Distributed File Systems

- Mainly based on Tanenbaum's 2002 "Distributed Operating Systems" Ch. 10 and Silberschatz & Galvin's Ch. 17
  - An overview of the main Distributed FS issues
  - We will see all the design options and the choices made by different distributed file systems

Overview

There are two distinct concepts:

- *File service*: the interface to file system functionality (primitives, parameters, semantics)
- *File server*: an implementation of this interface
  - file servers may be local or remote
 - file servers may be replicated

Usually there are two distinct components of file service:

- *True file service*: operations on files
- *Directory service*: operations on directories (or metafiles)

File system interface

- There are two main models for distributed file services:
  - the *upload/download model*: the entire file is transferred from server to client and vice versa. That is, only whole files are moved
    - simple
    - whole file transfer is efficient
    - more storage needed in the client, transferring entire file may be wasteful or impossible
  - the *remote access model*: the file system runs on servers and fine grained access is allowed
    - the file system is truly remote
    - little space is needed in the client

DFS Architecture
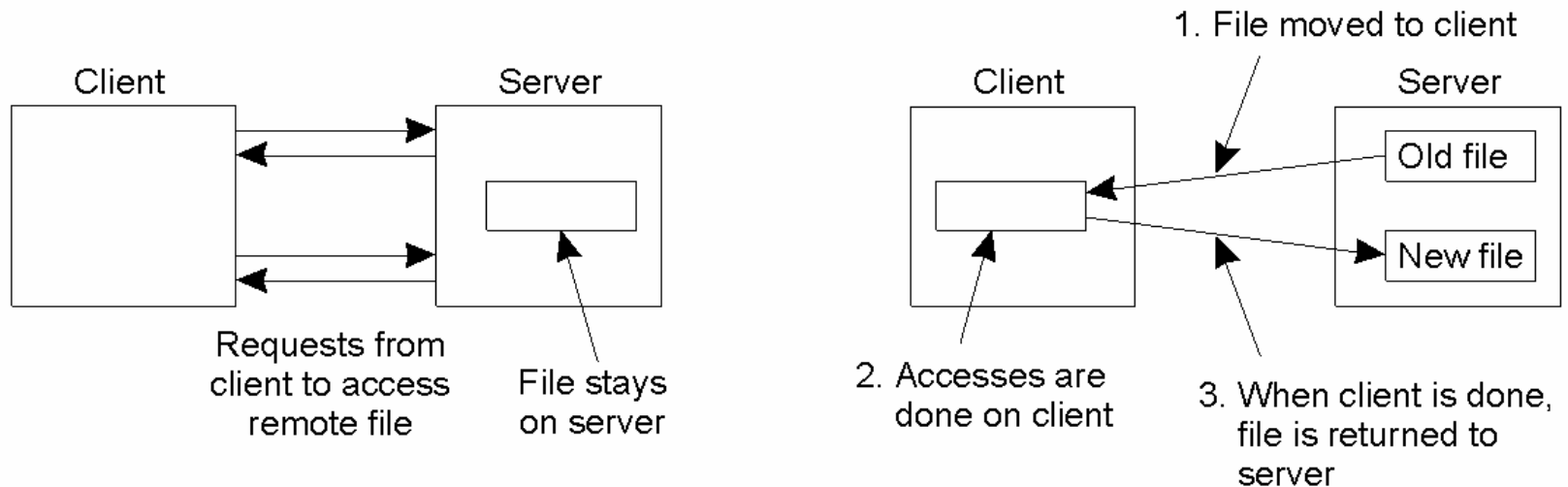
a)   The remote access model.
b)   The upload/download model

Directory Service Interface

- Main issue: do all machines have the same view of the file system?
  - are the paths to a file the same on all machines?
    - e.g., \\mymachine\c-disk on Windows
  - or can one "mount" a remote file system in different locations on the local tree?
- The former has the advantage of behaving like a local file system
- The latter has the advantage of an extra level of indirection in the naming scheme
  - e.g., one may decide to make /usr/local/ be remote on a Unix machine. The paths do not need to change
- With consistent naming across all machines, remote mounting is ideal

