Unit 8
Distributed file systems

Distributed Systems
Bachelor In Informatics Engineering
Universidad Carlos III de Madrid

FILES AND DIRECTORIES

- **Files**: data + attributes
  - Data accessible via read/write
  - Attributes held as single record with following info: file length, C/R/W/Attr timestamps, file type, owner, access ctrl list, etc

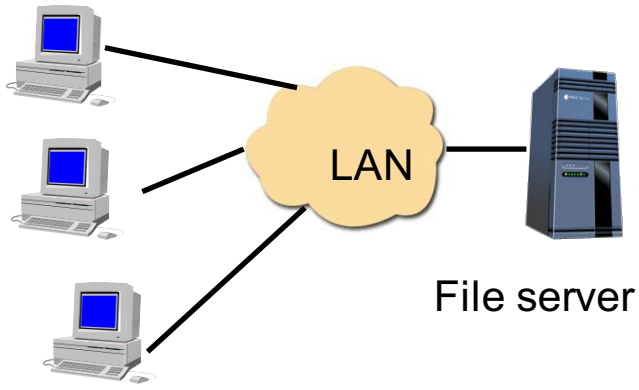
- **Directories**: are also files
  - Provide mapping between names and internal file ids
  - May include other directory names

- **Metadata**: extra info stored by a file system needed for managing files – file attr, directories, other persistent info

DISTRIBUTED FILE SYSTEMS

DEFINITIONS:

- A **Distributed File System** (DFS) is simply a classical model of a file system (as discussed before) distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- This is an area of active research interest today.
- The resources on a particular machine are **local** to itself. Resources on other machines are **remote**.
- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.



DISTRIBUTED FILE SYSTEMS

DEFINITIONS:

Clients, servers, and storage are dispersed across machines. Configuration and implmentation may vary

a) Servers may run on dedicated machines, OR
b) Servers and clients can be on the same machines.
c) The OS itself can be distributed (with the file system a part of that distribution.
d) A distribution layer can be interposed between a conventional OS and the file system.

Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

Performance is concerned with throughput and response time.

DISTRIBUTED FILE SYSTEM REQUIREMENTS

Initially access transparency and location transparency

Access transparency: clients unaware of the distribution of files – local/remote files access the same

Location transparency: clients must see uniform namespace. Files may be relocated without changing their pathnames.

Then: performance, scalability, concurrency control, fault tolerance, security

Concurrent file updates: most systems provide file- or record-level locking

Consistency: most systems require one-copy update semantics, modifications must propagate to all that have a copy

Security: use access control lists, authenticate client requests

DISTRIBUTED FILE SYSTEMS

Naming is the mapping between logical and physical objects.

- Example: A user filename maps to <cylinder, sector>.
- In a conventional file system, it's understood where the file actually resides; the system and disk are known.
- In a **transparent** DFS, the location of a file, somewhere in the network, is hidden.
- **File replication** means multiple copies of a file; mapping returns a SET of locations for the replicas.

Location transparency

a) The name of a file does not reveal any hint of the file's physical storage location.
b) File name still denotes a specific, although hidden, set of physical disk blocks.
c) This is a convenient way to share data.
d) Can expose correspondence between component units and machines.

DISTRIBUTED FILE SYSTEMS

Location independence

- The name of a file doesn't need to be changed when the file's physical storage
  location changes. Dynamic, one-to-many mapping.
- Better file abstraction.
- Promotes sharing the storage space itself.
- Separates the naming hierarchy from the storage devices hierarchy.

Most DFSs today:

- Support location transparent systems.
- Do NOT support **migration**; (automatic movement of a file from machine to machine)
- Files are permanently associated with specific disk blocks.

DISTRIBUTED FILE SYSTEMS

The ANDREW DFS AS AN EXAMPLE:

- Is location independent.
- Supports file mobility.
- Separation of FS and OS allows for disk-less systems. These have lower cost and convenient system upgrades. The performance is not as good.

NAMING SCHEMES:

There are three main approaches to naming files:

1. Files are named with a **combination** of host and local name.

   - This guarantees a unique name. NOT location transperent NOR location independent.
   - Same naming works on local and remote files. The DFS is a loose collection of independent file systems.

DISTRIBUTED FILE SYSTEMS

NAMING SCHEMES:

2. Remote directories are **mounted** to local directories.

   - So a local system seems to have a coherent directory structure.
   - The remote directories must be explicitñy mounted. The files are location independent
   - SUN NFS is a good example of this technique.

3. A **single global name structure** spans all the files in the system.

   - The DFS is built the same way as a local filesystem. Location independent.
   - Each machine has same view of all files.

DISTRIBUTED FILE SYSTEMS

IMPLEMENTATION TECHNIQUES:

- Can Map directories or larger aggregates rather than individual files.
- A **non-transparent** mapping technique:

  name ----> < system, disk, cylinder, sector >

- A **transparent** mapping technique:

  name ----> file_identifier ----> < system, disk, cylinder, sector >

- **Dynamic mapping**: OS must map same file name to different location at different time
- So when changing the physical location of a file, only the file identifier mapping need be modified. This identifier must be "unique" in the universe.

