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.
- 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**.
- Distributed file systems support the sharing of information in the form of files throughout the intranet.
- 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.

A distributed file system enables programs to store and access remote files exactly as they do on local ones, allowing users to access files from any computer on the intranet.

Performance is concerned with throughput and response time.

DISTRIBUTED FILE SYSTEMS

Naming and Transparency

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.

DISTRIBUTED FILE SYSTEMS

Naming and Transparency

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 name followed by its local name. The local name is Unix like path.
    - This guarantees a unique name. NEITHER location transparent 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 and Transparency

NAMING SCHEMES:

2. Remote directories are **mounted** to local directories.
   - So a local system seems to have a coherent/similar directory structure.
   - The remote directories must be explicitly 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 file system.
   - Location independent.

DISTRIBUTED FILE SYSTEMS

Naming and Transparency

IMPLEMENTATION TECHNIQUES:

•A **non-transparent** mapping technique:

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

•A **transparent** mapping technique:

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

•So when changing the physical location of a file, only the file identifier need be modified. This identifier must be "unique”.

DISTRIBUTED FILE SYSTEMS
Remote File Access

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.

DISTRIBUTED FILE SYSTEMS
Remote File Access

CACHING

- **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
Remote File Access

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".

DISTRIBUTED FILE SYSTEMS
Remote File Access

CACHE LOCATION:

- 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,

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.
- Caching works best on machines with considerable local store - either local disks or large memories.

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.
- 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.

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.

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.

stateful vs. stateless servers

Advantages of stateless servers	Advantages of stateful servers
Fault tolerance	Shorter request messages
No OPEN/CLOSE calls needed	Better performance
No server space wasted on tables	Read ahead possible
No limits on number of open files	Idempotency easier
No problem if a client crashes	File locking possible

File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:
  - A flat file service
  - A directory service
  - A client module.
- The relevant modules and their relationship is shown in Figure (next slide).