File Sharing Semantics

- As with all distributed systems, sharing is an important issue:
  - we need to avoid synchronization (serializing) as much as possible for performance reasons
    - e.g., to avoid having a single bottleneck by using caches
  - but files may explicitly be used for synchronization (e.g., one machine may write something that another will read)
- (In later lectures we will see similar sharing issues in other distributed mechanisms—e.g., distributed shared memory)
- Note that there is no strict ordering of unrelated events, anyway, because of the distributed character
 - whether event A occurred before B depends on the observer

- In practice, we will always have caches for efficiency reasons
- Propagating all writes back to the server as soon as they happen is inefficient
- Instead, a common policy is to propagate all changes only when the file is closed or the write buffer flushed explicitly (*session semantics*)
  - no surprise here: this is common for local file systems as well
- Another approach is to have transactions: atomic sequences of file operations
  - changes have the all-or-nothing property
- Note that the above talks about the *semantics*: the behavior that the user should expect. The implementation may be tricky!

Semantics of File Sharing (1)

a) On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.

b) In a distributed system with caching, obsolete values may be returned.
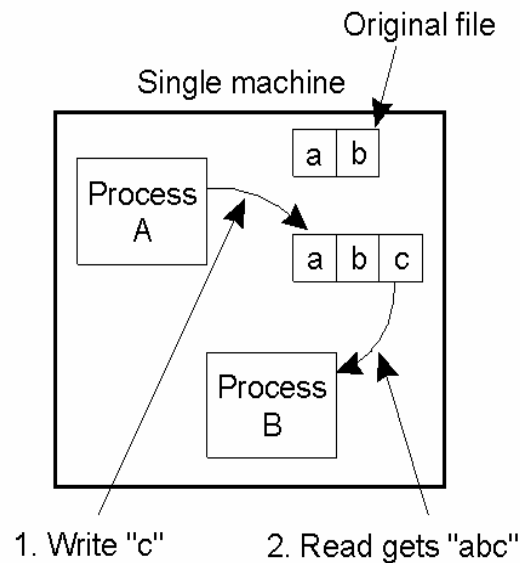
(a)
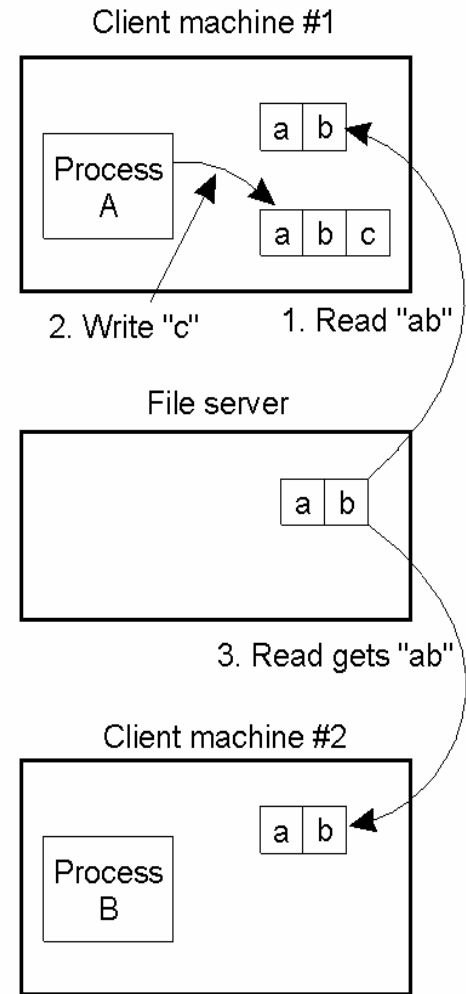
(b)

Semantics of File Sharing (2)

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

- Four ways of dealing with the shared files in a distributed system.

FS Implementation

- A good implementation should depend on the expected use. Usual properties of file access are:
  - most files are small (<10K). Maybe this is not true any more (either small or too big?)
  - reading is much more common than writing
  - random access is rare
  - most files don't live long
  - files are rarely shared
- All such properties need quantification ("how rare", "how long", etc.)
- Realistically, there have been very few good studies of the properties of file accesses
  - we will see the classical Sprite FS study in a later lecture. This has endured for years
  - only recently there have been some more studies in different environments

Axes of variability

There are many axes of variability in distributed file systems:

- Are servers special or can anybody export file services?
- Are file services and directory services combined?
  - is directory information cached?
- Can a server export file services for files that it itself imports?
- Can a directory hierarchy be partitioned? (So that server A asks server B, which asks server C, etc.?)
- Do servers maintain state information?

