NETWORK DISTRIBUTED SYSTEMS
DISTRIBUTED FILE SYSTEMS

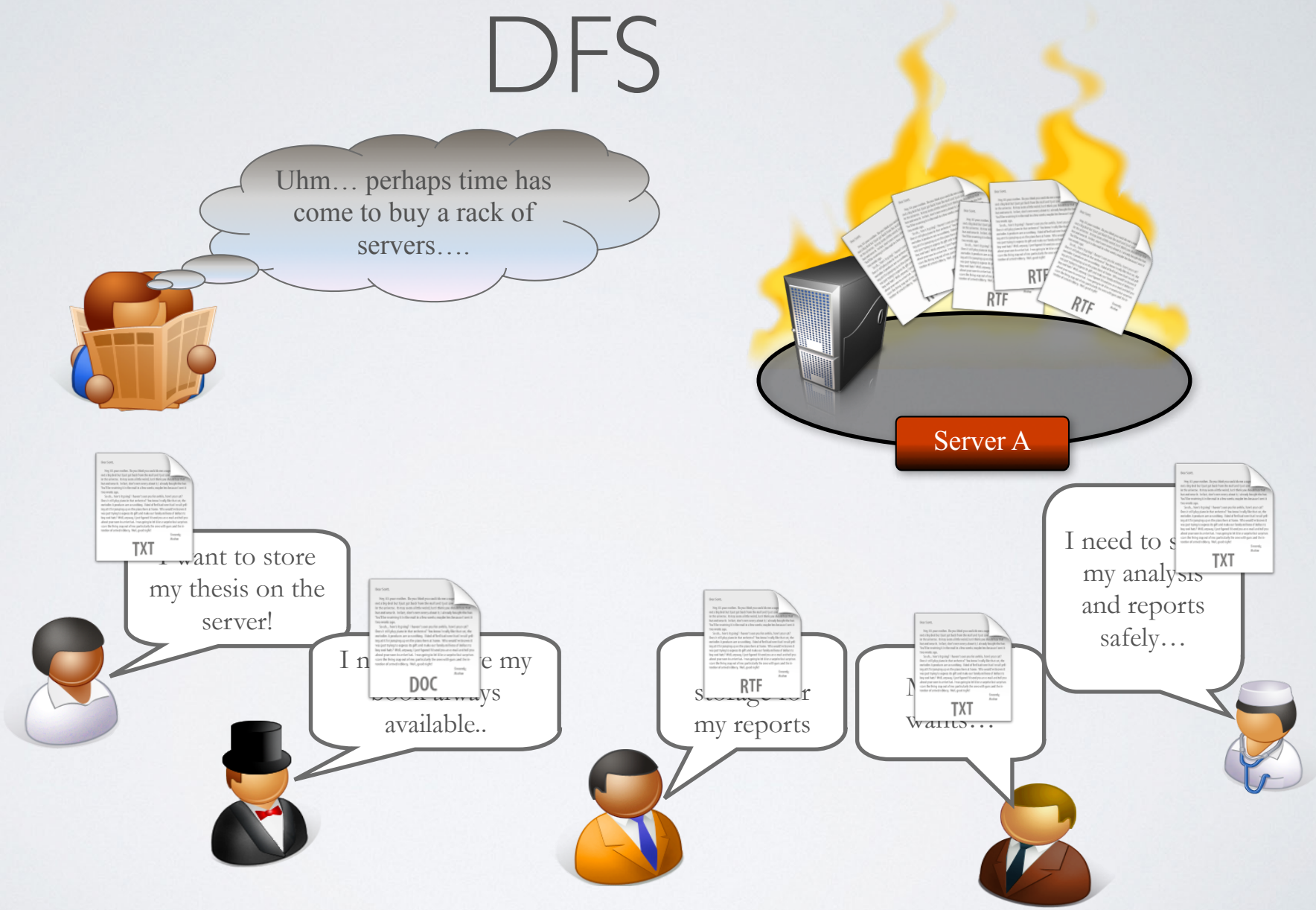
INTRODUCTION

- Why do we need a DFS?
  - Primary purpose of a Distributed System…

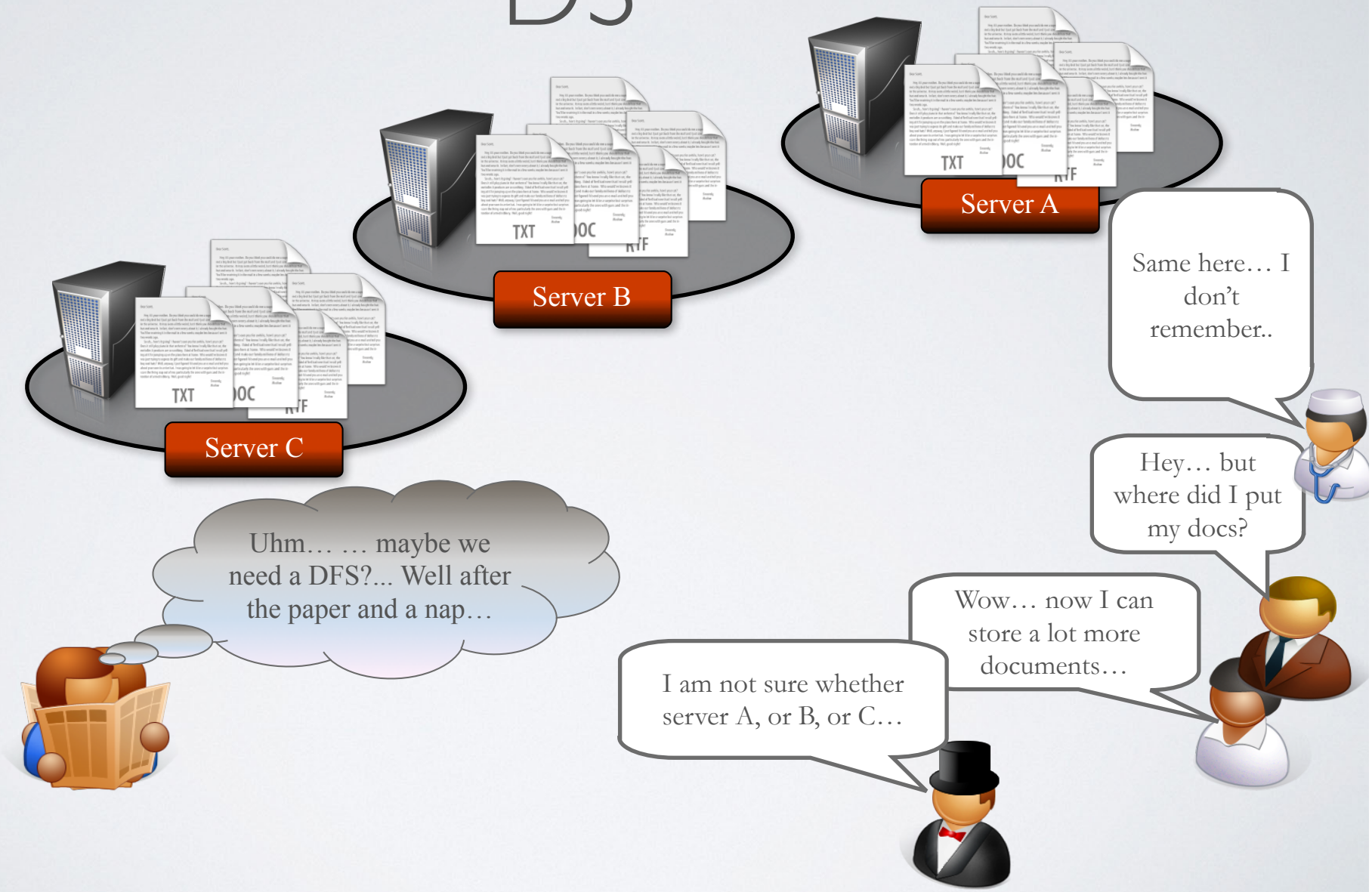
Connecting users and resources

  - Resources…
    - … can be inherently distributed
    - … can actually be data (files, databases, …) and…
    - … their availability becomes a crucial issue for the performance of a Distributed System