File Service Architecture

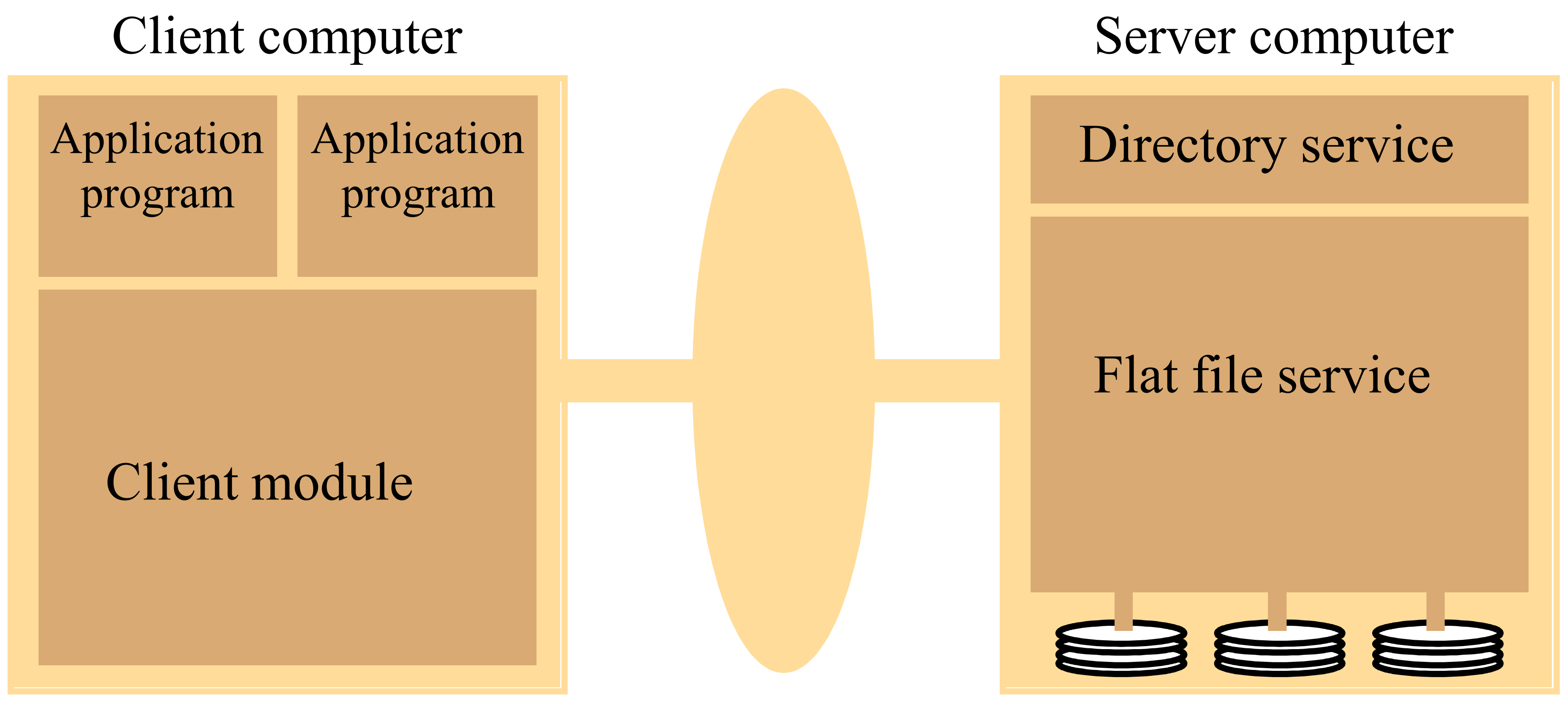


Figure: File service architecture

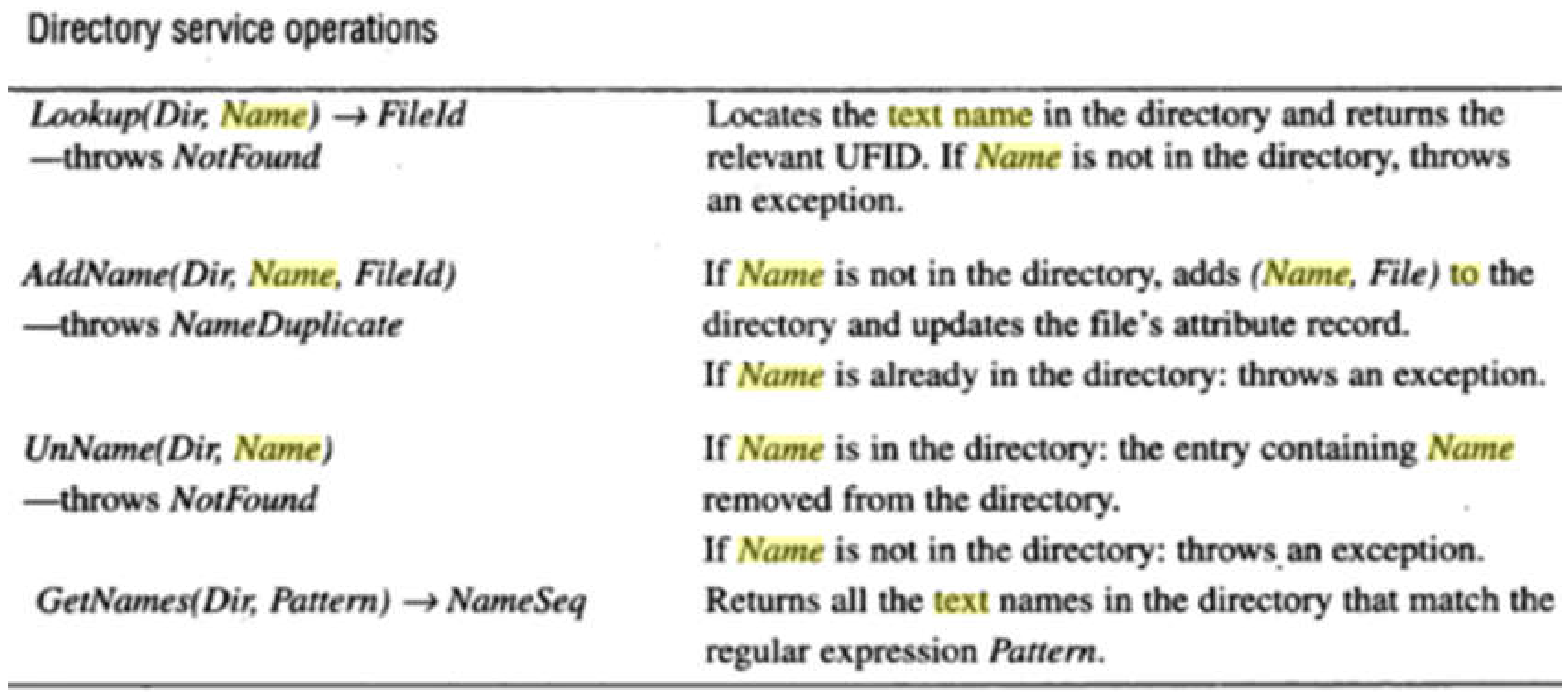
File Service Architecture

- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
  - Flat file service:
    - Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.

File Service Architecture

- Directory service:
  - Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.

File Service Architecture



Directory service operations

Lookup(Dir, Name) → *FileId* —throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) —throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) —throws *NotFound*	If *Name* is in the directory: the entry containing *Name* removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

File Service Architecture

- Client module:
  - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
  - It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

File Service Architecture

- Hierarchic file system

  In contrast to a hierarchical file system, in which there are directories and subdirectories and different files can have the same name as long as they are stored in different directories

- Flat file service

  In a flat file system every file must have a different name because there is only one list of files. Early versions of the Macintosh and DOS operating systems used a flat file system. Today's commercial operating systems use a hierarchical file system.

DFS: Case Studies

- NFS (Network File System)
    - Developed by Sun Microsystems (in 1985)
    - Most popular, open, and widely used.
    - NFS protocol standardized through IETF (RFC 1813)
- AFS (Andrew File System)
    - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
    - A research project to create campus wide file system.
    - Public domain implementation is available on Linux (LinuxAFS)
    - It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment