NFS File Operations

| Oper.    | 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                 |
| 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          |
| Open     | No  | Yes | Open a file                           |
| Close    | No  | Yes | Close a file                          |
| Lookup   | Yes | Yes | Look up a file by means of a name     |
| Readdir  | Yes | Yes | Read the entries in a directory       |
| Readlink | Yes | Yes | Read the path name in a symbolic link |
| Getattr  | Yes | Yes | Get the attribute values for a file   |
| Setattr  | Yes | Yes | Set one or more file-attribute values |
| Read     | Yes | Yes | Read the data contained in a file     |
| Write    | Yes | Yes | Write data to a file                  |

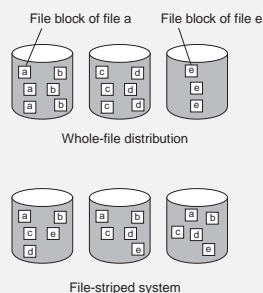
5 / 14

5 / 14

## Cluster-Based File Systems

### Observation

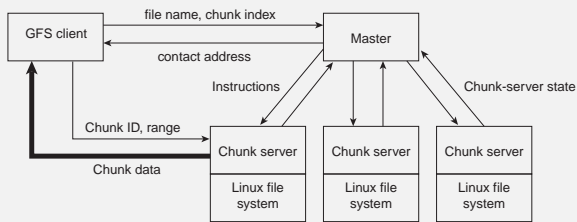
With very large data collections, following a simple client-server approach is not going to work  $\Rightarrow$  for **speeding up file accesses**, apply **striping** techniques by which files can be fetched in parallel.



6 / 14

6 / 14

## Example: Google File System



### The Google solution

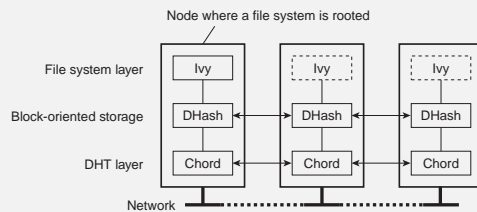
Divide files in large 64 MB chunks, and distribute/replicate chunks across many servers:

- The master maintains only a (file name, chunk server) table in **main memory**  $\Rightarrow$  minimal I/O
- Files are replicated using a **primary-backup** scheme; the master is kept **out of the loop**

7 / 14

7 / 14

## P2P-based File Systems



### Basic idea

Store data blocks in the underlying P2P system:

- Every data block with content  $D$  is stored on a node with hash  $h(D)$ . Allows for integrity check.
- **Public-key blocks** are signed with associated private key and looked up with public key.
- A local log of file operations to keep track of  $\langle \text{blockID}, h(D) \rangle$  pairs.

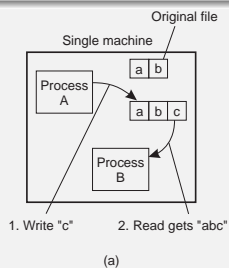
8 / 14

8 / 14

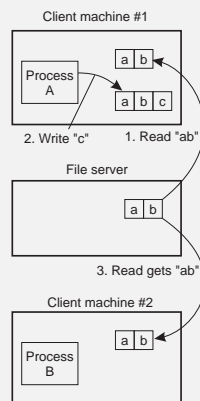
## File sharing semantics

### Problem

When dealing with distributed file systems, we need to take into account the ordering of concurrent read/write operations and expected semantics (i.e., consistency).



(a)



(b)

9 / 14

9 / 14

## File sharing semantics

## Semantics

- **UNIX semantics:** a *read* operation returns the effect of the last *write* operation  $\Rightarrow$  can only be implemented for remote access models in which there is only a single copy of the file
- **Transaction semantics:** the file system supports transactions on a *single* file  $\Rightarrow$  issue is how to allow concurrent access to a physically distributed file
- **Session semantics:** the effects of *read* and *write* operations are seen only by the client that has opened (a local copy) of the file  $\Rightarrow$  what happens when a file is closed (only one client may actually win)

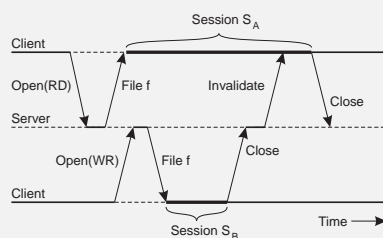
10/14

10/14

## Example: File sharing in Coda

## Essence

Coda assumes transactional semantics, but without the full-fledged capabilities of real transactions. **Note:** Transactional issues reappear in the form of "this ordering could have taken place."



11/14

11/14

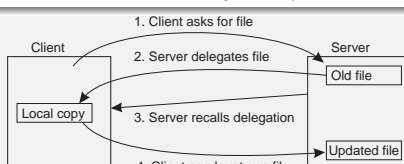
## Consistency and replication

## Observation

In modern distributed file systems, **client-side caching** is the preferred technique for attaining performance; **server-side replication** is done for fault tolerance.

## Observation

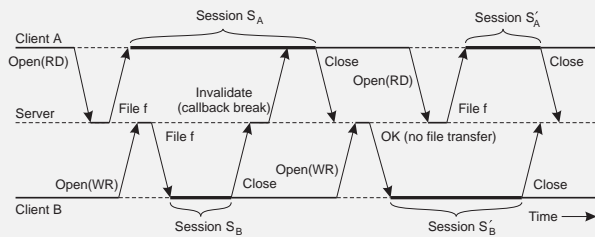
Clients are allowed to keep (large parts of) a file, and will be **notified** when control is withdrawn  $\Rightarrow$  servers are now generally **stateful**



12/14

12/14

## Example: Client-side caching in Coda

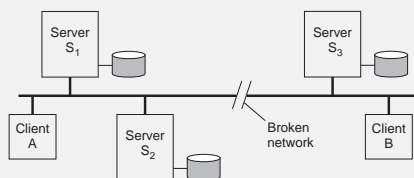
**Note**

By making use of [transactional semantics](#), it becomes possible to further improve performance.

13 / 14

13 / 14

## Example: Server-side replication in Coda

**Main issue**

Ensure that concurrent updates are detected:

- Each client has an [Accessible Volume Storage Group \(AVSG\)](#): is a subset of the actual VSG.
- Version vector**  $CVV_i(f)[j] = k \Rightarrow S_i$  knows that  $S_j$  has seen version  $k$  of  $f$ .
- Example:  $A$  updates  $f \Rightarrow S_1 = S_2 = [+1, +1, +0]$ ;  $B$  updates  $f \Rightarrow S_3 = [+0, +0, +1]$ .

14 / 14

14 / 14