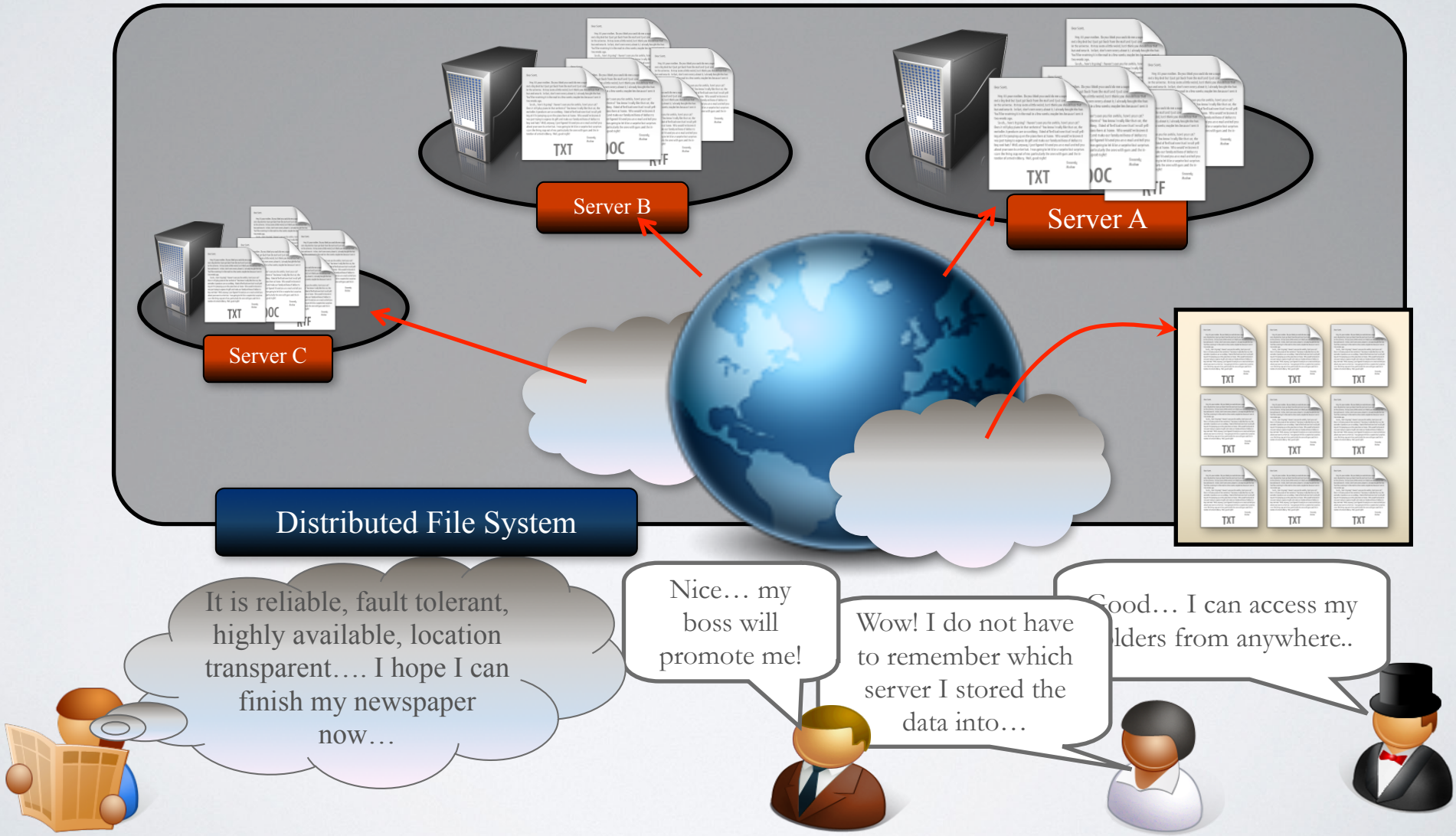
INTRODUCTION: A CASE FOR DFS



INTRODUCTION: A CASE FOR DS

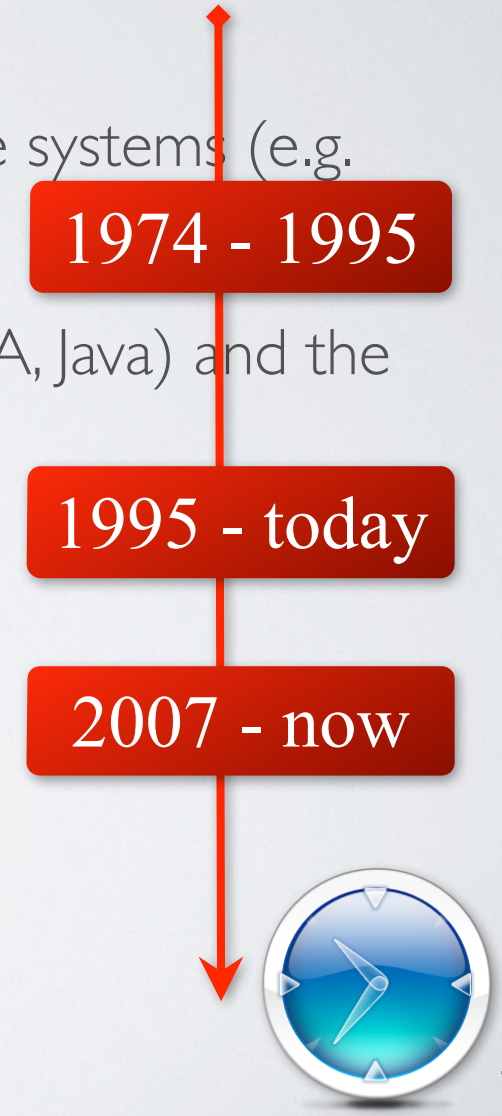


INTRODUCTION: A CASE FOR DFS



STORAGE SYSTEMS AND THEIR PROPERTIES

- In first generation of distributed systems (1974-95), file systems (e.g. NFS) were the only networked storage systems.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Current focus is on large scale, scalable storage.
  - Google File System
  - Amazon S3
  - Cloud Storage (e.g., DropBox)



STORAGE SYSTEMS AND THEIR PROPERTIES

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:

1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

WHAT IS A FILE SYSTEM? I

- Persistent stored data sets
- Hierarchic name space visible to all processes
- API with the following characteristics:
  - access and update operations on persistently stored data sets
  - Sequential access model (with additional random facilities)
- Sharing of data between users, with access control
- Concurrent access:
  - certainly for read-only access
  - what about updates?
- Other features:
  - mountable file stores
  - more? ...