CO-UTILIZATION SEMANTICS

- ***Session***: sequence of client accesses between an *open* and a *close*
- The co-utilization semantics specifies what happens when multiple processes access simultaneously the same file
- ***UNIX semantics***
  - Read sees the effects of all previous writes
  - The result of two subsequent writes is the second one
  - Processes may share the position pointer
  - Difficult to implement in distributed systems
    - Unique copy

CO-UTILIZATION SEMANTICS

- ***Session semantics:***
  - Modifications to an open file are visible solely to the process which is writing the file
  - Once the file is closed the modifications become visible to future sessions
  - Multiple file images
  - The effect of two concurrent sessions using the same file will be the effect of the **last one writing**
  - If two processes want to share data they **must open and close the file** to propagate the changes
    - Not reccommended for concurrent accesses
  - No shared pointers

CO-UTILIZATION SEMANTICS

- ***Immutable file semantics***
  - File content may not be modified
  - The file name may not be reused
  - File may only be shared for read operations

- ***Transaction semantics***
  - Access to file must happen between
    - *BEGIN TRANSACTION*
    - *END TRANSACTION*
  - The system ensures that the effect of two concurrent transactions is equivalent to executing them in one of the possible sequential orders

REMOTE ACCESS

- ***Download/upload model***
  - Full file transfer
  - Locally stored in memory/disk
  - Normally use session semantics
  - Efficient transfer
  - Open operation has high latency
  - Multiple file copies

- ***Remote service model***
  - Server performs all file operations
  - Block access
  - Client/server model

- ***Client cache***
  - Combines above models

CACHING

- Better performance
  - Exploits locality of references
    - Local proximity
    - Spatial proximity
  - Read ahead
  - Write back

- Other caches
  - Name caching
  - Metadata caching

CACHING

- ***In the server***
  - Reduces disk access

- ***In the client***
  - Reduces network traffic
  - Reduces server load
  - Scalable
  - Two possible locations
    - **On disk**
      - Bigger capacity
      - Slower
      - Not volatile, data not lost
    - **In main memory**
      - Smaller capacity
      - Faster
      - Volatile
  - Client caches introduce **coherence problems** b/c of the multiple copies, when a file is written

SOLUTION TO THE COHERENCE PROBLEM

- No client cache:
  - Trivial but bad: no reuse, no read ahead, no write back
- No client cache for data shared for writing (Sprite).
  - Remote access to unique copy ensures UNIX semantics
- Cache w/o data replication
  - Based on cooperation schemas which define a unique global space consisting of all system caches
  - Data goes through cache w/o replication
- Cache coherence protocols

CACHE COHERENCE PROTOCOLS

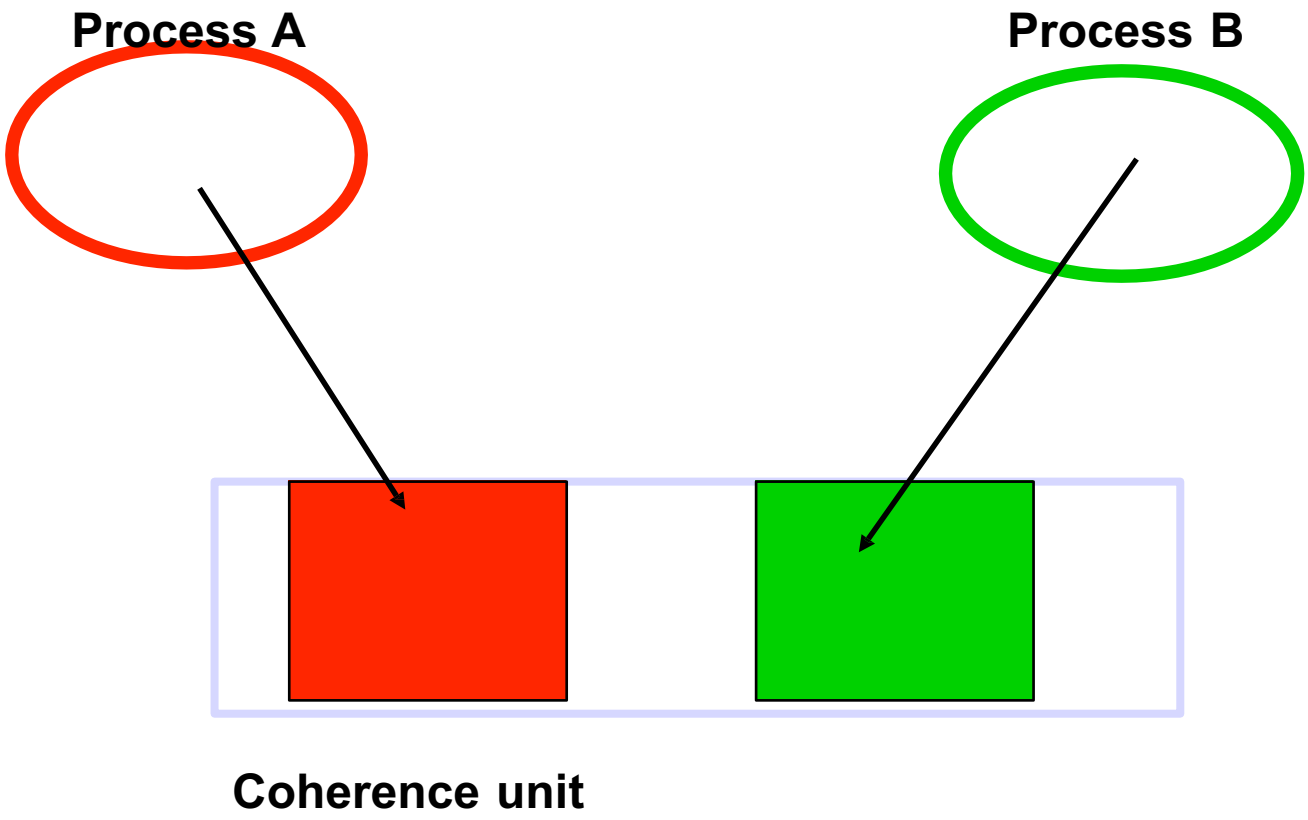
- Goals:
  - Allow the use of cache in the client while keeping the data coherent according to the co-utilization semantics for the file system
- Design must consider
  - Protocol granularity
  - Validation mechanism
  - Actualization mechanism
  - …

