Distributed Operating Systems FILE SYSTEM

Introduction

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.

DISTRIBUTED FILE SYSTEMS

DEFINITIONS:

- A **Distributed File System** (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- 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 implementation 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 SYSTEMS

Distributed file system support:

- **Remote Information Sharing**- Allows a file to be transparently accessed by processes of any node of the system irrespective of the file’s location
- **User Mobility-** User have flexibility to work on different node at different time
- **Availability-** better fault tolerance
- **Diskless Workstations**

DISTRIBUTED FILE SYSTEMS

Distributed File System provide following type of services:

- Storage Service
- True File Service
- Name Service

DISTRIBUTED FILE SYSTEMS

Desirable features of a good distributed file system

- Transparency
    - ✓ Structure transparency
    - ✓ Access Transparency
    - ✓ Naming Transparency
    - ✓ Replication Transparency
- User Mobility
- Performance
- Simplicity and ease of use
- Scalability
- High Availability
- High Reliability
- Data Integrity
- Security
- Heterogeneity

File Models

Criteria: Structure and Modifiability

Structured and Unstructured Files

- Structured Files: A file appear to the file server as an ordered sequence of records.
  - Files with Indexed Records
  - Files With non-indexed records
- Unstructured files: No substructure known to the file server

File Models

Mutable and Immutable Files

- Mutable
  - An update performed on a file overwrites on its old contents
  - A file is represented as a single stored sequence that is altered by each update operation.
- Immutable Files
  - A file cannot be modified once it has been created
  - File versioning approach used to implement file updates
  - It support consistent sharing therefore it is easier to support file caching and replication

File Accessing Models

File Accessing Models of DFS mainly depends on : Method used for accessing remote files and the unit of data access

Accessing Remote Files

- Remote Service Model:
  - Client's request processed at server's node
  - In this case Packing and communication overhead can be significant
- Data Caching Model:
  - Client's request processed on the client's node itself by using the cached data.
  - This model greatly reduces network traffic
  - Cache consistency problem may occur

LOCUS and NFS use the remote service model but add caching for better performance