Stateless vs Stateful

There are advantages in both approaches

- Stateless servers advantages:
  - fault tolerance:
    - if the server crashes and reboots, the clients can continue
    - if the client crashes, it does not burden the server with stale information (or no need for expiration protocols)
    - no space taken up by state, no limit in number of "open" files

- Stateful servers advantages:
  - convenience: file locking, performance (file state kept, shorter network requests, readahead)
  - *idempotency*: depending on the file *service* (i.e., the interface)
  - if the interface is not naturally idempotent (e.g., contains "Append") then state is needed to make it idempotent

Practicality

- Fault tolerance is important
- Performance is important: even stateless file systems can keep state that is strictly used for optimization
- Stateful servers have better support for efficient cache consistency with no stale data
  - expiration protocols can be used (see later)
- An idempotent interface is good (read, write, set_size operations, but not append)

Caching

- Having a cache in the server's RAM is completely transparent to clients
- Having a cache in the client's RAM is common
  - but not as important if networks are fast, and disks are the bottleneck
  - most likely in kernel space so that different processes can benefit
  - but with a system call overhead. With today's processors such overheads are negligible
- Caches have to be consistent: what "write" does a "read" see? Are all writes ordered sequentially?

Cache consistency

- The cache consistency issues for distributed file systems are similar to those for (multi-)processor caches
    - but the solutions are different: file use semantics is different than SMP memory
- Main consistency issue:
    - reading and writing should not conflict: a reader should get a consistent version (most recently written: *strong consistency*)
    - "consistency" can be relaxed
- Two performance issues:
    - writing should not propagate to the servers immediately for best performance
    - delayed writing: periodic writing or update on close/flush
    - reading should not poll the server every time to find out if the cached copy is current

Cache consistency

- Simple solution: strict (readers/writers) locking
  - but multiple clients may want to use a file (concurrency)
  - inconvenient semantics
- Two main variants for concurrent accesses: *server-driven* protocols and *client-driven* protocols
- Client-driven: clients poll servers to see if cached objects are current
  - have to sacrifice either performance or strong consistency
- Server-driven: servers notify clients of changes
  - more complicated: now clients need to "play server" and accept messages
  - fault tolerance is harder
  - the server has to be stateful

Server-driven consistency (leases)

- A well-known server-driven consistency protocol is based on *leases*

- Leases also solve some of the problems of stateful servers (esp. tolerance for client faults)

- Leases specify how long a client can hold on to cached data
  - after a lease expires, both the server and the client know that the data is not cached
  - the client must renew the lease by contacting the server
  - on a write, the server waits until the lease expires whenever clients are unreachable but their caches need updating

- Assuming that writes and network failures are rare, leases are great:
  - they offer both strong consistency and fault tolerance for client faults, network failures

Replication

- Distributed file systems often offer file replication
  - for fault-tolerance and availability: servers go down, disks go bad
  - for performance: distributing the load
- Replication can be transparent or user controlled
- Replication can be synchronous (all servers updated simultaneously) or lazy
- To avoid problems with crashes while replicating file changes, transaction techniques can be employed (all-or-nothing)
- Flexible, distributed replication protocols have been proposed (e.g., voting)
  - we won't cover them here