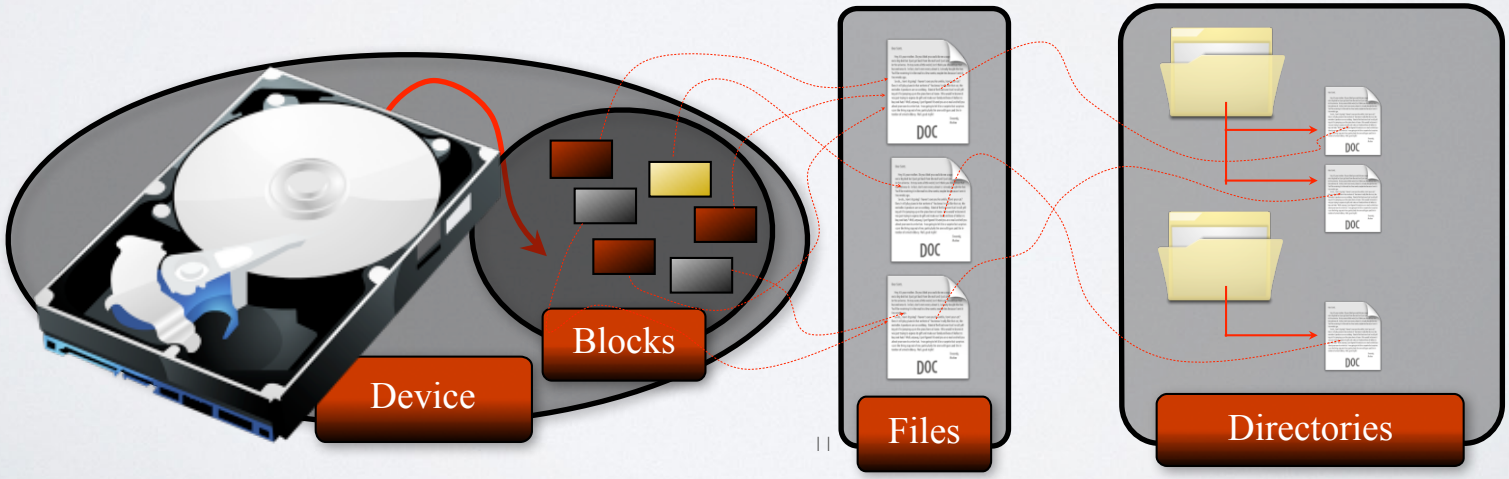
WHAT IS A FILE SYSTEM? 2

Figure 12.4 UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

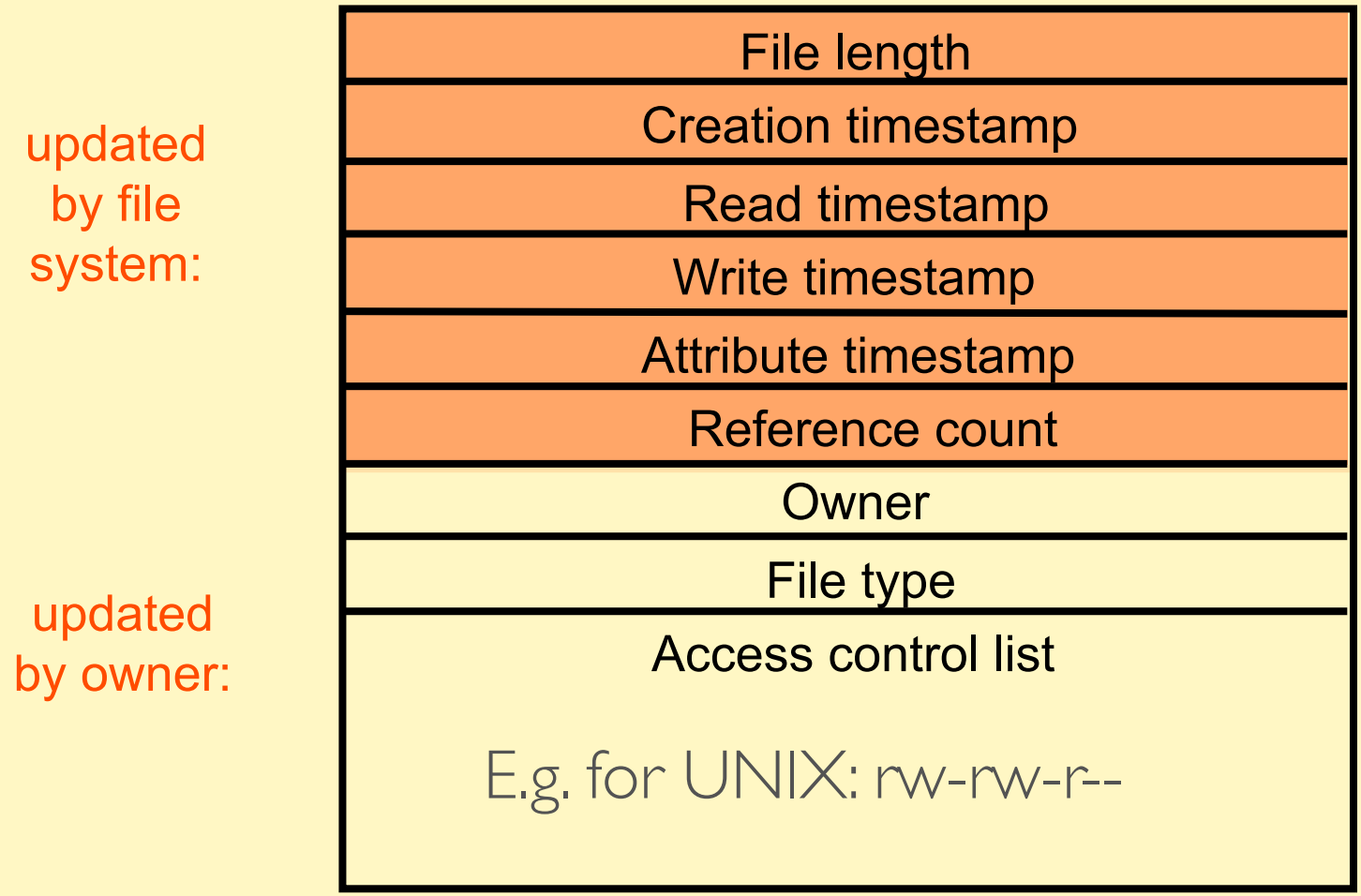
WHAT IS A FILE SYSTEM?
(A TYPICAL MODULE STRUCTURE FOR IMPLEMENTATION OF NON-DFS)

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering



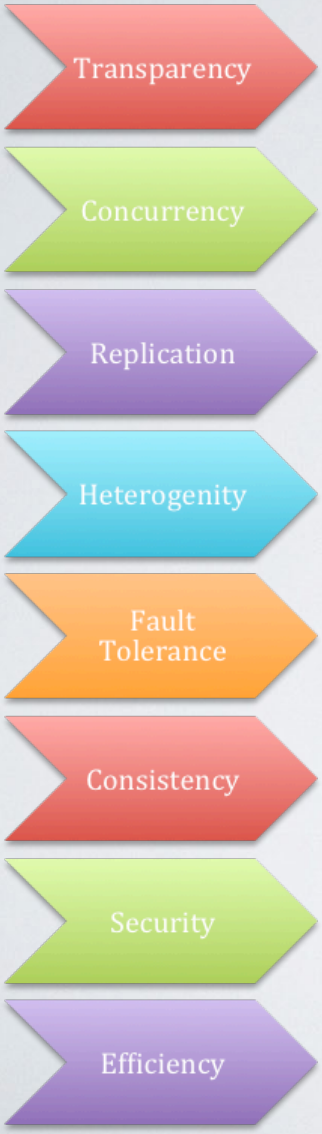
WHAT IS A FILE SYSTEM? 4

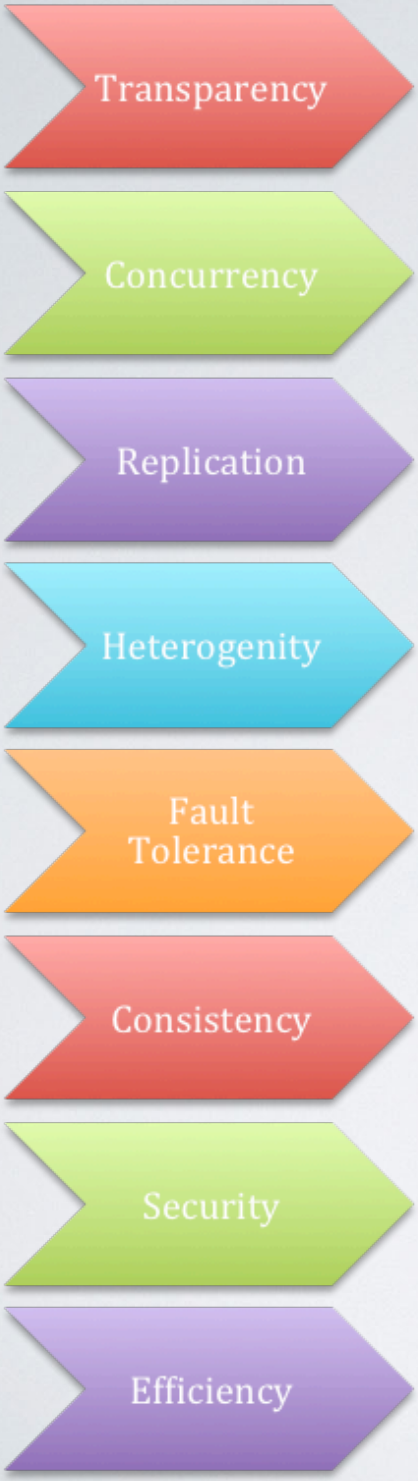
Figure 12.3 File attribute record structure



*

DISTRIBUTED FILE SYSTEM/ SERVICE REQUIREMENTS





Transparency

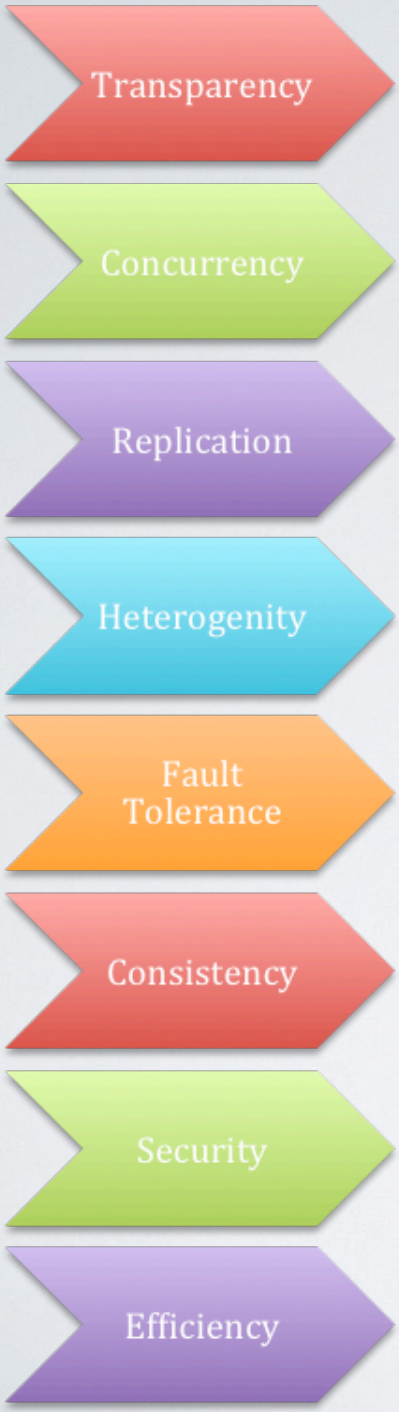
Access: Same operations (client programs are unaware of distribution of files)

Location: Same name space after relocation of files or processes (client programs should see a uniform file name space)

Mobility: Automatic relocation of files is possible (neither client programs nor system admin tables in client nodes need to be changed when files are moved).

Performance: Satisfactory performance across a specified range of system loads

Scaling: Service can be expanded to meet additional loads or growth.

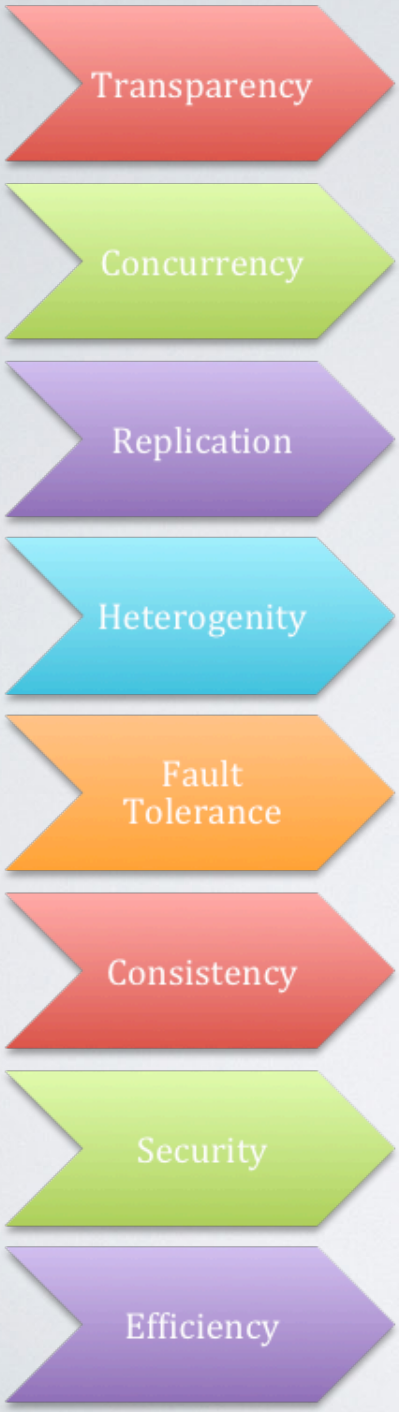


Concurrency

Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file.

Concurrency Properties

- Isolation
- File-level/record level locking
- Other forms of concurrency control to minimise contention



Replication

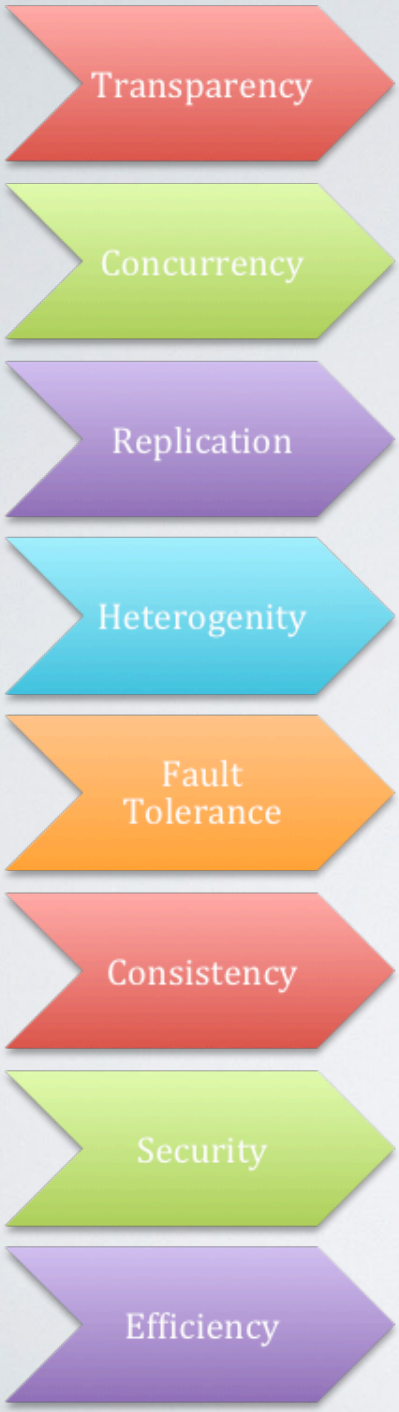
Properties

File service maintains multiple identical copies of files

- Load-sharing between servers makes service more scalable
- Local access has better response (lower latency)
- Fault tolerance

Full replication is difficult to implement.