PROTOCOL GRANULARITY

- Size of the data unit that coherence applies to one byte to whole file
- E.g:
  - *Sprite*: whole file
  - *Coda*: set of files (volumen)
  - *ParFiSys*: user defined

- In UNIX semantics, large units
  - May lead to false co-utilization

FALSE CO-UTILIZATION



UPDATING MECHANISMS

- When a copy is written an actualization is necessary to keep a coherent image

- Methods:
  - ***Update copies***
  - ***Invalidate copies***

UPDATING MECHANISMS

- Update all copies
  - Network traffic
  - Unrealistic for file systems
- Invalidate all copies; all future accesses done over a consistent copy.
  - Shorter messages, less traffic
  - Better scalability

LOCATION OF COPIES

- Must know which clients store copies
- Directory-based schemas. Every entry stores list of clients w copy in the cache.
- ***Centralized directories***: single point of management for ensuring data coherence
  - Possible bottleneck
- ***Distributed directories***: multiple managers
  - Scalability

DISTRIBUTED FILE SYSTEMS

CACHING

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.
- If required data is not already cached, a copy of data is brought from the server to the user.
- Perform accesses on the cached copy.
- Files are identified with one master copy residing at the server machine,
- Copies of (parts of) the file are scattered in different caches.

- **Cache Consistency Problem** -- Keeping the cached copies consistent with the master file.

- A remote service (RPC) has these characteristic steps:
  a) The client makes a request for file access.
  b) The request is passed to the server in message format.
  c) The server makes the file access.
  d) Return messages bring the result back to the client.

- This is equivalent to performing a disk access for each request.

DISTRIBUTED FILE SYSTEMS

CACHE LOCATION:

- Caching is a mechanism for maintaining disk data on the local machine. This data can be kept in the local memory or in the local disk. Caching can be advantageous both for read ahead and read again.
- The cost of getting data from a cache is a few HUNDRED instructions; disk accesses cost THOUSANDS of instructions.
- The master copy of a file doesn't move, but caches contain replicas of portions of the file.
- Caching behaves just like "networked virtual memory".
- What should be cached? << blocks <---> files >>. Bigger sizes give a better hit rate; smaller give better transfer times.
- **Caching on disk** gives:
  - Better reliability.

- **Caching in memory** gives:
  - The possibility of diskless work stations,
  - Greater speed,

- Since the server cache is in memory, it allows the use of only one mechanism.

DISTRIBUTED FILE SYSTEMS

CLIENT METHOD

Writes by the client do not result in immediate updating of cached copies

Usually need validity check before cache entry is used

Send blocks back to server if modified (asynchronously). When?

CACHE UPDATE POLICY:

- **Write through** cache has good reliability. But the user must wait for writes to get to the server – slower writes. Used by NFS.

- **Write back** - write requests complete more rapidly. Data may be written over the previous cache write, saving a remote write. Lower network traffic. Poor reliability on a crash.

- Alternatives:
  - Periodic flush sometime later tries to regulate the frequency of writes. (*Sprite*)
  - **Write on close** delays the write even longer.
  - Write-before-full (*ParFiSys*)
- Which would you use for a database file? For file editing?

DISTRIBUTED FILE SYSTEMS

CACHE CONSISTENCY:

The basic issue is, how to determine that the client-cached data is consistent with what's on the server.

- **Client - initiated approach – poll server**

  The client asks the server if the cached data is OK. What should be the frequency of "asking"? On file open, at fixed time interval, ...?

  - Validation frequency: on each access, on file open, periodically
  - If not write-through must maintain information about last validation
  - Higher network traffic, use of server CPU and increased time to serve requests

- **Server - initiated approach – keep clients informed of when their data becomes stale**

  Must keep track which clients cached which blocks: stateful servers not so fault tolerant

  Possibilities:

  •The server is notified on every open. If a file is opened for writing, then disable caching by other clients for that file.

  • Get read/write permission for each block; then disable caching only for particular blocks.

  - If a piece of data is used by more than one client, at least one in W mode, notify clients to update or invalidate their caches

  Breaks the client/server model

DISTRIBUTED FILE SYSTEMS

Remote File Access

COMPARISON OF CACHING AND REMOTE SERVICE:

- Many remote accesses can be handled by a local cache. There's a great deal of locality of reference in file accesses. Servers can be accessed only occasionally rather than for each access.
- Caching causes data to be moved in a few big chunks rather than in many smaller pieces; this leads to considerable efficiency for the network.
- Disk accesses can be better optimized on the server if it's understood that requests are always for large contiguous chunks.
- Cache consistency is the major problem with caching. When there are infrequent writes, caching is a win. In environments with many writes, the work required to maintain consistency overwhelms caching advantages.
- Caching works best on machines with considerable local store - either local disks or large memories. With neither of these, use remote-service.
- Caching requires a whole separate mechanism to support acquiring and storage of large amounts of data. Remote service merely does what's required for each call. As such, caching introduces an extra layer and mechanism and is more complicated than remote service.

DISTRIBUTED FILE SYSTEMS

Remote File Access

STATEFUL VS. STATELESS SERVICE:

Stateful: A server keeps track of information about client requests.

- It maintains what files are opened by a client; connection identifiers; server caches.

Stateless: Each client request provides complete information needed by the server (i.e., filename, file offset).

- The server can maintain information on behalf of the client, but it's not required.
- Useful things to keep include file info for the last N files touched.

DISTRIBUTED FILE SYSTEMS

Remote File Access

STATEFUL VS. STATELESS SERVICE:

Performance is better for stateful.

- Don't need to parse the filename each time, or "open/close" file on every request.
- Stateful can have a read-ahead cache.

Fault Tolerance: A stateful server loses everything when it crashes.

- Server must poll clients in order to renew its state.
- Client crashes force the server to clean up its encached information.
- Stateless remembers nothing so it can start easily after a crash.

DISTRIBUTED FILE SYSTEMS

Remote File Access

FILE REPLICATION:

- Duplicating files on multiple machines improves availability and performance.
- Placed on failure-independent machines (they won't fail together).

  Replication management should be "location-opaque".

- The main problem is consistency - when one copy changes, how do other copies reflect that change? Often there is a tradeoff: consistency versus availability and performance.
- Example:

  "Demand replication" is like whole-file caching; reading a file causes it to be cached locally. Updates are done only on the primary file at which time all other copies are invalidated.

- Atomic and serialized invalidation isn't guaranteed (message could get lost / machine could crash.)

DISTRIBUTED FILE SYSTEMS

SUN Network File System

OVERVIEW:

- Runs on SUNOS - NFS is both an implementation and a specification of how to access remote files.
- The goal: to share a file system in a transparent way.
- Can use heterogeneous machines - different hardware, operating systems, network protocols.
- Uses RPC on top of XDR for isolation - thus all implementations must have the same RPC calls. These RPC's implement the mount protocol and the NFS protocol.
- Uses client-server model (for NFS, a node can be both simultaneously.) Can act between any two nodes (no dedicated server.) Mount makes a server file-system visible from a client.

mount server:/usr/shared client:/usr/local

DISTRIBUTED FILE SYSTEMS

SUN Network File System

OVERVIEW:

mount server:/usr/shared client:/usr/local

- Then, transparently, a request for /usr/local/dir-server accesses a file that is on the server.
- The namespaces on the client and server are different.
- The mounting operation is NOT transparent, the client must know the name of the remote server.
- The mount is controlledby: (1) accessrights, (2) server specification of what's mountable.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

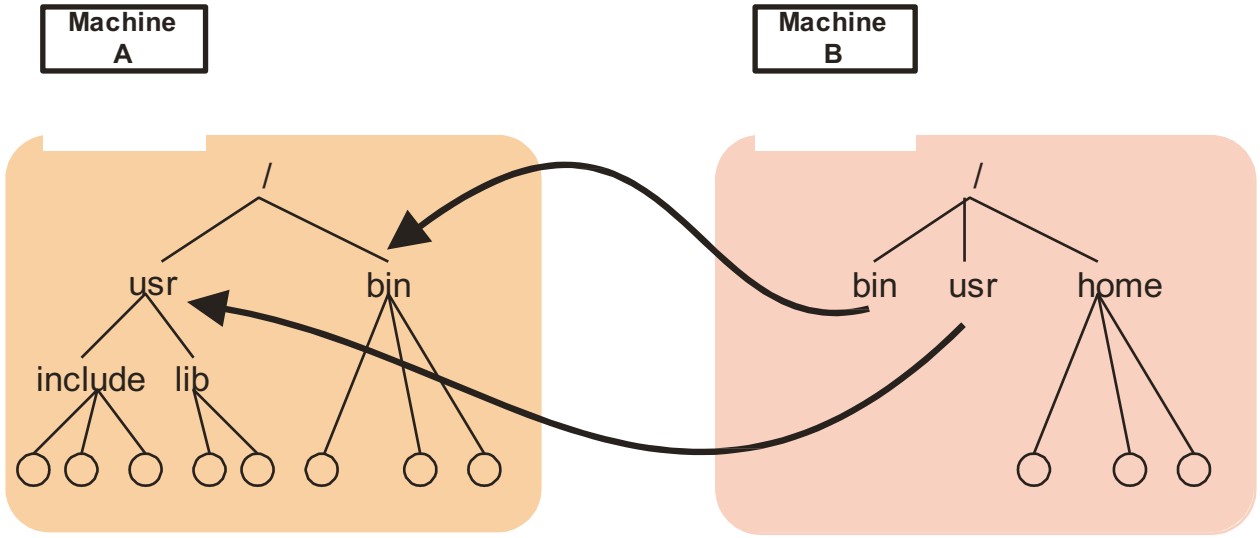
THE MOUNT PROTOCOL:

Establish a connexion between server and client.

Machine A exports */usr* and */bin*

On machine B run:

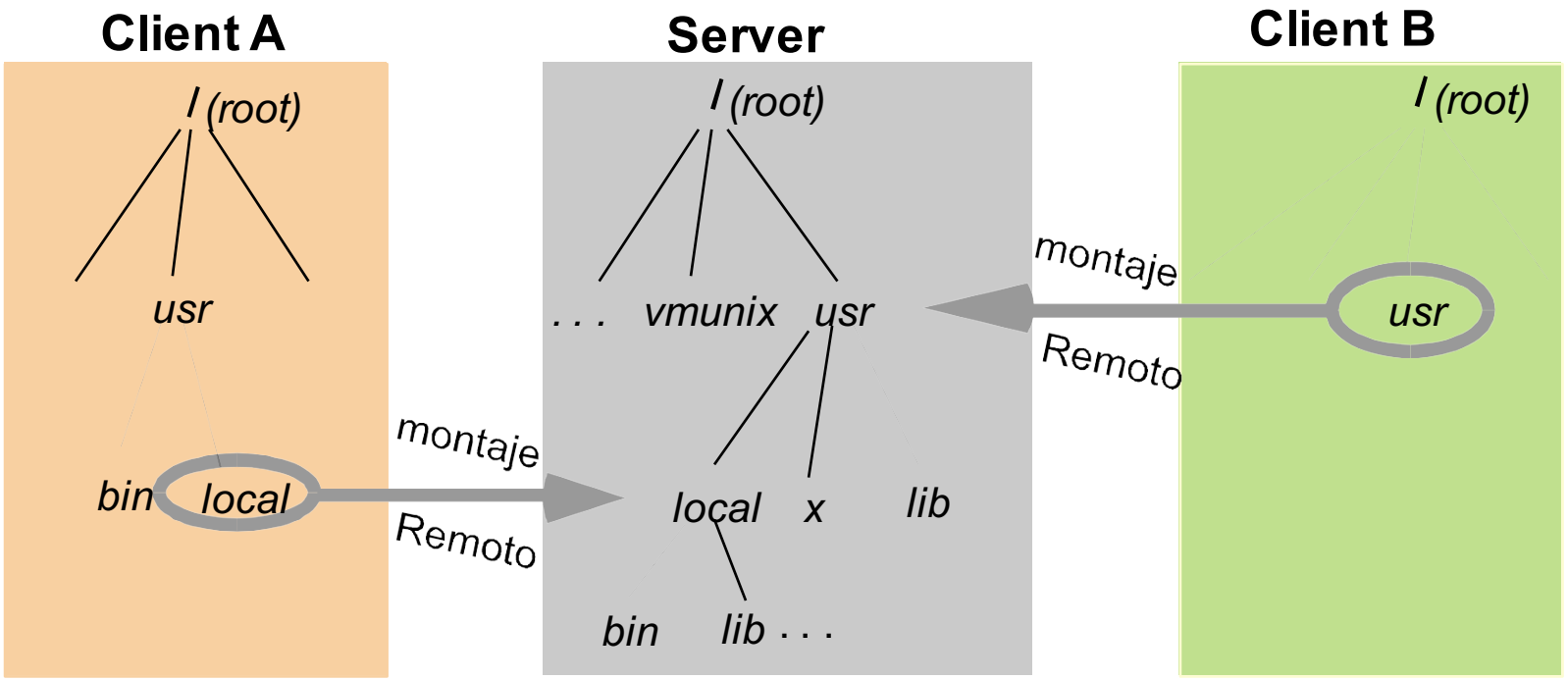
mount machineA:/usr /usr



DISTRIBUTED FILE SYSTEMS

SUN Network File System

The file system image may be different on different clients



DISTRIBUTED FILE SYSTEMS

SUN Network File System

THE MOUNT PROTOCOL:

The following operations occur:

1. The client's request is sent via RPC to the mount server (on server machine.)

2. Mount server checks export list containing

    a) file systems that can be exported,
    b) legal requesting clients.
    c) It's legitimate to mount any directory within the legal filesystem.

3. Server returns "file handle" to client.

4. Server maintains list of clients and mounted directories -- this is state information! But this data is only a "hint" and isn't treated as essential.

5. Mounting often occurs automatically when client or server boots.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

THE NFS PROTOCOL:

Offers a set of RPCs to support these remote file operations:

a) Search for file within directory.
b) Read a set of directory entries.
c) Manipulate links and directories.
d) Read/write file attributes.
e) Read/write file data.

Note:

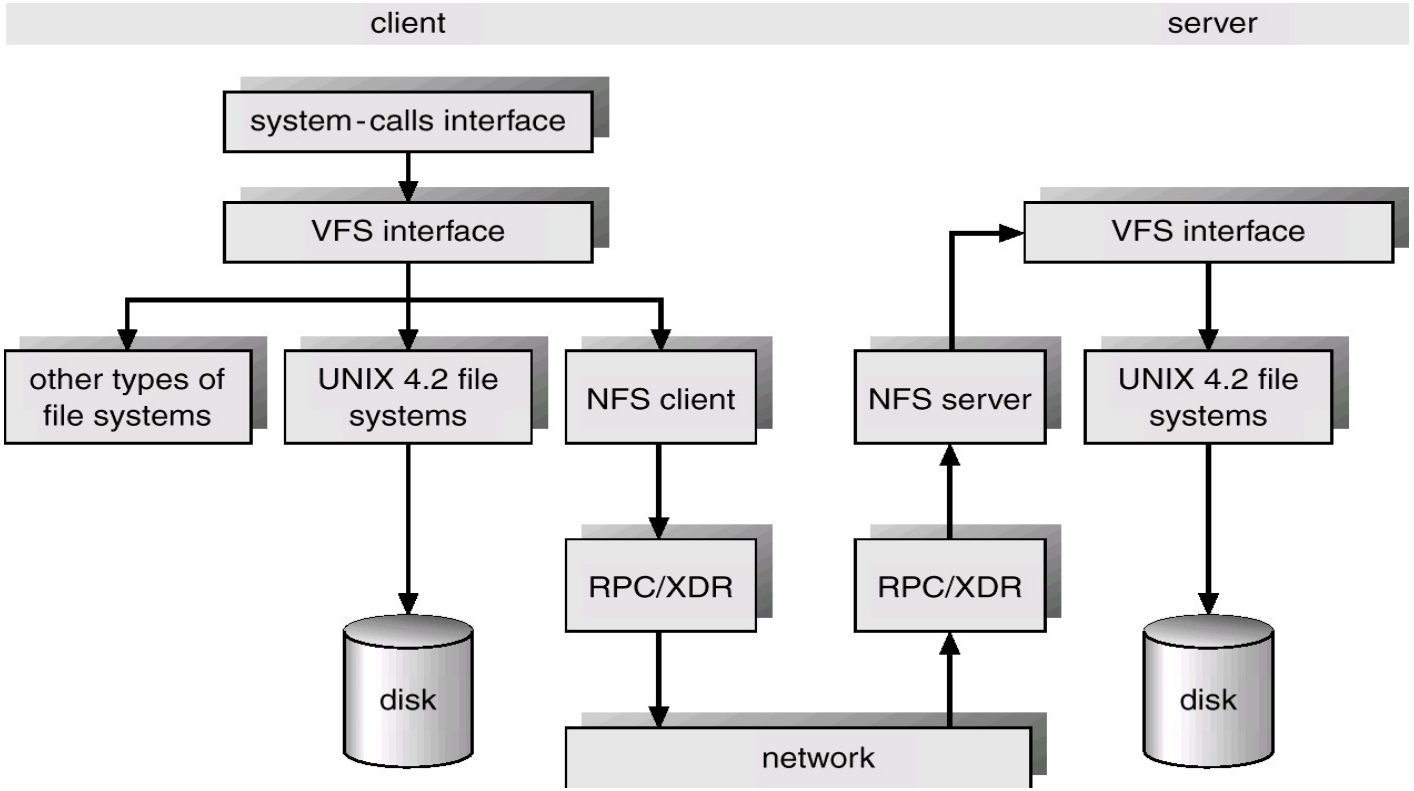
- Open and close are conspicuously absent from this list. NFS servers are **stateless**. Each request must provide all information. With a server crash, no information is lost.
- Modified data must actually get to server disk before client is informed the action is complete. Using a cache would imply state information.
- A single NFS write is **atomic**. A client write request may be broken into several atomic RPC calls, so the whole thing is NOT atomic. Since lock management is stateful, NFS doesn't do it. A higher level must provide this service.
  - NFS does not provide concurrency control mechanisms to implement, for instance, the UNIX semantics.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

NFS ARCHITECTURE:

Follow local and remote access through this figure:



DISTRIBUTED FILE SYSTEMS

SUN Network File System

NFS ARCHITECTURE:

1. UNIX filesystem layer - does normal open / read / etc. commands.

2. Virtual file system (VFS) layer
   a) Gives clean layer (interface) between user and filesystem.
   b) Independent of the actual file system.
   c) Acts as deflection point by using global vnodes (virtual nodes).
   d) Understands the difference between local and remote names.
   e) Keeps in memory information about what should be (mounted directories) and how to get to these remote directories.

3. System call interface layer
   a) Presents sanitized validated requests in a uniform way to the VFS.

FILE HANDLES

SUN Network File System

- Identifies a remote file
- In UNIX

File system id	i-node number	i-node generation number

  - File system id – unique nr corresponding to each filesystem when created
  - i-node generation number – incremented every time the i-node is reused

- VFS stores an entry for each open file (*v-node*) and one structure per mounted file system
- Each *v-node* points to either a local *i-node* or to a remote *filehandle*
- Client obtains first file handle for a remote file system when it mounts it

DISTRIBUTED FILE SYSTEMS

SUN Network File System

PATH-NAME TRANSLATION:

The **client** breaks the complete pathname into components. Pathnames cannot be translated at the server b/c name may cross a mount point at the client!!

When a v-node contains a file handle (i.e. it refers to a remote-mounted directory) the translation is done by a lookup request to the server.

File handles are opaque to clients! VFS responsible for resolving file handles to remote/local.

For each component, do an NFS lookup using the

component name + directory vnode.

After a mount point is reached, each component piece will cause a server access.

Can't hand the whole operation to server since the client may have a second mount on a subsidiary directory (a mount on a mount).

A directory name cache on the client speeds up lookups.

lookup(dir, name) *returns* **(fh, dir_attr)**

Fails if the client does not have permissions

DISTRIBUTED FILE SYSTEMS

SUN Network File System

PATH-NAME TRANSLATION:

Only one open call

fd = open(“/usr/joe/6360/list.txt”)

Results in several remote calls:

lookup(rootfh, “usr”) returns **(fh0, attr)**

lookup(fh0, “joe”) returns **(fh1, attr)**

lookup(fh1, “6360”) returns **(fh2, attr)**

lookup(fh2, “list.txt”) returns **(fh, attr)**

The *filehandle* **fh** identifies the file **list.txt** yand will be used by subsequent R/W operations

DISTRIBUTED FILE SYSTEMS
SUN Network File System

FILE ACCESS:

Using the *filehandle*:

read(fh, offset, cont) returns **(data, attr)**

write(fh, offset, cont, data) returns **(attr)**

The blocks transfered:

Version 2 – up tofff 8 KB

Version 3 - negotiable

Clients read ahead and write locally.

Blocks are sent to server when full or file closed.