NFS Architecture

- The basic NFS architecture for UNIX systems.
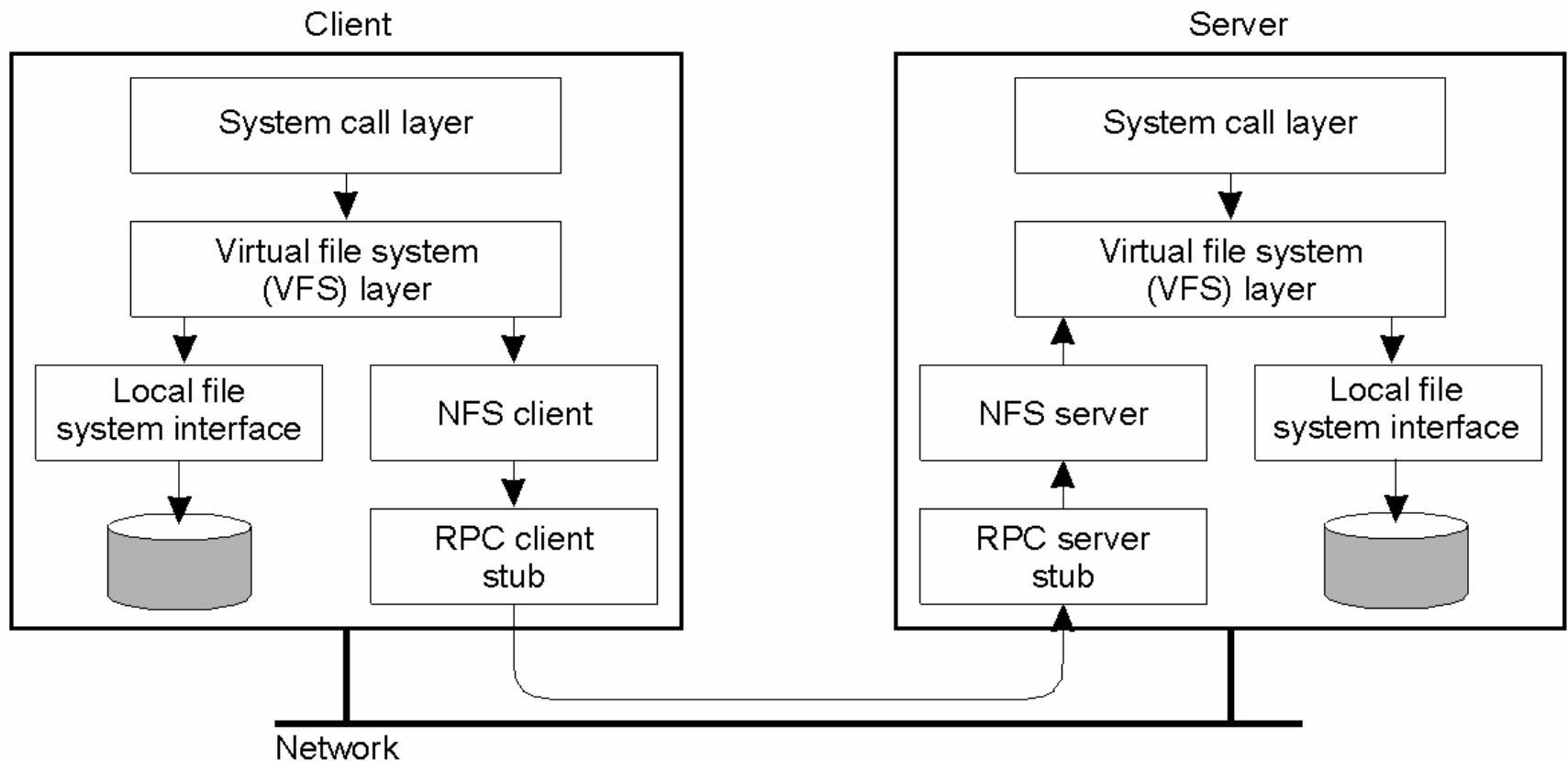
Case study NFS v.3

- NFS architecture: servers export directories, clients mount them wherever they want
  - any directory can be exported
  - exported directories "shadow" client directories
  - mounted directories are not exported when a parent directory is exported
- Automounting is supported
  - and offers some fault tolerance since multiple servers can be contacted
- NFS has no support for replication—it has to be done manually
  - possible inconsistencies with automounting

- NFS file service is similar to the Unix filesystem calls, with the exception of open and close
- because the server is stateless
  - read/write calls are self-contained
  - contain file handle, offset
  - file locking is not supported (although part of Unix)
- State is only kept for optimizations: readahead is supported and data are read in large chunks
- No strong cache consistency is supported, but periodic updates are issued
  - write buffers are flushed every 30 seconds
  - caches are invalidated every few seconds (3 secs for data blocks, 30 for directories?)
 - the server is polled for more recent versions on every "open" call