Caching (of all or part of a file) gives most of the benefits (except fault tolerance)



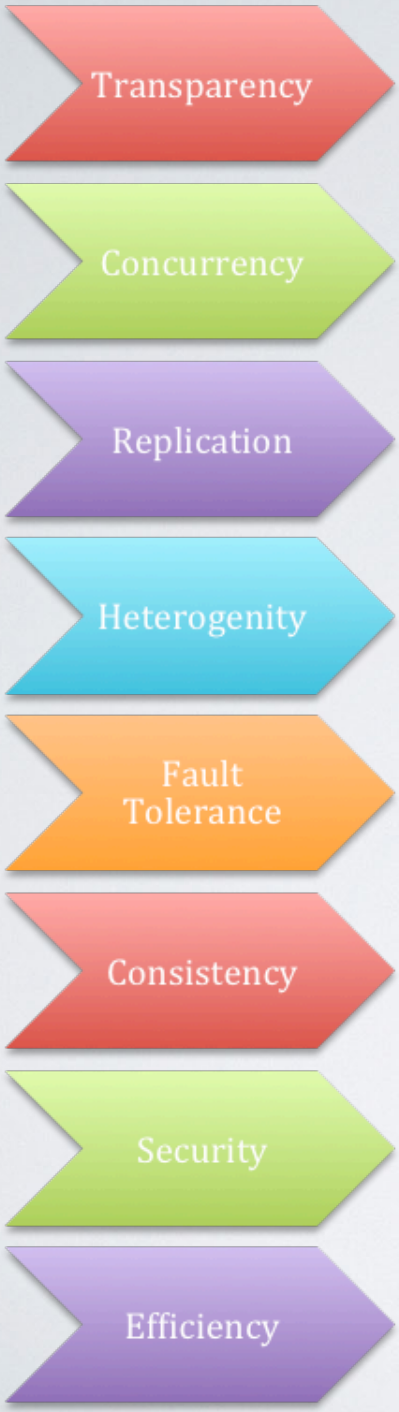
Heterogenity

Properties

Service can be accessed by clients running on (almost) any OS or hardware platform.

Design must be compatible with the file systems of different OSes

Service interfaces must be open - precise specifications of APIs are published.



Fault Tolerance

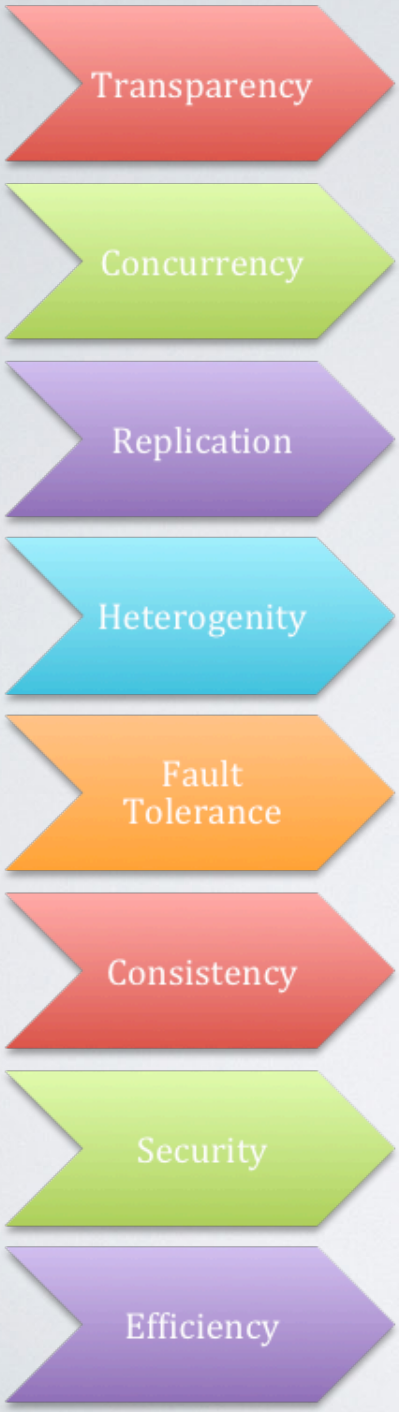
Fault Tolerance

Service must continue to operate even when clients make errors or crash.

- at-most-once semantics
- at-least-once semantics
- requires idempotent operations

Service must resume after a server machine crashes.

If the service is replicated, it can continue to operate even during a server crash.



Consistency

Unix offers one-copy update semantics for operations on local files

- caching is completely transparent.

Difficult to achieve the same for distributed file systems while maintaining good performance and scalability.

DISTRIB
SERVI

- Transparency
- Concurrency
- Replication
- Heterogenity
- Fault Tolerance
- Consistency
- Security
- Efficiency

Security

Must maintain access control and privacy as for local files.

- based on identity of user making request
- identities of remote users must be authenticated
- privacy requires secure communication

Service interfaces are open to all processes not excluded by a firewall.