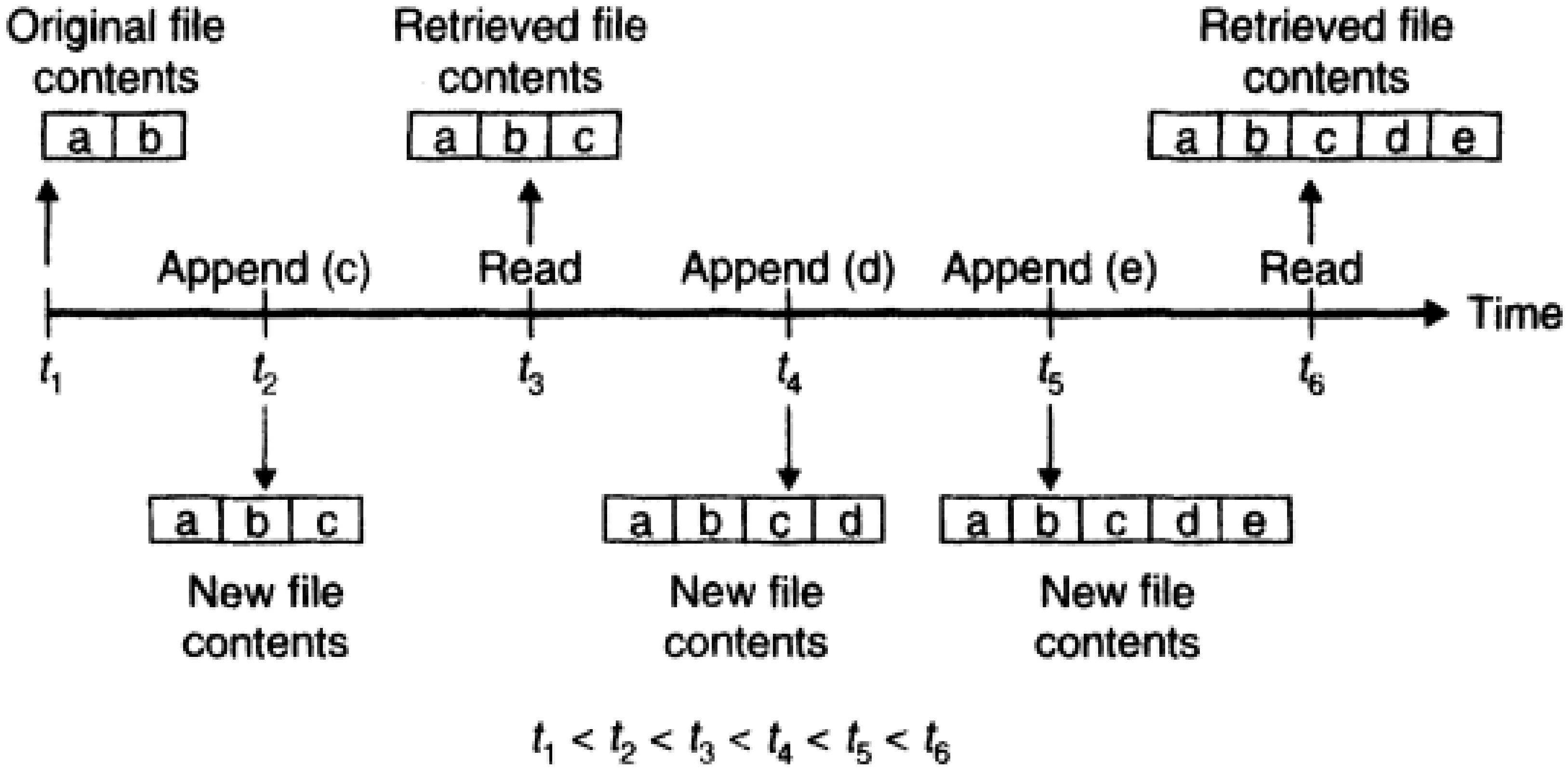
Sprite use data caching model but employs the remote service model under certain circumstances