WRITING TO SERVER CACHE:

Version 2: *write-through*

Version 3:

Write-through: write to disk before sending reply to client

Delayed-write: write to cache only

commit is the operation which forces write to disk. Normally the client force this operation on close.

DISTRIBUTED FILE SYSTEMS
SUN Network File System

CACHES OF REMOTE DATA ON THE CLIENT:

The client keeps the following info for *read*, *write*, *getattr*, *lookup* and *readdir* operations:

File block cache - (the contents of a file)

File attribute cache - (file header info (inode in UNIX)).

The local kernel hangs on to the data after getting it the first time. Cached attributes are thrown away after a few seconds.

For a file modified concurrently by two clients (which have it in their caches): A

doesn't see B's changes

B doesn't see A's changes

Inconsistencies -> cannot guarantee UNIX semantics

Mechanism has:

Server consistency problems. Good performance.

CACHES OF REMOTE DATA ON THE CLIENT: SOLUTION

The clients must:

Validate cache blocks before use (including when open):

If very recently validated or same timestamp as server go ahead use it.

Otherwise invalidate cache entry (i.e. discard block) and request valid data from server.

On write mark dirty and flush modified block asynchronously to server: when close

when client synch periodically

Ask server to validate data in the cache 3-30 s for files

30-60 s for directories

The attributes are discarded after

3 s for files

30 s for directories

DISTRIBUTED FILE SYSTEMS
Parallel File Systems

CHARACTERISTICS:

Distributes file data over multiple servers

Provides for concurrent access by multiple tasks

Servers offer the illusion of a unique global storage system

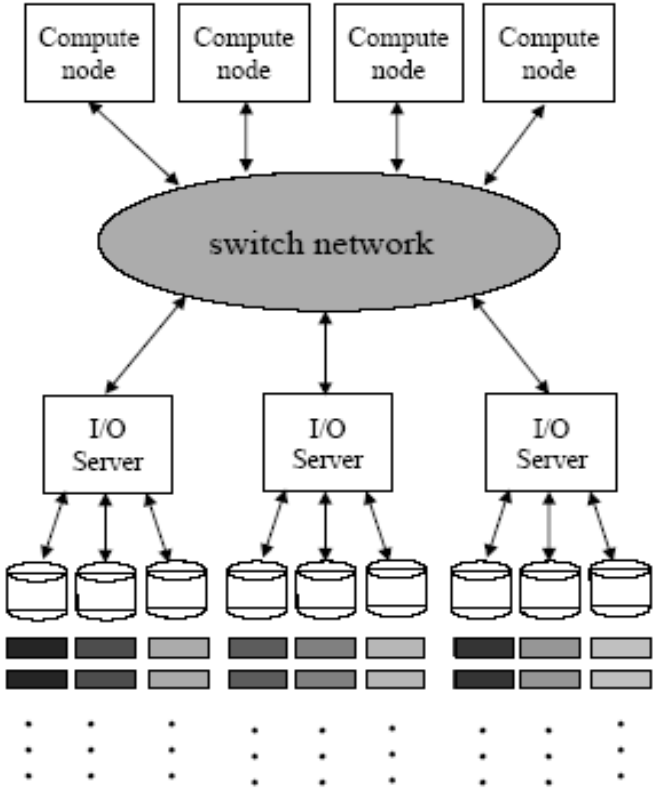
Servers must ensure data consistency when multiple clients access same data blocks

Used in cluster computing

E.g. PVFS:

High performance access to large data sets.

Client library provides for high performance access via the message passing interface (MPI).



DISTRIBUTED FILE SYSTEMS

TYPES:

Traditional - Direct Attached Storage (DAS): disks connected to servers

Storage networks

- Network Attached Storage (NAS)
- Storage Area Networks (SAN)

Shared storage file systems

DISTRIBUTED FILE SYSTEMS

DAS

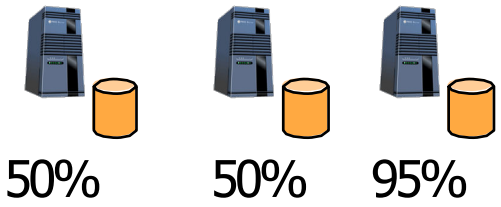
ADVANTAGES:

Independent services

DISAVANTAGES:

Admin and maintenance
Saturation – servers may become a bottleneck, no load balancing
Scalability
Cost
Performance
Bad resource utilization

SOLUTION:
More servers
Partitioning – Hard to get right!!



DISTRIBUTED FILE SYSTEMS

NAS

Uses standard protocols such as NFS, HTTP, …

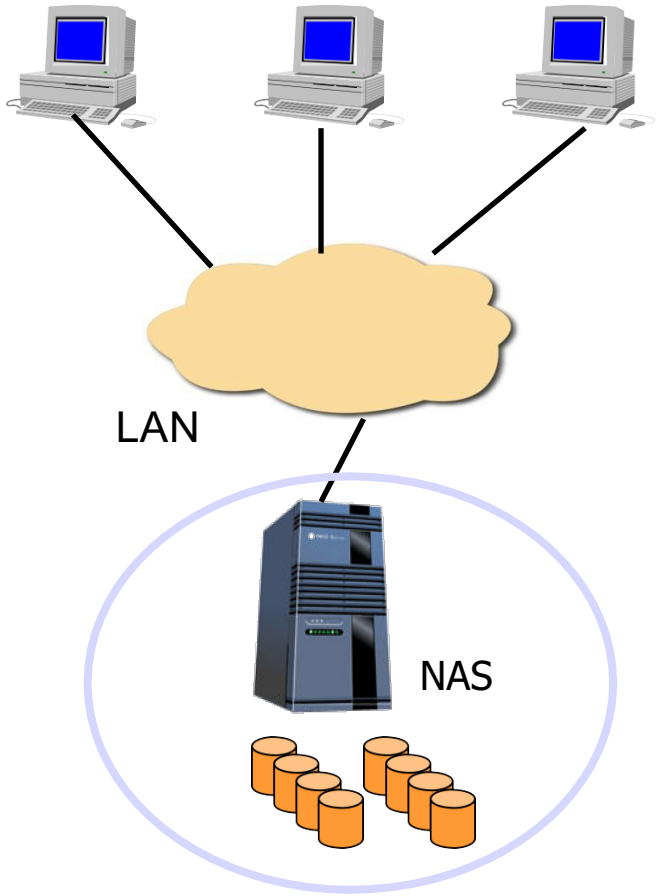
In general NAS only provides file-based data storage services to other devices on the network:
special purpose!