NSF v.4

- NFS v.3 is the specification upon which all common implementations are based
- v.4 is the next step
  - not clear if it will become as common
- Features:
  - stateful servers, leases for exclusive file access
  - "compound procedures" for performance (do lookup, open, read with a single network operation)
  - "lookup" of mounted directories can cross mount points
  - when a client looks up a remote directory that is itself mounted from another server, the client can tell that the directory is remote (by examining the file handle returned by the "lookup"). If needed, the client can mount the directory itself

Communication
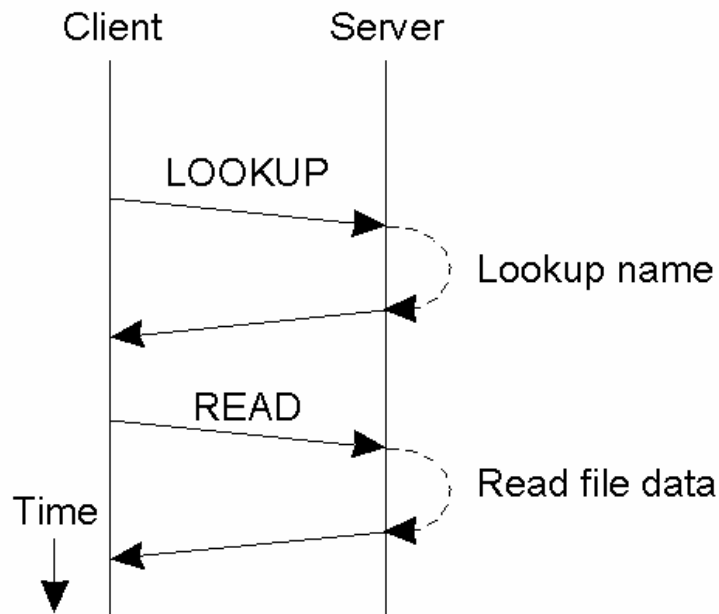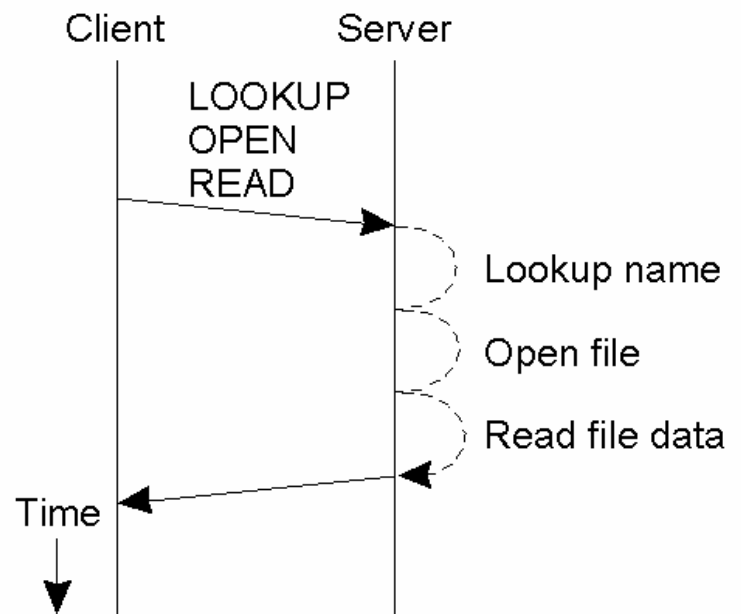
a) Reading data from a file in NFS version 3.
b) Reading data using a compound procedure in version 4.

- locking protocol (lease-based for fault tolerance) is supported
  - both polling and queueing until lock is acquired are supported
  - a readers/writers lock is also supported (called "share reservation")
  - works well with the Windows FS model where files cannot be written by multiple processes
  - more advanced security (both authentication and confidentiality)
  - based on "secure RPC"
  - better access control (Windows-like—a requirement for v.4 was interoperation with Windows file systems)
  - multiple groups, like "interactive", "anonymous", "network", etc.