File Accessing Models

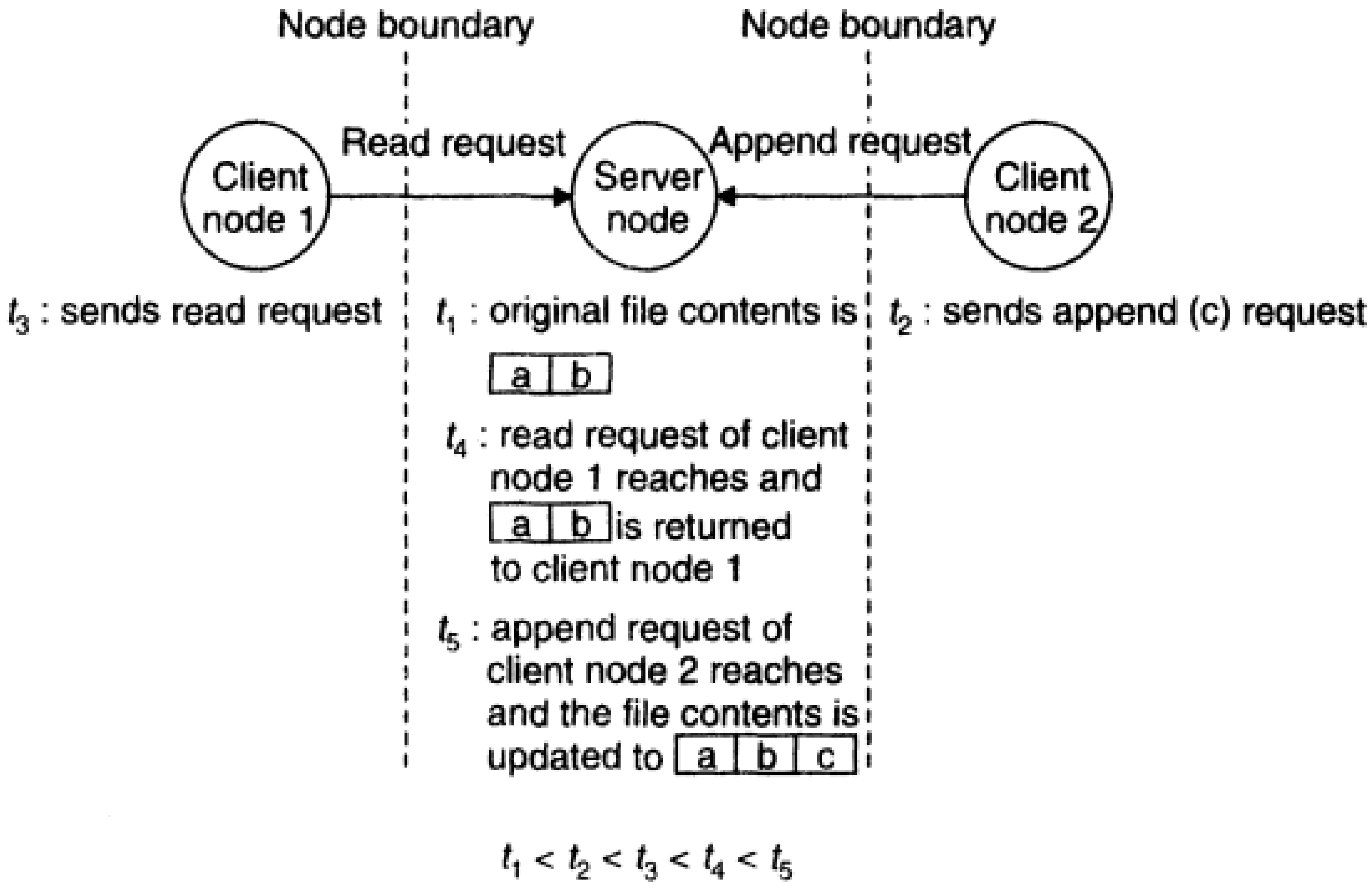
Unit of Data Transfer

- File Level Transfer Model(Ex. Amoeba, AFS)
  - The whole file is moved when an operation requires file data
  - It is simple, It has better scalability
  - Disk access routines on the servers can be better optimized
  - But it requires sufficient storage space on client's node
- Block Level Transfer Model(Ex. LOCUS, Sprite)
  - Data transferred in units of file blocks
  - It does not require client node to have large storage space
  - It can be used in diskless workstations
  - Network traffic may be significant
- Byte Level Transfer Model(Cambridge file server)
  - Data transfers in units of bytes
  - Low Storage requires but difficulty in cache management
- Record Level Transfer Model(Research Storage System)
  - Suitable for Structured model

File Sharing Semantics
1. Unix semantics



File Sharing Semantics
Problem due to Unix semantics


Node boundary
Node boundary
Read request
Append request
Client node 1
Server node
Client node 2
t_3 : sends read request
t_1 : original file contents is a | b
t_2 : sends append (c) request
t_4 : read request of client node 1 reaches and a | b is returned to client node 1
t_5 : append request of client node 2 reaches and the file contents is updated to a | b | c
$t_1 < t_2 < t_3 < t_4 < t_5$