- vulnerable to impersonation and other attacks

DISTRIB
SERVI

- Transparency
- Concurrency
- Replication
- Heterogenity
- Fault Tolerance
- Consistency
- Security
- Efficiency

Efficiency

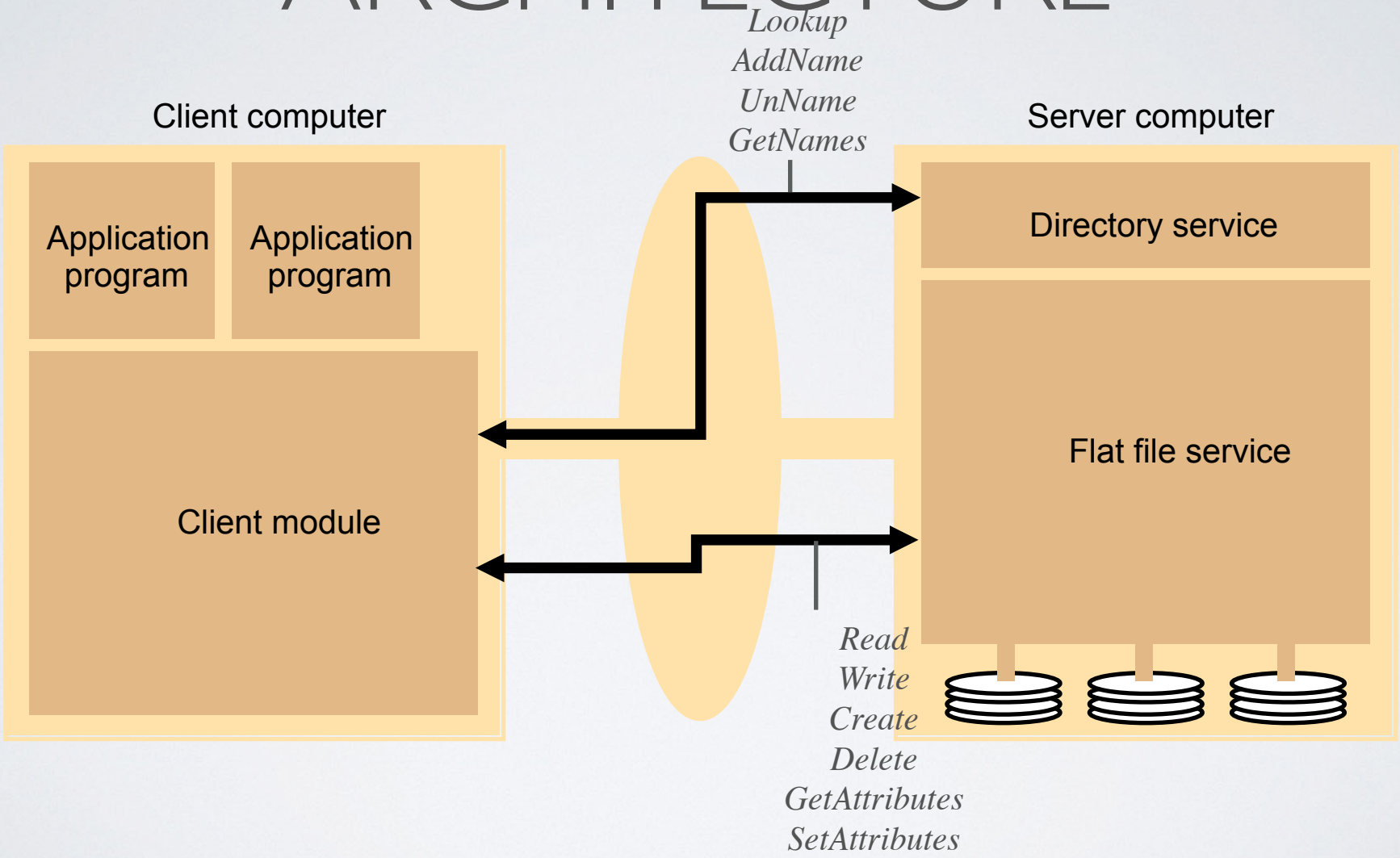
Goal for distributed file systems is usually performance comparable to local file system.

File service is most heavily loaded service in an intranet, so its functionality and performance are critical

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 Client module implements exported interfaces by flat file and directory services on server side.

MODEL FILE SERVICE ARCHITECTURE



RESPONSIBILITIES OF VARIOUS MODULES

Client

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.

Server

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.

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.

SERVER OPERATIONS/INTERFACES FOR THE MODEL FILE SERVICE

Figure 12.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file i
Create() → *FileId*	Creates a new file of length 0 and deliver
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attribut shaded in Figure 12.3).

Figure 12.7 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, removes the entry *Name* from the directory. If *Name* is not in the directory, throws an exce
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that regular expression *Pattern*.

Pathname lookup

Pathnames such as '/usr/bin/tar' are resolved by iterative calls to lookup(), one call for each component of the path, starting with the ID of the root directory '/' which is known in every client.

FileId

A unique identifier for files anywhere in the network. Similar to the remote object references described in Section 4.3.3.

HADOOP

- https://www.youtube.com/watch?v=9S9H6OzJ9N4

SUMMARY

- Distributed File systems provide illusion of a local file system and hide complexity from end users.
- Sun NFS is an excellent example of a distributed service designed to meet many important design requirements
- Effective client caching can produce file service performance equal to or better than local file systems
- Consistency versus update semantics versus fault tolerance remains an issue
- Most client and server failures can be masked

*