NAS provides both storage and a file system

File level access

ADVANTAGES:

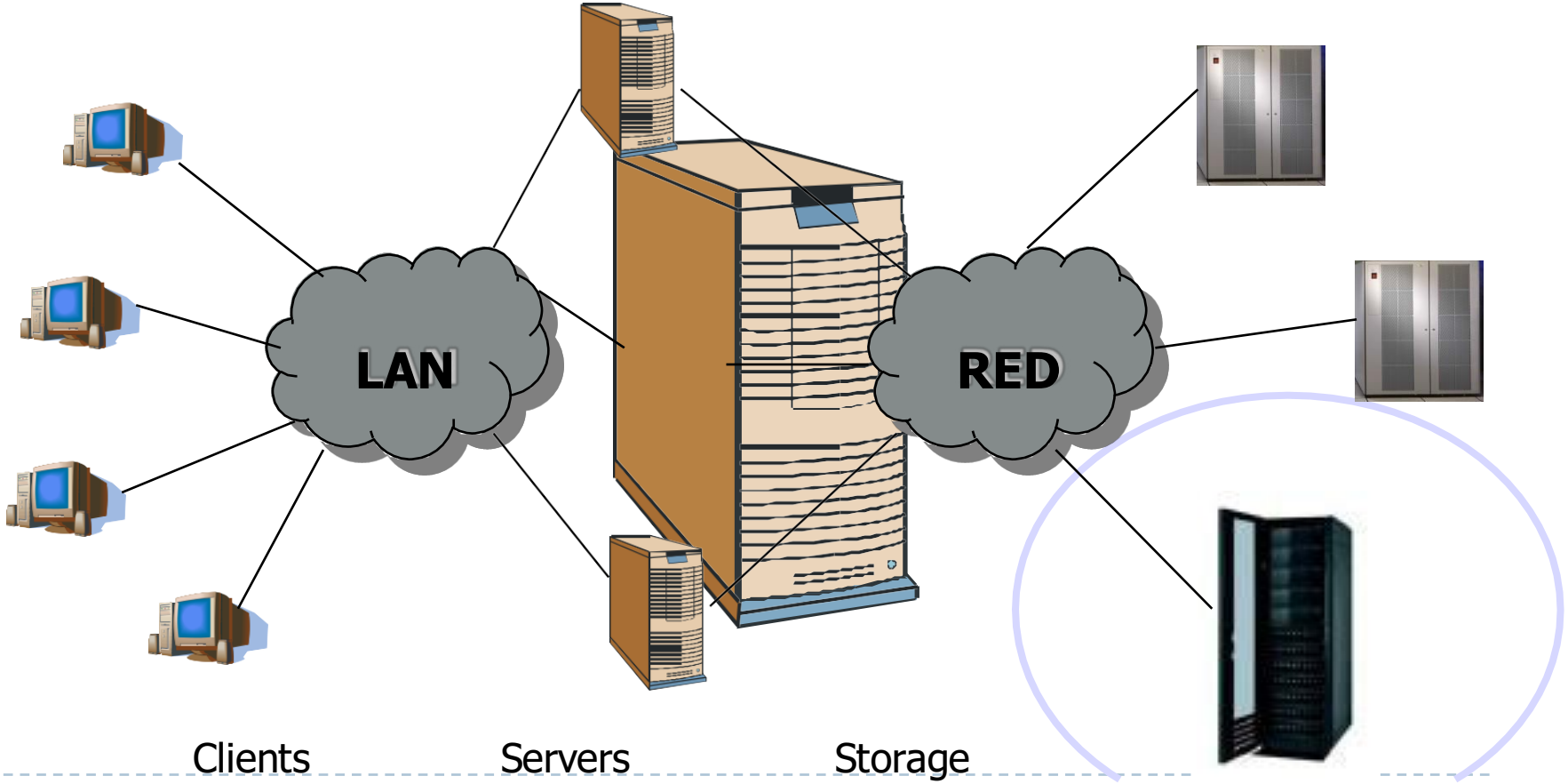
Faster data access (than DAS), easier administration, and simple configuration, load balancing



DISTRIBUTED FILE SYSTEMS

SAN

- Provides only block-based storage - block-level access Leaves file system concerns on the "client" side Protocols: SCSI, Fiber channel, …
- NAS appears to the client OS as a file server (the client can map network drives to shares on that server)
- A disk available through a SAN appears to the client OS as a disk, visible along client's local disks, and available to be mounted



DISTRIBUTED FILE SYSTEMS

Shared storage file systems

Clients have direct access to disks

Requires a metadata manager