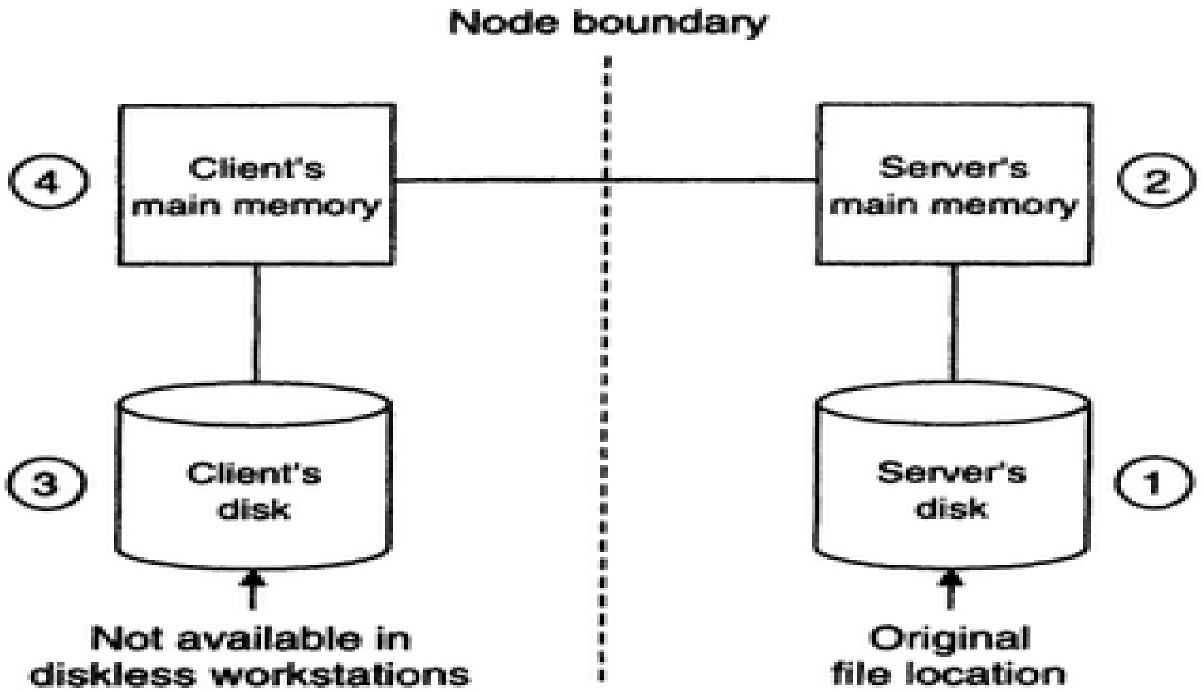
File Sharing Semantics

2. Session semantics
3. Immutable shared-files semantics
4. Transaction-like semantics

File-Caching Schemes

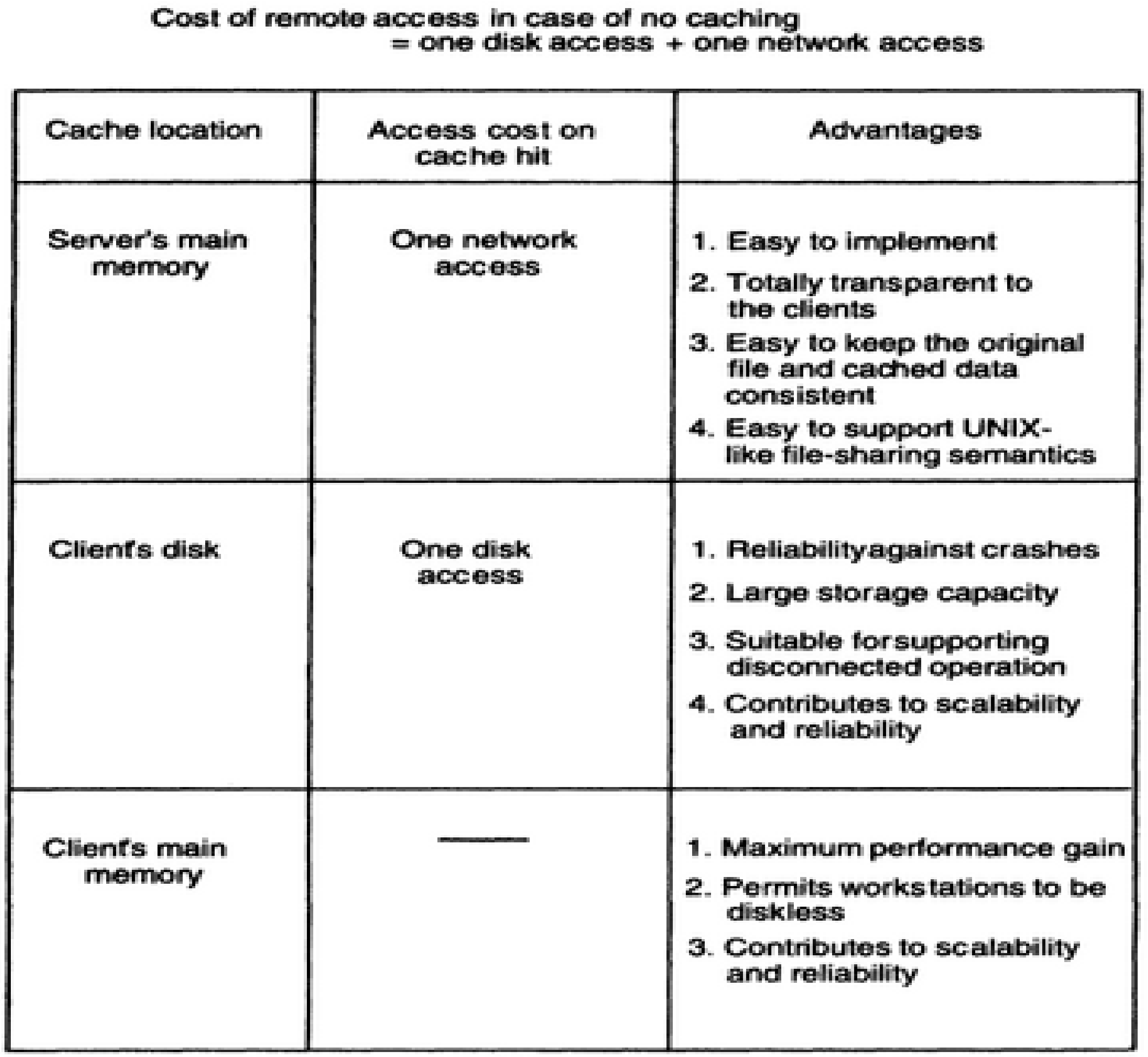
- 1. Cache location
- 2. Modification Propagation
- 3. cache validation

Possible Cache location



1. No caching
2. Cache located in server's main memory
3. Cache located in client's disk
4. Cache located in client's main memory

Summary of the relative advantages of the three cache location policies.



Cost of remote access in case of no caching
= one disk access + one network access

Cache location	Access cost on cache hit	Advantages
Server's main memory	One network access	1. Easy to implement 2. Totally transparent to the clients 3. Easy to keep the original file and cached data consistent 4. Easy to support UNIX-like file-sharing semantics
Client's disk	One disk access	1. Reliabilityagainst crashes 2. Large storage capacity 3. Suitable forsupporting disconnected operation 4. Contributes to scalability and reliability
Client's main memory	——	1. Maximum performance gain 2. Permits workstations to be diskless 3. Contributes to scalability and reliability

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 UPDATE POLICY:

A **write through** cache

- When a cache entry is modified, the new value is immediately set to server for updating master copy of file
- It has good reliability. But the user must wait for writes to get to the server. Used by NFS.

Delayed write

- Modified value written only to the cache and client make a note
- All update gathered and sent to server at a time
  - Write on ejection from cache
  - Periodic write
  - Write on close
- write requests complete more rapidly. Data may be written over the previous cache write, saving a remote write. Poor reliability on a crash.

DISTRIBUTED FILE SYSTEMS

CACHE CONSISTENCY: Cache Validation Schemes

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

- **Client - initiated approach -**

  The client asks the server if the cached data is OK. What should be the frequency of "asking"? Before every access, On file open, at fixed time interval, ...?
  1. Checking before every access.
  2. Periodic checking
  3. Check onfile open

Cache Validation Schemes

- **Server - initiated approach -**
- **In this method, a client informs the file server when opening a file, indicating whether the file is being opened for reading, writing, or both.**
- **The file server keeps a record of which client has which file open and in what mode.**
- **In this manner, the server keeps monitoring the file usage modes being used by different clients and reacts whenever it detects a potential for inconsistency.**
- **A potential for inconsistency occurs when two or more clients try to open a file in conflicting modes.**

DISTRIBUTED FILE SYSTEMS

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

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.
- Memory must be reclaimed when client closes file or when client dies.

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

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

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. It's both a definition and a specific instance.
- The goal: to share a file system in a transparent way.
- 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

- Then, transparently, a request for /usr/local/dir-server accesses a file that is on the server.
- The mount is controlled by: (1) access rights, (2) server specification of what's mountable.
- Can use heterogeneous machines - different hardware, operating systems, network protocols.
- Uses RPC for isolation - thus all implementations must have the same RPC calls. These RPC's implement the mount protocol and the NFS protocol.

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:

RPC's 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:

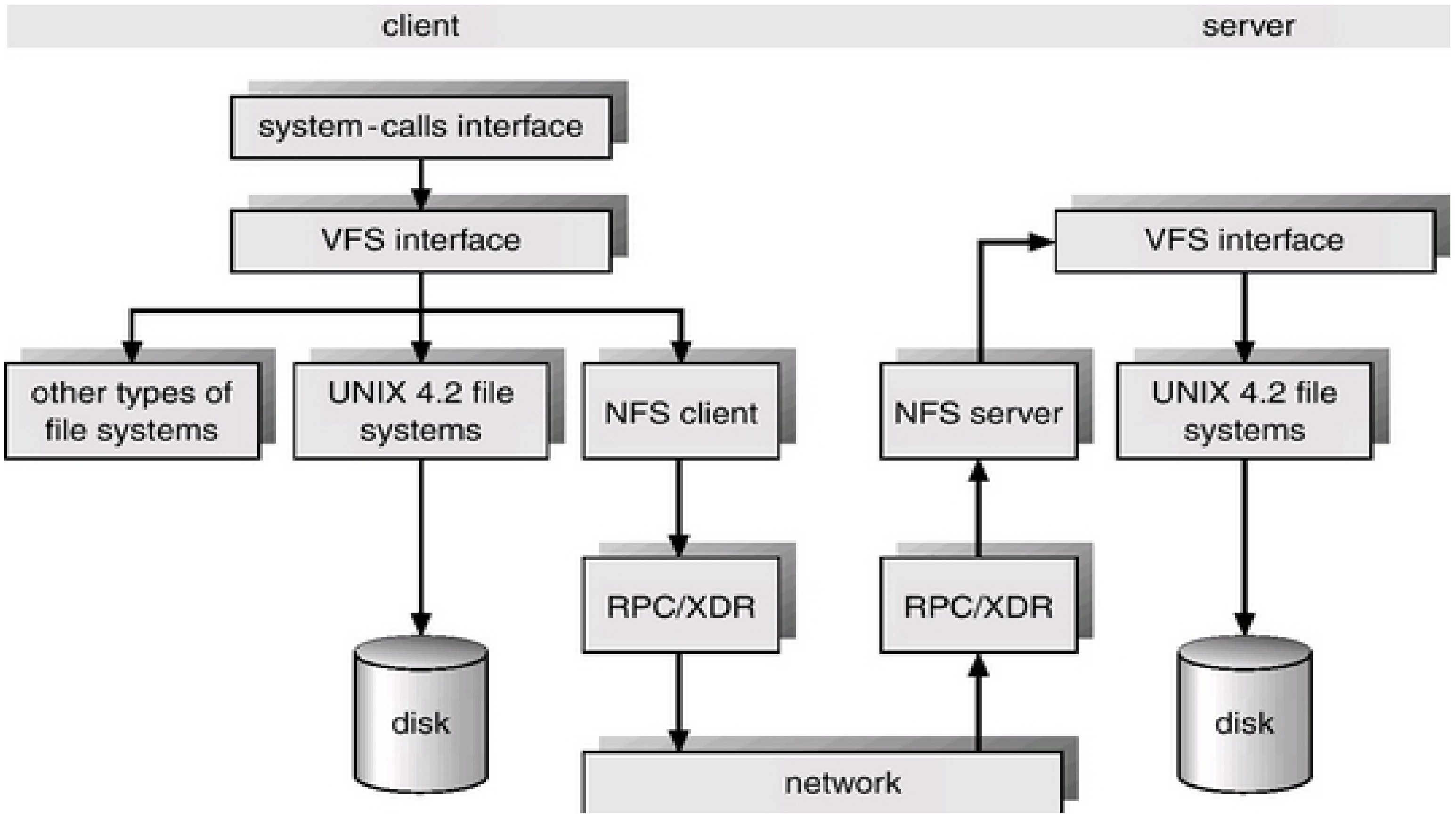
- Open and close are conspicuously absent from this list. NFS servers are **stateless**. Each request must provide all information. With a server crash, no information is lost.
- Modified data must actually get to server disk before client is informed the action is complete. Using a cache would imply 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.

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 between user and filesystem.

   b) Acts as deflection point by using global vnodes.

   c) Understands the difference between local and remote names.

   d) Keeps in memory information about what should be deflected (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.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

PATH-NAME TRANSLATION:

- Break the complete pathname into components.
- 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.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

CACHES OF REMOTE DATA:

- The client keeps:
  - 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.
- On an open, local kernel, it checks with server that cached data is still OK.
- Cached attributes are thrown away after a few seconds.
- Data blocks use read ahead and delayed write.
- Mechanism has:
  - Server consistency problems.
  - Good performance.

DISTRIBUTED FILE SYSTEMS

Andrew File System

A distributed environment at CMU. Strongest characteristic is scalability.

OVERVIEW:

- Machines are either servers or clients.
- Clients see a local name space and a shared name space.
- **Servers**

  run **vice** which presents a homogeneous, location transparent directory structure to all clients.

- **Clients** (workstations):

  Run **virtue** protocol to communicate with vice.
  Have local disks (1) for local name space, (2) to cache shared data.

- For scalability, off load work from servers to clients. Uses whole file caching.
- NO clients or their programs are considered trustworthy.

DISTRIBUTED FILE SYSTEMS

Andrew File System

SHARED NAME SPACE:

- The server file space is divided into volumes. Volumes contain files of only one user. It's these volumes that are the level of granularity attached to a client.
- A vice file can be accessed using a fid = <volume number, vnode >. The fid doesn't depend on machine location. A client queries a volume-location database for this information.
- Volumes can migrate between servers to balance space and utilization. Old server has "forwarding" instructions and handles client updates during migration.
- Read-only volumes (system files, etc.) can be replicated. The volume database knows how to find these.

DISTRIBUTED FILE SYSTEMS

Andrew File System

FILE OPERATIONS AND CONSISTENCY SEMANTICS:

- If a file is remote, the client operating system passes control to a client user-level process named
  Venus.
- The client talks to Vice server only during open/close; reading/writing are only to the local copy.
- A further optimization - if data is locally cached, it's assumed to be good until the client is told otherwise.
- A client is said to have a callback on a file.
- When a client encaches a file, the server maintains state for this fact.
- Before allowing a write to a file, the server does a callback to anyone else having this file open; all other cached copies are invalidated.
- When a client is rebooted, all cached data is suspect.
- If too much storage used by server for callback state, the server can break some callbacks.
- The system clearly has consistency concerns.

DISTRIBUTED FILE SYSTEMS

Andrew File System

IMPLEMENTATION:

- Deflection of open/close:
- The client kernel is modified to detect references to vice files.
- The request is forwarded to Venus with these steps:
- Venus does pathname translation.
- Asks Vice for the file
- Moves the file to local disk
- Passes inode of file back to client kernel.
- Venus maintains caches for status (in memory) and data (on local disk.)
- A server user-level process handles client requests.
- A lightweight process handles concurrent RPC requests from clients.
- State information is cached in this process.
- Susceptible to reliability problems.

DISTRIBUTED FILE SYSTEMS

Wrap Up

In this section we have looked at how files systems are implemented across systems. Of special concern is consistency, caching, and performance.